2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C) | 979-8-3503-2498-3/23/$31.00 ©2023 IEEE | DOI: 10.1109/MODELS-C59198.2023.00020

PyDaQu: Python Data Quality Code Generation Based on Data Architecture

1st Moamin Abughazala
DISIM Department
L'Aquila University
L'Aquila, Italy
moamin.abughazala@graduate.univaq.it

2nd Henry Muccini
DISIM Department
L'Aquila University
L'Aquila, Italy
henry.muccini@univaq.it

3rd Khitam Qadri
Quality Assurance Team Lead
KABi Technologies
Nablus, Palestine
khetam.q1990@gmail.com

***Abstract*—Accurate and dependable data is critical when making crucial business decisions. However, verifying the accuracy of complex and extensive datasets can be both error-prone and time-consuming when done manually. We developed a PyDaQu, our automated framework that creates data quality checks code based on a standardized template. PyDaQu offers a variety of quality assurance measures, including validation, completeness, and consistency checks. These measures ensure exceptional data quality while simultaneously streamlining your data management processes. With PyDaQu, creating data quality checks requires significantly less time and effort. We have thoroughly evaluated PyDaQu using data from two different industry domains.**

***Index Terms*—Data Quality Framework, Data-Driven, Data Architecture**

I. Introduction

The significance of ensuring high data quality cannot be overstated for data-driven applications that seek to thrive in today's data-driven business landscape [1] [2]. Reliable data is a critical asset that enables us to make well-informed decisions, boost operational efficiency, enhance customer satisfaction, comply with regulations, manage risks, save costs, facilitate data integration, and maintain trust and credibility. Accurate data is essential in streamlining processes, improving customer experiences, mitigating compliance risks, avoiding unnecessary expenses, and building stakeholder trust [3].

Great Expectations [4], Deequ [5], and Dbt [6] are powerful data quality enhancement tools. However, configuring and setting up the environment can be time-consuming and prone to human error. Furthermore, utilizing their predefined methods (expectations, constraints, ...) may be challenging, requiring additional code writing.

This paper introduces *PyDaQu*[1], a cutting-edge code generation framework that leverages the powerful DAT modeling framework, including data quality metrics. With PyDaQu, developers can easily convert their DAT models into Python code while incorporating the popular Great Expectation library. This results in a more streamlined workflow and the ability to efficiently produce high-quality code that meets their expectations.

Generating code for data quality checks automatically is a *time-saving* and highly efficient solution. It guarantees *consistency and standardization* across many datasets, thus enhancing accuracy and reliability. Promoting agility in adapting to changing data sources improves data governance. Moreover, it offers *documentation for traceability and compliance purposes*, streamlining the assessment process and enabling informed decisions based on reliable data.

The rest of this tool demo paper is organized as follows. The background is presented in Section 2. The related work is in Section 3. The Use Case of PyDaQu is described in Section 4. The PyDaQu tool, approach, and architecture are presented in Section 5. Evaluation is discussed in Section 6, while conclusions are drawn in Section 7.

[1]PyDaQu: **Py**thon **Da**ta **Qu**ality framework

II. Background

This section provides an overview of the DAT modeling framework and Great Expectation.

A. DAT Modeling Framework

DAT is a modeling tool for Data-Driven applications [7] that helps visualize the flow of data through the system and create a blueprint for it. Stakeholders can use it to describe high-level and low-level data architecture [8], including formats, processing types, storage, analysis types, and consumption methods. The tool is based on a structural and behavioral metamodel and supports the understanding and documentation of data-driven applications; it includes metrics for data quality. It uses a modeling language called Data Architecture Structural and Behavioral View (DAML), designed according to the IEEE/ISO/IEC standard [9].

B. Great Expectation

Great Expectations [4] is an open-source Python module that offers a comprehensive framework for evaluating and testing data quality in data pipelines and analytics workflows. Great Expectations offers automatic data validation and a customizable approach to defining expectations. It generates data documentation and profiling reports, promoting collaboration and ongoing data quality monitoring. Utilizing Great Expectations guarantees accurate data and enhances data-driven activities.

979-8-3503-2498-3/23/$31.00 ©2023 IEEE
DOI 10.1109/MODELS-C59198.2023.00020

III. Related Work

This section will examine studies pertinent to utilizing the most related research for the Data-Quality framework for data-driven applications.

PyDaQu automates generating data quality checks for any data source, mapping them to predefined expectations from Great Expectations, generating runnable Python code, and monitoring data quality. It can also generate Data Docs reports and take customizable actions based on evaluation outcomes to save time and effort while ensuring high-quality data.

BIGQA [10] simplifies data quality assessment for experts and management specialists across domains and contexts. It generates tailored reports and runs smoothly on parallel or distributed computing frameworks. It uses simple operators for large datasets and facilitates incremental assessment to save time. BIGQA creates plans for assessing data quality, while PyDaQu generates Python code for data quality that can be easily integrated into CI/CD processes. In [11] proposed A new method to measure dataset reliability introduces a "believability factor" by sampling a portion of the dataset and calculating execution time and Mean Absolute Error. PyDaQu can handle all data formats and creates Python code to verify the six primary data quality dimensions. In [12] proposed a Big Data Quality Management framework that works on the entire Big Data lifecycle. The framework relies on a Data Quality Profile enriched with valuable information as it progresses through various stages, such as Big Data project parameters, quality requirements, quality profiling, and quality rule proposals. PyDaQu uses DAT and can model the entire data life cycle. This allows PyDaQu to work on every stage and storage, regardless of format.

IV. Use Cases

This section provides a brief case description utilized by PyDaQu to test the generated code on actual industry data.

A. Case A: ISP - Internet Service Provider

As part of our research project, we carried out an investigation utilizing the raw data provided by an ISP Telecommunication company. The dataset featured a vast array of details about the internet usage patterns of their customers, all of which were stored in a complex relational database. Our experiment was conducted to check the quality of the data, which could be used to inform better decision-making strategies.

B. Case B: Errors Data Pipeline

This case outlines the data pipeline for error data from different printers. Files come in JSON format, including printer version, location, ink type, software version, and time. The data is saved on Amazon S3 and can be accessed using a query engine. After processing, the data is converted to parquet format, CSV, and relational database format for customer queries. We were able to run the quality check on raw data (same format of generation) and relational databases.

V. PyDaQu Framework

A. Approach

PyDaQu [2] [3] is a framework that automates the process of data quality checks and generates Python code to check for data quality issues such as missing values, outliers, and incorrect data types. Great Expectation is a tool that provides features to define and perform data quality checks, and DAT uses Great Expectations to generate the code for these checks. The DAT models are transformed into Great Expectations code, which can be integrated into existing data pipelines. By running this code, DAT can assess the correctness and usefulness of the data, providing insights into the quality of the data being analyzed. Overall, DAT and Great Expectations offer a powerful solution for automating data quality checks and improving the accuracy and reliability of data analysis.

B. Architecture

To fully comprehend the architecture of PyDaQu, it is crucial to examine each software component that comprises the system. Figure 1 offers an overview of the tool as an excellent reference point. In-depth explanations and demonstrations for each component will be provided to ensure a comprehensive understanding of PyDaQu.

DAT Model A new " Verify Data" data action has been added to ensure the model includes data quality metrics. This action can be attached to any data source. Figure 3 shows the part of the behavioral meta-model.

Parsing, The DAML model is typically stored in an XMI (XML Metadata Interchange) file, which contains various optional information that needs to be filtered out to extract a relevant subset of DAML model values conforming to the DAML meta-model.

To generate code from the DAML model, templates are defined for different DAML sub-models, such as data nodes and connectors, ports, and data. These templates define the code structure and logic based on the DAML meta-model.

The parser reads the DAML XMI file and extracts the necessary information based on the defined templates. This process involves creating objects representing each data element and a list of data quality dimensions linked to any data source. The resulting object of data architecture carries the DAML model description in Java, which can be used to generate code.

Template-based code generation is a common pattern in implementing code generators [13]. It involves defining templates that describe the code structure and logic based on the meta-model and using a parser to extract the necessary information from the input model to instantiate these templates. Developing a *coding template* involves creating reusable code snippets that can be saved as plain text files. The process includes identifying common patterns, crafting concise templates with placeholders, categorizing and organizing them, thoroughly testing, and keeping them up-to-date as necessary [14].

[2]PyDaQu Tool Source Code : https://github.com/khitam90/PyDaQu
[3]PyDaQu Tool Demo Video : https://youtu.be/697F6h0q7ss

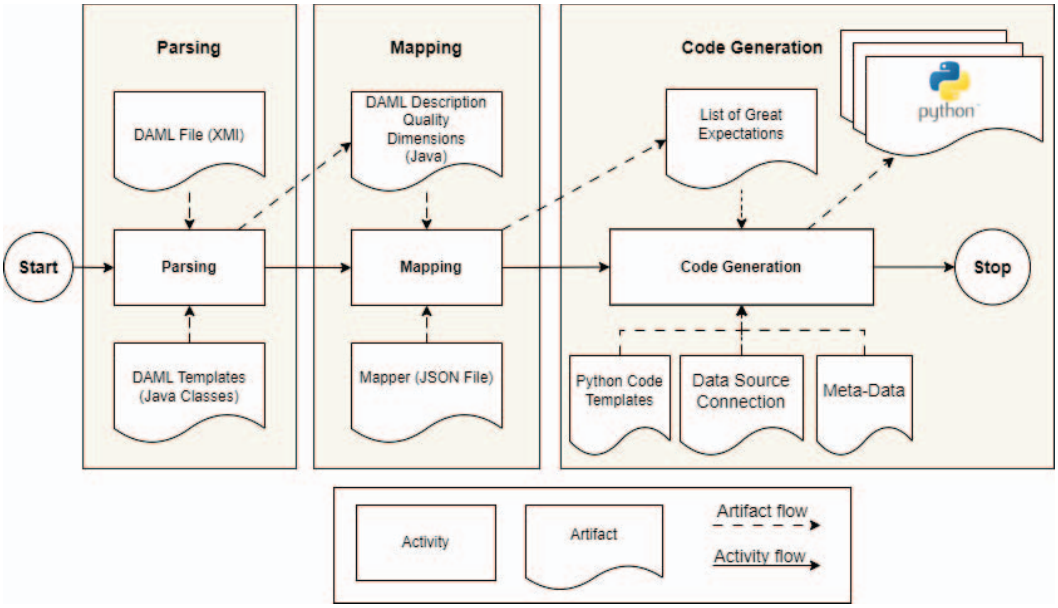

Fig. 1. PyDaQu automatic code generation framework

```
import great_expectations as ge

from Entities.batches import Batch
from Entities.checkpoints import CheckPointConfig
from Entities.datasources import DataSource
from Entities.expectationUtiles import ExpectationUtil
from Entities.expectationsuites import ExpectationSuite
from Entities.databases import DataBaseType


# Get Context
context = ge.get_context()

if __name__ == '__main__':

    # Data Source Code
    dataSource=DataSource({UserName},{Password},{IPAddress},{Port},{DatabaseName},{DatabaseType})
    context.add_datasource(**dataSource.{DatabaseConfigurationMethod}())

    condition =""
    list_of_expectation_configuration=[]

    {TableCode}
#-----------------------------------
    # List of expectations for each column
    {ExpectationCode}

    # add list of expectation configuration to test suite
    suite = ExpectationSuite.create_expectation_suite(tableName,context ,list_of_expectation_configuration)
    context.save_expectation_suite(suite)
    list_of_expectation_configuration.clear()

    # getting batch requests for the tables
    batch_request = Batch.get_batch(dataSource.dataSchema, tableName, condition)

    # running tests on tables
    print(tableName, " ", batch_request)
    CheckPointConfig.run_check_point_config(context,batch_request, tableName)
```

```
    expectation_configuration = ExpectationUtil.{expectationMethod}({ExpectationParameter})
    list_of_expectation_configuration.append(expectation_configuration)
```

Fig. 2. Templates Example

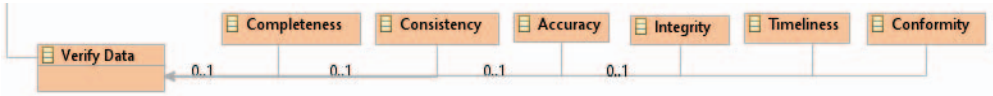

Fig. 3. "Verify Data" Action

Accordingly, the resulting data architecture object contains the data quality dimensions linked to each data source. **Mapping**, This activity inputs the data architecture object and the DAML model description in Java, resulting from the previous

```python
import great_expectations as ge
from Entities.batches import Batch
from Entities.checkpoints import CheckPointConfig
from Entities.datasources import DataSource
from Entities.expectationUtiles import ExpectationUtil
from Entities.expectationsuites import ExpectationSuite
# Get Context
context = ge.get_context()
if __name__ == '__main__':

    # Data Source Code
    dataSource=DataSource("root","root","127.0.0.1","3306","testdb",DataBaseType.MySQL)
    context.add_datasource(**dataSource.getMySQLDataCourceConfig())

    condition =""
    list_of_expectation_configuration=[]
    # Table Name
    tableName='userinfo'
#---------------------------------
    # List of expectations for each column
    #Expectations For Column [username]
    expectation_configuration = ExpectationUtil.expect_column_values_to_be_unique("username")
    list_of_expectation_configuration.append(expectation_configuration)

    expectation_configuration = ExpectationUtil.expect_column_values_to_not_be_null("username")
    list_of_expectation_configuration.append(expectation_configuration)

    # add list of expectation configuration to test suite
    suite = ExpectationSuite.create_expectation_suite(tableName,context ,list_of_expectation_configuration)
    context.save_expectation_suite(suite)
    list_of_expectation_configuration.clear()

    # getting batch requests for the tables
    batch_request = Batch.get_batch(dataSource.dataSchema, tableName, condition)

    # running tests on tables
    print(tableName, " ", batch_request)
    CheckPointConfig.run_check_point_config(context,batch_request, tableName)
```

Fig. 4. Example Of Generated Code

TABLE I
PART OF MAPPER

Quality Dimensions	Great Expectations
Uniqueness	expect_column_values_to_be_unique
Completeness	expect_column_values_to_not_be_null
	expect_column_values_to_be_null
Validity	expect_column_values_to_be_of_type
	expect_column_values_to_be_in_type_list
	expect_column_values_to_be_increasing
Consistency	expect_column_value_lengths_to_be_between
	expect_column_value_lengths_to_equal
	expect_column_values_to_match_regex
	expect_column_values_to_match_regex_list
Timeliness	expect_column_min_to_be_between
	expect_column_max_to_be_between
Accuracy	expect_column_values_to_be_in_set
	expect_column_values_to_not_be_in_set

parsing activity. This Mapper represents the core section of the code generation process and is responsible for mapping quality dimensions to their corresponding expectations for each data source. Table I shows the list of quality dimensions and corresponding expectations provided by Great Expectations.

Finally, the result of Mapper is a list of data sources defined with their list of quality dimensions and mapped predefined expectations (from Great Expectations). These objects will be an entry to the following code generation activity.

Generate Python Data Quality Checkpoints Code, This activity depends on four artifacts:

1) Previous activity's output, a list of data sources with related expectations.
2) Python code templates.
3) The connection details for the data source.
4) The meta-data of the data source.

During the code generation, we use the list of Python code templates. Every part of the code within braces "{}" will be replaced with values or generated code see Figure 2. We use the connection details and the meta-data to select which part of the data we need to run the quality checks (the list of expectations).

Figure 4, shows an example of generated code that connects to MySQL database, using table "userinfo" to get the batch of data to run the test; lines 22-26 show two of expectations ("expect_column_values_to_be_unique", "expect_column_values_to_not_be_null") will be applied to "username" column.

The results of code generation activity are runnable Python files that use the Great Expectation library to validate, document, and profile the data to maintain quality and improve team communication. Figure 5 shows the results of running the generated Python files.

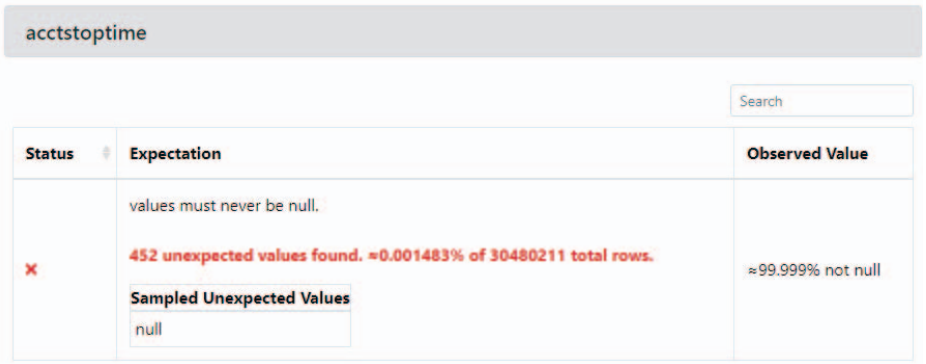

Fig. 5. Results of running the generated code

TABLE II
COMPARING THE TIME REQUIRED FOR MANUAL AND AUTOMATED CODE GENERATION.

Cases	Engineer Level	Manual	PyDaQu
Case A	Fresh	5 days	**10 minutes**
	Senior	3 days	
Case B	Senior	3 days	

VI. EVALUATION

Our recent publication [7] encompasses a thorough assessment of DAT in diverse domains. We have included one case study from our previous research and another from the telecommunications industry to enhance our argument. The cases were selected based on the data available to execute the code produced by PyDaQu to check the effectiveness of it.

For both cases, we engaged in the process of manually constructing Python code and integrating Great Expectations. After conducting a thorough evaluation, we found a significant difference in the time required to complete a data quality check when comparing manual Python coding versus utilizing PyDaQu to create and execute the code.

The table II shows the estimated time required to manually generate the code based on the engineer's seniority level. Seniors are expected to complete the task within three days, while juniors may require up to five days. On the other hand, it takes only ten minutes to generate and run the code using PyDaQu.

Using code generation templates helps developers create high-quality, error-free code in a structured and efficient manner, leading to faster development cycles and reliable software.

VII. CONCLUSION

In this tool demonstration paper, we have presented PyDaQu, an excellent Python code generator that employs the Great Expectation Library to assess data quality. PyDaQu proves to be an indispensable tool for engineers, as it helps them save time while developing data-checking code, regardless of their level of expertise. The generated code seamlessly integrates with any data pipeline, which makes it a valuable asset for teams working with data. Extensive testing on real-world data across various domains has demonstrated PyDaQu's efficacy in delivering reliable and accurate evaluations. PyDaQu sets the benchmark for quality data assessment and is a must-have tool for anyone working with data. Engineers can confidently create high-quality, error-free code efficiently and systematically by utilizing code generation templates. This approach results in faster development, better code, and more reliable software.

REFERENCES

[1] A. Karkouch, H. Mousannif, H. Al Moatassime, and T. Noel, "Data quality in internet of things: A state-of-the-art survey," *Journal of Network and Computer Applications*, vol. 73, pp. 57–81, 2016.

[2] S. Ji, Q. Li, W. Cao, P. Zhang, and H. Muccini, "Quality assurance technologies of big data applications: A systematic literature review," *Applied Sciences*, vol. 10, no. 22, p. 8052, 2020.

[3] L. Cai and Y. Zhu, "The challenges of data quality and data quality assessment in the big data era," *Data science journal*, vol. 14, 2015.

[4] G. Expectations. (2021) Great expectations. [Online]. Available: https://greatexpectations.io/

[5] A. Labs. (2018) Deequ - unit tests for data. [Online]. Available: https://github.com/awslabs/deequ

[6] DBT. (2022) Data build tool. [Online]. Available: https://www.getdbt.com

[7] M. Abughazala, H. Muccini, and M. Sharaf, "Dat: Data architecture modeling tool for data-driven applications," in *16th European Conference on Software Architecture (ECSA)*. Springer, 2022, p. http://dx.doi.org/10.13140/RG.2.2.23556.81286.

[8] M. Abughazala. (2022) Dat - data architecture modeling tool for iot. [Online]. Available: https://github.com/moamina/DAT

[9] ISO/IEC/IEEE, "ISO/IEC/IEEE 42010:2011 Systems and software engineering – Architecture description," 2011.

[10] H. Fadlallah, R. Kilany, H. Dhayne, R. e. Haddad, R. Haque, Y. Taher, and A. Jaber, "Bigqa: Declarative big data quality assessment," *ACM Journal of Data and Information Quality*, 2023.

[11] S. Soni and A. Singh, "Improving data quality using big data framework: A proposed approach," in *IOP Conference Series: Materials Science and Engineering*, vol. 1022, no. 1. IOP Publishing, 2021, p. 012092.

[12] I. Taleb, M. A. Serhani, C. Bouhaddioui, and R. Dssouli, "Big data quality framework: a holistic approach to continuous quality management," *Journal of Big Data*, vol. 8, no. 1, pp. 1–41, 2021.

[13] M. Voelter, "A catalog of patterns for program generation." in *EuroPLoP*, 2003, pp. 285–320.

[14] E. Syriani, L. Luhunu, and H. Sahraoui, "Systematic mapping study of template-based code generation," *Computer Languages, Systems Structures*, vol. 52, pp. 43–62, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1477842417301239