Backend Engineer Assessment

Text2SQL Analytics System with PostgreSQL

Optimization & QA Focused

Time Allocation: 7-8 hours over 3 days

Makebell Inc.

October 4, 2025

Quick Overview

Build a production-ready Text2SQL analytics system that converts natural language questions into SQL queries using **Google Gemini API (free tier)**, executes them against a **PostgreSQL database**, and returns accurate results. Demonstrate best practices in data normalization, testing, and secure AI integration using the publicly available **Northwind Database**.

Contents

1 Objectives & Evaluation Criteria

1.1 Primary Objectives

You will be evaluated on the following key areas:

1. **Data Engineering (15%):** Excel normalization and PostgreSQL schema design to 3NF
2. **Code Quality (20%):** Clean architecture, proper error handling, and comprehensive documentation
3. **AI Integration (10%):** Secure and restricted LLM interaction with database using Gemini API
4. **Testing Coverage (25%):** Pytest coverage with unit, integration, and accuracy tests (80%+ coverage required)
5. **Text2SQL Accuracy (25%):** Query generation accuracy and result validation using heuristic metrics
6. **Security & Restrictions (5%):** SQL injection prevention and query access restrictions

1.2 Bonus Points (+10%)

- Query result caching for performance optimization
- Query execution plan analysis and optimization insights
- RESTful API endpoint using FastAPI or Flask
- Query history tracking and learning mechanism
- Database performance monitoring dashboard

2 Dataset & Technical Stack

2.1 Dataset: Northwind Database

The **Northwind Database** is a classic business dataset that models a gourmet food supplier's operations. It contains rich relational data ideal for Text2SQL evaluation.

Dataset Sources:

- **Primary:** Microsoft SQL Server Samples
- **Alternative (Excel):** Maven Analytics
- **PostgreSQL Ready:** Northwind PostgreSQL

Dataset Structure:

- `Orders, Products, Customers, Employees, Suppliers`
- `Categories, Shippers, Order_Details`
- Rich foreign key relationships (14+ tables)
- Historical sales data spanning multiple years

2.2 Technology Stack

Component	Technology
Language	Python 3.10+
Database	PostgreSQL 14+
LLM API	Google Gemini API (Free Tier)
Testing	pytest, pytest-cov
Data Processing	pandas, openpyxl
Database Driver	psycopg2 / SQLAlchemy
Environment	python-dotenv
Type Checking	mypy (optional)
Code Quality	black, flake8 (optional)

Table 1: Required Technology Stack

3 System Architecture & Components

3.1 Core Components

3.1.1 A. Data Normalization Pipeline

```
# Expected functionality
- Load Excel/CSV files into pandas DataFrames
- Validate data types and constraints
- Handle NULL values appropriately
- Ensure referential integrity
- Create normalized schema (3NF minimum)
- Generate proper indexes for query optimization
- Measure and report normalization metrics
```

Listing 1: Data Normalization Requirements

3.1.2 B. Database Layer

Requirements for PostgreSQL schema:

- Primary keys on all tables
- Foreign key constraints with proper cascade rules
- Appropriate indexes (B-tree, GIN, etc.) with performance justification
- Data validation constraints (CHECK, NOT NULL, UNIQUE)
- Audit timestamps (`created_at`, `updated_at`)
- Read-only database user for query execution

3.1.3 C. Text2SQL Engine

Core features:

- Natural language to SQL conversion using Gemini API
- SQL sanitization and validation before execution

- Restricted query execution (SELECT only)
- Query result formatting (JSON, pandas DataFrame)
- Comprehensive error handling and logging
- Query timeout enforcement (5 seconds maximum)

3.2 Security & Restrictions

⚠ Critical Security Requirements

The AI must have **strictly restricted** database access:

✓ **Allowed Operations:**

- SELECT queries only
- Aggregations (COUNT, SUM, AVG, MAX, MIN, etc.)
- JOINs across multiple tables
- Subqueries and CTEs

✗ **Blocked Operations:**

- INSERT, UPDATE, DELETE, DROP, CREATE, ALTER
- System table access (pg_catalog, information_schema)
- User management operations
- Transaction control statements

Enforcement Mechanisms:

- Query timeout: 5 seconds maximum
- Result row limit: 1000 rows maximum
- SQL injection prevention testing required
- Database user with SELECT-only privileges

4 Testing Requirements (Pytest)

4.1 Test Categories & Distribution

Testing accounts for **50% of total grade** (25% coverage + 25% accuracy).

4.1.1 1. Unit Tests (30% of testing grade)

```
# test_excel_loader.py
- test_load_valid_excel_file()
- test_handle_missing_values()
- test_data_type_validation()
- test_foreign_key_detection()
- test_duplicate_row_detection()

# test_sql_sanitizer.py
- test_block_insert_statements()
- test_block_drop_statements()
- test_allow_select_statements()
- test_sql_injection_prevention()
- test_query_timeout_enforcement()
```

Listing 2: Unit Test Examples

4.1.2 2. Integration Tests (30% of testing grade)

```
# test_database_operations.py
- test_connection_pool_management()
- test_transaction_rollback()
- test_query_timeout_enforcement()
- test_result_set_limiting()
- test_concurrent_query_execution()

# test_text2sql_pipeline.py
- test_end_to_end_simple_query()
- test_multi_table_join_query()
- test_aggregate_query_generation()
- test_error_recovery_mechanism()
- test_invalid_question_handling()
```

Listing 3: Integration Test Examples

4.1.3 3. Accuracy Tests (40% of testing grade - Heuristic Based)

Create a test suite with **at least 20 analytics questions** covering:

Category	**Count**	**Example**
Simple Queries	5	Single table SELECT, WHERE clauses
Intermediate	10	JOINs (2-3 tables), GROUP BY, aggregations
Complex	5	Multi-level JOINs (4+ tables), subqueries

Table 2: Accuracy Test Distribution

Example Test Questions:

Simple Queries (5 questions)

1. How many products are currently not discontinued?
2. List all customers from Germany
3. What is the unit price of the most expensive product?
4. Show all orders shipped in 1997
5. Which employee has the job title 'Sales Representative'?

Intermediate Queries (10 questions)

1. What is the total revenue per product category?
2. Which employee has processed the most orders?
3. Show monthly sales trends for 1997
4. List the top 5 customers by total order value
5. What is the average order value by country?
6. Which products are out of stock but not discontinued?
7. Show the number of orders per shipper company
8. What is the revenue contribution of each supplier?
9. Find customers who placed orders in every quarter of 1997
10. Calculate average delivery time by shipping company

Complex Queries (5 questions)

1. What is the average order value by customer, sorted by their total lifetime value?
2. Which products have above-average profit margins and are frequently ordered together?
3. Show the year-over-year sales growth for each product category
4. Identify customers who have placed orders for products from all categories
5. Find the most profitable month for each employee based on their order commissions

4.2 Heuristic Evaluation Metrics

```
# Execution Accuracy (EX): 20%
execution_success = 1 if query executes without errors else 0

# Result Match: 40%
result_match = 1 if results match expected output else 0

# Query Quality Score: 40%
quality_metrics = {
    'uses_proper_joins': 0/1,       # No cartesian products
    'has_necessary_where': 0/1,     # Proper filtering
    'correct_group_by': 0/1,        # Appropriate grouping
    'efficient_indexing': 0/1,      # Uses indexes effectively
    'execution_time': 0/1           # < 1 second
}
query_quality = mean(quality_metrics.values())

# Final Accuracy Score
accuracy_score = (
    0.20 * execution_success +
    0.40 * result_match +
    0.40 * query_quality
)
```

Listing 4: Accuracy Scoring Formula

5 Repository Structure

```
text2sql-analytics/
          README.md
          requirements.txt
          .env.example
          .gitignore
          setup.py

          data/
                raw/
                      northwind.xlsx
                schema/
                    schema.sql

          src/
                __init__.py
                config.py
                data_loader.py
                database.py
                text2sql_engine.py
                query_validator.py
                utils.py

          tests/
                __init__.py
                conftest.py
                test_data_loader.py
                test_database.py
                test_query_validator.py
                test_text2sql_engine.py
                test_accuracy/
                    test_simple_queries.py
                    test_intermediate_queries.py
                    test_complex_queries.py

          notebooks/
                analysis.ipynb

          scripts/
              setup_database.py
              run_evaluation.py
```

Listing 5: Expected Project Structure

6 Deliverables

6.1 1. Working Code (40%)

- Complete implementation of all components
- Clean, well-documented Python code (docstrings, type hints)
- Proper error handling and structured logging
- Configuration management using .env files

- No hardcoded credentials or API keys in code

6.2 2. Testing Suite (30%)

- Minimum 80% code coverage (measured by pytest-cov)
- All test categories implemented (unit, integration, accuracy)
- Pytest fixtures for database setup/teardown
- Clear test documentation and naming conventions
- Test coverage HTML report included

6.3 3. Documentation (20%)

README.md must include:

- Project overview and architecture diagram
- Setup instructions (step-by-step)
- Database initialization guide
- API key configuration instructions
- How to run tests with examples
- Example usage with code snippets
- Accuracy metrics results table
- Known limitations and future improvements

Additional Documentation (If you want):

- Schema diagram (can use dbdiagram.io)
- API documentation (if implementing REST API)
- Test coverage report (HTML format)

6.4 4. Evaluation Report (10%)

Create `EVALUATION.md` containing:

- Test accuracy results breakdown by complexity level
- Query performance metrics (execution time distribution)
- Failed queries analysis with explanations
- Database optimization opportunities identified
- Lessons learned and challenges faced
- Time spent on each component

7 Security Checklist

Before submission, ensure all items are completed:

- No API keys in code or git history
- SQL injection prevention tested
- Read-only database user for query execution
- Query timeout enforcement (5 seconds)
- Result size limiting (1000 rows)
- No system table access allowed
- Input sanitization for all user inputs
- Error messages don't leak schema information
- Environment variables properly configured
- Database credentials not in version control

8 Helpful Resources

8.1 Text2SQL & LLM Resources

- **Awesome Text2SQL:**
  `https://github.com/eosphoros-ai/Awesome-Text2SQL`
- **Google Gemini API Documentation:**
  `https://ai.google.dev/docs`
- **Gemini Python SDK:**
  `https://github.com/google/generative-ai-python`
- **Text2SQL Evaluation Framework (QueryCraft):**
  `https://medium.com/towards-generative-ai/querycraft-evaluation-framework-for-nl2sql-ge`
- **Text2SQL Accuracy Metrics Guide:**
  `https://blog.premai.io/evaluating-llms-for-text-to-sql-with-prem-text2sql/`
- **Spider Text2SQL Benchmark:**
  `https://yale-lily.github.io/spider`
- **BIRD Text2SQL Benchmark:**
  `https://bird-bench.github.io/`

8.2 Pytest & Database Testing

- **Pytest Official Documentation:**
  `https://docs.pytest.org/`
- **Database Testing with Pytest and SQLAlchemy:**
  `https://pytest-with-eric.com/database-testing/pytest-sql-database-testing/`
- **Pytest Database Fixtures Guide:**
  `https://coderpad.io/blog/development/a-guide-to-database-unit-testing-with-pytest-and-`

- **Python Unit Testing Best Practices:**
  https://pytest-with-eric.com/introduction/python-unit-testing-best-practices/
- **Effective Python Testing (Real Python):**
  https://realpython.com/pytest-python-testing/
- **Testing with Pytest and PostgreSQL:**
  https://github.com/Pytest-with-Eric/pytest-db-testing-example

8.3 PostgreSQL & Database Normalization

- **PostgreSQL Normalization Guide:**
  https://www.compilenrun.com/docs/database/postgresql/postgresql-best-practices/postgresql-normalization/
- **Database Normalization (1NF-3NF):**
  https://www.freecodecamp.org/news/database-normalization-1nf-2nf-3nf-table-examples/
- **Normalization in SQL (DataCamp):**
  https://www.datacamp.com/tutorial/normalization-in-sql
- **PostgreSQL Schema Design Best Practices:**
  https://reintech.io/blog/best-practices-database-schema-design-postgresql
- **Database Normalization Visual Guide:**
  https://www.digitalocean.com/community/tutorials/database-normalization
- **PostgreSQL Official Documentation:**
  https://www.postgresql.org/docs/

9 Submission Guidelines

9.1 Submission Requirements

1. Push all code to a **public GitHub repository**
2. Ensure README.md has complete, tested setup instructions
3. Include test coverage report (HTML format in `htmlcov/`)
4. Add `EVALUATION.md` with results and analysis
5. Tag your final submission with version: `git tag v1.0`
6. Submit repository URL via designated submission portal
   (https://tally.so/r/n0NEV6)

9.2 Repository Must Include

- All source code in `src/` directory
- Complete test suite in `tests/` directory
- `requirements.txt` with all dependencies and versions
- `.env.example` with template for environment variables
- `README.md` with setup and usage instructions

- `EVALUATION.md` with test results and analysis
- Schema diagram (PNG/PDF or link to dbdiagram.io)
- Test coverage report

9.3 What NOT to Include

- `.env` file (actual credentials)
- `__pycache__/` directories
- `.venv/` or `venv/` directories
- `.idea/`, `.vscode/` IDE-specific folders
- Database files (`*.db`, dump files)
- API keys or secrets

✓ Sample .gitignore

```
# Python
__pycache__/
*.py[cod]
*$py.class
*.so
.Python
venv/
.venv/
ENV/

# Environment
.env
.env.local

# IDE
.vscode/
.idea/
*.swp
*.swo

# Testing
.pytest_cache/
.coverage
htmlcov/

# Database
*.db
*.sqlite
*.sql.gz
```

10 Tips for Success

1. **Start with schema design** – A well-normalized database makes everything easier and sets the foundation for accurate queries.
2. **Test early and often** – Write tests as you build features. This catches bugs early and ensures your code is testable.

3. **Prompt engineering matters** – Give Gemini clear, detailed context about your database schema. Include table relationships and example queries.
4. **Document as you go** – Don't leave documentation for the end. Write docstrings and README sections immediately after implementing features.
5. **Focus on core functionality first** – Ensure the basic Text2SQL pipeline works perfectly before attempting bonus features.
6. **Measure everything** – Track query performance, accuracy rates, and test coverage throughout development.
7. **Use version control properly** – Commit frequently with meaningful messages. This shows your development process.
8. **Handle edge cases** – Test with ambiguous questions, invalid inputs, and malformed queries to demonstrate robustness.
9. **Optimize iteratively** – First make it work, then make it better. Don't prematurely optimize.
10. **Review the rubric** – Before submission, verify you've addressed all evaluation criteria.

11 Frequently Asked Questions

11.1 Technical Questions

Q: Can I use SQLAlchemy ORM instead of raw psycopg2?
A: Yes, either is acceptable. SQLAlchemy provides better abstraction but adds complexity. Choose based on your comfort level.

Q: How should I handle the Gemini API rate limits?
A: The free tier allows 60 requests/minute, which is sufficient for testing. Implement basic rate limiting and exponential backoff for production-grade handling.

Q: What if Gemini generates syntactically correct but semantically wrong SQL?
A: This is expected! Document these cases in your evaluation report. Implement validation where possible, and consider them in your accuracy metrics.

Q: Should I normalize beyond 3NF?
A: 3NF is sufficient for this assessment. Higher normal forms (BCNF, 4NF) are bonus points if implemented correctly.

Q: Can I use Docker for the PostgreSQL database?
A: Yes, Docker is encouraged! Include a `docker-compose.yml` file and update setup instructions accordingly.

11.2 Testing Questions

Q: How do I test database operations without affecting my main database?
A: Use pytest fixtures to create a temporary test database, or use transactions with rollback. See the pytest resources section for examples.

Q: What constitutes a "passing" accuracy test?
A: A query passes if it executes without errors AND returns results matching the expected output. Partial credit for correct execution but wrong results.

Q: Should I test the Gemini API directly?
A: Mock the Gemini API for unit tests. Test actual API integration in integration tests with rate limiting.

11.3 Submission Questions

Q: What if I can't complete all features in 8 hours?
A: Focus on core functionality first. A well-implemented subset is better than incomplete features across the board. Document what you'd improve with more time.

Q: Can I use AI coding assistants (GitHub Copilot, ChatGPT, etc.)?
A: Yes, but all code must be your own work and you must understand it completely. You may be asked to explain any part of your implementation.

Q: How will my submission be evaluated?
A: Automated tests will check functionality and coverage. Manual review will assess code quality, documentation, and architecture decisions.

12 Contact & Support

For Questions or Clarifications:

- Create an issue in your repository with the `question` label
- Email: `asifsadek509@gmail.com`
- Response time: Within 12 hours on business days

Good luck! We're excited to see your solution! 🚀

Remember: We value clean, tested, well-documented code over flashy features.
Quality over quantity. Understanding over completion.