CMPE 492
An AI Companion for Developers to Tune Complex Technologies

Onur Dilsiz
Yusuf Suat Polat

Advisors:
Bahri Atay Özgövde
Yunus Durmuş

Contents

Chapter 1

INTRODUCTION

In today's development landscape, many developers struggle with underperforming applications on complex systems. To address these challenges, developers choose asking an LLM by giving a prompt that includes the problem description, code, and maybe configuration context. This approach is getting more popular day by day since the models can provide some suggestions, which may not be the best but at least solve the problem. LLMs can be trained for a specific purpose, which inspires us for this project.

1.1 Broad Impact

This project addresses a crucial technical challenge: optimizing Apache Spark, which is a distributed data processing engine. Many developers who use Apache Spark are unaware of its best practices, resulting in inefficient code that wastes resources and increases the execution time of the program. Consequently, companies need to hire external experts to fix and improve their Spark-based systems.
The technology we plan to use in this AI companion is a large language model, which can be trained for specific use cases. By implementing an AI companion for developers of Apache Spark, this project aims to spread the knowledge of best practices and performance optimization on Apache Spark, enhancing developer productivity and code efficiency. This tool has the potential to reduce dependency on external experts, promote efficient use of computational resources, and make high-performance computing accessible to a broader range of developers. Furthermore, this project could be a game changer that result in the development of similar AI companions that troubleshoot resource-intensive systems.

1.2 Ethical Considerations

This project requires paying attention to ethical considerations that are mostly raised by the usage of LLMs in the implementation.
This project needs Spark metrics from Spark UI and the related code in order to analyze the programs efficiency. Even though AI companion does not access directly the actual datasets that are used by the programs, these metrics and the code are mostly confidential for any company. Therefore, it is essential that any data collected is only used for optimizing and troubleshooting.
Since the main technology behind the project is an LLM, there can be bias towards to training data after training the model. In order to prevent this bias as much as possible, training data should be diverse and be tested across various applications. In addition, misinformation and hallucination may be a problem of this project due to the usage of LLMs.

Chapter 2

PROJECT DEFINITION AND PLANNING

2.1 Project Definition

This project is an AI assistant to support developers in optimizing and troubleshooting Apache Spark applications. Apache Spark is a widely used distributed data processing engine known for its ability to process large volumes of data in clustered computing environments. However, Spark's performance tuning is often complex, requiring developers to manage numerous configuration settings, understand detailed execution plans, and apply best practices effectively. Many developers struggle to implement optimized Spark applications, leading to inefficient use of resources, increased execution time, and, consequently, higher operational costs. Due to these complexities, companies need to rely on external experts to diagnose performance issues and tune their Spark applications, which leads to additional expenses.
This AI companion is designed to inspect the metrics from the Spark UI with the program's code, identify the bottlenecks of the program, and then provide solutions for optimization.

2.2 Project Planning

2.2.1 Project Time and Resource Estimation

This project is expected to take 8 months, which covers CmpE 491 and 492, including research, data collection , model selection, and deployment phases.
Research:
Initial phase for this project requires a remarkable research about LLMs,

Apache Spark, multiagent tools and architectures. Foundation of the knowledge about these technologies is expected to take approximately seven weeks. Furthermore, this project requires continuous research for the whole span of the project. In the beginning, the research subjects on LLMs are about how LLMs work, how they are implemented and how they are trained for a specific objective. On the other hand, Apache Spark part includes the structure and operation of the system, how the developer gets the metrics from the Spark UI, how the metrics should be considered, and the best practices for an Spark application. After that, multiagent architectures and frameworks should be studied continuously to meet our project needs.

Data Collection:
This phase is about the gathering the data about the best practices and the optimization techniques which developers can utilize in the Apache Spark. It is expected to take 2 weeks, which includes the both collection and the preprocessing of the high quality dataset for the training of the chosen LLM which we are going to use in AI companion.

Model and Method Selection:
It is planned to utilize the AI assistant with only one LLM at first. Different prompting techniques is going to be tested while choosing the model and assessing the performance of the models. We are going to analyze several LLMs for adaptibility to technical domains, and performance of suggesting code improvements. This part is expected to take 2 weeks.

Testing and Evaluation:
This part is expected to be at the end of the first term. The AI assistant is going to be tested with a diverse dataset in both identifying the bottlenecks and the effectiveness of the solution provided by the tool. Then, the results that demonstrates the success rate of the project are going to be documented. The areas needing improvement is going to identified.

Research:
After the evaluation, research is going to take the stage again. We are going to research about the applicable new strategies and potential improvements according to project's state at that timespan. The research is expected to be about different LLMs and multiagent tools. At this point, we need to study the frameworks and architectures continuously, to be able to use the most suitable ones for our use cases.

Models selection:
In the second term, it is planned to try more large language models in order to make the best choice out of them. According to the results of our research phase, the most suitable LLM for the improvement areas is going to be selected.

System Integration:
We are planning to have an AI companion that can interact with the Spark applications in real time by inspecting the code and the Spark UI metrics at the same time. Then, the assistant is going to produce an evaluation of the application, provide suggestions for bottlenecks if applicable.

Testing and Evaluation:
After the integration phase, we are going to test the our AI companion in real time, with a dedicated test dataset in order to assess the performance of identifying the bottlenecks and the success of the suggested solutions. If possible, we plan to conduct user testing, by providing the tool to the developers. We plan to documentate the project with the evaluation of the results.

2.2.2 Success Criteria

Accurate Problem Identification: The companion should accurately identify significant bottlenecks in 70% of tested cases.

Effective Recommendations: The companion should provide relevant optimization suggestions..

Performance Improvements: The AI companion's solution should provide at least a 5% improvement in resource usage or execution time for Spark applications.

User Satisfaction: The qualitative feedback on ease of use, efficiency of the AI companion from the developers.

2.2.3 Risk Analysis

Data Security and Privacy: The AI assistant needs to access to the code and Apache Spark metrics in order to look for optimization strategies. This data can contain confidential data of the company, which may raise concerns

about data privacy.
Mitigation Strategy : Implement data retention protocols which ensures that the data is only used for optimization temporarily.

Model Bias and Misinformation:
Because of the biases in the training data or limitations of the LLM, the AI companion might produce incorrect or biased suggestions. Hallucinations, which refers to false information that LLM provides as if it were true can be an important problem for this project.
Mitigation Strategy:
Prepare high quality training dataset as diverse as possible.

Computational Resource Constraints:
Large language models are computationally expensive to train and deploy, and real time integration may need significant computational resources.
Mitigation Strategy:
Create a plan for managing costs and resources, and use thresholds to keep computational costs under control.

Potential Limitations in Identifying Complex Bottlenecks:
There can be complex performance bottlenecks in Spark applications, and identifying the certain issues with AI assistant might be challenging.
Mitigation Strategy:
Prepare a high quality dataset which demonstrates these complex ones, and ensure that the users learn which data must be shared with the model.

2.2.4 Team Work

This project is being implemented by 2 students and 2 advisors. Students are in charge of research, implementation, testing, and deployment of the project. Students and advisors meet regularly, once or twice a week, in order to assess the progress of the project and discuss necessary actions for the improvement of the project.

Chapter 3

RELATED WORK

Throughout the research part of the project, a significant number of articles were read. By doing so, the aim was to become familiar and up-to-date with current standards and methods in similar tasks. In addition, there are numerous frameworks and tools that have tried to perform similar tasks as this project. Those tools and frameworks are also studied. Here is the list of the papers and other resources that are studied by students and suggested by the project advisors.

LangChain, https://www.langchain.com

SparkMeasure Library, https://github.com/LucaCanali/sparkMeasure

Dr. Elephant Library, https://github.com/linkedin/dr-elephant

Brown, T . B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P ., ... Amodei, D. (2020). Language models are few-shot learners. Advances in Neural Information Processing Systems, 33, 1877-1901. https://arxiv.org/abs/2005.14165

Mishra, S., Khashabi, D., Baral, C., Choi, Y . (2021). Cross-task generalization via natural language crowdsourcing instructions. arXiv preprint arXiv:2104.08773. https://arxiv.org/abs/2104.08773

Reynolds, L., McDonell, K. (2021). Prompt programming for large language models: Beyond the few-shot paradigm. arXiv preprint arXiv:2102.07350. https://arxiv.org/abs/2102.07350

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F ., ... Le,

Q. V. (2022). Chain of thought prompting elicits reasoning in large language models. arXiv preprint arXiv:2201.11903. https://arxiv.org/abs/2201.11903

Wang, X., Kordi, Y ., Mishra, S., McCoy, T ., Patel, R., Liu, S., ... Le, Q. (2022). Self-consistency improves chain of thought reasoning in language models. arXiv preprint arXiv:2203.11171. https://arxiv.org/abs/2203.11171

Zhao, W., Wallace, E., Feng, S., Klein, D., Singh, S. (2021). Calibrate before use: Improving few-shot performance of language models. In International Conference on Machine Learning (pp. 12697-12706). PMLR. https://arxiv.org/abs/2102.09690

Li, X. L., Liang, P . (2021). Prefix-tuning: Optimizing continuous prompts for generation. In Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers) (pp. 4582-4597). https://arxiv.org/abs/2101.00190

Fang, C., Miao, N., Srivastav, S., Liu, J., Zhang, R., Fang, R., ... Homayoun, H. (2024). Large Language Models for Code Analysis: Do LLMs Really Do Their Job?. In 33rd USENIX Security Symposium (USENIX Security 24) (pp. 829-846).

Yu, Y., Rong, G., Shen, H., Zhang, H., Shao, D., Wang, M., ... Wang, J. (2024). Fine-tuning large language models to improve accuracy and comprehensibility of automated code review. ACM transactions on software engineering and methodology.

Pornprasit, C., Tantithamthavorn, C. (2024). Fine-tuning and prompt engineering for large language models-based code review automation. Information and Software Technology, 175, 107523.

Wadhwa, N., Pradhan, J., Sonwane, A., Sahu, S. P., Natarajan, N., Kanade, A., ... Rajamani, S. (2024). Core: Resolving code quality issues using llms. Proceedings of the ACM on Software Engineering, 1(FSE), 789-811.

https://docs.agno.com/introduction

https://browser-use.com

https://github.com/unclecode/crawl4ai

Chapter 4

METHODOLOGY

4.1 Overview

The methodology of this project is centered on developing an AI companion capable of diagnosing and recommending solutions for performance issues in Apache Spark applications. The AI model, based on a large language model (LLM) framework, will be trained to recognize inefficiencies in Spark execution patterns and provide optimization suggestions. Our approach consists of three primary stages: data collection, prompt preparation, and evaluation. Each stage is designed to ensure that the AI companion not only provides accurate performance insights but also adheres to data security and the minimization of bias.

4.2 Data Collection

We opt for specific common bad practices of Apache Spark codes. We need a dataset corresponding the cases to develop our AI companion. Since we need a diverse dataset, we chose to get some pySpark code from GitHub, and generated some sentetic code using ChatGPT.

For the cases that we consider about analysis on Spark UI, we are going to generate some synthetic applications which we can evaluate the performance of our companion in terms of reading the metrics from UI, and suggesting correct optimizations.

4.3 Prompt Preparation

To select the most suitable prompts for our cases, we tried multiple prompt templates, which are modified to aim each case specifically. Throughout the experiments, it was observed to specify the output format as JSON in order to make the response evaluation more feasible.

For Spark UI cases, we are doing research about how to inserting the UI into our prompt. We have tried browser-use agent, direct screenshots and pdf versions of the web pages.

4.4 Evaluation and Testing

Our evaluation workflow is designed to systematically assess model performance. For each file in the dataset, the code contained within the file is embedded into predefined prompts. These prompts are then processed by various models. The response from each model is structured in JSON format, which is subsequently parsed to extract a binary detection result. This result is inserted another file that also includes the expected results for evaluating the models' metrics in detecting specific cases.

Similarly, for the Spark UI cases evaluation and testing are going to be done by the same methodology.

Chapter 5

REQUIREMENTS SPECIFICATION

This chapter presents the functional and non-functional requirements for the project "An AI Companion for Developers to Tune Complex Technologies." The primary goal of this AI companion is to help developers optimize Apache Spark applications by providing suggestions on code and configuration improvements based on the program's performance metrics. The requirements are specified below in detail, along with corresponding use case diagrams.

5.1 Functional Requirements

The functional requirements define the core capabilities that the AI companion should provide. These requirements include both high-level features and specific functionalities that the AI assistant must support.

5.1.1 Core Functionalities

(i) **Analyzing Spark Application Metrics**:

- 5.1.1.1.1 The AI companion shall retrieve and analyze performance metrics from the Apache Spark UI.
- 5.1.1.1.2 The AI companion shall identify potential bottlenecks based on metrics such as memory usage, CPU utilization, and job execution times.

(ii) **Providing Optimization Suggestions**:

- 5.1.1.2.1 The AI companion shall suggest code improvements based on identified bottlenecks.
- 5.1.1.2.2 The AI companion shall suggest configuration adjustments for better resource utilization.
- 5.1.1.2.3 The AI companion shall provide explanations for each suggestion, so developers understand the reason behind each recommendation.

(iii) **Learning from Feedback**:

- 5.1.1.3.1 The AI companion shall allow developers to provide feedback on the suggestions.
- 5.1.1.3.2 The AI companion shall adjust future recommendations based on collected feedback to improve accuracy and relevance.

5.2 Non-Functional Requirements

The non-functional requirements specify the quality attributes, constraints, and standards that the AI companion must adhere to. These requirements include performance, usability, reliability, security, and scalability.

5.2.1 Performance Requirements

- 5.2.1.1 The AI companion should respond to a developer's request for recommendations within 30 seconds.

5.2.2 Usability Requirements

- 5.2.2.1 The AI companion shall have a user-friendly interface with easy-to-understand suggestions and explanations.
- 5.2.2.2 The AI companion should allow developers to quickly provide feedback on each suggestion.

5.2.3 Reliability Requirements

- 5.2.3.1 The AI companion should have a suggestion accuracy rate of at least 70 percent for identifying meaningful performance improvements.
- 5.2.3.2 The AI companion should operate without crashes or significant bugs during its operation.

5.2.4 Security Requirements

- 5.2.4.1 The AI companion shall not retain any sensitive code or metrics unless explicitly allowed by the developer.
- 5.2.4.2 The AI companion shall follow secure data handling protocols to prevent unauthorized access to Spark application data.

5.2.5 Scalability Requirements

- 5.2.5.1 The AI companion should be able to handle multiple concurrent users without performance degradation.
- 5.2.5.2 The AI companion should be adaptable to various configurations of Apache Spark clusters.

5.3 Use Cases

This section provides detailed descriptions of the use cases for the AI companion. Each use case outlines the steps, actors involved, and success criteria.

5.3.1 Use Case 1: Analyzing Spark Application Metrics

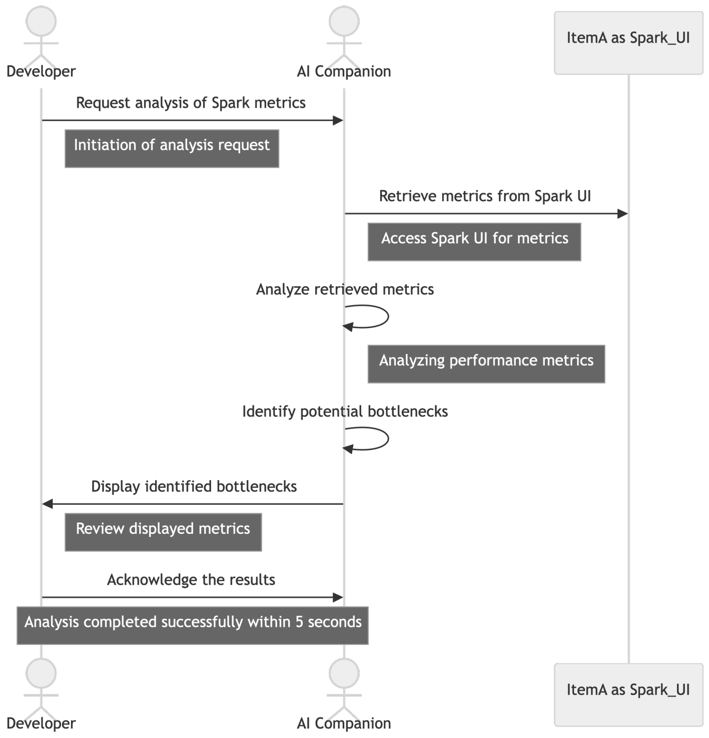

Developer
AI Companion
ItemA as Spark_UI
Request analysis of Spark metrics
Initiation of analysis request
Retrieve metrics from Spark UI
Access Spark UI for metrics
Analyze retrieved metrics
Analyzing performance metrics
Identify potential bottlenecks
Display identified bottlenecks
Review displayed metrics
Acknowledge the results
Analysis completed successfully within 5 seconds
Developer
AI Companion
ItemA as Spark_UI

Figure 5.1: Use Case Diagram 1 for AI Companion

Actors: Developer, AI Companion

Description: The developer requests the AI companion to analyze a Spark application's performance metrics. The AI companion retrieves the metrics and identifies any potential bottlenecks.

Steps:

(i) Developer initiates a request for analysis.

(ii) AI companion retrieves metrics from the Spark UI.

(iii) AI companion analyzes the metrics and identifies potential bottlenecks.

(iv) AI companion displays the identified bottlenecks to the developer.

Success Criteria: The AI companion successfully identifies and displays any major performance bottlenecks within 5 seconds.

5.3.2 Use Case 2: Providing Optimization Suggestions

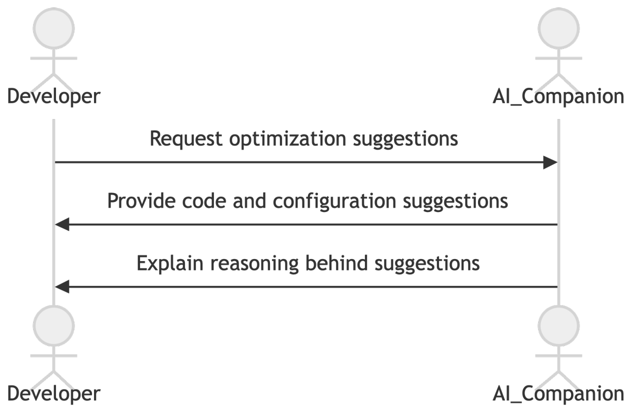

Developer
AI_Companion
Request optimization suggestions
Provide code and configuration suggestions
Explain reasoning behind suggestions
Developer
AI_Companion

Figure 5.2: Use Case Diagram 2 for AI Companion

Actors: Developer, AI Companion

Description: The developer requests optimization suggestions after viewing the performance analysis. The AI companion provides recommendations for code or configuration changes to address the bottlenecks.

Steps:

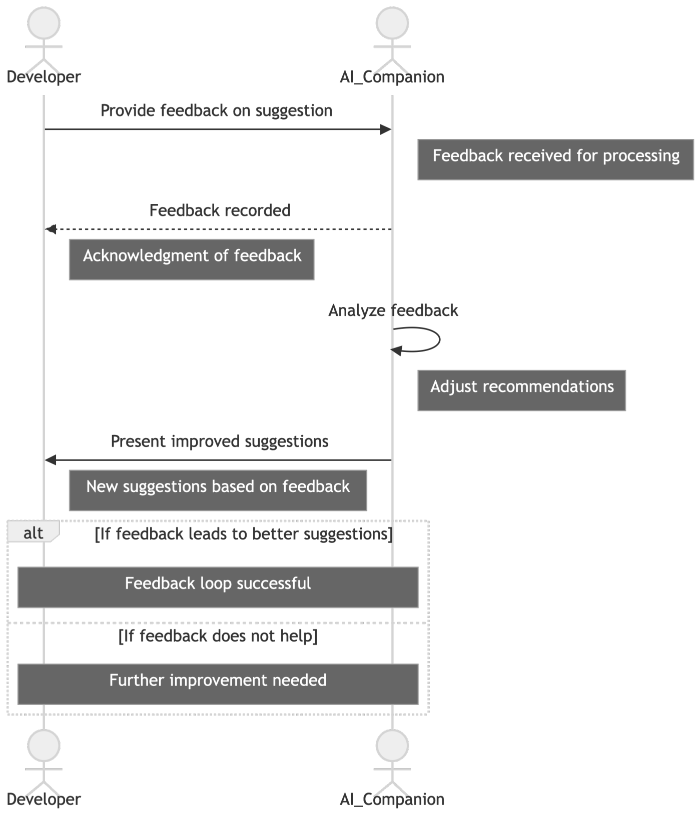
(i) Developer requests optimization suggestions.

(ii) AI companion provides suggestions for code and configuration improvements.

(iii) AI companion explains the reasoning behind each suggestion.

Success Criteria: The AI companion provides relevant suggestions with explanations, and the developer finds at least 70 percent of the suggestions useful.

5.3.3 Use Case 3: Learning from Feedback



Developer
AI_Companion
Provide feedback on suggestion
Feedback received for processing
Feedback recorded
Acknowledgment of feedback
Analyze feedback
Adjust recommendations
Present improved suggestions
New suggestions based on feedback
alt
[If feedback leads to better suggestions]
Feedback loop successful
[If feedback does not help]
Further improvement needed
Developer
AI_Companion

Figure 5.3: Use Case Diagram 3 for AI Companion

Actors: Developer, AI Companion **Description:** The developer provides feedback on the AI companion's suggestions. The AI companion incorporates this feedback to improve future recommendations. **Steps:**

(i) Developer provides feedback on a suggestion.

(ii) AI companion records the feedback.

(iii) AI companion adjusts future recommendations based on collected feedback.

Success Criteria: The AI companion successfully records feedback and improves the relevance of future suggestions based on similar scenarios.

Chapter 6

DESIGN

6.1 Information Structure

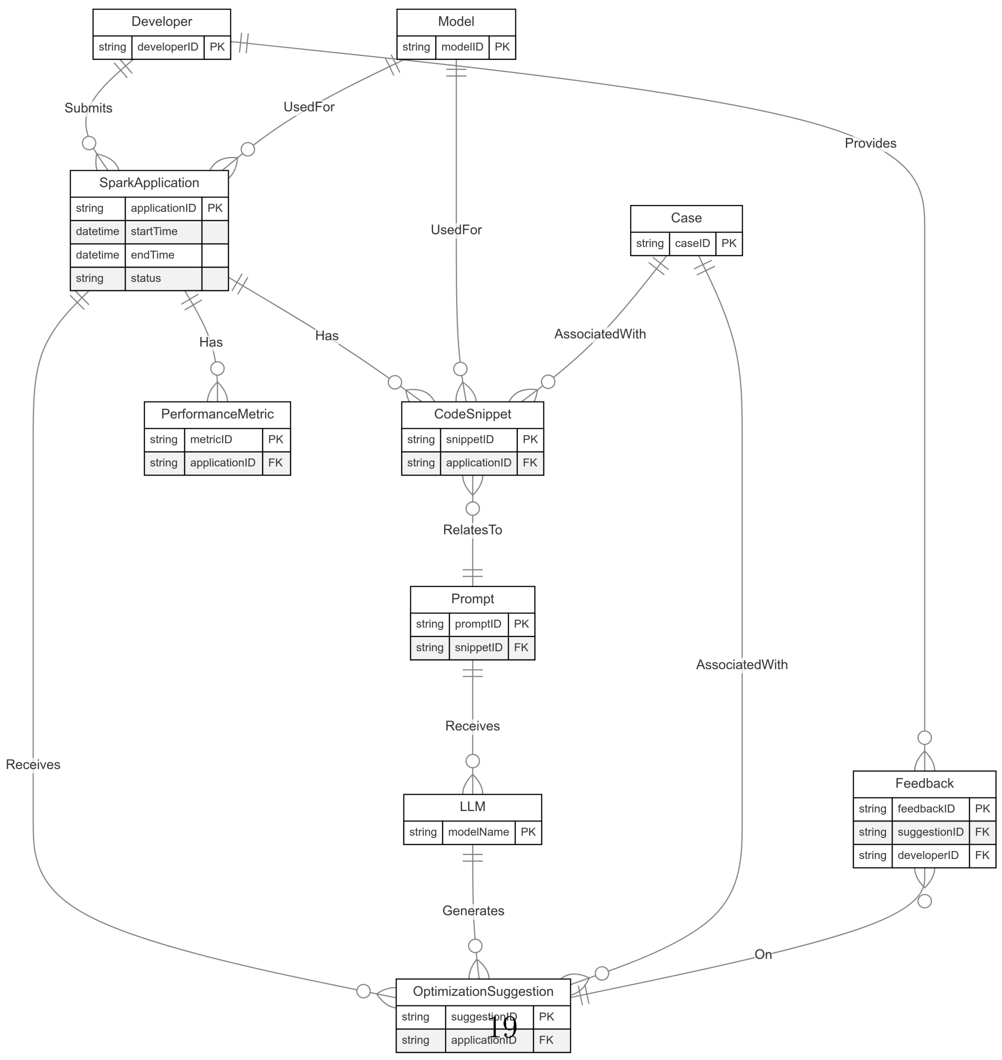


Figure 6.1: Entity Relation Diagram

6.2 Information Flow

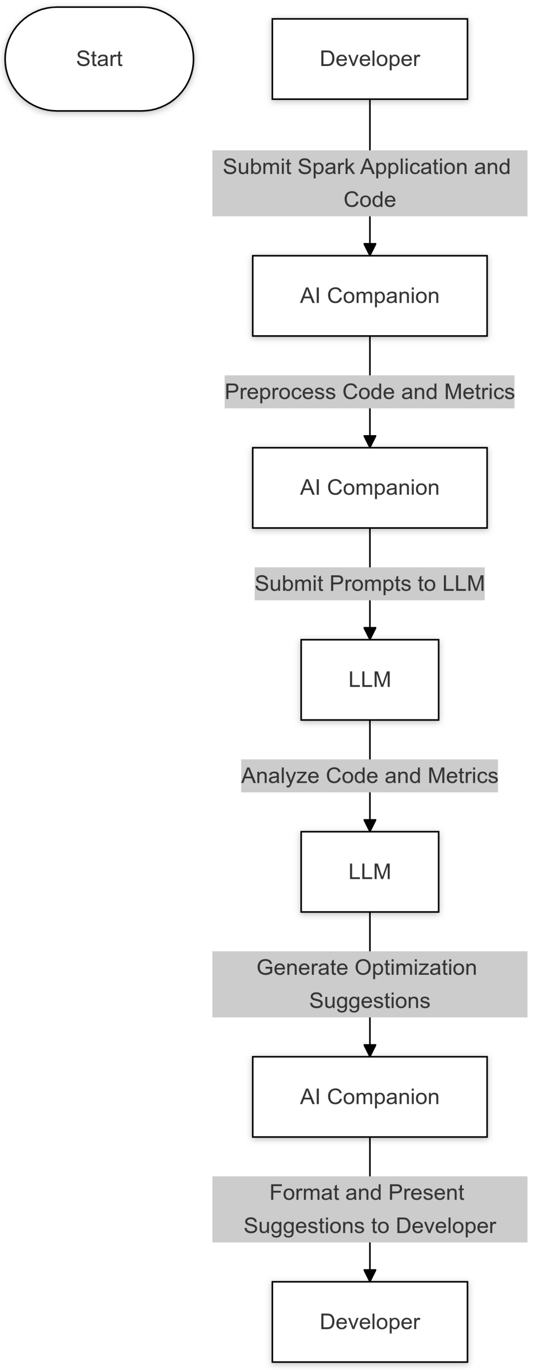
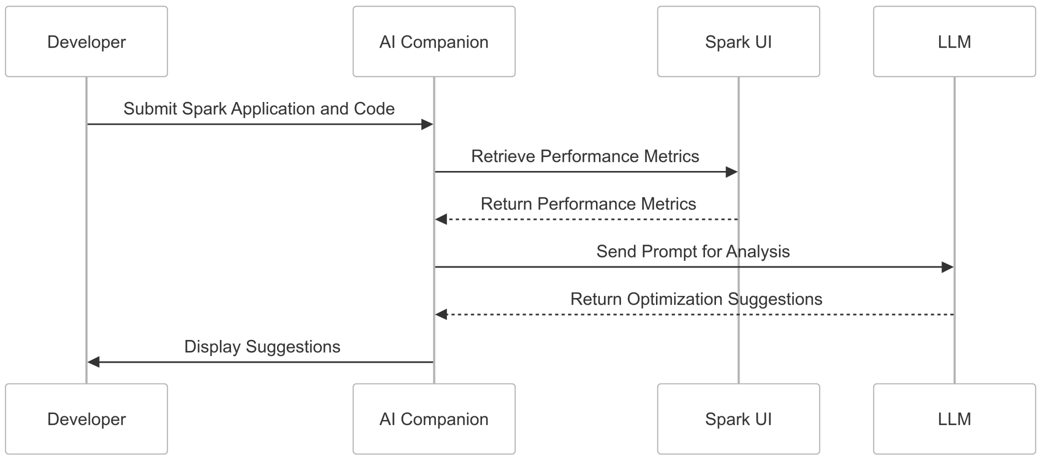

Start
Developer
Submit Spark Application and Code
AI Companion
Preprocess Code and Metrics
AI Companion
Submit Prompts to LLM
LLM
Analyze Code and Metrics
LLM
Generate Optimization Suggestions
AI Companion
Format and Present Suggestions to Developer
Developer

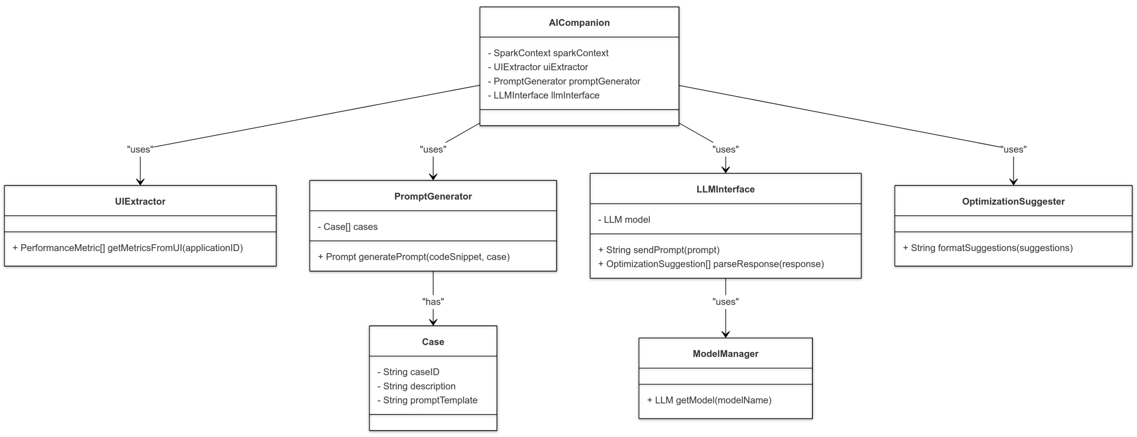
Figure 6.2: Activity Diagram



Developer
AI Companion
Spark UI
LLM
Submit Spark Application and Code
Retrieve Performance Metrics
Return Performance Metrics
Send Prompt for Analysis
Return Optimization Suggestions
Display Suggestions
Developer
AI Companion
Spark UI
LLM

Figure 6.3: Sequence Diagram

6.3 System Design



AICompanion
- SparkContext sparkContext
- UIExtractor uiExtractor
- PromptGenerator promptGenerator
- LLMInterface llmInterface
"uses"
"uses"
"uses"
"uses"
UIExtractor
+ PerformanceMetric[] getMetricsFromUI(applicationID)
PromptGenerator
- Case[] cases
+ Prompt generatePrompt(codeSnippet, case)
LLMInterface
- LLM model
+ String sendPrompt(prompt)
+ OptimizationSuggestion[] parseResponse(response)
OptimizationSuggester
+ String formatSuggestions(suggestions)
"has"
"uses"
Case
- String caseID
- String description
- String promptTemplate
ModelManager
+ LLM getModel(modelName)

Figure 6.4: Class Diagram

Chapter 7

IMPLEMENTATION AND TESTING

7.1 Implementation

The main focus of our project is to detect bad practices, which make the Spark application suboptimal. In the first version of our AI companion, we prioritized the bad practices which we can detect by looking at only the code of the application. We opt for these five cases:

- **Avoid UDFs:** Use Spark SQL functions for better performance and optimization.
- **Prefer `coalesce()` over `repartition()`:** Minimize shuffling when reducing partitions.
- **Use `mapPartitions()` over `map()`:** Initialize heavy resources once per partition for improved efficiency.
- **Use DataFrame/Dataset over RDD:** Leverage Catalyst and Tungsten optimizations to avoid expensive RDD serialization.
- **Use Optimized Formats (Parquet/Avro):** These formats are faster, compressed, and optimized for big data analytics.
- **Initial Executor/Autoscaling:** In addition to those cases, we specifically aim to detect cases that have room for improvement by using monitoring the suitable UI tabs.

Our project requires having prompts for any case in order to inspect it using LLM. Therefore, we first prepared prompts that include a code segment, which is filled by our AI companion taking the file from the user. The first step for our application is initializing a selected model using langchain libraries. Subsequently, the code file is expected from the user. After the user provided the code file, our AI companion inserts the code inside of the predefined prompts and sends these prompts to the chosen LLM. Then, LLM returns a response for each predefined prompt. The response has the following format:

- `"detected"`: A boolean value indicating whether the bad practice in the prompt was detected in the analyzed code.
- `"occurrences"`: An integer value representing the number of occurrences of that suboptimal usage.
- `"response"`: An array of objects, where each object includes:
  - `"operation"`: The specific operation detected, along with its location in the code (e.g., line number or a code snippet).
  - `"improvementExplanation"`: A detailed explanation of why this change should be done.
  - `"caseEquivalent"`: A code snippet demonstrating the equivalent transformation using the best practice.
  - `"benefits"`: A summary of the benefits of switching to ”caseEquivalent”.

7.2 Testing

In order to test our application, we needed a proper dataset which includes both sentetic and real life application codes. Thus, we searched for the code files on GitHub, and generated some files using LLMs.
We selected the following six models to assess our application and performance of different models for our purpose.

- `llama-3.1-8b-instruct-maas`
- `gpt-3.5-turbo-0125`
- `gpt-4o`
- `gemini-1.0-pro-002`
- `gemini-1.5-flash-002`

- `gemini-2.0-flash-exp`

For each file in our dataset, we send 6 prompts corresponding to the predefined cases and and an additional "all-in-one" prompt that evaluates multiple cases simultaneously to the models above, then, our AI companion writes the responses to one JSON and one TXT file per each prompt. Then, we have parsed JSON responses to a CSV which contains the fields below:

- **Model**
- **Case**
- **File**
- **Detection**: A boolean demonstrating whether the bad practice was detected.
- **Occurrences**: The number of detections the bad practice was found in the file.

We prepared a ground truth CSV file for our dataset which contains the same fields except the model. We calculated accuracy, precision, recall, and F1-score by comparing the model-generated CSV files with the ground truth.

We believe that the same method can be applied for our Apache Spark UI cases.

Chapter 8

RESULTS

Model	Accuracy	Precision	Recall	F1 Score
gemini-1.0-pro-002	0.6484	0.7117	0.4847	0.5766
gemini-1.5-flash-002	0.7727	0.7178	0.8896	0.7945
gemini-2.0-flash-exp	0.7879	0.7818	0.7914	0.7866
gpt-3.5-turbo-0125	0.6242	0.5970	0.7362	0.6593
gpt-4o	0.8000	0.8255	0.7546	0.7885
llama-3.1-8b-instruct-maas	0.6333	0.6129	0.6994	0.6533

Table 8.1: Model Evaluation Metrics for All in One Prompt

Model	Accuracy	Precision	Recall	F1 Score
gemini-1.0-pro-002	0.6212	0.5819	0.8282	0.6835
gemini-1.5-flash-002	0.7758	0.7259	0.8773	0.7944
gemini-2.0-flash-exp	0.7576	0.7128	0.8528	0.7765
gpt-3.5-turbo-0125	0.5000	0.4968	0.9571	0.6541
gpt-4o	0.8182	0.8121	0.8221	0.8171
llama-3.1-8b-instruct-maas	0.4667	0.4745	0.7423	0.5789

Table 8.2: Model Evaluation Metrics for Separate Prompts

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

Precision metric indicates how much of the detection should be detected in our case. For example, llama model is prone to detect when the cases are provided in separate prompts even if there is no error. However, gpt-4o does not show a big difference in precision.

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

Recall metric shows that how much of the detection could not be done by model. When all cases provided in one prompt, models are more prone to fail detection comparing the separate prompts. When the cases are given in separate prompts gpt3.5-turbo-0125 has 0.9571 score on recal, however, it has precision as 0.4968. It also shows that separate prompts leads to more detection for this model as well.

In these tables, we can see that gpt-4o outperforms other models since it has good balance between different scenarios with higher scores.

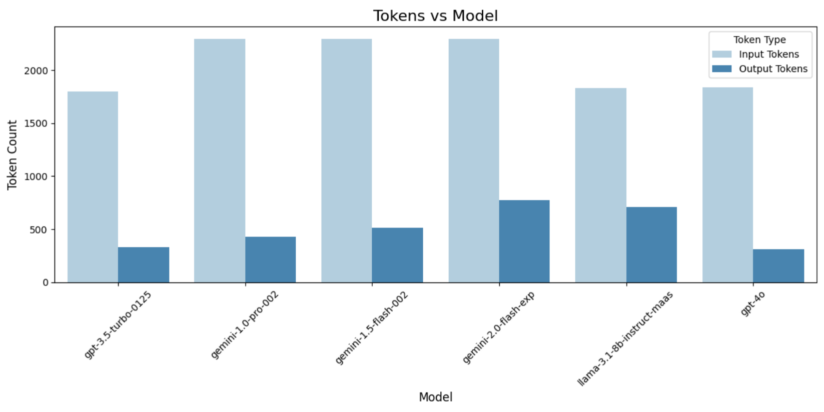


Figure 8.1: Average Input-Output Tokens used per prompt

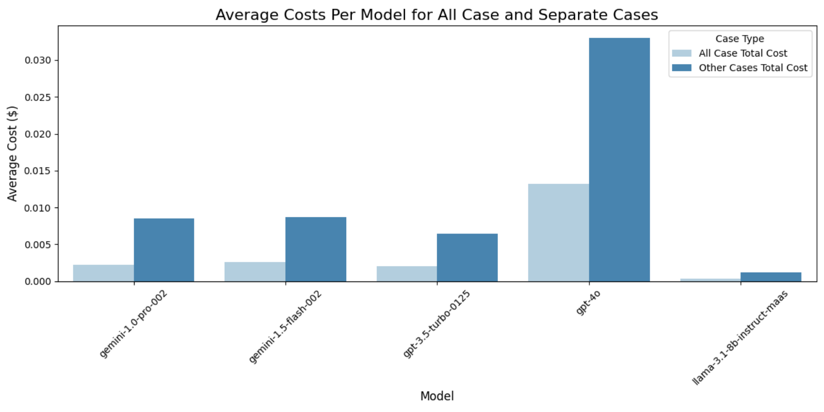


Figure 8.2: Average cost per case

We have inspected the cost per file for different models. We observed that having one prompt including all cases decreases the cost remarkably.

However, the cost per file for gpt-4o, which is the most expensive one, is just 3 cents which is most probably ignorable for the companies using Apache Spark.

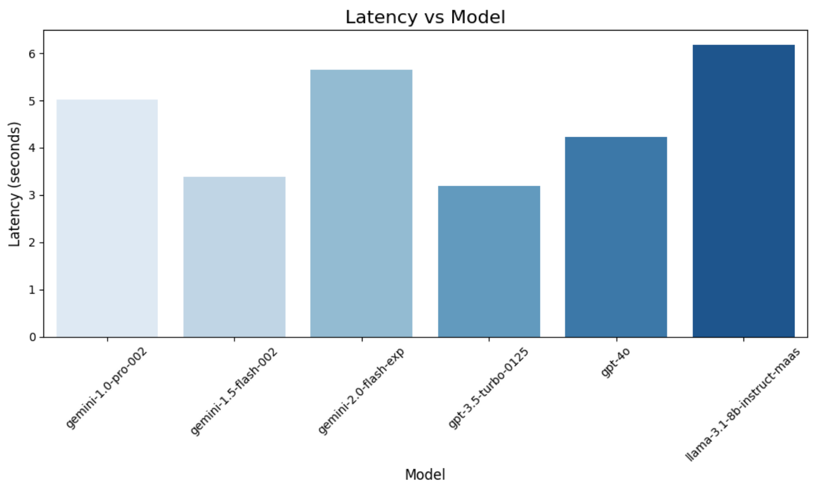


Figure 8.3: Average latency per separate prompt

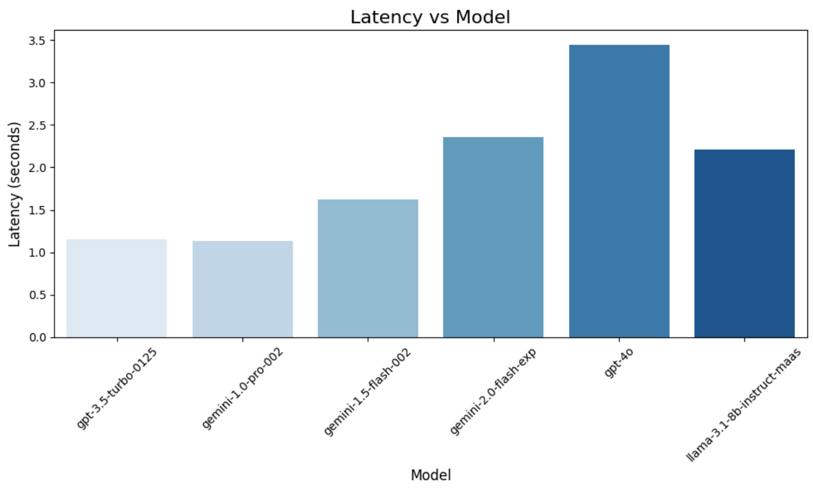


Figure 8.4: Average latency per case when all in one prompts used

Figure 8.4 shows the latency divided by 5 for all in one prompts since there are five cases that we inspected. We can observe that all in one scenario is a faster process as we expect. Comparing two graphs, we can see that longer prompts take more time in gpt-4o than the others.

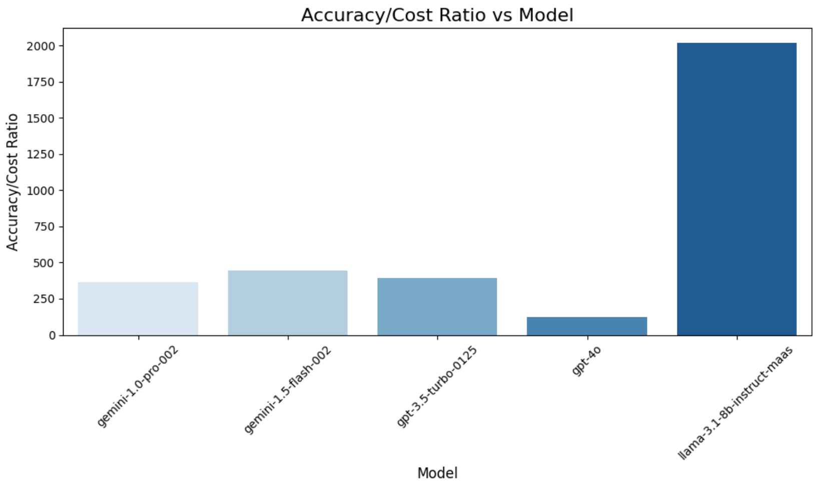


Figure 8.5: Accuracy/cost

This graph demonstrates the relationship between accuracy and cost by dividing accuracy by cost. Due to the low price of llama, it is the most profitable model as it is crystal clear in the chart.

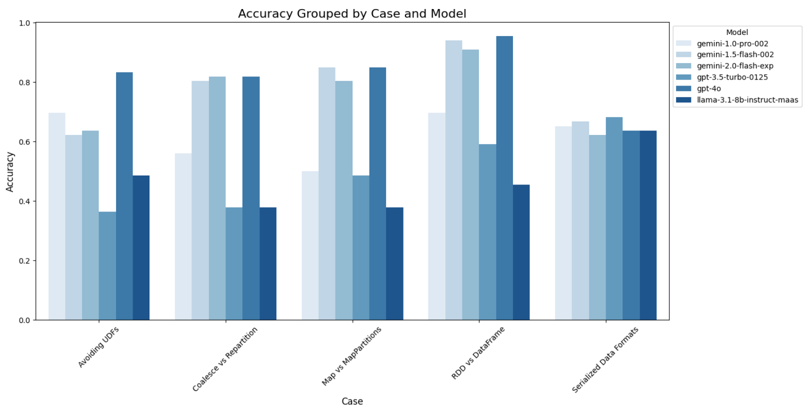


Figure 8.6: Accuracy grouped by case for separate prompts

We have observed that LLMs are not as successful in Serialized Data Formats as in other cases. In our perspective, the reason for this can be the popularity of CSV. LLMs can interpret it as normal despite having instructed, Parquet should be used.

Chapter 9

CONCLUSION

In this project, we focused on the challenge of optimizing complex technologies using a Large Language Model (LLM)-based AI companion specializing on Apache Spark. By focusing on common bad practices and performance metrics from Spark applications, we aimed to provide suggestions to developers for improving code efficiency and performance. The methodology combined Spark UI metric extraction, and structured prompts to guide the LLM in identifying bottlenecks and generating optimization recommendations.

Our experiments has shown that recently developed models like GPT-4o achieves high accuracy, as high as 80 percent for our predefined cases. According to our results, generic prompts are more effective in terms of cost and faster than using specific prompts for each case, making them a better choice. However, the study also highlighted key trade-offs between latency, cost, and accuracy. While models like GPT-4o delivered the best performance, they incurred slightly higher costs.

For the future, this work builds a foundation for building more robust tools for developers. Future directions might include developing an IDE extension or GUI with real-time code suggestions, integrating dynamic Spark UI analysis, and enabling developers to log and manage recurring optimization cases. Further research into multi-agent systems, prompting strategies, and more diverse datasets can improve the companion's accuracy and flexibility across a broader range of technologies, reducing time, costs, and resource inefficiencies in complex technologies like Apache Spark.