Dynamic ETL Pipeline for Unstructured Data — Detailed Testing Flow

Overview

This document defines a comprehensive, automated, and manual testing flow for evaluating participant submissions for the Dynamic ETL Pipeline challenge. The goal: verify a system that ingests .txt/.pdf/.md files containing mixed unstructured content (JSON, HTML, free text, tables), extracts and cleans important fields, generates a database schema (SQL/NoSQL/object), evolves the schema on each upload while maintaining backward compatibility, and exposes endpoints that allow queries (via LLM-translated queries) to run against the stored data.

Test Objectives

1. Validate ingestion of file types: .txt, .pdf, .md
2. Validate robust parsing: ability to detect and extract mixed formats (JSON, HTML, CSV, key-value, raw text) from a single file.
3. Validate data cleaning and canonicalization for extracted fields.
4. Validate automatic schema generation with explicit metadata describing target DB compatibility (e.g., PostgreSQL, MongoDB, Neo4j, object JSON schema).
5. Validate schema evolution across repeated uploads, with backward compatibility and migration strategy.
6. Validate re-ingestion of modified files results in appropriate schema updates and data migrations.
7. Validate API contract: endpoints for upload, schema retrieval, query execution.
8. Validate the LLM-driven query flow: end-to-end from natural language query -> LLM generates DB query -> platform executes query -> returns results.
9. Validate robustness, performance, logging, error handling, and security.

Test Environment & Tools

- Host (participant) provides a running service with these endpoints:
    - `POST /upload` - accepts multipart/form-data file upload and optional `source_id` and `version` fields.
    - `GET /schema?source_id=<id>` - returns current schema metadata and compatibility.
    - `GET /schema/history?source_id=<id>` - returns past schemas and change diffs.
    - `GET /records?source_id=&query_id=` - fetch results for executed queries if async.
    - Optional: `POST /migrate` for explicit migrations.
- **You can use multiple databases.**
- Versioning & storage:
    - Each `source_id` must map to a stable namespace. Files must produce deterministic schema updates for the same content.
    - System should produce change diff (add/remove/modify fields) with timestamped migrations.

Test Data Sets

Create multiple tiers of input files to exercise increasing complexity.

Tier A — Simple

1. Plain `.txt` containing `key: value` lines, small JSON blob, and a raw sentence paragraph.

2. `.md` file with headings, code block containing JSON, and an HTML snippet inside a code block.

Tier B — Mixed, messy

1. `.txt` mixing JSON fragments, an embedded HTML table, and CSV-like lines.

2. `.md` with frontmatter (YAML), nested lists, and inline HTML.

Tier C — Real-world noise

1. Scraped website save containing malformed HTML, comments, JS snippets, repeated fields.

2. PDF containing scanned text (OCR noise) plus embedded tables and footers/headers.

Tier D — Evolving changes

Start with a base file (Tier C). Produce incremental variants where:

- New fields are added (simple and nested)
- Types change (string -> number, number -> string)
- Field semantics change (e.g., `price_usd` renamed to `price` with currency field added)
- Arrays become objects and vice versa
- Deleted fields

Provide at least 6 variants per base to test multiple evolution steps.

Acceptance Criteria

- **Parsing**: The system correctly extracts structured fragments (JSON/CSV/HTML tables) and identifies likely key-value pairs from free text.

- **Schema generation**: The generated schema must declare the database types it supports and include field types, nested structures, primary keys (if any), index suggestions, and confidence scores for each field.

- **Schema evolution**: On re-upload, the schema history must show diffs, and data migration must either preserve prior data or provide a clear migration plan. Backward compatibility must be demonstrable by successfully running historical queries against current storage when feasible.

Note: exact numeric thresholds are configurable depending on difficulty level.

Detailed Test Flow

This is the step-by-step execution plan for the automated evaluator.

Phase 0 — Sanity & Connectivity

1. Validate endpoints exist and follow the required contract.

2. Check MIME handling: `upload` must accept `application/pdf`, `text/plain`, `text/markdown`, and `multipart/form-data`.

Phase 1 — Ingest & Parse Tests

For each test file in Tier A/B/C:

1. Call `POST /upload` with `source_id` and file.

2. Expect HTTP 200/201 and a JSON response: `{source_id, file_id, schema_id, parsed_fragments_summary}`.

3. Verify parsed_fragments_summary lists detected fragments (e.g., `json_fragments: 2`, `html_tables: 1`, `kv_pairs: 12`) and locations/offsets in file.

4. Pull extracted records using `GET /records?source_id=&query_id=raw_extract` or equivalent API. Validate the extracted structured fragments against ground truth (for synthetic tests). Score extraction precision/recall.

5. For PDFs, ensure OCR is applied (or clearly flagged if OCR not supported). If OCR used, validate text quality threshold. **Now this depends on how you've designed your system.**

Phase 2 — Schema Generation Tests

1. After upload verify `GET /schema?source_id=` returns schema metadata in a canonical representation. Schema should include:
    - `schema_id`, `generated_at`, `compatible_dbs`: ["postgresql","mongodb"],
    - `fields`: list with `name`, `path`, `type`, `nullable`, `example_value`, `confidence`, `source_offsets`, `suggested_index`
    - `primary_key_candidates`
    - `migration_notes` (if applicable)
2. Validate types mapping: e.g., date-like strings flagged as `DATE` for SQL and `date` for NoSQL with iso format suggestion.
3. Validate the provider's claim about `compatible_dbs` by attempting to ingest converted data into those DBs or by using a provided SQL DDL / NoSQL collection schema.

Phase 3 — Evolving Schema Tests

1. Upload variant file `v2` that includes structural changes.
2. Expect new schema returned. Call `GET /schema/history?source_id=` and verify diff includes added/removed/changed fields with semantic notes.
3. Validate backward compatibility strategy:
    - If the system applies migrations, confirm that previous records remain queryable with older schema semantics, or
    - If system uses versioned collections/tables, ensure queries referencing older schema versions are routed correctly.

4. Run regression queries that were valid before `v2`. Check results remain correct (or flagged with graceful degradation when incompatible).

Phase 4 — Type-change & Ambiguity Tests

1. Upload variant where a field `amount` changes type from string to numeric and sometimes contains `N/A`.

2. The system must:

    - Detect mixed-types and create a union type or canonical type with nullable/nullable reason.

    - Add `data_quality` metadata for fields with ambiguous types.

    - Provide a migration or normalization plan (e.g., `attempt_cast` rules or `coalesce` logic).

3. Validate queries that filter by `amount` handle both types after normalization.

Phase 5 — Schema Mapping to Target DBs

1. For each `compatible_db` declared in the schema metadata, the system must provide either:

    - SQL DDL (for relational) and/or

    - NoSQL collection JSON schema (for Mongo-like) and/or

    - Graph schema (for Neo4j) and/or

    - Object schema (JSON Schema / Avro / Protobuf)

2. The evaluator will attempt to create artifacts on test DBs. Confirm successful ingestion and ability to run the expected queries.

Phase 6 — LLM-driven Query Flow

1. Tester will issue a set of natural language queries for each dataset (search, filter, aggregations, joins, nested access).

2. The test harness will send NL to an LLM (configurable). LLM returns a DB query in the target dialect (based on schema compatibility). Optionally, participants may provide their own LLM or prompt templates.

3. The harness posts `{source_id, nl_query}` to `POST /query` (or provides `llm_query` directly if LLM run externally) and expects:

    - immediate synchronous result with rows OR

    - an async `query_id` that can be polled with `GET /records?source_id=&query_id=`.

4. Validate correctness of returned rows against expected results. Score success rate.

Phase 7 — Re-upload & Idempotency Tests

1. Re-upload same file (same content) multiple times. System must not create duplicate schema versions unnecessarily. Should return same `schema_id` or increment version deterministically.

2. Upload file with trivial reformatting (whitespace, character encoding changes). Verify no schema churn.

Phase 8 — Stress & Performance (Optional advanced)

1. Batch upload 100 files concurrently. Measure ingestion latency and schema generation time.

2. Large file test: 100MB with many embedded fragments. Monitor memory usage and failure modes.

Phase 9 — Logging, Observability & Error Handling

1. System must provide structured logs for parsing, schema generation, and migration steps.

Phase 10 — Security & Privacy

1. Validate that uploaded files are stored securely and deleted according to retention policy.
2. Check for injection vulnerabilities in parsing and query execution. E.g., an embedded `DROP TABLE` inside extracted SQL must not execute.
3. Validate access controls for schema and record endpoints.

Automated Evaluation Script (Reference)

Provide a recommended harness outline (Python pseudocode) participants should implement or the evaluator will use. This harness will:

1. Spin up test DB containers (Postgres, Mongo, Neo4j) or use test equivalents.
2. For each sample file:
    - Upload to `/upload`
    - Poll `GET /schema` until schema appears
    - Validate schema fields vs ground truth
    - Run a predefined set of NL queries: send to LLM, get DB query, call `/query`, compare results
3. Keep a scoreboard with weighted scoring across categories: parsing(30%), schema(25%), evolution(20%), LLM-query correctness(15%), robustness/logging/security(10%).

Include sample pseudocode for LLM integration. Suggest using a small deterministic LLM simulator for offline grading (e.g., map certain NL templates -> canned SQL) to avoid API variability.

Expected API Contracts (Examples)

POST /upload request (multipart):

POST /upload
Content-Type: multipart/form-data
fields: source_id (string), file (binary), metadata (json optional)

POST /upload response example:


```
{
  "status": "ok",
  "source_id": "site_abc",
  "file_id": "file_20251115_001",
  "schema_id": "schema_v7",
  "parsed_fragments_summary": {"json_fragments":2, "html_tables":1, "kv_pairs":12}
}
```


GET /schema?source_id=site_abc response example (canonical):


```
{
  "schema_id":"schema_v7",
  "generated_at":"2025-11-15T08:00:00Z",
  "compatible_dbs":["postgresql","mongodb"],
  "fields":[
    {"name":"title","path":"$.title","type":"string","nullable":false,"example":"Widget
A","confidence":0.98},

{"name":"price","path":"$.pricing.price","type":"decimal","nullable":true,"example":9.99,"confidenc
e":0.87}
  ],
  "primary_key_candidates":["id","slug"],
  "migration_notes":"Added pricing.price as decimal, previous price stored as string in
legacy_price"
}
```

Common Failure Modes & Grading Mitigations

- **Overfitting to test files**: Use multiple unseen randomized test variants during evaluation.

- **LLM nondeterminism**: Use canonical prompt templates or a simulated LLM for objective grading.
- **Schema churn for insignificant changes**: Require participants to implement semantic change detection vs syntactic.
- **Mixed-type chaos**: Expect union types with normalization strategies; penalize systems that silently drop data.

Appendix: Example Test Cases (Concrete)

1. **Simple JSON embed**: `.txt` with inline JSON object and trailing paragraph. Expect schema with fields matching object keys.
2. **HTML table + CSV**: `.txt` containing an HTML `<table>` and a CSV section. Extract both into separate record collections/tables and reference them in schema metadata.
3. **PDF with OCR noise**: PDF with dates like `12/13/2025` and `13/12/2025`—system must infer correct date format or provide confidence & normalization policy.
4. **Evolution - rename**: `v1` has `price_usd`; `v2` has `price` + `currency`. System must update schema and show migration notes mapping `price_usd` -> `price` with currency `USD`.
5. **Type flip with nulls**: `v1` `views` are integers; `v2` `views` contains `N/A` and numbers. System must support union/nullable and normalization.

Example Mock Data

```
# mock_input.txt
# Source: example-scrape
# scraped_at: 2025-11-10T14:22:00+05:30

--- METADATA (key: value lines)
source: https://example.com/product/widget-a
scraper: simple-scraper-v1
lang: en
```

```
publisher: Example Corp
contact: support@example.com

--- RAW PARAGRAPH
Widget A is a compact device for everyday use. It comes in multiple
colors and often appears in scraped pages with noisy markup and
unrelated text like promotional banners, comments, or code snippets.
The price information is inconsistent in these pages and sometimes
appears as "9.99 USD", "$9.99", or "9,99". Dates observed in
different locales: 12/13/2025 and 13/12/2025 (ambiguous). Some text
contains OCR errors from PDFs: \"l0cation\" instead of \"location\"
and \"O\" vs \"0\" confusion.

--- INLINE JSON (well-formed)
{
  "id": "prod-1001",
  "title": "Widget A",
  "slug": "widget-a",
  "pricing": {
    "price_usd": "9.99",
    "inventory": 120,
    "currency_hint": "USD"
  },
  "tags": ["gadget","home","widget"],
  "dimensions": {"w_mm": 120, "h_mm": 45, "d_mm": 30},
  "release_date": "2025-11-01"
}

--- MALFORMED JSON FRAGMENT (common in scraped text)
{ "id": "prod-1001-b", "title": "Widget B", "specs": { "color":
"red", "weight": "0.5kg", }  "notes": "missing comma and trailing
comma issues"

--- HTML SNIPPET (embedded table)
<div class="reviews">
  <h3>Customer Reviews</h3>
  <table>

<thead><tr><th>author</th><th>rating</th><th>comment</th><th>date</th
></tr></thead>
    <tbody>
      <tr><td>Alice</td><td>5</td><td>Excellent
product.</td><td>2025-10-20</td></tr>
      <tr><td>Bob</td><td>4</td><td>Good value for
money.</td><td>20/10/2025</td></tr>
```

```
      <tr><td>Charlie</td><td>3</td><td>Okay, but packaging was
bad.</td><td>Oct 19, 2025</td></tr>
    </tbody>
  </table>
</div>

--- CSV-LIKE SECTION
author,rating,helpful_votes,date
Dave,4,2,2025-10-18
Eve,2,0,18-10-2025
Mallory,5,10,2025/10/17

--- KEY-VALUE KVP BLOCK (no fixed structure)
title: Widget A - Special Edition
price: $9.99
currency: USD
availability: In Stock
tags: gadget;home;clearance

--- JSON-LD (schema.org style)
<script type="application/ld+json">
{
  "@context": "http://schema.org/",
  "@type": "Product",
  "name": "Widget A",
  "image": [
    "https://example.com/images/widget-a-1.jpg",
    "https://example.com/images/widget-a-2.jpg"
  ],
  "description": "A versatile widget for the modern home.",
  "sku": "WA-1001",
  "offers": {
    "@type": "Offer",
    "priceCurrency": "USD",
    "price": "9.99",
    "availability": "http://schema.org/InStock",
    "url": "https://example.com/product/widget-a"
  }
}
</script>

--- INLINE CSV TABLE (tabular HTML converted text captured)
ProductID,Name,Color,Stock
prod-1001,Widget A,black,120
prod-1001,Widget A,white,80
```

```
prod-1002,Widget B,red,50

--- FREE TEXT WITH NOISE & JS SNIPPET
<!-- Some scraped page contains inline scripts and comments -->
<script>var config = {id: 'prod-1001', price: '9.99', promo:
true};</script>
Random footer text - Contact us at (555) 123-4567. Promo code:
SAVE10.
Note: DO NOT RUN SQL: \"DROP TABLE users;\" included as sample text
from scraped code blocks.

--- OCR-LIKE PAGE FOOTER
Page 3 of 12
Document title: Product catalog - Example Corp
l0cation: Warehouse 9
Total items: one hundred and twenty (120)

--- SQL-LIKE SNIPPET (should be extracted as code, not executed)
-- Example of raw SQL shown on a page, should not be executed
SELECT id, title, price FROM products WHERE price < 20;

--- REPEATED FIELDS (conflicting values across page)
price_usd: 9.99
price: "$9.99"
legacy_price: "9.9900"
currency_hint: USD

--- AMBIGUOUS TYPES / MISSING DATA
views: "1024"
views: "N/A"
sold: "12"
rating_avg: "4.2"
is_featured: "true"
is_limited: 0

--- VARIANT / EVOLUTION BLOCK (v2) - show how source may change over
time
=== VARIANT v2 START ===
# New upload on 2025-11-20: product schema changed
{
  "id": "prod-1001",
  "title": "Widget A",
  "slug": "widget-a",
  "pricing": {
    "price": 9.99,
```

```
    "currency": "USD"
  },
  "tags": ["gadget","home","widget","sale"],
  "dimensions": {"w_mm": 120, "h_mm": 45, "d_mm": 30},
  "release_date": "2025-11-01",
  "metadata": {
    "imported_from": "example-scrape-v2",
    "ingested_at": "2025-11-20T09:10:00+05:30"
  },
  "views": "N/A",              # type flip: previously numeric, now
sometimes N/A
  "ratings": {
    "avg": 4.1,
    "count": 235
  }
}

# Extra CSV row introduced in v2
ProductID,Name,Color,Stock,warehouse
prod-1001,Widget A,black,120,W-9
prod-1001,Widget A,white,80,W-9
prod-1003,Widget C,blue,30,W-12

# New front-matter like YAML (in v2 some pages include it)
---
title: "Widget A - 2025"
published: 2025-11-18
authors:
  - "Editorial Team"
tags:
  - "gadget"
  - "featured"
---

=== VARIANT v2 END ===

--- END OF FILE ---
```