ChatGPT

BI Documentation Tool – AI Assistant Prompt & Development Guidelines

Objective and Scope

Develop a **Business Intelligence (BI) documentation tool** that can parse **Power BI** and **Tableau** files to extract key metadata and produce structured documentation. The tool should handle **Power BI** `.pbix` **files** and **Tableau** `.twb/.twbx` **workbooks**, extracting elements like datasets, fields, measures, data sources, and the visual layout structure. Outputs should be generated in **Markdown** (for human-readable docs) and **JSON** (for integrations). The solution will be delivered as a **command-line interface (CLI)** application and a **Docker container**, following best practices for modular design, logging, and future extensibility.

Prompt for the AI Assistant

Implement a CLI tool (e.g., in Python) with the following core functionality and requirements:

1. **File Parsing**: Support importing **Power BI** `.pbix` files and **Tableau** `.twb/.twbx` files. The tool should detect the file type and invoke the appropriate parser module for each. Ensure that packaged Tableau files (`.twbx`) are handled by unpacking the zipped contents to access the `.twb` XML [1].

2. **Metadata Extraction**: For each file, extract key metadata elements:

3. **Datasets & Tables**: List all data tables or data sources present. For Power BI, include all tables in the data model (whether imported or via DirectQuery), and for Tableau, include all data sources and their tables.
4. **Fields & Calculations**: Catalog all fields/columns in each table, including data types and any calculated fields or measures. For Power BI, capture DAX measures and calculated columns (with their DAX formulas) [2] [3]. For Tableau, capture calculated fields (with formulas) and dimensions/measures, as the Tableau XML or API provides [4].
5. **Measures**: Record all measures or calculations – DAX measures in Power BI (including their formula expressions) and calculated fields in Tableau (including formulas and aggregation).
6. **Data Sources & Connections**: Identify data source connections (e.g., SQL databases, Excel files, etc.), including connection details like server, database, etc. For Tableau, use the workbook's datasource and connection information (e.g., via Tableau's Document API, `Workbook.datasources` and `Datasource.connections` properties [5] [6]). For Power BI, list the data source types and connection strings if available (Power Query connections, etc.).

7. **Visual Layout Structure**: Reconstruct a tree or list of the report's visual elements. For Power BI, enumerate report pages and within each page, list visuals (charts, tables, etc.) along with the fields or measures they use. *Hint:* In a `.pbix`, the **Report/Layout** file (accessible by renaming `.pbix` to `.zip`) contains JSON that describes pages, visuals, and the fields they bind to [7]. Parse this to

extract *Report Name* → *Visual Name* → *Fields used* . For Tableau, enumerate worksheets and dashboards: list each worksheet's name, what datasource it uses, and which fields are present in the view. Tableau's XML (or API) can provide the list of fields used in each worksheet [8] .

8. **Output Formats**: Generate documentation in two formats:

9. **Markdown**: A human-readable Markdown report documenting all the above elements (e.g., sections for "Data Sources", "Tables and Fields", "Measures", "Visuals by Page/Dashboard"). Use proper Markdown formatting (headings, lists, tables) for clarity.

10. **JSON**: A machine-readable JSON structure containing the extracted metadata (e.g., a nested JSON where keys might include `tables`, `measures`, `visuals`, etc., with arrays of metadata objects). Ensure the JSON follows a clear schema so it can be consumed by other tools.

11. **CLI Interface**: Design the tool as a command-line app (e.g., `bidoc-cli`) that accepts input file(s) and output directory/path options. The CLI should allow flags or subcommands such as: input file path(s), output format selection (or both), and verbosity level for logging. Example usage:

```
bidoc-cli --input sample.pbix --output docs/ --format all
```

This should parse `sample.pbix` and produce both `sample.md` and `sample.json` in the `docs/` folder. Support batch processing if multiple input files are provided.

12. **Logging**: Implement logging with an appropriate level of detail. At minimum, log progress steps (e.g., "Parsing Power BI model...", "Extracted 5 tables, 12 measures", "Generating Markdown output..."). Use a standard logging library (such as Python's `logging` module) and allow a verbose flag (`-v/--verbose`) for debug-level logs. Ensure that errors are reported clearly (e.g., if a file can't be parsed, catch exceptions and log an error message without crashing the entire run).

13. **Packaging and Docker**: Organize the project for ease of deployment:

14. Provide a `setup.py` or similar if distributing via pip, and make the CLI entry point accessible (using `console_scripts` or a small wrapper).
15. Write a Dockerfile to containerize the application. The Docker image should be based on a lightweight runtime (e.g., Python slim or Alpine for Python) and include all dependencies. The container entrypoint should be the CLI, so users can run the documentation tool by invoking the container (e.g., `docker run ... bidoc-cli --input ...`). Ensure the container can accept mounted volumes for input/output (to integrate with Git-based workflows, CI pipelines, etc.).

16. The CLI should be stateless (no GUI), making it suitable for CI/CD. Also, design output paths so that documentation can be version-controlled (e.g., outputs to a `docs/` directory that can be committed to Git).

17. **Code Structure & Quality**: Emphasize a clean, modular architecture:

18. Separate the parsing logic for Power BI and Tableau into distinct modules or classes (e.g., `PowerBIParser` and `TableauParser`). Each should handle the specifics of reading the file and extracting the metadata.
19. Separate the output formatting logic into its own module (or modules for each format). For instance, a `DocumentationFormatter` class with methods `to_markdown(data)` and `to_json(data)` that take the extracted metadata and produce the formatted output. This enforces a clear separation between data extraction and presentation.
20. Include placeholder interfaces or stub functions for **future enhancements**. For example, prepare for **AI-driven summarization** by defining a function `generate_summary(metadata)` that could call an AI service to create narrative insights from the metadata. For now, this function can just return a static note or be left unimplemented (with a `TODO` comment), ensuring the core design can accommodate such features later without major changes.
21. Follow best practices for a production-grade CLI tool: clear function names, error handling (with user-friendly messages), and comments/docstrings for maintainability.

Make sure to handle edge cases (unsupported file types, empty or corrupt files, etc.) gracefully, and include basic unit tests for the parsing functions if possible. The output should be accurate and comprehensive in capturing the BI assets in the workbook.

Development Guidelines

Architecture and Modular Design

To achieve **modularity and extensibility**, structure the project with clear separation of concerns:

- **Core Parsers**: Implement dedicated parser components for each platform: one for Power BI (`pbix_parser`) and one for Tableau (`tableau_parser`). Each parser encapsulates the logic to open the file, extract metadata, and return a structured representation (e.g., a Python dict or custom data classes with sections for tables, measures, visuals, etc.). This separation allows updates or additions (e.g., supporting a new BI tool in the future) without affecting other parts of the codebase.

- **Output Generators**: Create output modules, e.g., `markdown_generator` and `json_generator`, which know how to take the structured metadata from parsers and format it into Markdown and JSON respectively. This decoupling means the parsers do not need to know about output format details. It also allows easy addition of new output formats (HTML, PDF, etc.) later.

- **CLI Interface**: A main CLI module (e.g., `cli.py` or an entry-point script) should handle argument parsing (using Python's `argparse` or a library like `click`) and orchestrate the workflow: determine file type, invoke the correct parser, then call the output generator(s), all while handling user options (like output path, format selection, etc.). This main module can also initialize logging based on verbosity flags.

- **Extensibility Hooks**: Define interfaces or placeholders for advanced features:

- **AI Summaries**: For example, an `ai_summary.py` module with a function `summarize_metadata(metadata)` that returns a text summary. In the MVP, this could simply return an empty string or a note that this feature is not implemented, but having the hook in place

(perhaps called by the CLI if a `--with-summary` flag is given) means integrating actual AI in the future will not require redesigning the tool.
- **Integrations**: If future integration with other systems is expected, design the JSON output schema thoughtfully, and consider adding stub functions or classes that could, for instance, push the JSON to an API or database. Keep these as no-ops or simple print statements in the MVP, with clear `TODO` comments.

This design aligns with clean **separation of parsing vs formatting**. It ensures that upgrading one part (e.g., switching to a different Markdown library or updating for a new version of Power BI) can be done in isolation [9].

Folder Structure and Boilerplate

A recommended project structure (assuming a Python implementation) is as follows:

```
bi-doc-tool/                # Root directory
├── README.md
├── setup.py                # Packaging config (if publishing as a package)
├── requirements.txt        # Dependencies (pbixray, tableau SDK, etc.)
├── Dockerfile              # For containerizing the app
├── bidoc                   # Application source code package
│   ├── __init__.py
│   ├── __main__.py         # Allows "python -m bidoc" to run CLI
│   ├── cli.py              # CLI argument parsing and main entry logic
│   ├── pbix_parser.py      # Power BI parsing logic
│   ├── tableau_parser.py   # Tableau parsing logic
│   ├── markdown_generator.py   # Markdown output formatting
│   ├── json_generator.py       # JSON output formatting
│   ├── ai_summary.py       # (Stub for future AI summary features)
│   └── utils.py            # Utility functions (e.g., logging setup, common
helpers)
└── tests/                  # Unit tests for parsers and output (if applicable)
```

In this structure: - The `bidoc` package contains all logic. The `__main__.py` or CLI script will parse arguments and dispatch to `pbix_parser` or `tableau_parser` based on file extension. - The output modules produce the final strings/JSON. For Markdown, consider creating nicely formatted sections (perhaps using Markdown templates or f-strings). For JSON, ensure it's well-structured; you might define a JSON schema or at least document the format in `README.md`. - Logging configuration can be done in `cli.py` (e.g., configuring the root logger format and level).

This layout makes it clear where each piece of functionality resides, aiding maintainability. It is also compatible with packaging the tool (so that `pip install bi-doc-tool` could be possible in the future) and with containerization.

Parsing Power BI (`.pbix`) Files

Power BI `.pbix` files are not officially documented by Microsoft for external parsing (there is no public SDK for reading `.pbix` content structure [10]). However, a `.pbix` is essentially a zipped archive containing various components of the Power BI report. Key parts include: - The **data model** (which might be stored in a `DataModel` file, potentially as an Analysis Services database serialization). - The **report layout** (`Report/Layout`), which is a JSON (possibly encoded in UTF-16) describing the report pages, visuals, and their properties [7]. - Other resources (like cached datasets, etc., which might not be needed for just documentation of schema).

For extracting metadata from Power BI, you have a few approaches: 1. **Use an open-source parser library**: Incorporating an existing library can save time: - *PBIXRay*: An open-source Python library specifically designed to parse `.pbix` files and extract info [11]. PBIXRay can retrieve tables, columns, relationships, DAX measures, and even Power Query (M) code from a `.pbix` [12] [2]. Using PBIXRay, you can get a structured list of all tables (`model.tables`), then for each table its columns (`model.schema` gives table/column data types [13]) and DAX measures (`model.dax_measures` gives measure formulas [2]). It even can list relationships between tables [14]. Leveraging this library will cover much of the "dataset, tables, fields, measures" extraction for Power BI. - *pbi-tools*: A CLI tool by Mathias Thierbach that can **decompile a `.pbix`** into its internal components for source control [15]. It's open source (MIT license) and provides a way to export the Power BI file into a folder (sometimes called a "PbixProj") which includes a `database.json` (representing the data model) and `Report` layout files. While using pbi-tools directly might be outside the scope of having our own parser, understanding its output can guide what to extract. For example, pbi-tools' `database.json` is essentially the Tabular model (with tables, columns, measures) [15]. You could use pbi-tools in the background (via a subprocess call) to decompile and read the JSON, but that adds complexity and a dependency on an external binary. Alternatively, since PBIXRay covers similar ground in pure Python, it might be preferable for integration. - *Manual approach*: If not using an external library, you can manually unzip the `.pbix` (Python's `zipfile` module) and parse the `Report/Layout` JSON. As noted on Stack Overflow, you must handle encoding and extraneous control characters in this JSON [16]. A known trick is to decode it as UTF-16 and remove certain control characters (0x00, 0x1C, 0x1D, 0x19) before loading as JSON [16]. Once loaded, you can navigate the JSON structure to find visual containers, their names, and bound field names. This is more complex, but certainly doable if fine-grained control is needed. For the data model (tables, columns, measures), without a library like PBIXRay, it's quite difficult because the data model may be stored in a proprietary format. However, if the `.pbix` was saved as a `.pbit` (template), it contains no data and might have a `DataModelSchema` or similar to parse. Given the time, using PBIXRay is a strong choice for MVP.

Recommendation: Use **PBIXRay** to extract the Power BI model metadata programmatically [11]. After installing it (`pip install pbixray`), the tool can do something like:

```python
from pbixray import PBIXRay
model = PBIXRay('path/to/report.pbix')
tables = model.tables              # list of table names
schema = model.schema             # pandas DataFrame of table, column, datatype [13]
measures = model.dax_measures     # DataFrame of measures with formulas [2]
```

You can convert these outputs into plain Python lists/dicts for our documentation purposes. PBIXRay also provides Power Query M scripts (`model.power_query`), which could be included in the documentation (perhaps as an advanced detail or appendix, since the prompt doesn't explicitly ask for M code but it might be a nice addition for completeness). It also can list **calculated tables** (`model.dax_tables`) and **calculated columns** (`model.dax_columns`) [17] [3], which should be documented under measures/ calculations.

For the **visual layout**, since PBIXRay focuses on data model, you should manually parse the `Layout` JSON. After obtaining the JSON (using technique described above or borrowing from existing scripts like the one by `grenzi` on GitHub [18]), extract the hierarchy of **report pages → visuals → fields used**. In the JSON, visuals often reference fields by an identifier; you might need to map those to actual field names (which are likely present in the JSON's `resourcePackages` or similar section). For MVP, listing the field names as seen in visuals is valuable to users documenting reports.

Parsing Tableau (`.twb/.twbx`) Workbooks

Tableau workbook files are easier to parse because they are XML-based and Tableau provides some tooling. A `.twb` is an XML file, and a `.twbx` is a zipped package that contains the `.twb` plus any external resources (data extracts, images) [1]. The core of the Tableau metadata (datasources, worksheets, fields, etc.) is in the XML.

Approaches for Tableau parsing: 1. **Tableau Document API (Python)**: Tableau has an official (though not formally supported) **Document API** in Python. This can read a workbook and provide structured access to its contents. For instance, using the Document API, one can do:

```
from tableaudocumentapi import Workbook
wb = Workbook('sample.twbx')
datasources = wb.datasources   # list of Datasource objects [19]
worksheets = wb.worksheets     # list of Worksheet objects/names [19]
```

Each **Datasource** object gives access to fields, including calculated fields and their formulas [20]. Each **Worksheet** can list which fields it uses (the Document API might not directly list fields per worksheet, but the field objects have a property indicating which worksheets use them [8]). The Document API can thus retrieve: - Data source connections (server, database, etc.) [21]. - Field definitions (name, type, whether it's a dimension/measure) [22]. - Formulas for calculated fields (`Field.calculation` property gives the formula [23]). - Worksheet names and dashboard names. You may have to manually map which worksheets belong to which dashboard (the XML structure has `<dashboard>` tags referencing sheets).

Using this API can simplify development – the library handles the XML parsing under the hood and adapts to differences in Tableau version formats. Since it's open-source [24], you can include it in the project (`tableaudocumentapi` on pip).

1. **Direct XML parsing**: Alternatively, you can parse the XML with Python's `xml.etree.ElementTree` or `lxml`:

2. The XML structure of `.twb` includes sections like `<datasources>` (with `<connection>` details and `<column>` elements for fields) and `<worksheet>` sections which define each sheet's structure and the fields used in it.
3. By parsing XML, you can extract all `<column>` entries (fields) along with attributes like `name`, `datatype`, `role` (dimension/measure), and any `<calculation>` formula inside the column tag (for calculated fields).
4. You can also find `<connection>` tags to list the data source type and server/database names (for instance, `Connection` class in Document API corresponds to these tags [21]).
5. To get the layout of worksheets and dashboards, look for `<worksheet name="...">` tags and for `<dashboard name="...">` tags. Dashboards contain references to the worksheets they include (in Tableau XML, a dashboard is a collection of layout items that reference worksheets by name).

6. This approach is more verbose but does not rely on external libraries. It may require maintenance when Tableau updates their format (though basic metadata tends to remain accessible).

7. **Existing open-source tools**: There are some projects that have tackled Tableau workbook parsing:

8. **Tableau XML Parse (Python)**: e.g., Dave Rintoul's tool which extracts field metadata from Tableau workbooks [4]. It focuses on field names, types, and calculations, then outputs to Excel. You can use similar logic but output to Markdown/JSON instead. This tool demonstrates handling both `.twb` and `.twbx` (by unzipping) and iterating through data sources [4].
9. **TWB Ruby gem**: An older Ruby library (`twb` gem by Chris Gerrard) exists for parsing Tableau workbook XML [25], but unless you plan to call Ruby from your tool (unlikely), it's more of a reference that parsing Tableau XML is feasible.
10. **tabtosql / twp**: Tools that extract SQL from Tableau (for workbooks using custom SQL) [26]. If documenting underlying SQL queries is in scope, you might consider integrating such logic. However, our focus is on metadata (fields, measures).

For an MVP, using the **Tableau Document API for Python** is a quick win. It will give you structured objects to iterate: for each datasource, loop through fields and list their attributes and formulas; list all parameters if any; list all worksheets and note which datasource they use. You can augment this by directly reading the XML if needed to get additional info (like the exact layout of dashboards or to verify which fields are used in which worksheet – though the Document API's `Field.worksheets` property indicates where each field is used [8]).

Data to extract for Tableau: - **Datasources**: for each datasource, get the connection info (type of database, server, etc.), and list the fields in it. - **Fields**: for each field, note if it's a dimension or measure (Tableau roles), its data type, and if it's calculated (if so, include the formula). For example, a calculated field's formula can be obtained via the API [27] or by the `<calculation>` XML in the field definition. - **Worksheets and Dashboards**: list all worksheet names. Optionally, list dashboards and which worksheets they contain. Also, to satisfy the "visual tree or layout structure", you can outline something like: Dashboard "Sales Overview" contains worksheets "Sales by Region" and "Top Products", etc., and perhaps list key filters or parameters used on them (that might be too detailed for MVP, depending on time). - **Mappings**: it's useful to document which fields are used in which worksheet (to know field usage). The Document API provides this by each Field having a `.worksheets` list [8]. So you could, for each datasource's field, note the worksheets where it appears.

Output Generation (Markdown and JSON)

Markdown Documentation: Aim for a clean, organized Markdown output. Here's a suggested outline for the Markdown structure, incorporating both Power BI and Tableau cases (the tool should output whatever is relevant depending on the file type):

- **Title**: Use the workbook/report name as the title (e.g., "Documentation for `<ReportName>`" as an H1).
- **Overview**: A short description (one can be generated or left for the user) and possibly a table of contents with links to sections (optional).
- **Data Sources**: List each data source:
- For Power BI: list the data source kinds (e.g., "SQL Server - connection string", "Excel file - path", or "Web API – URL") if accessible via Power Query queries or model metadata. If using PBIXRay, some of this might be in `model.metadata` or Power Query definitions [28] (the M queries often contain the connection info).
- For Tableau: list each connection (the Document API's `Connection` properties give server/database details [21]). E.g., "Superstore.xlsx (Excel file)" or "SQL Server – Server: X, Database: Y".
- **Tables and Fields**:
- For each table (or each Tableau datasource), present a **table schema**: You can do a subheading per table/datasource, then a bullet list or table listing fields. Include field name, data type, and notes if it's a key or has special role. You can annotate calculated fields (e.g., mark them with asterisks or list their formula right below).
- If using a table format in Markdown, columns could be: Field Name | Type | Description. The "Description" could include "Calculated: <formula>" or "(calculated field)" as a flag, or if you want to be thorough, you might list formulas in a code block below the table.
- **Measures/Calculations**:
- For Power BI, list DAX measures separately (since they belong to tables, you could list them under their respective table or in a consolidated section). Each measure: Name, formula (possibly in a Markdown code block for readability), and perhaps the format string if available.
- For Tableau, calculated fields are already covered in the fields list, but if you want a special section, you could highlight all calculations as well.
- **Visualizations (Report Structure)**:
- For Power BI: list each page (report tab). Under each page, list visuals by name or type (if names aren't available, perhaps by type index, e.g., "Bar Chart 1") and the fields used. For example:
    - **Page: Sales Overview**
    - *Visual*: Clustered Bar Chart – **Fields**: [Sales.Amount], [Product.Category], [Date.Month]
    - *Visual*: Card – **Field**: [Sales.Amount (YTD)]
      This gives a tree of what fields feed each visual, which is often what documentation requires (knowing which calculations are used where).
- For Tableau: list each dashboard and worksheet. For each worksheet, you could list the fields in use (dimensions and measures in that view). If a worksheet has filters or parameters, documenting those could be useful too. If multiple worksheets feed a dashboard, you can reflect that hierarchy.
- **Additional Metadata**:
- **Relationships** (Power BI): If the model has relationships between tables, you can include a section describing them (e.g., "Table A [Column X] $\longrightarrow$ Table B [Column Y] (Many-to-One, bidirectional)" to capture joins). PBIXRay provides this info [14] .

- **Parameters**: Both Power BI (via M parameters [29]) and Tableau (parameters in workbook) may have user-defined parameters. List them if present, including default values.
- **Refresh Schedule / Last Update**: Not applicable unless connected to service, but you might include "Last updated time" if the info is in the file (Power BI's metadata might have last saved time).
- **Lineage**: Possibly list the original data source for each field if that's meaningful (e.g., in Power BI, field X comes from SQL table Y.column Z).

When writing the Markdown, use consistent formatting. E.g., use backticks for field names or code (to differentiate from prose), use **bold** for section labels like visual names or table names to make them stand out. Aim for readability (since the user specifically wants easy scanning). Avoid overly dense paragraphs; prefer itemized lists for fields and measures.

JSON Output: The JSON should capture the same information in a structured way. For example, you could structure it like:

```
{
  "file": "ReportName.pbix",
  "type": "Power BI",
  "data_sources": [
    {
      "name": "SQLServerDataset",
      "connection": "mssql://server/database",
      "tables": [
        {
          "name": "Sales",
          "columns": [
            {"name": "SalesId", "data_type": "Int64"},
            {"name": "Amount", "data_type": "Decimal"},
            ...
          ],
          "measures": [
            {"name": "Total Sales", "expression": "SUM(Sales[Amount])"}
          ]
        },
        ...
      ]
    }
  ],
  "relationships": [
    {"from_table": "Sales", "from_column": "ProductId", "to_table": "Products",
"to_column": "ID", "cardinality": "Many-to-one"}
  ],
  "visualizations": [
    {
      "page": "Sales Overview",
      "visuals": [
        {
          "type": "Bar Chart",
          "title": "Sales by Category",
          "fields": ["Product.Category", "Sales.Amount"]
        },
        ...
      ]
    }
  ]
}
```

The above is an illustrative snippet. The actual schema is up to you – the key is to include all relevant info in a hierarchical JSON. Ensure consistency, e.g., if multiple data sources exist, separate them; if it's a single data model (like Power BI usually has one model), you might not need a nested data source array, just use tables at root. Document the schema so users know where to find each piece.

One advantage of JSON output is that it can be used in automated checks or even fed into visualization tools (including potentially feeding back into Tableau or Power BI for a self-documenting report!). Keep the JSON keys intuitive and lowercase (e.g., `data_sources`, `tables`, `measures`, etc.).

CLI Implementation and Logging

Implement the CLI using a robust library or the built-in `argparse`: - **Arguments**: `--input` (or positional argument) for input file path. Possibly allow multiple inputs by repeating `--input` or accepting a directory. `--output` for output directory (default could be current directory or a `docs/` folder). `--format` to specify `markdown`, `json`, or `all`. `--verbose` or `--debug` for logging verbosity. Maybe `--no-docker` is not needed because Docker usage is external (the user would choose to run in Docker or not). - **Example**: `bidoc parse --format markdown --input report.pbix --output ./docs/report.md` – you can design as subcommands like `parse` or just have options on the base command. - Use `argparse` to display help and usage. Make sure to handle errors like "file not found" (print a clean error message).

For **logging**, configure at the start of execution:

```
logging.basicConfig(level=logging.INFO, format="%(levelname)s: %(message)s")
```

If verbose flag is set, adjust level to DEBUG. Throughout the code, use `logger.info()` for normal progress and `logger.debug()` for detailed internal state (like "Parsed 10 fields from Sales table"). Use `logger.error()` to report issues. Avoid excessive logging for large items (like printing every field name in debug might be okay, but not in info level).

If an error occurs (exception during parsing), catch it in the CLI layer, log an error, and continue to the next file (if multiple inputs). Also consider exit codes: return a non-zero exit code if any file fails to process, so that CI pipelines can catch failures.

Packaging and Docker Details

To facilitate use in different environments: - **Python Package**: Even if not publishing to PyPI, setting up `setup.py` or `pyproject.toml` with entry points means a user can install the tool in a virtualenv and run `bidoc-cli`. Use the console_scripts entry point like:

```
entry_points={
    'console_scripts': [
        'bidoc-cli = bidoc.cli:main'
    ]
}
```

This will map the command to your `main` function in `cli.py`.

- **Dockerfile**: Write a Dockerfile that:
- Uses a base image like `python:3.11-slim` for small footprint.
- Sets working directory, copies in the relevant files (you might use multi-stage build if compiling anything, but here mostly Python).
- Installs dependencies (you'll need system libs if any of the parsers require them. PBIXRay uses `kaitaistruct` and `apsw`, which might require SQLite or such; ensure those install via pip or apt as needed).
- Sets the entrypoint to the CLI command. For instance:

  ```
  ENTRYPOINT ["bidoc-cli"]
  ```

  so running the container followed by arguments will pass into the tool.
- Consider adding a specific user in the container for security (to avoid running as root), though for simplicity, it's optional.
- Document how to use the Docker container, e.g., in README: "`docker run -v $(pwd):/data bidoc-image --input /data/report.pbix --output /data/docs`".

Testing the Docker container is important – ensure it runs and can access files via mounted volumes.

Because the user expects to integrate this in **Git-based workflows**, the idea is that the output Markdown/JSON can be committed to a repository for version tracking. The tool could be run as part of a CI pipeline (for example, whenever a `.pbix` is updated, run doc generation, then perhaps commit or send as an artifact). By providing the Docker image, users can easily plug it into GitHub Actions or Azure DevOps pipelines without worrying about Python environment setup.

Best Practices and Future-Proofing

Following best practices ensures the tool is maintainable: - **Code Quality**: use linting (pylint/flake8) and possibly type hints for clarity. This is not directly exposed to user, but important if multiple contributors. - **Performance**: Parsing large workbooks can be heavy. PBIXRay and the Tableau API are reasonably efficient, but be mindful if a workbook has thousands of fields. Use efficient data structures. For instance, PBIXRay returns pandas DataFrames for some properties – if not too large, that's fine to iterate or convert; otherwise, ensure that any heavy computation (like reading a large JSON) is done in a streamed manner if possible. - **Extensibility**: The architecture should allow adding new features: - Adding support for **additional output format** (say HTML or direct Confluence wiki page upload) should be achievable by adding a new module without touching core parsing. - Adding support for another BI tool (e.g., Looker or Qlik) in the future would involve writing a new parser and hooking it in. - The AI summary feature could later use an API call to an LLM to generate text. By having a stub, later one can implement: `summary = openai.summarize(metadata)` for example, and then append the summary to the Markdown (maybe in an "Insights" section). - **Documentation**: Provide a README with usage examples. Also, within the code, docstrings on functions like `parse_pbix(file_path)` explaining what it returns (so future developers or AI assistants working on it can understand context).

Lastly, ensure to cite or credit any open-source libraries you use in your documentation or comments. For example, if using PBIXRay, mention it in the README (it's MIT licensed, so just a note is fine). Similarly for the Tableau Document API.

By adhering to these guidelines, the developed tool will be robust, easy to maintain, and ready for future enhancements. It will effectively serve the need of generating documentation from BI files, in a way that's automated and integrable into various workflows.

Open-Source Parsers and SDKs for `.pbix` and `.twb/.twbx`

When building this tool, it's advantageous to leverage existing libraries and SDKs where possible:

- **Power BI (`.pbix`)**:
- **PBIXRay (Python)** – *Open-source parser library.* As noted, PBIXRay can extract a wealth of information from `.pbix` files, including tables, columns, DAX measures, and even the underlying VertiPaq engine data (column statistics) [11] [30]. It's a Python package, making it easy to integrate. Given its capabilities (and active development as of 2025), it is a top choice for handling `.pbix` parsing in our tool. Using it aligns with the requirement to **extract datasets, fields, measures, etc.** from Power BI.
- **pbi-tools (CLI, .NET)** – *Open-source DevOps tool.* PBI-Tools is a command-line tool that deconstructs `.pbix` files for version control [15]. It's written in .NET and can output the Power BI file contents (including a `database.json` for the model and the report layout). We might not call it directly from our Python tool, but it's good to know. In scenarios where using a Python library is insufficient (or if one wanted to implement this in a .NET environment), pbi-tools is invaluable. It also has a **Docker image** available [31], showing the emphasis on CI/CD. Even if not used in code, the **concept** from pbi-tools of splitting the model and report is something our tool mimics.
- **Tabular Object Model (TOM) / Analysis Services** – (mention for completeness) Microsoft's official way to programmatically interact with Power BI models is via the Tabular Object Model library (C#) or the XMLA endpoint for deployed datasets. For .pbix files, these are not directly accessible offline (except through external tools like Tabular Editor). Since our focus is offline `.pbix` files, we stick with PBIXRay or similar. But if one were to extend the tool, connecting to the Power BI Desktop process via external tools or using the XMLA endpoint for datasets on the service could retrieve model definitions as well.

- **Community Scripts**: The Stack Overflow example [18] and projects like `powerbi-model-utilization` are small open-source snippets that show how to handle the Layout JSON. These are not full libraries but can be referenced for guidance on JSON parsing of visuals.

- **Tableau (`.twb/.twbx`)**:

- **Tableau Document API (Python)** – *Official API, open-source on GitHub.* This library provides a high-level interface to Tableau workbook content. It handles both `.twb` and `.twbx` packaging details [32] and gives Python objects for workbook, datasources, connections, fields, etc. It's not full coverage of everything in the XML, but it supports common tasks (and was intended for modifying workbooks like updating connections). For our needs, it covers listing worksheets, data sources, and fields (including calculated field formulas) [27] [19]. Installable via pip (`tableau-document-api`).
- **TWB Ruby Gem** – *Community SDK.* The `twb` gem by Chris Gerrard is a Ruby tool for reading Tableau workbook content [25]. It can be a reference or alternative if someone were to use Ruby. Since our context is likely Python, we won't use it directly, but it validates that parsing Tableau XML is feasible and has been done externally.
- **Tableau Metadata API** – *Server/Online API.* Note: Tableau also has a Metadata GraphQL API (for Tableau Server/Cloud) that can query content (including lineage, fields, etc.) [33]. This requires the workbook to be published to a Tableau Server, so it's outside our offline scope. But it's a potential future integration point (imagine documenting differences between what's in the file vs what's published).
- **Custom XML parsing** – Libraries like Python's built-in `xml.etree` are sufficient. The open-source `tableau-xml-parse` project demonstrates using `xml.etree.ElementTree` to read `.twb` files and extract field metadata into Python objects and ultimately Excel [34] [35]. We can use a similar approach if not using the Document API.

Recommendation summary: For the MVP, **PBIXRay** [11] and **Tableau's Document API** [32] are suggested, as they will handle the heavy lifting of parsing and let us focus on formatting the output. Both are open-source and can be added to our tool's dependencies. By using these, we ensure we're not reinventing the wheel and we leverage tested parsers. If those libraries cover 80% of needs, our code can simply transform their outputs into the desired documentation structure.

In conclusion, armed with these libraries and guidelines, the AI coding assistant (or a human developer) should be well-equipped to implement the BI documentation tool. The design is modular, the approach is informed by current best practices and tools, and the end result will be a comprehensive documentation utility that benefits developers and analysts working with Power BI and Tableau. With placeholders for AI enhancements, the tool is also ready to evolve, possibly incorporating automated insights in the future.

Sources:

- Power BI `.pbix` file parsing: PBIXRay library overview [11] and feature list (tables, metadata, DAX, etc.) [2].
- pbi-tools for Power BI (decompiling .pbix to folder/JSON) [15].
- Stack Overflow on `.pbix` report layout JSON extraction [7] [16].
- Tableau workbook format (XML in `.twb`, zipped in `.twbx`) [1] and open-source "twb" library (Ruby) reference [25].

- Tableau Document API usage (Workbook class and properties) [32] [19] and Field properties (e.g., calculated field formula, usage in worksheets) [36] [8].
- Example of extracting Tableau field metadata via XML (drintoul's tableau-xml-parse) [4].
- Tableau SQL extract tool (twp) indicating approach to parse workbook XML for queries [26].
- Talend note on lack of public Power BI SDK [10] (underlying reason for using community tools).

---

[1] [24] [25] What Tableau API does InterWorks uses in their Workbook SDK? - Stack Overflow
https://stackoverflow.com/questions/30136030/what-tableau-api-does-interworks-uses-in-their-workbook-sdk

[2] [3] [11] [12] [13] [14] [17] [28] [29] [30] GitHub - Hugoberry/pbixray: PowerBI (pbix) file parser. Surfaces metadata and VertiPaq tables from PowerBI DataModel.
https://github.com/Hugoberry/pbixray

[4] [9] [34] [35] GitHub - drintoul/tableau-xml-parse: Process Tableau Workbooks' XML programmatically
https://github.com/drintoul/tableau-xml-parse

[5] [6] [8] [19] [20] [21] [22] [23] [27] [32] [36] API reference
https://tableau.github.io/document-api-python/docs/api-ref

[7] [16] [18] python - Power BI(PBIX) - Parsing Layout file - Stack Overflow
https://stackoverflow.com/questions/66831049/power-bipbix-parsing-layout-file

[10] Microsoft Power BI (File) - Import | Talend Data Catalog Help
https://help.qlik.com/talend/en-US/talend-data-catalog/8.0/Content/MIRMicrosoftPowerBIFileImport.htm

[15] Converting a Power BI Dataset .bim to a .pbix with pbi-tools — DATA GOBLINS
https://data-goblins.com/power-bi/download-pbix

[26] GitHub - PrecisionNutrition/twp: Simple Tableau SQL Extract Tool
https://github.com/PrecisionNutrition/twp

[31] Welcome to pbi-tools | pbi-tools
https://pbi.tools

[33] Introduction to Tableau Metadata API
https://help.tableau.com/current/api/metadata_api/en-us/index.html