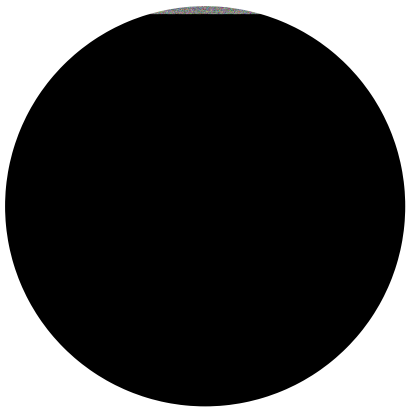
From Text to SQL Agent

Smart Query in Action

Eric Julianto

About Me



x.com/algonacci

github.com/algonacci

linkedin.com/eric-julianto

instagram.com/eric.juliantooo

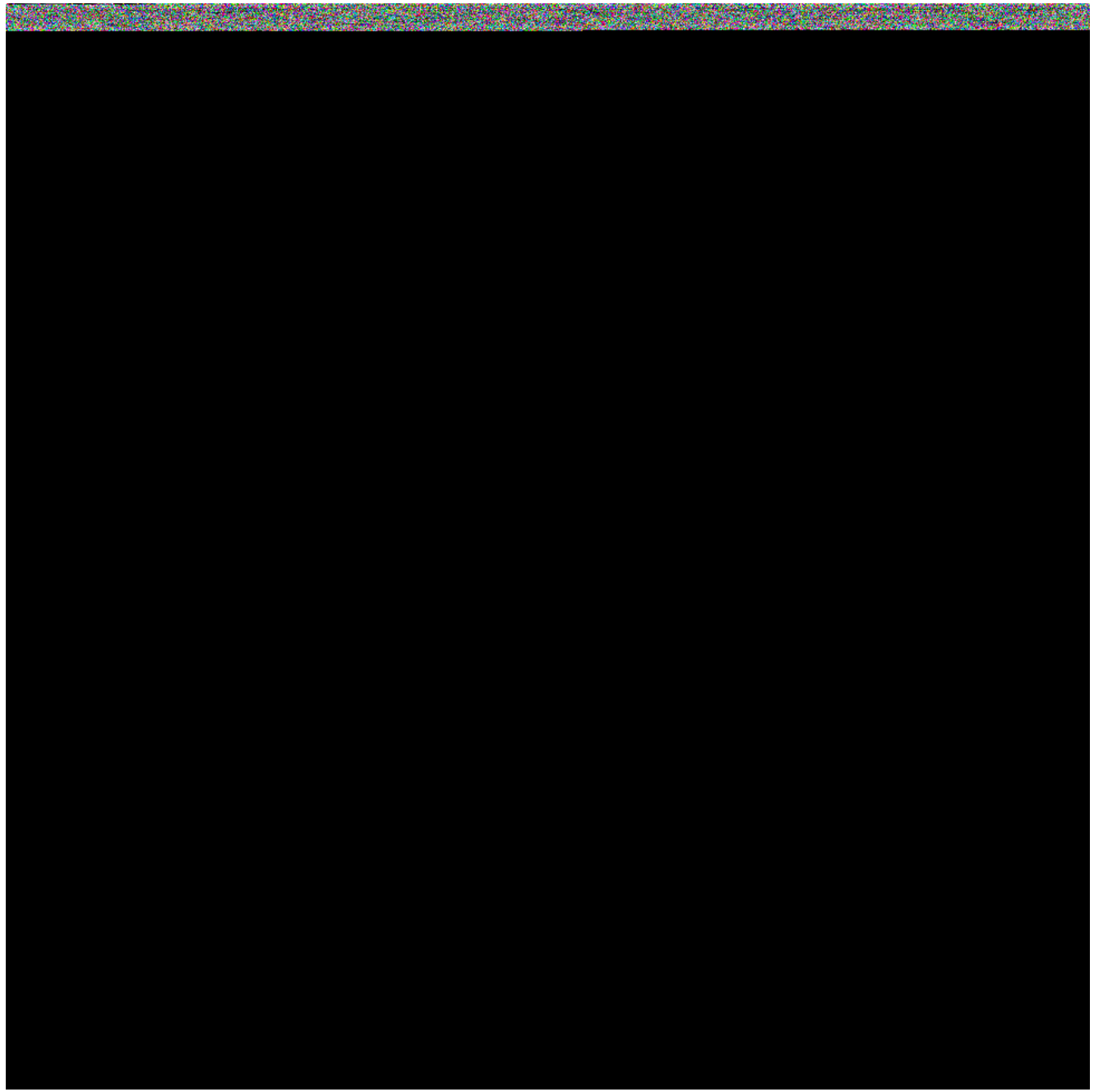
Work Experience

- Research Analyst at Braincore
  (Dec 21 - Now)
- Machine Learning Mentor at Bangkit Academy
  (Feb 23 - Jan 24)
- SEO Intern at Dibilabs by Dibimbing.id
  (Mar 23 - Jun 23)
- AI Developer Intern at ZettaByte
  (May 22 - Aug 22)

Education

- Hospitality & Tourism at Universitas Bunda Mulia
- Computer Science at Universitas Esa Unggul

Let’s talk about SQL Agent



From Text to SQL Agent

Smart Query in Action

Mengubah pertanyaan dalam bahasa natural menjadi SQL query menggunakan LLM

Problem Statement

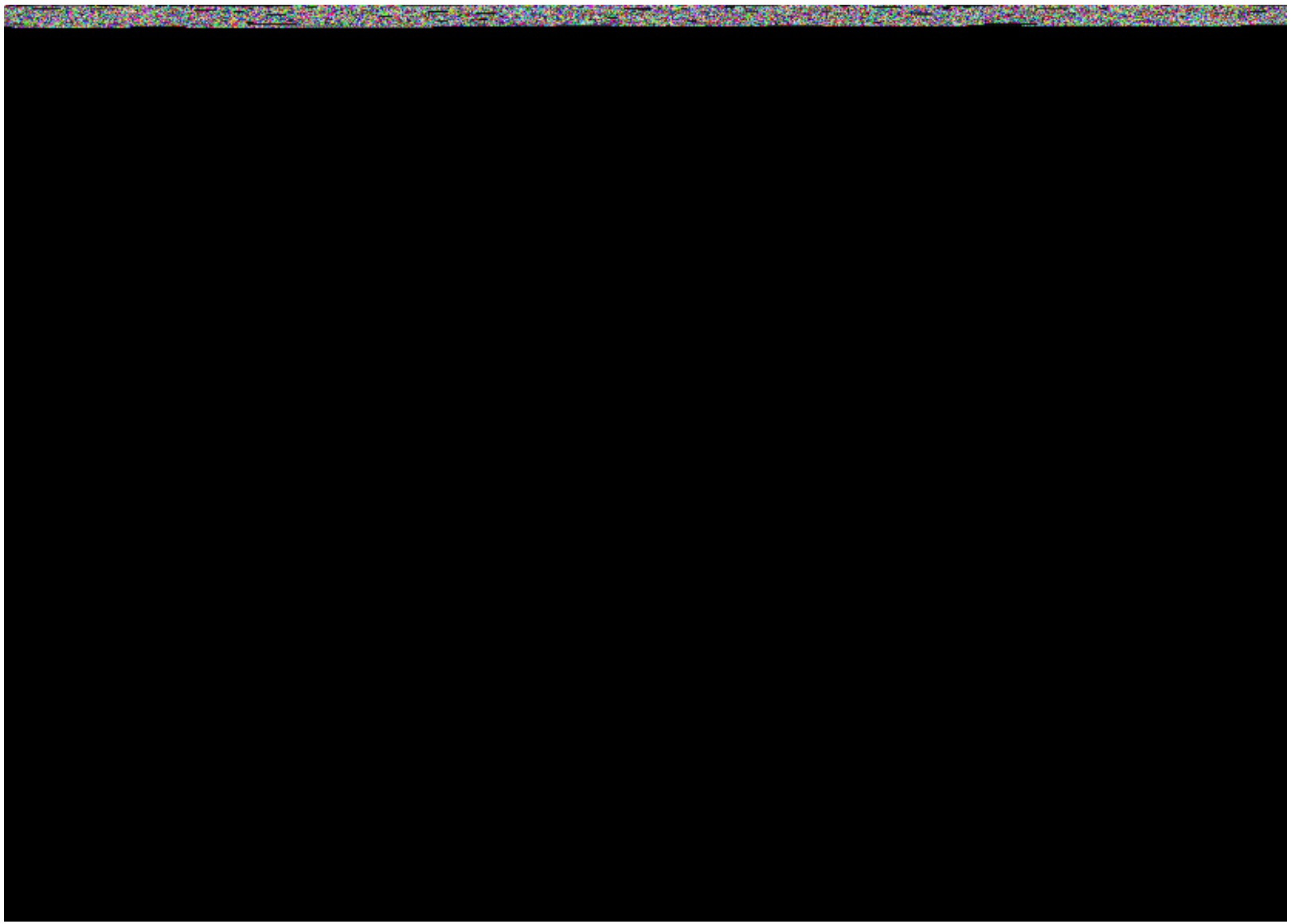
Tantangan:

- SQL syntax sulit dipelajari
- Tidak semua orang familiar dengan database schema
- Query kompleks butuh waktu lama
- Sering typo atau syntax error

Solusi:

- Natural language → SQL
- AI memahami struktur database
- Generate query otomatis
- User-friendly interface

Arsitektur Sistem



How It Works

1. **Input** - User mengetik pertanyaan dalam bahasa natural
2. **Intent Classification** - LLM mengklasifikasi: query atau schema_info
3. **Generation** - LLM membuat SQL query atau info schema
4. **Execution** - Execute query di database (read-only)
5. **Display** - Tampilkan hasil dalam format rapi

Demo Time

Quick Setup Guide

1. Clone Repository

```
git clone https://github.com/algonacci/from-text-to-sql-agent
cd from-text-to-sql-agent
cd scripts
```

2. Setup Environment

```
cp .env.example .env
```

3. Install Dependencies

```
uv sync
```

4. Run the Agent

```
uv run simple.py
```

Tips and Tricks

- Prompt engineering
- Context engineering
- Only query `SELECT` statement
- Validate the generated query statement
- Structured prompt
- Role and persona
- Chain-of-Thought (CoT)
- One shot example
- Few shot example
- Explicit instruction
- Output formatting
- Fallback behavior

Challenges and Limitations

1. **LLM Accuracy**
   - Kadang generate query yang tidak optimal
   - Butuh prompt engineering yang baik
2. **Complex Queries**
   - JOIN multi-table masih challenging
   - Aggregasi kompleks perlu tuning
3. **Database Specific**
   - Dialect SQL berbeda per database
   - Perlu testing per platform

Future Improvements

Features:

- Query history & caching
- Query optimization suggestions
- Data visualization
- Export hasil (CSV, Excel)
- Multi-language support

Technical:

- Unit testing
- Query validation
- Web UI (FastAPI + React)
- User authentication
- Mobile app

Use Cases

1. **Business Analytics**
   - Non-technical users query data
   - Quick insights tanpa SQL
2. **Data Exploration**
   - Explorasi database baru
   - Understand schema cepat
3. **Prototyping**
   - Rapid query testing
   - Data validation
4. **Education**
   - Belajar SQL dari generated queries
   - Understand best practices

Advanced Case

What if...

What if...

- **Multi Database**
  - Lebih dari 1 database
  - Cross-database queries
- **Dialek Berbeda**
  - PostgreSQL, MySQL, SQL Server
  - Syntax compatibility issues
- **Scale Challenge**
  - Jumlah row sudah jutaan
  - Performance optimization needed
- **Complex Schema**
  - Jumlah tabel sudah ratusan
  - Dependency mapping
- **Security & Privacy**
  - Sensitive data handling
  - Row-level security
- **Real-time Analytics**
  - Streaming data
  - Live dashboard queries

Cara Tackle Advanced Case

Hybrid Approach: Template-based + LLM Fallback

Template Pipeline (~5-10s)

Pattern Matching → SQL Templates → Execute

LLM Pipeline (~30-60s)

Intent → Routing → Schema Filter → SQL Gen → Format

Key Solutions

Multi-DB: Session registry mapping

Scale: 2-stage schema filtering

Security: SQL sanitization guards

Performance: Template caching

Complexity: Foreign key graph

SQL Template Example

Query: “Tampilkan user yang aktif dengan role admin”

Pattern: USER_LIST_FILTER → **Variables**: `{status: "active", role: "admin", limit: 50}`

```sql
SELECT
    u.id, u.name, u.email,
    u.role, u.status,
    u.created_at
FROM users u
WHERE u.status = '{status}'
  {role_filter}
  AND u.deleted_at IS NULL
ORDER BY u.created_at DESC
LIMIT {limit};
```

Filled Template: `{role_filter}` → `AND u.role = 'admin'`

Result: Fast, consistent, predictable SQL generation

Deep Dive: Schema Filtering

Problem: Ratusan tabel → LLM hallucination & slow

Solution: 2-Stage Filtering

Stage 1: Graph-based

Required tables + neighbors via FK graph

200 tables → ~30 tables

Stage 2: LLM Refinement

Chain-of-Thought reasoning + confidence scoring

30 tables → 5-10 tables

Best Practices Applied

Structured prompt

Few-shot examples

Chain-of-Thought

Role & persona definition

Explicit instructions

Fallback behavior

Security: SQL Sanitization

Guards Module

```python
def sanitize_sql(sql: str):
  # Only allow SELECT/WITH
  if not starts_with_select():
    raise ValueError()

  # Block dangerous keywords
  if has_forbidden_tokens():
    raise ValueError()

  # Prevent SQL injection
  if has_multiple_statements():
    raise ValueError()

  # Force LIMIT
  ensure_limit(default=500)
```

Protection Against

SQL Injection

DROP TABLE attacks

Multi-statement execution

Unbounded queries

Comment-based bypasses

Read-only enforcement pada database level

Multi-Database Handling

Session Registry Pattern

```
SESSION_REGISTRY = {
  "db_1": SessionMySQL,
  "db_2": SessionMariaDB,
  "db_3": SessionPSQL,
  "db_4": SessionMariaDB
}
```

Auto-routing based on

Intent classification

Keyword mapping

Table requirements

Dialect-aware SQL Templates

PostgreSQL: `DATE_TRUNC`

MySQL: `YEAR()`, `DATE_FORMAT`

MariaDB: JSON functions

Dynamic template selection

berdasarkan detected database

Performance Optimization

Template-based Pipeline

17 pre-built SQL templates

Pattern matching ~2-3s

Total query time ~5-10s

99%+ consistency

Pattern Definitions

JSON-based pattern library

Variable extraction rules

Default value fallbacks

Optimization Techniques

Forced LIMIT on queries

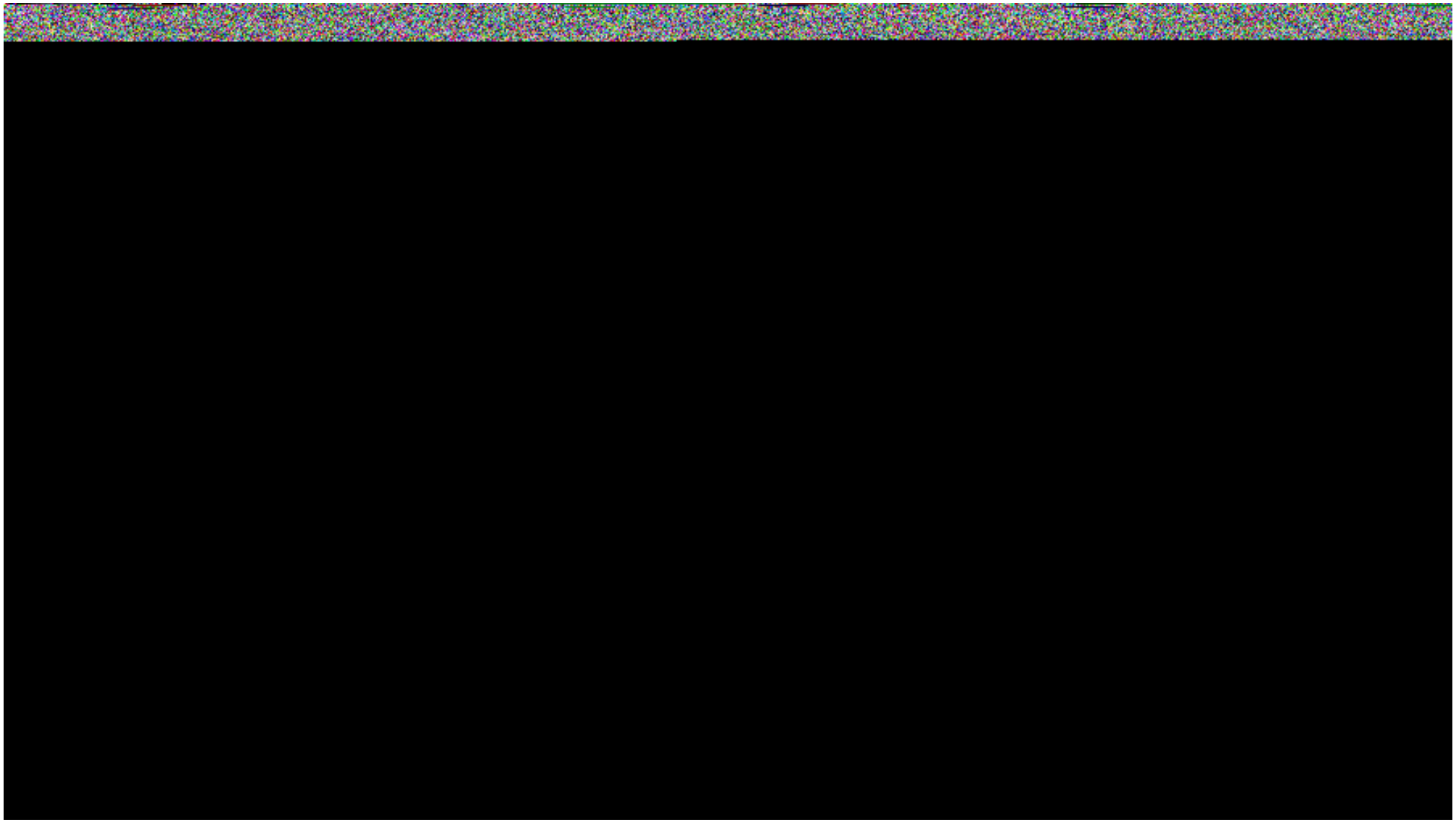
Schema pre-filtering

Foreign key graph caching

Confidence-based routing

Lazy LLM fallback

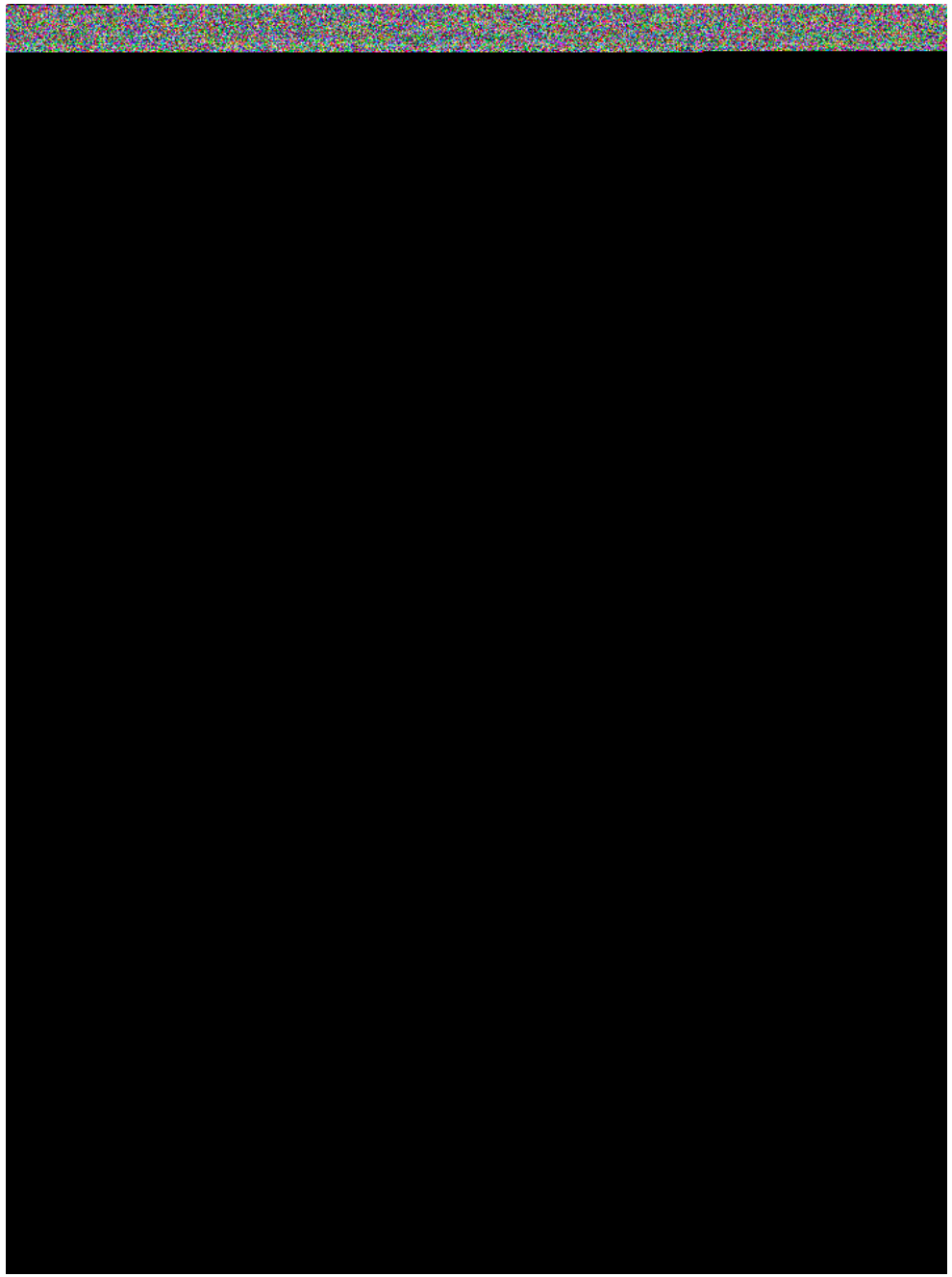
Trade-off: Speed vs Flexibility



Demo Time (Part 2)

Key Takeaways

1. **LLM + Database = Powerful Combo**
   - Natural language accessibility
   - Reduce learning curve
2. **Modular Architecture Matters**
   - Easy to maintain
   - Easy to extend
3. **User Experience is Key**
   - Error handling
   - Helpful messages
   - Smooth interactions



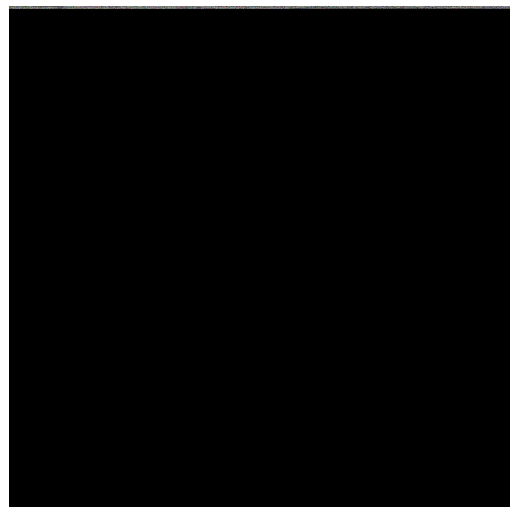
Resources

Project Repository:

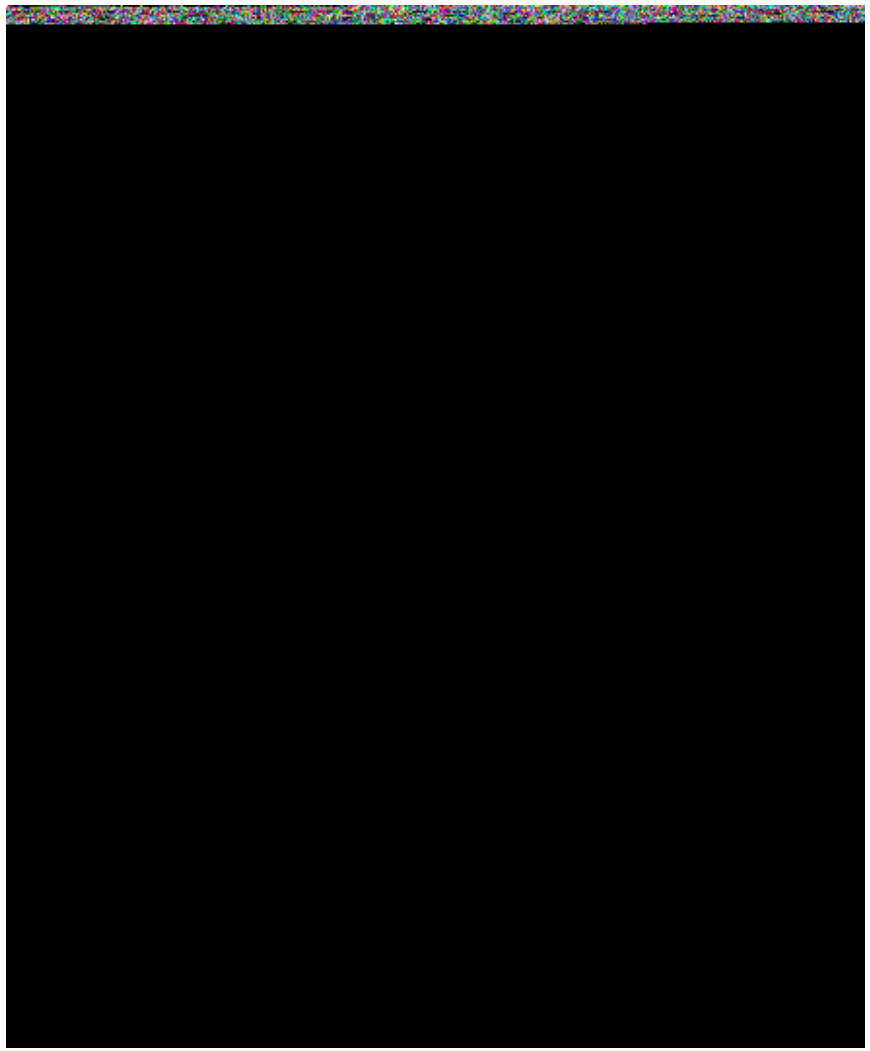
https://github.com/algonacci/from-text-to-sql-agent

Short Link:

https://s.id/sql-agent-bandungpy



Q&A



Thank you for listening!