

PyxGen
PORTFOLIO

Lowell Buschert

Interoperability Engineer



1-321-352-9679 lbuschert@pyxgen.com

PyxGen Healthcare ETL Pipeline

A Decoupled, Containerized Platform for Healthcare Data Transformation

Overview

Healthcare data integration is rarely a simple matter of moving a file from one system to another.

Healthcare environments contain multiple data formats, legacy interfaces, inconsistent source structures, and large volumes of information that must be processed reliably without allowing computational workloads to interfere with the user interface or API layer.

The PyxGen Healthcare ETL Pipeline addresses this problem through a deliberately decoupled architecture.

The current platform separates the system into three independent layers:

1. **Web Dashboard — the presentation and operator interface**
2. **FastAPI Engine & Router — the API and orchestration layer**
3. **Docker Worker Engine — the isolated computational environment**

This separation allows the web interface to remain responsive while data processing occurs independently in an isolated worker environment.

The architecture is designed for scalability, operational reliability, data isolation, and deployment into customer-controlled infrastructure.

Architectural Separation of Concerns

The central architectural principle is simple:

> **The interface should not perform the work, and the API should not perform the heavy computation.**

The system therefore separates presentation, orchestration, and computation.

LAYER 1: WEB DASHBOARD

HTML5 / CSS3 / Vanilla JavaScript

Operator Interface • Job Status • Analytics

POST /run-pipeline

LAYER 2: FASTAPI ENGINE & ROUTER

REST API • Validation • Job UUID • Routing

Background Processing • Status Management

Docker Socket

LAYER 3: WORKER ENGINE

Docker Sandbox

Python • Pandas • OpenPyXL • HL7 → FHIR

Analysis • Output Generation • Ephemeral Execution

This three-layer architecture is the foundation of the current system.

Layer 1 — Web Dashboard

The presentation layer is intentionally lightweight.

Technology

- HTML5/ CSS3
- Vanilla JavaScript

The dashboard provides the operator with the controls required to select pipeline configurations and input data. It also provides the user-facing view of processing activity and results.

The browser does not execute the ETL workload. Instead, it communicates with the API layer through REST requests for commands and receives real-time updates via a persistent uni-directional stream. This is an important architectural distinction. The browser maintains only the state necessary for its current session and is not responsible for processing the underlying healthcare data.

Non-Blocking Status Monitoring (via Server-Sent Events)

One of the problems addressed by the current architecture is the traditional browser timeout and the inefficiencies of frequent short-polling. A large ETL process should not require a browser connection to block, nor should it flood the API with repetitive HTTP requests.

PyxGen uses an asynchronous job model paired with **Server-Sent Events (SSE)**. The browser submits the job via a standard REST request and receives a job identifier immediately. It then establishes a long-lived, push-based connection to the API to stream status updates in real time.

Browser -> POST /run-pipeline

FastAPI -> returns jobId

Browser -> GET /status/{job_id}/stream (EventSource connection)

FastAPI -> pushes "Event: RUNNING" (immediately on change)

FastAPI -> pushes "Event: SUCCESS / ERROR" (closes stream)

Layer 2 — FastAPI Orchestration Engine

The second layer is the API and orchestration engine.

Technology

- **Python 3.11**
- **FastAPI**
- **Uvicorn**
- **asynchronous request handling**

The FastAPI server performs several functions:

- **exposes REST endpoints**
- **validates incoming requests**
- **accepts pipeline configuration**
- **identifies the client**
- **generates job identifiers**

- **creates and tracks job state**
- **launches background processing**
- **starts worker containers**
- **monitors worker completion**
- **retrieves worker output**
- **exposes job status to the dashboard**

The API layer is deliberately lightweight.

It should coordinate processing rather than become the processing engine itself.

The current architecture is designed to return a job tracking token rapidly, allowing the heavy workload to be handled outside the HTTP request cycle.

Job-Based Execution

Every pipeline execution receives a unique UUID.

The initial request establishes a job state similar to:

```
{    "status": "RUNNING",    "result": null }
```

The API immediately returns the job identifier to the browser.

For example:

```
{  "jobId": "47c77d6b-31ea-4b4d-9422-955d9db024fa",   "job_id":
"47c77d6b-31ea-4b4d-9422-955d9db024fa" }
```

This identifier becomes the correlation key for the entire processing operation.

It connects:

- **the original request**
- **the worker process**
- **the processing state**
- **the resulting output**
- **the dashboard status request**

This is a conventional and reliable pattern for asynchronous processing systems.

Layer 3 — Computational Worker Sandbox

The actual ETL processing occurs in a separate Docker container.

The worker environment contains the processing runtime and executes the actual data transformation workload.

Current technology

- **Docker**
- **Python**
- **Pandas**
- **OpenPyXL**
- **healthcare transformation logic**

The worker is responsible for:

- **reading the input data**
- **executing the configured pipeline**
- **processing HL7 streams**
- **transforming data into FHIR structures**
- **checking input data for anomalies**
- **generating analytical results**
- **writing `result.json`**
- **recording execution information**

The API server does not perform these computational tasks itself.

That distinction is one of the most important aspects of the current architecture.

Ephemeral Worker Architecture

The worker container is designed to be ephemeral.

A worker is created for the processing task, performs its workload, writes its results, and exits.

The container is then removed.

Conceptually:

Job Request ->

Create Worker

Process Input ->

Generate Output ->

Exit Code 0

Retrieve Result ->

Remove Worker

This creates a clean boundary around individual processing operations.

Rather than keeping a long-running computational process loaded with customer data, the system creates an execution environment for the workload and disposes of it when processing is complete.

Data Processing Lifecycle

The current execution model follows a defined hand-off sequence.

1. Trigger

The operator selects the required pipeline and input data from the dashboard.

The browser submits:

POST /run-pipeline

with the required configuration and client identifier.

2. Tokenization

FastAPI generates a UUIDv4 job identifier and registers the initial job state.

3. Immediate Response

The API returns the job identifier to the browser without waiting for the ETL workload to complete.

4. Worker Creation

The API instructs Docker to create the worker container.

Input paths are translated into the appropriate container mounts.

5. Processing

The worker executes the pipeline and produces the output data.

6. Result Collection

When the worker exits successfully, the API verifies the generated result and loads the output into the job state.

7. Dashboard Update

The browser's next status request receives the completed result.

The polling cycle terminates and the dashboard renders the resulting analytics.

This complete lifecycle is explicitly defined in the current implementation.

API Interface

The current API provides a deliberately small interface between the dashboard and the processing engine.

Start a Pipeline

POST /run-pipeline

The request identifies:

- **the client**
- **the pipeline configuration**
- **the input file or data source**

Example:

```
{  "pipeline": "healthcare_ingest_config",   "params": {

   "input_file": "data/raw/clientA/dirty_hl7_test.txt"   }}
```

The API immediately returns the tracking identifier.

Monitor a Pipeline

GET /status/{job_id}

While processing:

```
{  "status": "RUNNING",   "result": null,   "error": null }
```

Once processing completes:

```
{ "status": "SUCCESS",   "result": { "records_processed": 150000,
"anomalies_detected": 14, "anomaly_rate_percent": 0.0093,  "output_file_path":
"data/output/result1.json"  },   "error": null }
```

This gives the dashboard a simple contract while keeping the actual ETL implementation behind the API boundary.

Healthcare Data Transformation

The computational layer performs the actual healthcare data processing.

One of its principal workflows is the transformation of HL7 data into FHIR structures.

The architecture therefore provides a pathway from legacy healthcare messaging environments toward standards-based FHIR output.

Conceptually:

```
HL7v2  ->
```

Ingestion ->

Parsing ->

Transformation ->

FHIR ->

Analysis / Output

The architecture is designed so that the API and user interface do not need to understand the internal details of each healthcare transformation.

That logic remains inside the worker processing environment.

Data Quality and Anomaly Detection

The worker is also responsible for examining incoming data for anomalies.

Rather than simply reporting that a pipeline completed, the resulting payload can provide operational information such as:

- **records processed**
- **anomalies detected**
- **anomaly rate**
- **output location**
- **execution status**

This allows the pipeline to function as both a transformation engine and a data-quality processing layer.

For example:

Records processed: 150,000

Anomalies detected: 14

Anomaly rate: 0.0093%

These measurements provide a useful operational view of the quality of the incoming dataset.

Deployment Model

The platform is designed to support deployment into customer-controlled infrastructure.

The current deployment model targets Linux servers with Docker.

Supported target environments described by the current deployment package include:

- **Ubuntu 22.04+**
- **Red Hat Enterprise Linux 8+**
- **Docker Engine CE**
- **Docker Compose V2**

This makes the platform suitable for organizations that want to keep the processing environment within their own infrastructure.

The deployment package can be installed under: /opt/etl_runner/

with the primary deployment components consisting of:

/opt/etl_runner/

docker-compose.yml

.env

etl-pipeline-bundle.tar

The current deployment documentation defines this as the expected on-site installation structure.

On-Premise Deployment

The deployment architecture is intentionally self-contained.

A customer can receive the application package and container image bundle and install the platform on its own Linux server.

The Docker Compose configuration provides:

- **the API/dashboard service**
- **raw data mounts**
- **output mounts**
- **Docker socket access for worker creation**
- **automatic service restart**
- **container logging controls**

The worker image can also be loaded from an offline image archive.

This provides a deployment path for environments where the customer wants greater control over the location and handling of its healthcare data.

File and Data Boundaries

The deployment separates raw input, generated output, and working space.

Conceptually:

/opt/etl_runner/

── data/

── raw/

── output/

── work/

── docker-compose.yml

── .env

── etl-pipeline-bundle.tar

The raw data directory provides the input channel.

The output directory receives generated analytical results.

The work directory provides temporary processing space.

This creates a clear operational boundary around the ETL workload.

Operational Resilience

The deployment configuration is intended to automatically restart the primary service if the host experiences a service interruption.

Container logging is also configured with limits on individual log-file size and the number of retained log files.

This prevents unrestricted container logging from gradually consuming server disk capacity.

These details are part of the current production deployment configuration rather than merely theoretical architectural considerations.

In-Memory Job Management

The orchestration layer currently maintains job state in memory rather than introducing a separate database dependency solely for job tracking.

This keeps the architecture lightweight.

However, an in-memory job registry has an obvious operational constraint: historical job records cannot be allowed to grow indefinitely.

The current design therefore incorporates a retention mechanism that limits the number of historical job records retained in memory.

The proposed implementation maintains the most recent 100 job footprints while avoiding removal of jobs that are still actively running.

This provides a simple bounded-memory model without introducing another persistence dependency.

Why the Architecture Is Decoupled

The architectural decisions are not simply about software organization.

They solve specific operational problems.

Browser reliability

The browser is not held open waiting for an ETL process to finish.

API responsiveness

FastAPI accepts the job and returns a tracking identifier rather than performing the computation synchronously.

Computational isolation

The ETL workload runs inside a separate Docker environment.

Data boundaries

Input and output are passed through controlled filesystem mounts.

Scalability

The presentation and orchestration layers are separated from the computational workload.

Maintainability

The API does not need to contain the implementation of every ETL transformation.

Deployment flexibility

The system can operate within customer-controlled Linux and Docker infrastructure.

Architectural Philosophy

The platform deliberately favors a straightforward engineering model.

There is no requirement for a large collection of proprietary services.

There is no requirement for an LLM to determine how the pipeline should execute.

There is no requirement for the web browser to participate in computation.

Instead, the architecture follows a conventional separation of responsibilities:

Presentation

↓

Orchestration

↓

Computation

Each layer has a defined responsibility.

That simplicity is an advantage.

A healthcare data pipeline must ultimately be predictable, observable, supportable, and explainable.

From Prototype to Commercial Platform

The current architecture represents a significant evolution from an experimental ETL prototype.

The emphasis has shifted toward:

- **operational separation**
- **asynchronous processing**
- **containerized execution**
- **explicit API boundaries**
- **job tracking**
- **data-quality metrics**
- **controlled deployment**
- **customer-managed infrastructure**

The result is no longer simply a Python ETL script.

It is an execution platform for configurable healthcare data pipelines.

The distinction matters.

A script performs a task.

A platform provides the infrastructure in which many tasks can be executed reliably.

Commercial Use Cases

The architecture can support a number of healthcare integration scenarios.

Healthcare Data Conversion

Transform legacy healthcare data into modern structured representations.

HL7-to-FHIR Processing

Process HL7 streams and generate FHIR structures for downstream interoperability applications.

Data Quality Assessment

Process large datasets and identify anomalies before the data reaches downstream systems.

Healthcare Analytics Preparation

Transform raw datasets into structured output suitable for analytics.

Customer-Specific ETL Pipelines

Deploy customer-specific pipeline configurations without changing the fundamental platform architecture.

On-Premise Healthcare Processing

Provide organizations with a self-contained processing environment operating inside their own infrastructure.

The PyxGen Value Proposition

PyxGen is built around a simple proposition:

Healthcare data processing should be modular, isolated, observable, and deployable

without forcing every customer into the same infrastructure model.

The current architecture provides:

A responsive interface

through a lightweight browser dashboard.

An asynchronous API

through FastAPI and Uvicorn.

Isolated computation

through ephemeral Docker workers.

Healthcare transformation

through Python-based processing and HL7-to-FHIR workflows.

Operational visibility

through job status and analytical metrics.

Customer-controlled deployment

through Docker Compose and on-premise Linux installation.

A clean architectural boundary

between presentation, orchestration, and computation.

Current Platform Architecture at a Glance

PYXGEN

HEALTHCARE ETL PLATFORM CLIENT

▼

WEB DASHBOARD (HTML / JS / CSS)

POST /run-pipeline



FASTAPI ORCHESTRATION

Job UUID Status

Routing

Docker Socket



DOCKER WORKER

Python Pandas

HL7 Processing FHIR Transform

Analytics



RESULT result.json

Metrics Anomalies



STATUS / RESULT

▼

WEB DASHBOARD DISPLAY

Conclusion

The PyxGen Healthcare ETL Pipeline has evolved into a deliberately decoupled processing platform.

Its architecture separates the user interface from API orchestration and separates both from the computational workload.

The result is a system that can accept a processing request, immediately return a job identifier, execute the workload inside an isolated Docker worker, collect the resulting analysis, and report the outcome back to the dashboard without requiring the browser or API connection to remain attached to the computation.

The architecture is intentionally practical.

It uses established technologies—Python, FastAPI, Docker, Pandas, REST APIs, and standard container orchestration—to solve a very real healthcare integration problem.

The objective is not to make the architecture unnecessarily sophisticated.

It is to make healthcare data processing reliable, isolated, observable, and deployable.

That is the foundation of the current PyxGen platform.