TestDataService Framework - Technical Architecture & Structure

1. Overview

TestDataService is a modular, configuration-driven framework designed to intelligently fetch and validate test data across Oracle and SQL Server (DWH) databases using Large Language Model (LLM) assistance. It automates SQL generation, schema validation, and cross-DB data correlation — driven entirely by feature files, rules, and config settings, with zero code modification.

---

2. Folder & File Structure

```
TestDataService/
├── .env.example
├── .gitignore
├── README.md
├── requirements.txt
├── config.json
├── rules.json
│
├── features/
│   └── user_login.feature
│
├── schema/
│   ├── oracle_schema.json
│   └── dwh_schema.json
│
├── history/
│   └── query_history.json
│
├── output/
│   ├── oracle/
│   └── dwh/
│
└── src/
    ├── app.py
    ├── connectors/
    │   ├── oracle_connector.py
    │   └── dwh_connector.py
    ├── schema_extractors/
    │   ├── oracle_schema_extractor.py
```

```
    │   └── dwh_schema_extractor.py
    ├── executors/
    │   ├── oracle_executor.py
    │   └── dwh_executor.py
    ├── query_generators/
    │   ├── oracle_query_generator.py
    │   └── dwh_query_generator.py
    ├── validators/
    │   ├── oracle_query_validator.py
    │   └── dwh_query_validator.py
    ├── utils/
    │   ├── io_utils.py
    │   ├── sql_utils.py
    │   └── schema_utils.py
    ├── parsers/
    │   └── feature_parser.py
    └── services/
        └── llm_client.py
```

---

3. Conceptual Architecture

```
Feature File (.feature)
        │
        ▼
Feature Parser ──────► Extract member_type + criteria
        │
        ▼
app.py (Orchestrator)
  │
  ├── Oracle Flow
  │     ├─ Connect
  │     ├─ Generate SQL via LLM
  │     ├─ Validate
  │     ├─ Execute & Save JSON
  │
  └── DWH Flow
        ├─ Use Oracle output
        ├─ Generate DWH SQL via LLM
        ├─ Validate
        ├─ Execute & Save JSON

All runs logged → history/query_history.json
```

4. Core Configuration Files

File	Purpose	Editable
`.env`	All connection, owner, schema, table, batching, LLM & DWH settings	✅yes
`config.json`	Holds example SQL templates (active_members, registered_members, dwh_query)	✅yes
`rules.json`	Member type mapping: DB/fund code pairs + condition definitions	✅yes
`.env.example`	Template for reference	✅yes
`schema/*.json`	Auto-generated schema snapshots	❌no
`history/query_history.json`	Auto-maintained execution log	❌no

5. Execution Flow (Full Run)

Command:

```
python -m src.app --feature features/user_login.feature
```

Step-by-step:

1. Parse `.feature` file for `member_type` and `member_criteria`
2. Load `.env`, `rules.json`, `config.json`
3. Auto-extract schemas if missing (Oracle + DWH)
4. Generate & validate SQL (Oracle → Active → Registered)
5. Use Oracle results as input for DWH query
6. Execute DWH SQL → Fetch results → Save JSON
7. Append summary to `history/query_history.json`

6. Partial Execution Modes

Flag	Description
`--test-oracle`	Tests Oracle connectivity
`--test-dwh`	Tests SQL Server (DWH) connectivity

Flag	Description
`--extract-oracle-schema`	Extract Oracle schema only
`--extract-dwh-schema`	Extract DWH schema only
`--fetch-active --member-type accum`	Fetch only active members

---

7. Output Structure

Folder	Description	Auto-created
`output/oracle/`	Active and registered Oracle results	✅
`output/dwh/`	DWH data results	✅
`history/query_history.json`	Run metadata log	✅

Example:

```
output/
├── oracle/
│   ├── oracle_candidates_example1.json
│   └── oracle_active_accum_20251016_102311.json
└── dwh/
    ├── dwh_result_example1_20251016_102356.json
history/
└── query_history.json
```

---

8. Data Flow Summary

Step	Input	Processor	Output
1	`.feature` file	`feature_parser`	member_type + criteria
2	`.env`, `rules.json`, `config.json`	`app.py` orchestrator	environment setup
3	Oracle connection	`oracle_connector`, `oracle_query_generator`	active + registered members
4	DWH connection	`dwh_connector`, `dwh_query_generator`	insurance or condition results

Step	Input	Processor	Output
5	History logging	`io_utils.append_history`	cumulative record JSON

---

9. Extensibility & Scalability

- 💹Add new databases: just create new connector/extractor/validator.
- Add rules in `rules.json` for new member types.
- 9 Add templates in `config.json` for new query cases.
- 🛑Integrate with CI/CD by using CLI flags.

Future Enhancements:

- Query caching for repetitive requests.
- Parallel/async batch execution.
- Automatic schema refresh jobs.
- Web dashboard or API for query execution.

---

10. Design Principles

Category	Principle
Structure	Modular (connectors, executors, validators separated)
Config-driven	No hardcoded SQL — all templates in config/rules/.env
LLM intelligence	Query generation adapts to schema, rules, and examples
Self-managing	Auto-creates folders, schema, and history files
Portable	Works across Windows/Linux, Oracle/SQL Server
