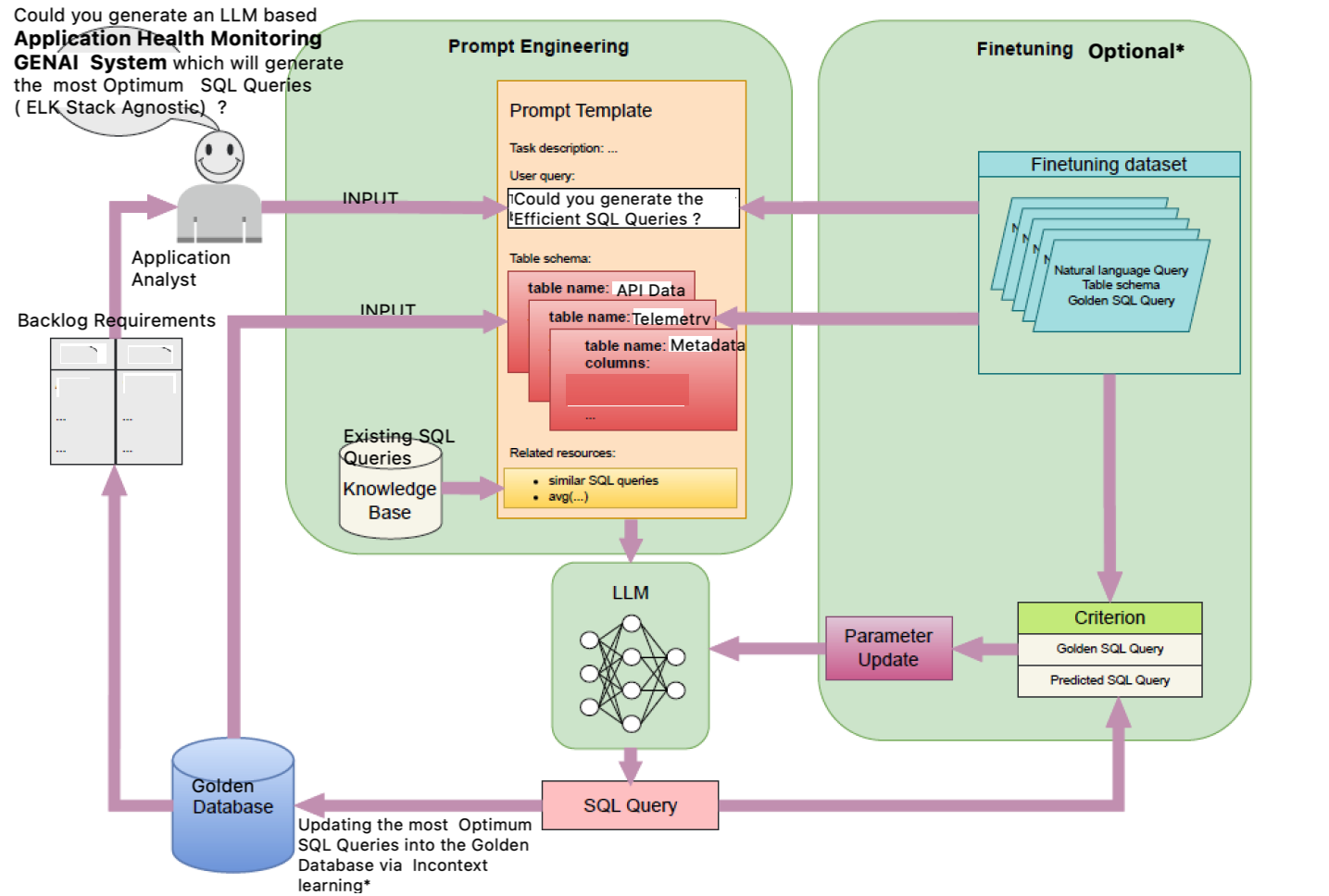
# LLM-Assisted Application Health: Solutions and Reasoning

* **Akram Sheriff**

To address the Problem explained in the Exercise document I will be leveraging LLMs to automate the creation of SQL queries and improve existing ones. We will also ensure compatibility with potential future migrations to Kibana Query Language or PromQL or other SQL variants.

**HIGH LEVEL SOLUTION APPROACH:**



**Existing Solution LLM Selection and Technical Rationale**

* **Model Choice:** Given the team's experience with GPT-4o and Claude3.5, we suggest using these models but with enhanced prompt engineering and validation strategies to ensure better interpretation of requirements.
* **Why LLMs:** These models are capable of understanding complex requirements and generating code snippets, including SQL queries, making them suitable for our needs.

**3. Solution Approach**

* **Automated Query Generation:** Use LLMs to generate SQL queries from backlog requirements. Implement a feedback loop where generated queries are reviewed by a Human SQL expert (HIL workflow) to ensure accuracy.
* **Query Improvement:** Analyze existing SQL queries to identify inefficiencies or unclear objectives. Use LLMs to suggest improvements and clarify purposes.
* **Migration Readiness:** Develop a strategy for rewriting queries in Kibana Query Language or PromQL. This includes understanding the syntax and semantics of these languages and ensuring our LLM-assisted process can adapt accordingly.

**4. Tools and Libraries**

* **LLM Access:** Use APIs or libraries that provide access to GPT-4o and Claude3.5.
* **Database Management:** Use SQL tools to test and validate queries.
* **Monitoring Frameworks:** Implement tools like Grafana or Kibana for visual monitoring post-query execution.

**5. Problem Solving**

* **Incorrect Interpretations of SQL:** Improve prompt clarity and provide additional context\* (can be done via Prompt templates or Prompt Fine tuning or Prompt engineering) to the LLM to reduce misinterpretation of the SQL queries & its variants.
* **Query Validation:** Establish a robust SQL query validation process involving manual reviews and automated testing to catch errors early.

**6. Product Backlog Requirements and Solutions:**

## Challenge 1: Create a query to alert when any API endpoint experiences a 50% increase in average response time compared to the previous hour's baseline.

**SQL Query**:

WITH Previous\_Hour\_Baseline AS (  
 SELECT endpoint,  
 AVG(response\_time) AS avg\_response\_time  
 FROM api\_requests  
 WHERE timestamp BETWEEN NOW() - INTERVAL 2 HOUR AND NOW() - INTERVAL 1 HOUR  
 GROUP BY endpoint  
),  
Current\_Hour AS (  
 SELECT endpoint,  
 AVG(response\_time) AS avg\_response\_time  
 FROM api\_requests  
 WHERE timestamp >= NOW() - INTERVAL 1 HOUR  
 GROUP BY endpoint  
)  
SELECT c.endpoint, c.avg\_response\_time AS current\_avg, p.avg\_response\_time AS previous\_avg  
FROM Current\_Hour c  
JOIN Previous\_Hour\_Baseline p ON c.endpoint = p.endpoint  
WHERE c.avg\_response\_time > p.avg\_response\_time \* 1.5;

### Reasoning:

1. Data Segmentation: Split data into two intervals (current and previous hour) using WITH clauses. Depending on the Type of the Data Split ratio the efficiency would change.

2. Comparison: Calculate average API response times for both intervals and compare them.

3. Threshold Identification: Use a condition to identify when the current average exceeds 150% of the previous average.

## Challenge 2: Detect anomalous user session patterns indicating account compromise.

SQL Query:

WITH Frequent\_Logins AS (  
 SELECT user\_id, COUNT(\*) AS login\_count  
 FROM user\_sessions  
 WHERE start\_time >= NOW() - INTERVAL 1 HOUR  
 GROUP BY user\_id  
 HAVING login\_count > 10  
),  
Concurrent\_Sessions AS (  
 SELECT user\_id, COUNT(\*) AS concurrent\_sessions  
 FROM user\_sessions  
 WHERE status = 'active' AND end\_time IS NULL  
 GROUP BY user\_id  
 HAVING concurrent\_sessions > 3  
)  
SELECT f.user\_id  
FROM Frequent\_Logins f  
JOIN Concurrent\_Sessions c ON f.user\_id = c.user\_id;

### Reasoning:

1. Frequent Logins: Identify users with abnormally high login frequency.

2. Concurrent Sessions: Detect users with multiple active sessions.

3. Anomaly Detection: Combine results to flag users exhibiting both behaviors, which may indicate compromise.

**Challenge 3: API Error Rate Monitoring:**

#### ****Monitor for scenarios where error rates exceed 5% of total requests per endpoint while also having high response times (>2s).****

WITH Total\_Requests AS (

SELECT endpoint, COUNT(\*) AS total\_count

FROM api\_requests

WHERE timestamp >= NOW() - INTERVAL 15 MINUTE

GROUP BY endpoint

),

High\_Response\_Errors AS (

SELECT endpoint, COUNT(\*) AS error\_count

FROM api\_requests

WHERE timestamp >= NOW() - INTERVAL 15 MINUTE AND response\_code >= 400 AND response\_time > 2

GROUP BY endpoint

)

SELECT t.endpoint,

h.error\_count,

t.total\_count,

(h.error\_count \* 100.0 / t.total\_count) AS error\_rate

FROM Total\_Requests t

JOIN High\_Response\_Errors h ON t.endpoint = h.endpoint

WHERE (h.error\_count \* 100.0 / t.total\_count) > 5;

**Reasoning:**

1. **Error Rate Calculation:** Track the percentage of requests that fail.
2. **High Response Time Filtering:** Focus on errors with response times exceeding 2 seconds.
3. **Alert Criteria:** Set a threshold of 5% error rate for generating alerts.

**Challenge 4: Understand Application Slowdowns:**

SELECT a.timestamp,

a.endpoint,

AVG(a.response\_time) AS avg\_response\_time,

COUNT(\*) AS total\_requests,

s.cpu\_usage,

s.memory\_usage,

s.active\_connections

FROM api\_requests a

JOIN system\_performance s ON a.timestamp = s.timestamp

WHERE a.timestamp BETWEEN '2024-12-31 12:00:00' AND '2024-12-31 14:00:00'

GROUP BY a.timestamp, a.endpoint, s.cpu\_usage, s.memory\_usage, s.active\_connections

ORDER BY avg\_response\_time DESC;

**Reasoning:**

1. **Performance Correlation:** Link API request metrics with system performance data.
2. **Peak Hour Focus:** Analyze performance during high user activity periods.
3. **Bottleneck Detection:** Identify endpoints with the highest average response times.

**Challenge 5: Proactive Resource Utilization Monitoring:**

WITH Utilization\_Trend AS (

SELECT resource\_type,

server\_id,

AVG(current\_usage / max\_capacity \* 100) AS usage\_trend

FROM resource\_utilization

WHERE timestamp >= NOW() - INTERVAL 1 DAY

GROUP BY resource\_type, server\_id

)

SELECT u.resource\_type, u.server\_id, u.usage\_trend

FROM Utilization\_Trend u

WHERE u.usage\_trend > 75;

**Reasoning:**

1. **Trend Analysis:** Calculate average resource usage trends over a day.
2. **Threshold Monitoring:** Flag servers exceeding 75% utilization.
3. **Proactive Alerts:** Allow early intervention to avoid capacity issues.

### Tools and Frameworks

1. **LLMs:** GPT-4 for interpreting requirements and generating precise SQL queries.
2. **Frameworks:**
   * LangChain for query orchestration.
   * Grafana for visualizing monitoring outputs.

### Challenges and Solutions

1. **Ambiguity in Requirements:** Utilized prompt engineering to clarify ambiguous statements.
2. **Prototype Adaptability:** Designed modular queries for future migration to Kibana Query Language or PromQL.

### Future Work if additional Time Provided:

1. Prompt Template Fine Tuning using an LLM Model
2. DataSet Fine Tuning with Instruction FT technique
3. Real-time anomaly detection using streaming data.
4. Predictive analytics for resource usage trends.
5. Seamless integration with CI/CD pipelines for automated monitoring updates.

**CHALLENGES & LIMITATIONS OF LLM TEXT-TO-SQL**:

There are 4 fundamental major challenges associated with the LLM text-to-SQL approach:

#### 1. Schema awareness

LLMs need to be aware of relevant database schema to generate accurate SQL queries. While this is simple for smaller databases, it's challenging for massive databases with hundreds or thousands of different tables and columns. Additionally, enterprise data often exists in multiple, inconsistent versions across different systems.

To be effective, LLMs must be aware of and use trusted data sources like master data management systems in conjunction with [active retrieval-augmented generation](https://www.k2view.com/blog/active-retrieval-augmented-generation/) to ensure they generate SQL queries that provide accurate and consistent results based on both publicly available (Internet) and private (enterprise) data.

#### 2. Accuracy of results

AI-generated SQL queries face several challenges in producing accurate results, including [AI hallucinations](https://www.k2view.com/what-are-ai-hallucinations/), erroneous column names, misunderstood schemas, and poorly executed [AI prompt engineering](https://www.k2view.com/blog/ai-prompt-engineering/).

Real-world enterprise systems and databases present additional complexities due to highly complex schemas, which may exceed your LLM’s prompt limits. AI systems also struggle with nuanced contexts, such as understanding the specific meaning of generic column names (e.g., "date" in different contexts).

Without proper context, LLMs risk misinterpreting data structures, potentially leading to misleading or inaccurate query results in complex enterprise environments.

#### 3. Performance

It’s not easy to calculate the efficiency of AI-generated SQL, since very different SQL statements can arrive at the same result.

Extensive schemas and ambiguous column names pose additional difficulties. When dealing with unstructured data through APIs, issues of latency and resource usage arise, which could lead to unexpected costs.

The disconnect between applying the latest [prompt engineering techniques](https://www.k2view.com/blog/prompt-engineering-techniques/) and SQL expertise can complicate efforts to generate queries and may necessitate multi-disciplinary teams for effective implementation.

#### 4. Security

An [enterprise LLM](https://www.k2view.com/blog/enterprise-llm) can pose significant security risks. While the ability to generate SQL from natural language interfaces can democratize data access, it's important to note that it can expose data to significant security risks.

Concern over public LLMs exposing confidential data has already led many companies to ban their use. However, even self-hosted LLMs can compromise security if given unrestricted access to enterprise systems. In customer service applications, LLMs could be manipulated to reveal sensitive user information or be exploited in phishing schemes.

While protective measures such as [dynamic data masking](https://www.k2view.com/blog/dynamic-data-masking/) and multi-factor authentication do exist, LLM security doesn’t happen automatically. Implementing these guardrails requires careful planning and execution.

**OTHER LLM MODEL SELECTION IN FUTURE TECHNICAL REFERENCE:**

Reasoning models like o1 and o3 have been pushing the boundaries of language model evaluations with bold claims of being a direct path to AGI through what's called "scaling test-time compute" - essentially allocating more computational resources to the text generation phase rather than model training itself to push performance.

The last few years of AI progress were dominated by pre-training scaling laws - GPT-2 at 1.5 billion parameters, GPT-3 at 175 billion, and GPT-4 reportedly exceeding 1 trillion, with significant improvements between each GPT generation. However, this approach has hit two key limitations: larger models are showing diminishing returns, and we're running into constraints with quality web data availability. While there have been (and continue to be) improvements in training through techniques like data filtering, distillation methods, and synthetic data generation approaches, recent research focus has shifted to the other side of the pipeline: inference.

The core concept is straightforward: hard problems require a lot of thought. The harder the problem, the more time you would likely spend exploring potential solutions on your way to a final answer. This has been applied to the recent series of reasoning models, including OpenAI's O1, Google's Gemini 2.0 Flash thinking, DeepSeek R1, QwQ, and more. While the exact way to teach a language model how to learn to reason and correct itself isn't fully known yet, research suggests it's possible through novel multi-turn reinforcement learning techniques. These train models to learn an advanced self-correction process where they don't just answer directly but work towards an improved final response with a lengthened generation process.

In essence, these models perform a sort of search in their internal knowledge by generating and exploring potential solutions iteratively before finding the right approach and answer. This idea of search and verification can be extracted to augment existing language models in the second approach to scaling test time compute: search-based techniques. While these search approaches take different forms, they typically rely on a special model called a process reward model that's able to accurately score the steps taken to solve a problem. You might generate hundreds of candidates then use the model to score and aggregate top answers, having this be your final output. Or you can help guide the LLM's generation step by step, generating many first-step candidates, seeing which are on the right track, and expanding those highly scored ones until you reach a more improved final answer. Either way, implementations of this test-time compute scaling method have pushed the performance of small models up past those even 25 times its size!.

**SQL Query AI Research Reference:**

<https://arxiv.org/pdf/2403.20014> - SQL Writer Example Reference