Line-by-Line Explanation of `run_bronze_load.py`

As a fellow data professional, you'll see this script is structured into three distinct parts:

1. **Configuration:** Setting up logging and loading credentials.
2. **Helpers:** Reusable functions for writing to our `etl_log` table.
3. **Main Logic:** The core `try...except...finally` block that orchestrates the entire ETL process.

Here is the detailed breakdown.

Imports

These are the libraries (tools) we need from Python's standard library and our `requirements.txt` .

```
import mysql.connector  # The main driver that lets Python talk to MySQL.
import configparser     # (No longer used, but was in the previous version for .ini)
import logging          # Python's built-in logging module.
import time             # Used to time operations for bottleneck analysis.
import sys              # Used for `sys.exit(1)` to signal an error to the OS.
import os               # Used for `os.getenv()` (getting env variables) and `os.path.r
from datetime import datetime   # Used for precise timestamps in the log table.
from dotenv import load_dotenv  # The function that reads our .env file.
```

Section 1: Configuration & Logging

This section prepares the script to run.

`setup_logging()` Function

This function's role is to set up a robust logger. A key best practice here is **dual logging**:

1. **Console (StreamHandler):** For you to see real-time status.
2. **File (FileHandler):** For a persistent file (`bronze_load.log`) you can review for errors after an automated run.

```
def setup_logging():
    # Gets or creates a logger instance. Using a name prevents
    # conflicts with other libraries' "root" loggers.
    logger = logging.getLogger('bronze_etl')
    # We set the minimum level to log. INFO is good for production.
    # (Use DEBUG for more verbose troubleshooting).
    logger.setLevel(logging.INFO)

    # This is a critical line to prevent duplicate log messages if
    # the script is run in an environment (like Jupyter)
    # where the logger might persist.
    logger.propagate = False
    if logger.hasHandlers():
        logger.handlers.clear()

    # --- Console Logger ---
    console_handler = logging.StreamHandler()
    # This defines the format: Time - Level - Message
    console_format = logging.Formatter('%(asctime)s - %(levelname)s - %(message)s')
    console_handler.setFormatter(console_format)
    logger.addHandler(console_handler)

    # --- File Logger ---
    file_handler = logging.FileHandler('bronze_load.log') # The log file name
    file_format = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message
    file_handler.setFormatter(file_format)
    logger.addHandler(file_handler)

    return logger # Returns the configured logger object.
```

`read_config()` Function

This function's role is to securely load all configuration from the `.env` file. It keeps our credentials and paths out of the code.

```
def read_config(env_file='.env'):
    logger = logging.getLogger('bronze_etl')

    # This one function finds the .env file and loads all its
    # variables into the environment for `os.getenv()` to read.
    if not load_dotenv(env_file):
        # "Fail-fast" principle: If config is missing, stop immediately.
        logger.error(f"CRITICAL: Environment file '{env_file}' not found.")
        sys.exit(1) # Exits the script with an error code.

    logger.info(f"Reading configuration from {env_file}...")

    # We map the string-based env variables into a structured dictionary.
    # This is clean and makes our `main` function easy to read.
    db_config = {
        'host': os.getenv('DB_HOST'),
        'user': os.getenv('DB_USER'),
        'password': os.getenv('DB_PASSWORD'),
        'database': os.getenv('DB_DATABASE')
    }

    # We do the same for our file paths.
    paths_config = {
        'crm_cust_info': os.getenv('PATH_CRM_CUST_INFO'),
        # ... (all 6 paths) ...
        'erp_px_cat_g1v2': os.getenv('PATH_ERP_PX_CAT_G1V2')
    }

    # More "fail-fast" validation. `all()` checks if any value is
    # None (i.e., the env variable was missing).
    if not all(db_config.values()):
        logger.error("CRITICAL: One or more DB_... variables are missing from .env file
        sys.exit(1)

    if not all(paths_config.values()):
        logger.error("CRITICAL: One or more PATH_... variables are missing from .env fi
        sys.exit(1)

    # We return the two dictionaries to the `main` function.
    return db_config, paths_config
```

Section 2: Database Logging Functions

These are helper functions to keep our `main` loop clean. Their only job is to write to the `etl_log` table.

`log_etl_start()` Function

```
def log_etl_start(connection, process_name):
    start_time = datetime.now()
    query = """
        INSERT INTO etl_log (process_name, start_time, status, log_message)
        VALUES (%s, %s, 'In Progress', 'Bronze load started.')
    """
    # Best Practice: `with connection.cursor() ...`
    # This is a context manager. It automatically creates the cursor
    # and (most importantly) closes it, even if an error occurs.
    with connection.cursor() as cursor:
        # We pass parameters securely (as a tuple) to prevent SQL injection.
        cursor.execute(query, (process_name, start_time))
        # INSERTs/UPDATEs must be explicitly committed.
        connection.commit()
        # This is how we get the new `log_id` (which is AUTO_INCREMENT)
        # to pass to the `log_etl_end` function.
        return cursor.lastrowid, start_time
```

`log_etl_end()` Function

```
def log_etl_end(connection, log_id, start_time, status, message):
    end_time = datetime.now()
    # We calculate the duration in seconds for our performance log.
    duration = (end_time - start_time).total_seconds()
    query = """
        UPDATE etl_log
        SET end_time = %s, duration_sec = %s, status = %s, log_message = %s
        WHERE log_id = %s
    """
    # Again, we use a context manager for the cursor.
    with connection.cursor() as cursor:
        cursor.execute(query, (end_time, duration, status, message, log_id))
        connection.commit()
```

Section 3: The `main()` Function (The Core Orchestrator)

This is where the entire process is executed, using the "Orchestrator's Try-Catch" pattern.

```
def main():
    # --- Part 1: Setup ---
    logger = setup_logging()
    db_config, paths = read_config()

    # This list makes our code "data-driven." If you need to add
    # a 7th table, you just add a new tuple here. The `for` loop
    # will handle it automatically.
    tables_to_load = [
        ('crm_cust_info', paths.get('crm_cust_info')),
        ('crm_prd_info', paths.get('crm_prd_info')),
        ('crm_sales_details', paths.get('crm_sales_details')),
        ('erp_cust_az12', paths.get('erp_cust_az12')),
        ('erp_loc_a101', paths.get('erp_loc_a101')),
        ('erp_px_cat_g1v2', paths.get('erp_px_cat_g1v2'))
    ]
    # ... (validation check) ...

    # We initialize these to None so the `finally` block
    # can safely check if they were ever created.
    connection = None
    log_id = None
    process_start_time = datetime.now()

    # --- Part 2: The "TRY" Block ---
    # This is the main "happy path." We *try* to do all our work here.
    try:
        logger.info(f"Connecting to database...")

        # `**db_config` is a Python trick to "unpack" the dictionary
        # into keyword arguments. It's the same as writing:
        # host=db_config['host'], user=db_config['user'], ...
        connection = mysql.connector.connect(
            **db_config,
            allow_local_infile=True  # The magic key to bypass secure_file_priv!
        )

        if not connection.is_connected():
            # ... (fail-fast check) ...

        logger.info("Database connection successful.")

        # We log our start to the DB. Now we have a `log_id`.
        log_id, process_start_time = log_etl_start(connection, 'bronze_load_python')

        logger.info(f"Starting Bronze load process (Log ID: {log_id})...")
        total_start_time = time.time() # For total bottleneck timing.

        # We create one cursor to use for the whole loop.
        with connection.cursor() as cursor:
            # We loop through our `tables_to_load` list.
            for table_name, file_path in tables_to_load:
                # --- This is the bottleneck timing ---
                table_start_time = time.time()
                logger.info(f"Processing table: {table_name}...")

                # 1. TRUNCATE: Fast, preserves permissions.
                cursor.execute(f"TRUNCATE TABLE {table_name}")

                # 2. LOAD DATA

                # This small block is a safety-check for Windows.
                # It converts "C:/path" to "C:\\/path"
                # which is what the MySQL driver expects.
                safe_file_path = os.path.normpath(file_path).replace('\\', '\\\\')

                # The core SQL command, using an f-string to insert
                # the (now-safe) table and path variables.
                load_query = f"""
                    LOAD DATA LOCAL INFILE '{safe_file_path}'
                    INTO TABLE {table_name}
                    FIELDS TERMINATED BY ','
                    OPTIONALLY ENCLOSED BY '"'
                    LINES TERMINATED BY '\\r\\n'
                    IGNORE 1 LINES
                """
                cursor.execute(load_query)

                # We commit *after each table*. This is a design choice.
                # It means if table 4 fails, tables 1-3 are still loaded.
                # This is a "partial success" model.
                connection.commit()

                # --- Bottleneck timing complete ---
                table_duration = time.time() - table_start_time
                # `cursor.rowcount` gives us the rows affected by the LOAD.
                logger.info(f"Successfully loaded {cursor.rowcount} rows into {table_nc

        # --- Log Success ---
        total_duration = time.time() - total_start_time
        success_message = f"All {len(tables_to_load)} tables loaded successfully in {tc
        logger.info(success_message)
        # We update our `etl_log` record from "In Progress" to "Success".
        log_etl_end(connection, log_id, process_start_time, 'Success', success_message)

    # --- Part 3: The "CATCH" Blocks ---
    # This block only runs if something in the `try` block fails.
    except mysql.connector.Error as err:
        # This *specifically* catches database errors (bad query,
        # connection lost, table not found, etc.)
        error_message = f"MySQL Error: {err.errno} - {err.msg}"
        logger.error(f"ETL FAILED. {error_message}")

        # We try to update our `etl_log` table to "Error"
        if log_id and connection and connection.is_connected():
            log_etl_end(connection, log_id, process_start_time, 'Error', error_message)
        sys.exit(1) # Tell the OS the job failed.

    except Exception as e:
        # This is a "catch-all" for *any other* error
        # (e.g., FileNotFoundError, a Python typo like `1/0`).
        error_message = f"Non-DB Error: {str(e)}"
        logger.error(f"ETL FAILED. {error_message}")

        if log_id and connection and connection.is_connected():
            log_etl_end(connection, log_id, process_start_time, 'Error', error_message)
        sys.exit(1)

    # --- Part 4: The "FINALLY" Block ---
    # This block runs *no matter what* (success or failure).
    # Its job is to clean up resources.
    finally:
        if connection and connection.is_connected():
            connection.close()
            logger.info("Database connection closed.")
```

Section 4: The Script Entry Point

This is a standard Python idiom.

```
if __name__ == "__main__":
    main()
```

- `__name__` is a special variable.
- When you run `python3 run_bronze_load.py` , Python sets `__name__` to `"__main__"` , so the `main()` function is called.
- If you were to `import` this file into another Python script, `__name__` would be `"run_bronze_load"` , so `main()` would *not* run.