datapilot-cli

Release 0.0.10

Altimate Inc.

Apr 21, 2024

CONTENTS

CHAPTER

ONE

OVERVIEW

docs	
tests	
	|scrutinizer|
package	
	0.0.10 **|wheel| |supported-versions| |supported-implementations| |commits-since|**

Assistant for Data Teams

- Free software: MIT license

1.1 Installation

```
pip install altimate-datapilot-cli
```

You can also install the in-development version with:

```
pip install https://github.com/AltimateAI/datapilot-cli/archive/main.zip
```

1.2 Documentation

https://datapilot.readthedocs.io/

1.3 Development

To run all the tests run:

```
tox
```

Note, to combine the coverage data from all the tox environments run:

Windows	`set PYTEST_ADDOPTS=--cov-append` `tox`
Other	`PYTEST_ADDOPTS=--cov-append tox`

CHAPTER

TWO

INTRODUCTION TO DATAPILOT

2.1 What is DataPilot?

DataPilot is an innovative tool designed to be an AI-powered assistant for data engineers and analysts working with SQL and dbt (data build tool). It integrates seamlessly into the development environment, providing real-time insights and suggestions to uphold best practices and enhance the quality of data projects.

With DataPilot, teams can automate the review process for their SQL queries and dbt models, ensuring that their data transformations are efficient, well-documented, and maintainable. It also facilitates organization-wide consistency by enforcing project standards through integration with version control systems and continuous integration/continuous deployment (CI/CD) pipelines.

2.2 Key Features

DataPilot comes with a host of features aimed at improving data project management:

- **Insightful Analysis:** DataPilot performs in-depth analysis of SQL code and dbt projects, highlighting areas of concern such as model fanouts, hard-coded references, and potential duplications.
- **Seamless Integration:** It can be easily integrated into local development environments as well as Git workflows and CI/CD pipelines, making it a versatile tool for teams of all sizes.
- **Early Detection:** By identifying potential issues early in the development cycle, DataPilot helps prevent costly and time-consuming fixes down the line.
- **Best Practice Enforcement:** DataPilot encourages the adoption of best practices in SQL and dbt project development, aiding in the maintenance of high-quality data models.
- **Automated Checks:** The tool includes a range of automated checks for detecting unused sources, ensuring dependency integrity, and encouraging comprehensive testing and documentation.

2.3 How DataPilot Works

DataPilot operates by scanning your SQL and dbt project files, identifying patterns and structures that indicate potential problems or deviations from best practices. Once an issue is detected, it provides feedback and recommendations on how to address it.

For dbt projects, DataPilot makes use of the manifest and catalog files generated by dbt to perform its analysis. This ensures that the insights provided are based on the most up-to-date view of your project's state.

CHAPTER

THREE

INSTALLATION

3.1 Prerequisites

Before installing DataPilot, ensure you have the following prerequisites met:

- Python 3.7 or higher installed on your machine.
- Access to a command-line interface (CLI) to execute pip commands.
- An existing dbt project to analyze with DataPilot.

3.2 Installation

To install DataPilot, open your CLI and run the following command:

```
pip install altimate-datapilot-cli
```

This command will download and install the latest version of DataPilot along with its dependencies.

3.3 QuickStart

Once DataPilot is installed, you can set it up to work with your dbt project.

Execute the following command to perform a health check on your dbt project:

```
datapilot dbt project-health --manifest-path /path/to/manifest.json --catalog-path /path/
↪to/catalog.json
```

After running the command, DataPilot will provide you with insights into your dbt project's health. Review the insights and make any necessary adjustments to your project.

CHAPTER

FOUR

DBT

4.1 project-health

The `project-health` feature in DataPilot is a comprehensive tool designed to analyze and report on various aspects of your dbt project. This feature is currently supported for dbt version 1.6 and 1.7.

4.1.1 How to Use

To use the `project-health` feature, run the following command in your dbt project directory:

Step 1: Generate a manifest file for your dbt project.

```
dbt compile
```

This command will generate a manifest file for your dbt project under the configured `target` directory. The default location for this directory is `target/manifest.json`.

Step 2: Generate a catalog file for your dbt project.

```
dbt docs generate
```

This command will generate a catalog file for your dbt project under the configured `target` directory. The default location for this directory is `target/catalog.json`.

Step 3: Run the `project-health` command.

```
datapilot dbt project-health --manifest-path ./target/manifest.json --catalog-path ./
↪target/catalog.json
```

The catalog path is optional. If you do not provide a catalog path, the command will still run, but the catalog-related insights will not be available.

You can also select specific list of models to run the health check on by providing the '–select' flag. For example:

```
datapilot dbt project-health --manifest-path ./target/manifest.json --select "path:dir1␣
↪path:dir2 model1 model2"
```

This will run the health check on all the models in the 'dir1' and 'dir2' directory. It will also run the health check on the 'model1' and 'model2' models. As of now, the '–select' flag only supports filtering based on model path and model name. We will add support for other filters and make it compatible with the dbt comands soon.

CHAPTER

FIVE

INSIGHTS

The following insights are available in DataPilot:

5.1 1. Modelling Insights

Name	Description	Files Required	Overrides
source_staging_model_inte	Ensures each source has a dedicated staging model and is not directly joined to downstream models.	Manifest	None
down-stream_source_dependence	Evaluates if downstream models (marts or intermediates) are improperly dependent directly on a source. This check ensures that all downstream models depend on staging models, not directly on the source nodes.	Manifest	None
Duplicate_Sources	Identifies cases where multiple source nodes in a dbt project refer to the same database object. Ensures that each database object is represented by a single, unique source node.	Manifest	None
hard_coded_references	Identifies instances where SQL code within models contains hard-coded references, which can obscure data lineage and complicate project maintenance.	Manifest	None
rejoin-ing_upstream_concepts	Detects scenarios where a parent's direct child is also a direct child of another one of the parent's direct children, indicating potential loops or	Manifest	None

5.2 2. Performance Insights

Name	Description	Files Required	Overrides
chain_view_linking	Analyzes the dbt project to identify long chains of non materialized models (views and ephemerals). Such long chains can result in increased runtime for models built on top of them due to extended computation and memory usage.	Manifest	None
exposure_parent_bad_materializ	Evaluates the materialization types of parent models of exposures to ensure they rely on transformed dbt models or metrics rather than raw sources, and checks if these parent models are materialized efficiently for performance	Manifest	None

5.3 3. Governance Insights

Name	Description	Files Required	Overrides
documentation_on_stale_columns	Checks for columns that are documented in the dbt project but have been removed from their respective models.	Manifest, Catalog	None
exposures_dependent_on_private	Detects if exposures in the dbt project are dependent on private models. Recommends using public, well documented, and contracted models as trusted data sources for downstream consumption.	Manifest	None
public_models_without_contra	Identifies public models in the dbt project that are accessible to all downstream consumers but lack contracts specifying data types and columns.	Manifest	None
missing_documentation	Detects columns and models that don't have documentation.	Manifest, Catalog	None
undocumented_public_models	Identifies models in the dbt project that are marked as public but don't have documentation.	Manifest	None

5.4 4. Testing Insights

Name	Description	Files Required	Overrides
missing_primary_key_tests	Identifies dbt models in the project that lack primary key tests, which are crucial for ensuring data integrity and correctness.	Manifest	None
dbt_low_test_coverage	Identifies dbt models in the project that have tests coverage percentage below the required threshold.	Manifest	min_test_coverage_percent

5.5 5. Project Structure Insights

Name	Description	Files Required	Overrides
model_directory_structure	Checks for correct placement of models in their designated directories. Proper directory structure is essential for , organization, discoverability, and maintenance within the dbt project.	Manifest	None
model_naming_convention_	Ensures all models adhere to a predefined naming convention. A consistent naming convention is crucial for clarity, understanding of the model's purpose, and enhancing navigation within the dbt project.	Manifest	None
source_directory_structure	Verifies if sources are correctly placed in their designated directories. Proper directory placement for sources is important for organizationand easy searchability.	Manifest	None
test_directory_structure	Checks if tests are correctly placed in the same directories as their corresponding models. Co locating tests with models aids in maintainability and clarity.	Manifest	None

5.6 6. Check Insights

Name	Description	Files Required	Overrides
column_descriptions_are_same	Checks if the column descriptions in the dbt project are consistent across the project.	Manifest	None
column_name_contract	Checks if the column names in the dbt project abide by the column name contract which consists of a regex pattern and a series of data types.	Manifest, Catalog	None
check_macro_args_have_de	Checks if the macro arguments in the dbt project have descriptions.	Manifest	None
check_macro_has_desc	Checks if the macros in the dbt project have descriptions.	Manifest	None
check_model_has_all_colur	Checks if the models in the dbt project have all the columns that are present in the data catalog.	Manifest, Catalog	None
check_model_has_valid_me	Checks if the models in the dbt project have meta keys.	Manifest	None
check_model_has_propertie	Checks if the models in the dbt project have a properties file.	Manifest	None
check_model_has_tests_by_	Checks if the models in the dbt project have tests by name.	Manifest	None
check_model_has_tests_by_	Checks if the models in	Manifest	None

CHAPTER

SIX

PERFORMANCE OF PRE-COMMIT HOOK

6.1 Overview

The primary objective is to ensure the pre-commit hook operates swiftly and efficiently, preventing any delay in the development workflow. To achieve this, various optimizations have been applied, focusing on minimizing the time and resources required during execution.

6.2 Optimizations

1. **Partial Catalog Fetching**: Instead of retrieving the entire catalog schema, the pre-commit hook is optimized to fetch only the schema of the files being committed. This approach significantly reduces the fetching time and the amount of data processed.
2. **Cost-effective Commands**: The hook utilizes commands that avoid activating the warehouses in Snowflake, enhancing cost effectiveness. Specifically, it avoids the use of *dbt docs generate*, which retrieves columns from the information schema and requires warehouse activation, thereby incurring higher costs.

6.3 Timing Results for the_tuva_project

The following timing results illustrate the efficiency of the pre-commit hook across different scenarios, with varying numbers of files changed in the commit:

- **1 file changed**: - DataPilot: 15 seconds - Checkpoint: 60 seconds
- **5 files changed**: - DataPilot: 16 seconds - Checkpoint: 54 seconds
- **10 files changed**: - DataPilot: 19 seconds - Checkpoint: 54 seconds
- **15 files changed**: - DataPilot: 24 seconds - Checkpoint: 45 seconds
- **20 files changed**: - DataPilot: 19 seconds - Checkpoint: 56 seconds
- **Check on all files (309 files)**: - DataPilot: 42 seconds - Checkpoint: 71 seconds

6.4 Conclusion

The optimized pre-commit hook demonstrates a consistent performance improvement, effectively balancing the speed of development against the necessity of maintaining code quality and cost efficiency.

CHAPTER

SEVEN

ADVANCED USAGE

7.1 Project Health Configuration

You can configure the project health settings by providing a configuration file. The configuration file is a YAML file that contains the following fields:

```yaml
version: v1

# Insights to disable
disabled_insights:
  - source_staging_model_integrity
  - downstream_source_dependence
  - Duplicate_Sources
  - hard_coded_references
  - rejoining_upstream_concepts
  - model_fanout
  - multiple_sources_joined

# Define patterns to identify different types of models
model_type_patterns:
  staging: "^stg_.*"       # Regex for staging models
  mart: "^(mrt_|mart_|fct_|dim_).*"  # Regex for mart models
  intermediate: "^int_.*"  # Regex for intermediate models
  base: "^base_.*"         # Regex for base models

# Configure insights
insights:
  # Set minimum test coverage percent and severity for 'Low Test Coverage in DBT Models'
  dbt_low_test_coverage:
    min_test_coverage_percent: 30
    severity: WARNING

  # Configure maximum fanout for 'Model Fanout Analysis'
  model_fanout.max_fanout: 10

  # Configure maximum fanout for 'Source Fanout Analysis'
  source_fanout.max_fanout: 10

  # Define model types considered as downstream for 'Staging Models Dependency Check'
  staging_models_dependency.downstream_model_types:
    - mart
```

7.2 Key Sections of the config file

- disabled_insights: Insights that you want to disable
- model_type_patterns: Regex patterns to identify different model types like staging, mart, etc.
- insights: Custom configurations for each insight. For each insight, you can set specific thresholds, severity levels, or other parameters.

Severity can have 3 values -> INFO, WARNING, ERROR

7.3 Overriding default configs for the insights

To change the severity level or set a threshold, modify the corresponding insight under the insights section. For example:

```
insights:
  dbt_low_test_coverage:
    severity: WARNING
```

For insights with more complex configurations (like fanout thresholds or model types), you need to specify the insight name and corresponding parameter under insights. For example:

```
insights:
  model_fanout.max_fanout: 10
```

CHAPTER

EIGHT

CONTRIBUTING

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

8.1 Bug reports

When reporting a bug please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

8.2 Documentation improvements

datapilot could always use more documentation, whether as part of the official datapilot docs, in docstrings, or even on the web in blog posts, articles, and such.

8.3 Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/AltimateAI/datapilot/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that code contributions are welcome :)

8.4 Development

To set up *datapilot* for local development:

1. Fork datapilot (look for the "Fork" button).
2. Clone your fork locally:

```
git clone git@github.com:YOURGITHUBNAME/datapilot.git
```

3. Create a branch for local development:

```
git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

4. When you're done making changes run all the checks and docs builder with one command:

```
tox
```

5. Commit your changes and push your branch to GitHub:

```
git add .
git commit -m "Your detailed description of your changes."
git push origin name-of-your-bugfix-or-feature
```

6. Submit a pull request through the GitHub website.

8.4.1 Pull Request Guidelines

If you need some code review or feedback while you're developing the code just make the pull request.

For merging, you should:

1. Include passing tests (run `tox`).
2. Update documentation when there's new API, functionality etc.
3. Add a note to `CHANGELOG.rst` about the changes.
4. Add yourself to `AUTHORS.rst`.

8.4.2 Tips

To run a subset of tests:

```
tox -e envname -- pytest -k test_myfeature
```

To run all the test environments in *parallel*:

```
tox -p auto
```

CHAPTER

NINE

AUTHORS

- Anand Gupta - www.altimate.ai

CHAPTER

TEN

CHANGELOG

10.1 0.0.0 (2024-01-25)

- First release on PyPI.