On Building an End-to-end Prototype System for Harvesting Performance Characteristics of Code Snippets

Michał Bodziony
IBM Poland Software Lab Kraków
Kraków, Poland *michal.bodziony@pl.ibm.com*

Robert Wrembel
Poznan University of Technology
Poznań, Poland *robert.wrembel@put.poznan.pl*

Oleksii Bulenok
Jagiellonian University
Kraków, Poland *a.bulenok@student.uj.edu.pl*

Anastasiia Ganusina
Jagiellonian University
Kraków, Poland *anka.ganusina@student.uj.edu.pl*

Wiktor Prządka
Jagiellonian University
Kraków, Poland *wiktor.przadka@student.uj.edu.pl*

Adrian Suwała
Jagiellonian University
Kraków, Poland *adrian.suwala@student.uj.edu.pl*

Abstract

End-to-end solutions for running data engineering experiments and getting insights from them are gaining more and more interest from research communities. The insights are typically learned from applying machine learning algorithms on experimental data. In this context, experiments repeatability and open access experimental data become new important trends. In this paper we propose an end-to-end prototype architecture for collecting and analyzing performance characteristics of code snippets. The system was originally built, deployed, and tested for the problem of building performance models of user defined functions in data integration processes.

Keywords: performance experiments, architecture design, containerization

1. Introduction

The most recent trends in data engineering are based on excessive performance experiments and on applying machine learning (ML) techniques on collected experimental data, e.g., [1, 2, 5, 6]. ML algorithms are run for the purpose of getting insights from experimental data and for building various performance, prediction, and optimization models, e.g., [3, 4]. To this end, experiments repeatability and open access experimental data become new important requirement. Moreover, curated experimental data are valuable research assets.

In this paper we **contribute** an end-to-end experimental prototype architecture for data engineering and data science. The proposed architecture supports the automation of: (1) the process of data collection, (2) data pre-processing for ML algorithms, (3) building of ML models, and (4) the analysis of results. Our architecture is based on micro-services. It was originally designed and deployed for building performance models of user defined functions (UDFs) provided as black-boxes, within data integration processes.

2. The End-to-end Prototype Architecture

The developed prototype architecture to facilitate data harvesting and model training is based on micro-services, designed as an orchestration of several self-contained services. The architecture is shown in Figure 1. The main functionality is included in two components: (1) experiment execution and (2) results analysis. The architecture is build on the software components outlined in this section. All of them run as micro-services.

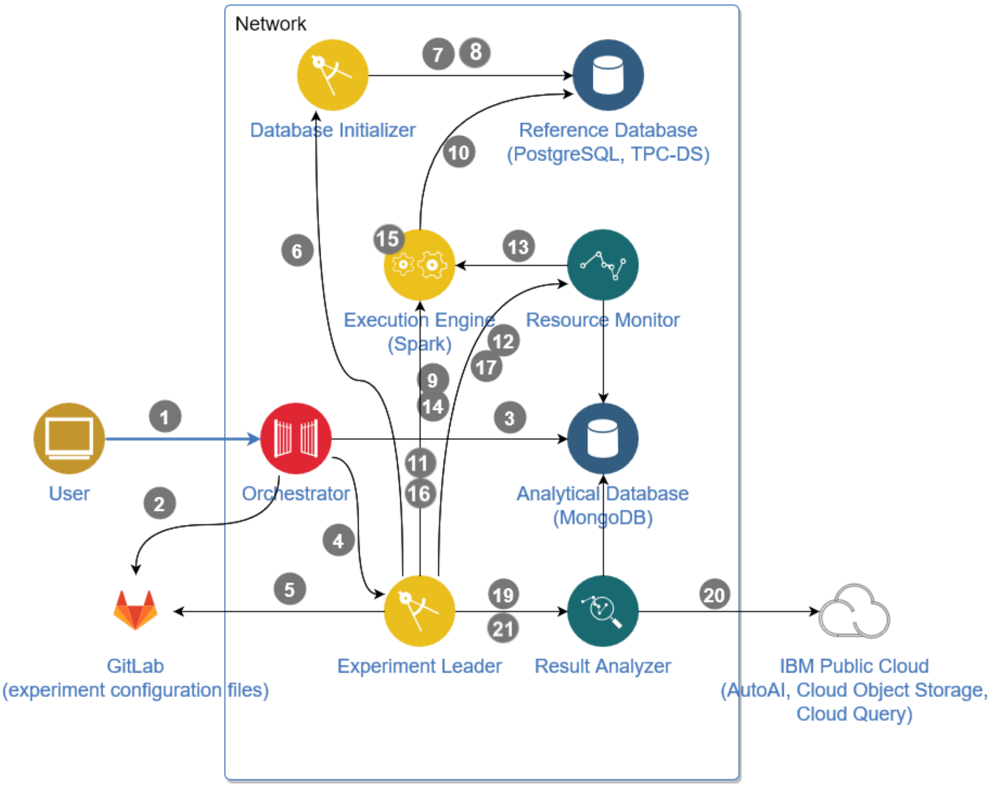

Fig. 1. The architecture for harvesting experimental data - components and their interactions

Orchestrator is an entry point to the execution environment. It initiates tasks involving experiments and analyses, by sending requests to *Experiment Leader*. Upon receiving a request to start a task, it utilizes a GitLab API token to retrieve the contents of a configuration file containing definitions of experiments and analyses. The parameters from a user request are forwarded to *Experiment Leader*, including: order ID, link to the configuration file, and GitLab API token. *Orchestrator* responds to the client with a task ID and a list of experiment IDs in the current task.

Experiment Leader manages all runs of experiments and analyses. It operates through exposed endpoints for receiving requests from *Orchestrator*. Upon receiving a command to start a task, *Experiment Leader* retrieves the configuration file from GitLab using a link and a private access token received from *Orchestrator*. The configuration file contains definitions of experiments and analyses. A special type of experiment called a *warm-up* is typically executed before main experiments to perform a system warm-up (e.g., populating caches) on *Referenced Database* engine as well as on *Execution Engine*. Analyses are done by sending for each analysis a request with data from a configuration file to *Result Analyzer*. *Experiment Leader* makes available a callback endpoint to receive a notification from *Result Analyzer* when an analysis is finished.

Reference Database is used to store data processed by code snippets under test. The database is implemented in PostgreSQL. During deployment, prepared databases are copied and mounted as a volume to a container. The system allows to use multiple databases, in which case, each of them is spawned from the same image as a separate container.

Database Initializer spawns and destroys containers for *Reference Databases*. Its functionality follows a prototype design pattern, where different configurations of databases used in experiments are instantiated (generated and loaded) before runtime. The service creates containers to which copies of databases are mounted as data volumes. *Database Initializer* manages the creation and removal of these copies and containers, whenever needed. To facilitate the creation and deletion of containers, the Docker SDK for Python is used. Additionally, a volume is attached to manage directories located on the host, pointing to the directory where the original databases and their copies are located.

Execution Engine plays a key role in running code snippets on the data from *Reference Database*. Apache Spark is used as a data processing tool. Prior to executing experiments, *Execution Engine* undergoes preparation, i.e., the service receives connection properties and required database tables for execution. Spark establishes multiple connections to the database via JDBC, creating temporary views for the tables. Having finished the preparation step, the execution of a specific code snippet can commence upon receiving a request from *Experiment Leader*, along with the code snippet. Dedicated *Experiment Leader* callback endpoints are used by *Execution Engine* to notify when the execution/ preparation is finished.

Resource Monitoring observes system resource usage during the execution of code snippets in *Execution Engine*. Collected data are added to a batch and sent as a group of documents to *Analytical Database*. When monitoring begins, two threads are created to handle a data flow. The first one is responsible for gathering data from Docker and adding them to a queue, and the second thread transforms them and adds timestamps to batches. It also sends batches when their size reaches a certain threshold.

Result Analyzer is responsible for post-data-gathering tasks. Upon receiving a request from *Experiment Leader*, a list of task identifiers supplied in the request is used to determine data to collect for analysis. *Result Analyzer* retrieves all measurement data matching these order IDs and can further limit data analysis to a subset of experiments defined in particular tasks. After filtering the data, *Result Analyzer* transforms them from an array of documents delivered by *Analytical Database* to a list of records in a CSV file. Following data transformation to the CSV format, additional pre-processing takes place, as defined in optional settings passed in the body of a request to start an analysis. Finally, the resulting CSV file is stored in IBM Cloud Object Storage, and its path is then provided as input to *AutoAI* for further processing. The returned training report in JSON format is saved in *Analytical Database* along with the analysis ID, for manual inspection.

AutoAI is used to optimize model training and evaluation. It automates the entire pipeline of model optimization. To efficiently explore the search space of models, *AutoAI* employs the *RBfOpt* algorithm, designed for solving box-constrained mathematical optimization problems with costly evaluation of an objective function. Feature engineering in *AutoAI* is automated using the Cognito engine. *AutoAI* evaluates generated models using typical metrics, i.e., accuracy, precision, recall, and F1. All these metrics are compiled into a report returned upon finishing a run and saved in a Cloud Object Storage bucket.

Analytical Database stores among others data harvested by resource monitoring during experiments, labels, code snippets for executed experiments, and training reports returned by *AutoAI*. MongoDB is used as the analytical database due to its compatibility with documents read from the Docker Engine and the flexibility of its document-based data model.

2.1. The Main Flow

The main flow of the processing is described as an enumerated sequence of 21 stages, providing a general intuition of how the services act and communicate. The sequence of actions is depicted in Figure 1.

- (1) *Orchestrator* receives a request to start an order, along with details concerning the number and type of repetitions for warm-ups and experiments.
- (2) *Orchestrator* downloads the configuration file from GitLab.
- (3) *Orchestrator* loads experiment labels (code snippets) into *Analytical Database*.
- (4) *Orchestrator* starts a task by sending a request with the link to the configuration file, the GitLab API token, order ID, and repetition details to *Experiment Leader*.
- (5) *Experiment Leader* loads the configuration file using GitLab API.
- (6) *Experiment Leader* sends a request to *Database Initializer* to prepare the database.
- (7) *Database Initializer* prepares *Reference Database*.
- (8) *Experiment Leader* receives notification on the dedicated callback endpoint from *Database Initializer* after the database has been prepared. The request contains the connection properties required to establish the connection to the *Reference Database*.
- (9) *Experiment Leader* sends a request to *Execution Engine* to prepare data for the experiment. The request contains connection properties and tables required for the execution of a code snippet.
- (10) *Execution Engine* prepares data, establishing connections to *Reference Database*.
- (11) *Experiment Leader* receives notification on the dedicated callback endpoint from *Execution Engine* after the data were prepared, with the status of the data preparation.
- (12) *Experiment Leader* sends a request to *Resource Monitoring* to start monitoring (not applicable for warm-ups).
- (13) *Resource Monitoring* monitors resources used by *Execution Engine* and streams the collected data into *Analytical Database* (not applicable for warm-ups).
- (14) *Experiment Leader* sends a request to *Execution Engine* to start the execution. The request includes a code snippet.
- (15) *Execution Engine* executes the code snippet using Spark SQL.
- (16) *Experiment Leader* receives notification on the dedicated callback endpoint from *Execution Engine* after the execution finished, with the status of the execution.
- (17) *Experiment Leader* sends a request to *Resource Monitoring* to stop monitoring (not applicable for warm-ups).
- (18) Steps 6-17 are repeated for each experiment.
- (19) *Experiment Leader* sends a request to *Result Analyzer* to start the analysis.
- (20) *Result Analyzer* analyzes the collected data.
- (21) *Experiment Leader* receives a notification on the callback URL from *Result Analyzer* after its analysis was finished, with the status of the analysis.

2.2. Deployment and Evaluation

Both Docker and Docker Compose are used to simplify the deployment of the entire architecture. Each service runs in a separate container. Docker Compose facilitates the injection of environment variables into each component. The use of IBM AutoAI requires a properly configured Watson Studio and Machine Learning service instance in IBM Cloud, along with credentials providing both read and write access to a Cloud Object Storage bucket. Additionally, cloud pre-processing requires an SQL Cloud Query instance.

The proposed architecture was deployed to run experiments on UDFs, to collect their performance characteristics (CPU, RAM, I/O). Based on these characteristics, performance models of UDFs were built by means of ML algorithms. The experiments were conducted on a server with 12 CPU cores and 31.27 GiB of RAM, under Ubuntu 18.04.5. The Docker Engine in version 19.03.6 with 1.40 API version was running on the server.

3. Conclusions

The data harvesting environment proposed in this paper allows collecting performance data and creating models in a mostly automated way, with all settings grouped in a single configuration file. Additional work, mainly in the space of database generation and management, would allow this environment to be scaled and effectively used in a larger scope as part of a bigger cluster. The experiments show that building performance models of UDFs, using ML can be done, with acceptable accuracy. To this end, methods for multi-class classification are suitable. In our work we used ML automation tool IBM AutoAI, which allowed to build performance models without any manual model selection or tuning.

Acknowledgements. The work of Michał Bodziony is related to his employment at IBM Polska Sp z o.o. Additionally, his work is supported by the Applied Doctorate grant no. DWD/4/24/2020 from the Polish Ministry of Education and Science.

References

[1] Baeza-Yates, R.: The Limitations of Data, Machine Learning and Us. In: *Int. Conf. on Management of Data (SIGMOD/PODS)*. ACM, 2024.

[2] Cong, G., Yang, J., and Zhao, Y.: Machine Learning for Databases: Foundations, Paradigms, and Open problems. In: *Int. Conf. on Management of Data (SIGMOD/PODS)*. ACM, 2024.

[3] Kraska, T., Li, T., Madden, S., Markakis, M., Ngom, A., Wu, Z., and Yu, G. X.: Check Out the Big Brain on BRAD: Simplifying Cloud Data Processing with Learned Automated Data Meshes. In: *Proc. VLDB Endowment* 16.11 (2023).

[4] Njoku, U. F., Abelló, A., Bilalli, B., and Bontempi, G.: Finding Relevant Information in Big Datasets with ML. In: *Int. Conf. on Extending Database Technology (EDBT)*. 2024, pp. 846–849.

[5] Tarvo, A., Sweeney, P. F., Mitchell, N., Rajan, V. T., Arnold, M., and Baldini, I.: CanaryAdvisor: a statistical-based tool for canary testing (demo). In: *Int. Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2015.

[6] Wrembel, R.: Optimizing Data Integration Processes with the Support of Machine Learning - Is it really possible? In: *Int. Workshop on Design, Optimization, Languages and Analytical Processing of Big Data (DOLAP)*. Vol. 3653. CEUR Workshop Proceedings. 2024.