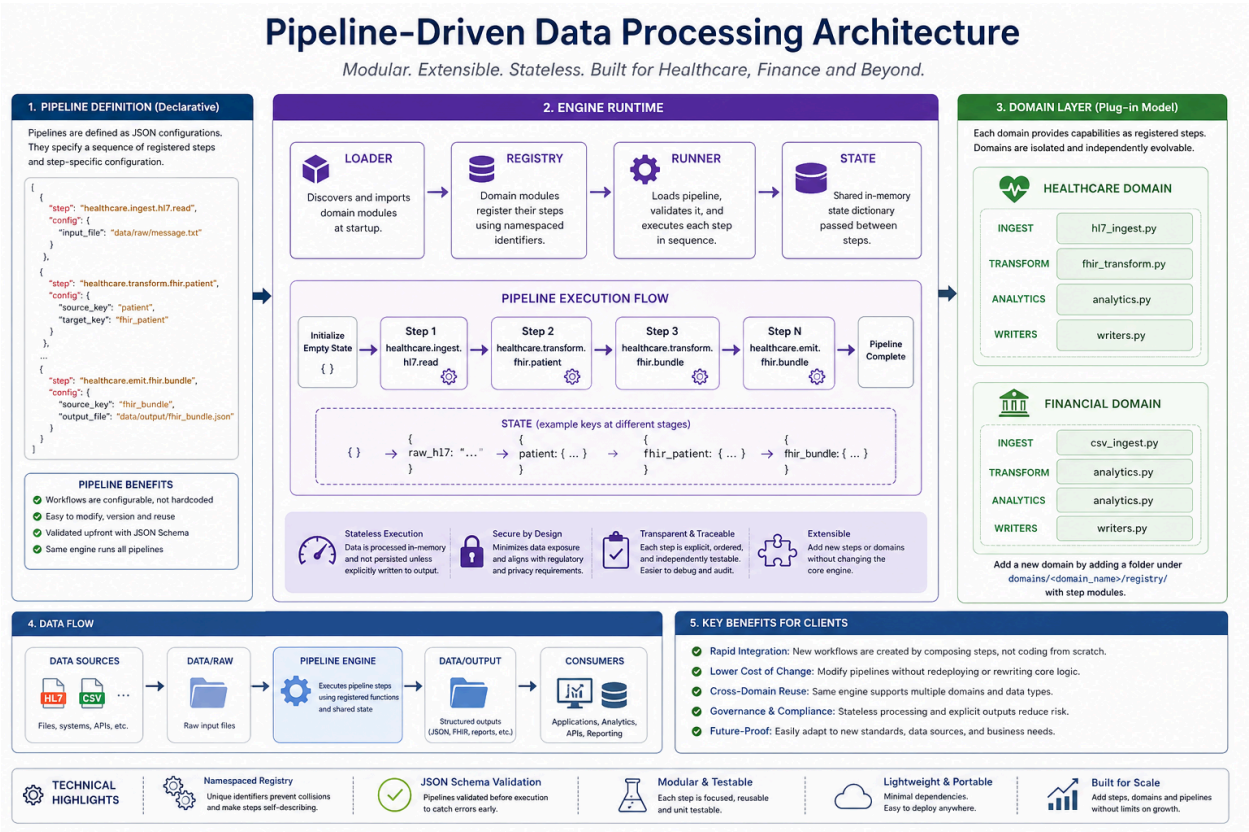
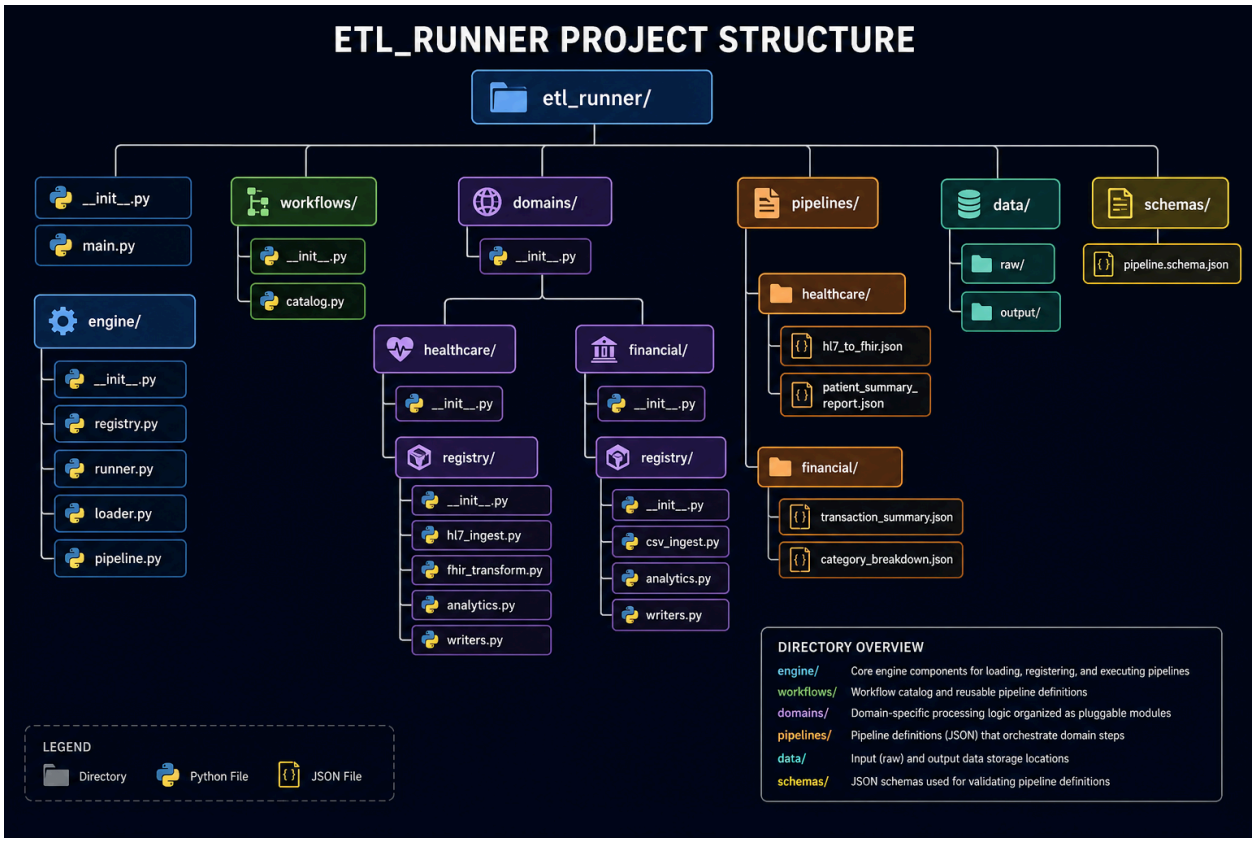
# Pyxgen Analytic Pipeline - New Direction




Pyxgen has built a **modular, stateless transformation engine** that separates orchestration, domain logic, and execution into clean, independent layers. At its core, the architecture revolves around the idea that data processing should be **pipeline-driven and declarative**, rather than

hardcoded. The `main.py` entry point acts as a thin orchestrator: it loads domain capabilities, reads a pipeline definition, and hands execution off to the engine. This keeps the control surface minimal while allowing the underlying system to evolve without constant changes to the entry point.

The app Structure




The **engine layer** is the backbone of the system. The `registry.py` module provides a centralized mechanism for registering processing functions (steps) under fully qualified, namespaced identifiers (e.g., `healthcare.ingest.hl7.read`). This registry acts as a lookup table that decouples pipeline definitions from implementation details. The `loader.py` dynamically discovers and imports domain modules, triggering registration of all available steps at runtime. This enables a plug-in model where new capabilities can be added simply by dropping new modules into a domain registry folder. The `runner.py` is responsible for executing pipelines sequentially: it iterates through each step defined in a pipeline JSON file, retrieves the corresponding function from the registry, and invokes it with a shared `state` object and step-specific configuration. The optional `pipeline.py` and schema layer formalize the structure of pipelines, enabling validation and consistency across executions.

The **domain layer** encapsulates all business and data-specific logic. Each domain (e.g., healthcare, financial) maintains its own `registry` submodule containing discrete,

single-responsibility processing steps such as ingestion, transformation, analytics, and output writing. These modules register their capabilities with the global engine registry using consistent naming conventions. This design ensures strict separation of concerns: domain teams can evolve their logic independently without impacting other domains or the core engine. Namespacing prevents collisions and makes pipelines self-describing, while the folder structure reinforces ownership boundaries and reduces the risk of unintended cross-domain coupling.

The **pipeline layer** defines workflows as external JSON configurations stored under `pipelines/`. Each pipeline is a declarative sequence of steps, where each step specifies a registered function and its configuration. This allows workflows to be composed, modified, and reused without touching code. Pipelines effectively act as the "program" while the engine is the "runtime." Because pipelines are data-driven, the same engine can execute vastly different workflows across domains—HL7-to-FHIR transformations in healthcare or transaction analytics in finance—without modification. The inclusion of a JSON schema (`schemas/pipeline.schema.json`) provides a mechanism for validating pipeline structure upfront, reducing runtime errors and enforcing consistency.

Data flows through the system via a shared, mutable **state object**, which acts as transient working memory. Each step reads from and writes to this state using well-defined keys (e.g., `raw_hl7`, `patient`, `fhir_patient`, `fhir_bundle`). This approach eliminates the need for persistent intermediate storage, aligning with a stateless processing model where data is ingested, transformed, and emitted within a single execution context. The `data/` directory serves only as an ingress/egress boundary—raw inputs are read from `data/raw/`, and results are written to `data/output/`—but the engine itself does not maintain long-term storage. This design is particularly well-suited for environments with strict data governance or privacy requirements, as data is not retained beyond processing.

The **workflow catalog layer** (`workflows/catalog.py`) provides a higher-level abstraction over pipelines, enabling grouping, discovery, and reuse of common processing patterns. This allows organizations to standardize workflows (e.g., patient normalization, financial reporting) while still leveraging the same underlying execution engine. It also opens the door to orchestration features such as scheduling, chaining, or conditional execution without altering the core engine.

Mechanically, the system operates as follows: when execution begins, the loader dynamically imports all domain registry modules, populating the global registry with available steps. The pipeline definition is then loaded and optionally validated against a schema. The runner initializes an empty state object and processes each step in order, passing state and configuration into each registered function. Each function performs a discrete transformation—such as parsing HL7, mapping to FHIR, aggregating financial data, or writing output—and returns the updated state. Once all steps complete, the pipeline terminates, having transformed input data into structured, domain-specific outputs without persisting intermediate artifacts.

Overall, this architecture achieves **extensibility, isolation, and reusability** through a combination of dynamic registration, namespaced functions, and declarative pipelines. It avoids tight coupling between components, minimizes changes to core infrastructure, and enables rapid development of new data workflows across domains. In practical terms, it functions as a lightweight integration engine—capable of ingesting heterogeneous data sources, normalizing them into analytical formats, and emitting structured outputs—while remaining simple enough to reason about and extend.