# Oracle Data Loading Framework — Design & Implementation

**Purpose:** A production-ready, memory-efficient Python framework to read files and load data into an Oracle database, with automatic table creation from DDL/schema files and robust operational features.

## Executive Summary

This document describes an optimized approach and a ready-to-run framework for loading large datasets into Oracle DB using Python. It focuses on reliability, performance, and maintainability: connection pooling, chunked/bulk loading, automatic schema creation from DDL/JSON, structured logging, and configurable behavior.

## Key Optimization Features

* Connection Pooling — reuse DB connections to reduce overhead and improve throughput.
* Bulk Insert with executemany — insert rows in configurable batches (default: 10,000).
* Chunked Processing — read and process files in chunks to keep memory constant.
* Context Managers — ensure deterministic cleanup of connections and file handles.
* Multi-format Support — CSV, Excel, JSON(.jsonl), Parquet, TXT.
* Automatic Schema Creation — create tables from DDL files or JSON schema definitions.
* File Tracking & Error Handling — move processed/failed files and keep detailed logs.

## Recommended Folder Structure

oracle\_loader/  
├── config/  
│ └── config.yaml # DB credentials & global settings  
├── schemas/ # DDL and JSON schema files (\*.sql, \*.json)  
├── data/  
│ ├── input/ # Incoming files to load  
│ ├── processed/ # Successfully loaded files  
│ └── failed/ # Failed files + error artifacts  
├── logs/ # Application logs  
├── src/ # Core modules  
│ ├── db/  
│ │ ├── connection.py  
│ │ └── ddl\_parser.py  
│ ├── ingestion/  
│ │ ├── reader.py  
│ │ └── loader.py  
│ ├── utils/  
│ │ ├── file\_mover.py  
│ │ └── schema\_utils.py  
│ └── config.py  
└── main.py # CLI / entry point

## Quick Start

**Prerequisites:** Python 3.9+, Oracle Instant Client or cx\_Oracle/ oracledb configured, and dependencies installed.  
  
**Install dependencies:Configure** config/config.yaml with your Oracle connection and file settings, place DDL/schema files in schemas/, drop files into data/input/ and run:

pip install oracledb pandas openpyxl PyYAML sqlparse python-dotenv python-docx

python main.py

## Example config/config.yaml

oracle:  
 user: ORA\_USER  
 password: ORA\_PASSWORD  
 dsn: host:1521/service\_name  
 min\_sessions: 1  
 max\_sessions: 5  
  
file\_settings:  
 chunk\_size: 10000  
 batch\_insert\_size: 5000  
 processed\_folder: data/processed  
 failed\_folder: data/failed  
  
logging:  
 level: INFO  
 file: logs/oracle\_loader.log

## Automatic Schema Creation (DDL / JSON)

The framework supports two ways to create tables automatically before loading data:  
1. Raw DDL SQL files (.sql) placed in the schemas/ directory. The framework will parse and execute CREATE TABLE statements.  
2. JSON schema definitions describing columns and types (useful for programmatic table creation when DDL is not available).  
  
A DDL parser module will sanitize SQL, ensure idempotence (CREATE TABLE IF NOT EXISTS semantics emulated when needed), and apply type mappings from input file types to Oracle types.

## Core Implementation — Select Snippets

Below are representative, ready-to-use code snippets for the most important modules. These are concise; full code is recommended to live in src/ as per the folder structure.

### src/db/connection.py

import oracledb  
from contextlib import contextmanager  
import yaml  
  
class OraclePool:  
 def \_\_init\_\_(self, cfg):  
 oracledb.init\_oracle\_client() # optional if using Instant Client  
 self.pool = oracledb.SessionPool(  
 user=cfg['oracle']['user'],  
 password=cfg['oracle']['password'],  
 dsn=cfg['oracle']['dsn'],  
 min=cfg['oracle']['min\_sessions'],  
 max=cfg['oracle']['max\_sessions'],  
 increment=1,  
 threaded=True  
 )  
  
 @contextmanager  
 def get\_conn(self):  
 conn = self.pool.acquire()  
 try:  
 yield conn  
 finally:  
 try:  
 self.pool.release(conn)  
 except Exception:  
 conn.close()

### src/ingestion/reader.py (chunked reading)

import pandas as pd  
  
def read\_file\_in\_chunks(path, chunk\_size=10000):  
 path = str(path)  
 if path.endswith('.csv') or path.endswith('.txt'):  
 for df in pd.read\_csv(path, chunksize=chunk\_size, iterator=True, dtype=str, keep\_default\_na=False):  
 yield df  
 elif path.endswith('.parquet'):  
 import pyarrow.parquet as pq  
 for batch in pq.ParquetFile(path).iter\_batches(batch\_size=chunk\_size):  
 yield batch.to\_pandas()  
 elif path.endswith('.json') or path.endswith('.jsonl'):  
 for chunk in pd.read\_json(path, lines=True, chunksize=chunk\_size):  
 yield chunk  
 elif path.endswith('.xlsx') or path.endswith('.xls'):  
 # Excel doesn't support streaming well; read in one shot, then yield slices  
 df = pd.read\_excel(path, dtype=str)  
 for i in range(0, len(df), chunk\_size):  
 yield df.iloc[i:i+chunk\_size]  
 else:  
 raise ValueError('Unsupported file type: ' + path)

### src/ingestion/loader.py (bulk insert with executemany)

def insert\_dataframe(conn, table\_name, df, batch\_size=5000):  
 cols = list(df.columns)  
 col\_list = ','.join(cols)  
 placeholders = ','.join([':'+str(i+1) for i in range(len(cols))])  
 sql = f"INSERT /\*+ APPEND \*/ INTO {table\_name} ({col\_list}) VALUES ({placeholders})"  
  
 data = [tuple(None if x=='' else x for x in row) for row in df.itertuples(index=False, name=None)]  
 cursor = conn.cursor()  
 for i in range(0, len(data), batch\_size):  
 batch = data[i:i+batch\_size]  
 cursor.executemany(sql, batch)  
 conn.commit()  
 cursor.close()

### Schema Execution (ddl\_parser.py)

def execute\_ddl(conn, sql\_text):  
 # sanitize and split; sqlparse can help to split safely  
 import sqlparse  
 stmts = sqlparse.split(sql\_text)  
 cur = conn.cursor()  
 for s in stmts:  
 s = s.strip()  
 if not s:  
 continue  
 try:  
 cur.execute(s)  
 except Exception as e:  
 # Ignore "table already exists" or rethrow depending on policy  
 if 'ORA-00955' in str(e):  
 # table exists  
 continue  
 else:  
 raise  
 cur.close()

## main.py (entry point)

from pathlib import Path  
from src.db.connection import OraclePool  
from src.ingestion.reader import read\_file\_in\_chunks  
from src.ingestion.loader import insert\_dataframe  
import yaml  
  
def main():  
 cfg = yaml.safe\_load(open('config/config.yaml'))  
 pool = OraclePool(cfg)  
  
 input\_dir = Path('data/input')  
 for file\_path in input\_dir.iterdir():  
 try:  
 # optionally run DDL if a schema file is present  
 schema\_file = Path('schemas') / (file\_path.stem + '.sql')  
 if schema\_file.exists():  
 with pool.get\_conn() as conn:  
 sql = schema\_file.read\_text()  
 from src.db.ddl\_parser import execute\_ddl  
 execute\_ddl(conn, sql)  
  
 # stream and insert  
 for chunk in read\_file\_in\_chunks(file\_path, chunk\_size=cfg['file\_settings']['chunk\_size']):  
 with pool.get\_conn() as conn:  
 insert\_dataframe(conn, file\_path.stem, chunk, batch\_size=cfg['file\_settings']['batch\_insert\_size'])  
  
 # move file to processed  
 from src.utils.file\_mover import move\_to\_processed  
 move\_to\_processed(file\_path, cfg['file\_settings']['processed\_folder'])  
 except Exception as e:  
 from src.utils.file\_mover import move\_to\_failed  
 move\_to\_failed(file\_path, cfg['file\_settings']['failed\_folder'], error=str(e))  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 main()

## Scalability, Monitoring & Best Practices

* Tune chunk\_size and batch\_insert\_size according to available memory and database capacity.
* Use Oracle indexing and partitioning strategies for large tables to improve load and query performance.
* Implement retries with exponential backoff for transient errors (network, DB throttling).
* Integrate observability: structured logs, Prometheus metrics, and alerts on failures or high-latency operations.
* Secure credentials using vaults or environment variables; avoid committing secrets to source control.
* Add schema validation to map incoming columns to the expected table columns before writing.