Cureus
Part of SPRINGER NATURE

Open Access Technical Report

PyMAUDE: A Python Library for Local and Reproducible Analysis of the US Food and Drug Administration's Manufacturer and User Facility Device Experience (MAUDE) Database

Jacob H. Schwartz [1], Maya M. Almoussa [1], Nicole Blattman [1], Mina S. Makary [2]

1. Medical School, University of Michigan, Ann Arbor, USA 2. Department of Radiology, The Ohio State University Wexner Medical Center, Columbus, USA

Corresponding author: Jacob H. Schwartz, jaschwa@umich.edu

Review began 09/23/2026
Review ended 10/27/2026
Published 11/01/2026

© Copyright 2026
Schwartz et al. This is an open access article distributed under the terms of the Creative Commons Attribution License CC-BY 4.0., which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

DOI: 10.7759/cureus.118938

Abstract

The US Food and Drug Administration's (FDA) Manufacturer and User Facility Device Experience (MAUDE) database is the primary national repository of medical device adverse event reports. Although the FDA provides several access routes to MAUDE, each either limits query complexity or requires researchers to write study-specific code to process and query the data, complicating verification and reuse. Furthermore, the FDA does not release discrete versions of MAUDE and instead updates it in place and without version labels, making it difficult to link a given analysis to a specific state of the data. We developed PyMAUDE, a free and open-source Python package that builds a local, date-stamped copy of the MAUDE database in a few lines of code and detects revisions to previously downloaded FDA files by checksum. A built-in archiving function preserves the database as a set of Parquet files alongside a bundle of the raw source files, together suitable for upload to an online repository for later citation. PyMAUDE additionally provides advanced query capabilities; through Python function calls, users can match device fields, combine search term sets with AND/OR logic, or execute custom structured query language (SQL) queries. Retrieval from MAUDE was assessed by comparing the unique report keys of all 26.6 million reports in the database produced by PyMAUDE to the openFDA bulk export, finding that 26,557,462 (99.998%) of 26,558,073 unique report keys present in either source were present in both, with all 611 discrepant report keys confirmed to reflect differences in field content or presence of reports in the FDA's distribution channels rather than errors in PyMAUDE data processing. Across five exact-match queries returning 774 to 129,149 reports, PyMAUDE and the openFDA application programming interface (API) returned identical report-key sets in two cases and differed in each of the remaining three, with each difference attributable to PyMAUDE's case-insensitive matching discovering more records than the API or to data-channel differences. In our benchmarks on a consumer laptop, full database construction from pre-downloaded source files took a median of 9 minutes and 14 seconds, resulting in a database size of 57.7 GB on disk. Once built, exact-match queries on device fields each completed in less than 7.0 seconds, and ungrouped device-name searches combining terms with AND/OR logic each completed in less than 5.3 seconds. Archiving the complete database took a median of 3 minutes and 19 seconds, producing a 13.7 GB archive. PyMAUDE is available on GitHub and via PyPI under the MIT license, with source code and a prebuilt database snapshot as of October 21, 2026 archived on Zenodo. By pairing a programmatic query interface with a citable database snapshot, PyMAUDE enables device safety researchers to perform custom, complex, and reproducible analyses of MAUDE.

Categories: Other, Healthcare Technology
Keywords: adverse event reporting, maude database, medical device safety, post-market surveillance, reproducible research

Introduction

The Manufacturer and User Facility Device Experience (MAUDE) database, maintained by the Food and Drug Administration (FDA), is the foremost repository of medical device adverse event reports in the United States. As of October 21, 2026, by our analysis, MAUDE contains 26.6 million medical device reports (MDRs). These are submitted by mandatory reporters, including manufacturers, importers, and device user facilities, as well as voluntary reporters, such as physicians, patients, and consumers [1]. Coverage dates back to 1991 for user facility reports, June 1993 for voluntary reports, and August 1996 for manufacturer reports [2].

MAUDE is a passive surveillance system, and as a result it has several well-known limitations. First, MAUDE includes only reports of adverse events, without information about overall device use that occurs without incident. As such, MAUDE does not provide a denominator for the estimation of the incidence or prevalence of any kind of device failure [2]. Furthermore, individual MDRs establish only that a device was in use when an event occurred, not whether the device caused it [2]. MDRs are additionally subject to under-reporting of events and inaccuracies in reports [2,3]. Structured fields are often populated inconsistently; brand names often appear in generic device name fields (and vice versa), device and manufacturer names often vary in spelling, abbreviation, and corporate designation, and product codes often appear inconsistently or are absent [4]. As a result, identifying a complete set of reports for a given device requires searching several fields for multiple spelling and naming variants, while excluding the unrelated devices that such broad searches inevitably return [5].

How to cite this article
Schwartz J H, Almoussa M M, Blattman N, et al. (November 01, 2026) PyMAUDE: A Python Library for Local and Reproducible Analysis of the US Food and Drug Administration's Manufacturer and User Facility Device Experience (MAUDE) Database. Cureus 18(11): e118938. DOI 10.7759/cureus.118938

The FDA provides MAUDE to the public via four distinct routes: 1) Online search tool [6]. This is the most accessible route, providing the ability to query quickly without any setup. However, it queries only the last 10 years of data, caps the results at 500 per query, and provides only a small subset of MAUDE fields in its downloadable results [4]. This route also does not support compound logic or multi-stage queries. A study of query reproducibility via this route found that only 14 of 60 executable queries from published MAUDE studies reproduced the originally reported number of reports within 5% [7]; 2) Downloadable flat text files [1]. These contain the full archive but require custom scripting to join and query. Furthermore, they are poorly documented and contain inconsistencies, increasing both the complexity of custom code and the likelihood of error [4]. Such code is rarely published, making analyses difficult to reproduce or verify, and additionally this interface is inaccessible to investigators without programming experience; 3) openFDA application programming interface (API) [3]. While the API exposes MAUDE data programmatically without requiring text-parsing code or a full-database download, it is designed for real-time queries rather than bulk device safety analysis. API requests using standard paging (the default) return at most 26,000 MDRs per query, with larger result sets requiring a cursor-based workaround [8]; 4) openFDA Dataset Download [9]. The data underlying the openFDA API are also available for download as a set of JSON files. This avoids the API's rate limits and the workarounds required for large result sets, but still requires custom code to parse and query. Additionally, these files provide openFDA's harmonized representation of MAUDE rather than the native structure of the source files.

Notably, while a query by any of the above four routes can be reported as having been run on MAUDE as of a specific date, none of them can retrieve a prior version of MAUDE. Here, a "version" of MAUDE means a fixed, identifiable state of the complete set of reports from a specific date. The FDA does not publish such versions; instead, it revises MAUDE in place [1,2]. These revisions occur at two levels. First, the database is updated monthly at the record level: new records are added as new events are reported, and individual records change as supplemental reports are received and as corrections are applied. Second, at the structural level, the FDA has repeatedly made major changes to file layouts and use of table fields over the years (Appendix 1). A retrieval date alone thus does not identify the data an analysis used, and a MAUDE-based analysis cannot be re-executed against its original data unless the investigators retained their own copy of the source files. Such a copy requires careful data management, and even then, if it is kept only as files on a local drive, it cannot be externally verified as matching MAUDE on that date. Furthermore, code written against one release of files or one access route might not run against a later one if the source data format changes.

Access to MAUDE itself is in transition. In March 2026, the FDA launched the Adverse Event Monitoring System (AEMS), a centralized adverse event platform that consolidates adverse event reports across FDA-regulated product categories, and announced that MAUDE and its historical data would be among the legacy systems integrated into AEMS by May 2026 [10]. As of October 2026, legacy (pre-AEMS) access to MAUDE remains available by the aforementioned four routes; nevertheless, the long-term availability and future file formats of MAUDE are uncertain, compounding the problem above. A date-stamped copy of the full archive, deposited in a public repository, would address both problems, preserving a citable version of MAUDE regardless of subsequent changes to FDA infrastructure.

Several tools have previously been developed to address certain limitations of MAUDE. MAUDEMetrics retrieves records on-demand through the openFDA API, automates the API's cursor-based pagination to allow users to easily bypass the 26,000-record ceiling, provides a no-code desktop interface for query construction, and standardizes fields for export [11]. This maximizes accessibility, but MAUDEMetrics is distributed as an application rather than as a library, and additionally only provides a small subset of MAUDE fields to query; together these limit its utility in a larger analytic pipeline. It also operates against the live API without versioning the database, limiting reproducibility over time given ongoing updates and revisions to MAUDE. Another tool, MAUDE-Dash, parses MAUDE's downloadable flat text files into a DuckDB database, then presents the data in an interactive dashboard for filtering, trend visualization, and narrative text analysis [12]. Building its own database removes the dependence on FDA's online tools after the initial download; however, MAUDE-Dash does not archive or checksum that database, and it does not expose a documented query interface intended for programmatic use. Furthermore, MAUDE-Dash does not deliver narrative or problem code data from reports received prior to 2015 even though these exist in the source MDR Data Files [1,12]. Earlier efforts include various scripts for downloading and parsing MAUDE text files without a query layer [13,14] and classification pipelines applied to MAUDE event text narratives [5,15].

PyMAUDE is a Python library that provides programmatic access to a local copy of MAUDE. PyMAUDE downloads the MAUDE flat text files for a user-specified year range and set of tables, and loads them into a local DuckDB database [16]. Downloaded files are tracked by SHA256 checksum, so re-running a build reloads only files whose contents have changed. The resulting database is date-stamped, and PyMAUDE can produce an archived copy for deposit in a repository such as Zenodo for citation alongside a published analysis. Queries with PyMAUDE are written as Python function calls and return results as pandas DataFrames [17]. Devices may be selected by exact match on product code, brand name, generic name, or manufacturer, or by case-insensitive substring matching across all three device name fields at once, addressing MAUDE's inconsistent field population described above. The library additionally supports multi-stage queries with compound logic (e.g. AND, OR), as well as grouped multi-cohort searches. Retrieved cohorts can be enriched with patient outcomes data, device problem codes, or patient problem codes; results can also be joined to free-text event narratives or summarized by year, all while the underlying DuckDB connection enables arbitrary structured query language (SQL) commands if preferred by the user. PyMAUDE

can be found on GitHub at https://github.com/jhschwartz/PyMAUDE and downloaded via PyPI at https://pypi.org/project/pymaude under the MIT license, with source code and a prebuilt database snapshot as of October 21, 2026 archived on Zenodo [18,19]. This paper describes the underlying architecture of PyMAUDE, details usage of its database management and query functions, validates retrieval against the openFDA bulk downloads and API, and benchmarks database construction and query performance on a consumer laptop.

Technical Report

PyMAUDE is released as a Python package, intended for use in a scripting or notebook environment. It provides functions to 1) download and process the MAUDE database, or a temporal subset of interest, into a local queryable SQL database; 2) search that database with simple or complex queries; and 3) return results as pandas DataFrames for further analysis or export. These capabilities are described below and are further illustrated in example Jupyter notebooks provided in both the GitHub repository and archived codebase.

Database configuration

PyMAUDE constructs its local database from files downloaded from the FDA MDR Data Files webpage [1]. These files contain data underlying six tables: master, device, patient, device_problem, patient_problem, and text. PyMAUDE mirrors this structure in a local DuckDB database [16], and the contents of each table are summarized in Table *1*.

Table	One row per	Columns	Example fields	Relationship to master
master	Event report	87	MDR_REPORT_KEY, REPORT_NUMBER, DATE_REPORT, REPORTER_OCCUPATION_CODE, EVENT_TYPE, source_file*	Base table
device	Device in a report	53	MDR_REPORT_KEY, DEVICE_EVENT_KEY, GENERIC_NAME, MANUFACTURER_D_NAME, DEVICE_AGE_TEXT, DEVICE_NAME_CONCAT*, source_file*	One-to-many
text	Narrative record associated with a report	7	MDR_REPORT_KEY, MDR_TEXT_KEY, TEXT_TYPE_CODE, FOI_TEXT, source_file*	One-to-many
patient	Patient in a report	11	MDR_REPORT_KEY, PATIENT_SEQUENCE_NUMBER, PATIENT_AGE, PATIENT_SEX, source_file*	One-to-many
device_problem	Device problem code in a report	3	MDR_REPORT_KEY, DEVICE_PROBLEM_CODE, source_file*	One-to-many
patient_problem	Patient problem code in a report	6	MDR_REPORT_KEY, PATIENT_PROBLEM_CODE, source_file*	One-to-many

TABLE 1: Contents and relationships of the PyMAUDE database tables

Description of tables in the PyMAUDE database, including the unit represented by each row, number of columns, example fields, and relationship of each table to the master event-report table. The master table contains one row per event report and serves as the base table, joining to the others on MDR_REPORT_KEY. Device, text, patient, device-problem, and patient-problem data have one-to-many relationships with the master table, because an individual event report may involve multiple devices, patients, narrative records, or problem codes. The DEVICE_NAME_CONCAT field in device is generated by PyMAUDE during preprocessing and is not present in the source; it combines the generic name, brand name, and manufacturer name into a single searchable field to facilitate efficient searches across device-name fields. The text table stores narrative information associated with event reports; narratives may be updated over time, and additional text records may be added and linked back to the original report. Where numbers of columns in tables have changed in the source data over time, PyMAUDE retains the union number of columns, leaving any years that did not have a given column specified with null values for that column. *Column counts include all columns from the FDA source files in addition to "source_file" in every table, which is added by PyMAUDE to record which source file each row came from, as well as "DEVICE_NAME_CONCAT" in the device table. MDR: Medical Device Report.

The master table is the base table, with one row per report and a unique key, MDR_REPORT_KEY, joining to the other five tables. Among its 87 columns are the date the report was received, the type of reporter, manufacturer contact information, remedial actions, reporter state and country, date added, and date last modified. Rows in master have a one-to-many relationship with rows in the other tables: a single report may describe an event affecting one or more patients, involving one or more devices, and attributed to one or more device or patient problems.

PyMAUDE's add_years(...) function abstracts the downloading, decompression, and loading of the requested tables into a single call for a user-specified range of years. It additionally repairs malformed lines encountered during parsing, the process of which is described in Appendix 1. The source files vary in naming, partitioning, temporal coverage, and column structure, both across tables and across years within a single table. add_years(...) automatically resolves these differences according to a per-table metadata file in the PyMAUDE codebase that specifies each table's file-naming patterns, partitioning, start year, and column

names. Any columns absent in some years' files for a given table are retained but left null for those years. Figure *1* shows example usage of add_years(...) and other database management, and further detail on the data ingestion process is provided in the table in Appendix 1.

```python
from pymaude import MaudeDatabase

# add_years(...) downloads, repairs malformed lines, and loads MAUDE tables
# into a local DuckDB database for a user-specified range of years.
db = MaudeDatabase(db_path='./maude.duckdb', data_dir='./maude_data', memory_limit='4GB')
db.add_years(
  years='2015-2025',
  tables=['master', 'device', 'text', 'patient', 'device_problem', 'patient_problem'],
  download=True
)

# update(...) re-downloads every currently-loaded source file and reloads
# only those whose checksum has changed since the last build.
db.update(force_download=True)

# archive(...) writes a checksum manifest alongside a citable snapshot of the
# database and its raw source files, for deposit in a repository such as Zenodo.
db.archive(output_dir='./maude_archive', include_raw=True)
```

FIGURE 1: Example PyMAUDE database initialization, updating, and archiving

Example code demonstrating the four core database management functions. MaudeDatabase(…) opens or creates a local database, specifying the database path, the directory to hold downloaded source files, and an application memory limit. add_years(…) downloads and loads the listed tables (the year range applies to the yearly device and text tables; the other four tables are cumulative files and are always loaded in full). update(…) re-downloads each currently loaded source file, compares its checksum against the value stored the last time the source file was ingested, and reloads the file only if the checksum is not equal, indicating the source file has changed. archive(…) writes the compressed database (as Parquet files), a checksum manifest, and the raw source files to a specified directory; the resulting format is intended for sharing or deposit. Line-by-line explanation of analogous examples is provided in the example Jupyter notebooks released with PyMAUDE.

Checksums, database updates, and archiving

During database initialization, PyMAUDE stores the SHA256 checksum of each imported text file in a metadata table. This metadata table records, for each imported file, the corresponding table, the report year, the source file name, its checksum, the number of rows loaded, and the timestamp at loading. Together these fields document which files were used to build the local database and when the build occurred.

The update(...) function keeps the local database current with the FDA's published files. It downloads each source file, regardless of whether a copy persists on disk from prior processing, and compares the new download's checksum against the value stored for that file. If the checksums differ, the corresponding data are reloaded from the new file; if they match, no further action is taken, limiting reprocessing to files that have actually changed.

Because the FDA revises MAUDE in place without version labels, the contents of a local PyMAUDE database depend on when the source files were retrieved. The metadata table provides a record of data retrieval, and the archive(...) function creates an archived copy of the database for potential upload to a repository such as Zenodo. Specifically, the function exports the database as Parquet files alongside a checksum manifest and a tar archive of the raw source files. Parquet was chosen for format stability and broad external support, so archived snapshots remain readable independent of DuckDB and PyMAUDE. Corresponding import function from_archive(...) restores a database from an archived snapshot. Such an archive is a locally created version in the sense defined above, because the manifest records the load time and SHA-256 checksum of every source file. A published analysis can therefore reference a specific, checksum-verifiable copy of MAUDE rather than referencing the online FDA database, whose contents at a given prior date cannot be reliably reconstructed.

Query interface

PyMAUDE provides two structured search functions and a function for arbitrary SQL access. Both structured search functions query the device table and return joined master and device records. Data from the patient, device_problem, patient_problem, and text tables can be attached after querying as needed. Figure *2* shows representative use of the query interface.

```python
# query_device(...): exact-match search on one or more device fields.
angiojet = db.query_device(brand_name='ANGIOJET', start_date='2015-01-01')

# search_by_device_names(...): case-insensitive substring search over the brand,
# generic, and manufacturer name fields, with user-specified boolean logic.
term_or     = db.search_by_device_names(['argon', 'cleaner'])
term_and_or = db.search_by_device_names([['venous', 'thrombectomy'], ['vein', 'angiojet']])

# query(...): arbitrary SQL for anything outside the two functions above.
reports_per_year = db.query(
    "SELECT year(DATE_RECEIVED) AS year, COUNT(*) AS n "
    "FROM master GROUP BY year ORDER BY year"
)

# Attach patient outcomes, problem codes, and narratives, then filter on any of them.
results = db.enrich_with_patient_data(term_and_or)
results = db.enrich_with_device_problems(results)
narratives = db.get_narratives(results['MDR_REPORT_KEY'])
deaths = db.filter_by_outcome(results, outcome='D')

# Results remain pandas DataFrames throughout, ready for analysis or export.
trends = db.get_trends_by_year(results)
trends.to_csv('thrombectomy_trends.csv', index=False)
db.close()
```

FIGURE 2: Example queries, result enrichment, and result filtering using PyMAUDE functions

Example code demonstrating PyMAUDE's three query functions: exact (case-insensitive) device-field matching with query_device(…), substring matching across device name fields with search_by_device_names(…), and arbitrary SQL via query(…). Results are then enriched with patient outcome data and device problem codes, joined to event narratives, and filtered by patient outcome code. Because all query, enrichment, and filtering functions return pandas DataFrames, results at any point can be summarized or written to file using standard pandas methods. Line-by-line explanation of analogous examples is provided in the example Jupyter notebooks released with PyMAUDE.

query_device(...) matches exactly on one or more of the device fields (product code, brand name, generic name, and/or manufacturer name), and additionally can be restricted to a range of dates reports were received. Matching is case-insensitive (with the exception of product code), and specifying multiple fields invokes combination of requirements by AND logic. Note that query_device(...) does not perform substring matching, so, for example, a search for manufacturer name "Boston Scientific" would not match a report with manufacturer name "Boston Scientific Corporation".

search_by_device_names(...) instead performs case-insensitive substring matching across the combined contents of brand name, generic name, and manufacturer name fields, addressing the inconsistent field usage for device naming, previously described in the Introduction. Search terms may be combined with user-specified boolean logic: a list of terms matches reports containing any of the terms in the list (e.g. term1 OR term2), while a nested list requires all terms within the nested list to be present (e.g. term1 AND term2). The user may also choose to pass a dictionary of named search term sets to perform a grouped search, which returns a single DataFrame with an additional column specifying the search group to which each resulting row corresponds. Reports matching more than one group are assigned to the first matching group in the order specified in the initial query, so each search result row is assigned only one search group name. Figure *3* provides an example of such a query.

```
results = db.search_by_device_names({
    'venous thrombectomy': [
        ['venous', 'thrombectomy'],       # (venous AND thrombectomy)...
        ['venous', 'angiojet'],           # OR (venous AND angiojet)...
        ['venous', 'argon', 'cleaner'],   # OR (venous AND argon AND cleaner)...
        ['vein', 'thrombectomy'],         # ...etc.
        ['vein', 'angiojet'],
        ['vein', 'argon', 'cleaner'],
        ['dvt', 'thrombectomy'],
        ['dvt', 'angiojet'],
        ['dvt', 'argon', 'cleaner'],
    ],

    # next group matches only reports not matched by the first group
    'arterial thrombectomy': [
        ['arterial', 'thrombectomy'],
        ['arterial', 'angiojet'],
        ['arterial', 'argon', 'cleaner'],
        ['artery', 'thrombectomy'],
        ['artery', 'angiojet'],
        ['artery', 'argon', 'cleaner'],
        ['coronary', 'thrombectomy'],
        ['coronary', 'angiojet'],
        ['coronary', 'argon', 'cleaner'],
    ],

    # third group matches only reports not matched by the first two groups
    'ambiguous thrombectomy': [
        ['thrombectomy'],
        ['angiojet'],
        ['argon', 'cleaner'],
    ],
})
```

FIGURE 3: Example use of PyMAUDE for a grouped compound query

Example code illustrating how a user might apply PyMAUDE's grouped compound query via the search_by_device_names(…) function to define multiple compound device cohorts in one call. A dictionary of named search term sets is defined as input to the query, with each dictionary key naming a group and each dictionary value a nested list of terms to define the group's query. The first group, keyed "venous thrombectomy," matches reports containing any of nine (AND) combinations of venous terms (e.g. "venous") with device terms (e.g. "argon"). The second group uses the same structure to match arterial thrombectomy reports. The third group, "ambiguous thrombectomy," captures reports with desired device names that were not previously matched by the first two groups; this illustrates that in grouped compound queries, groups are evaluated in the order user-specified, and each report is assigned only the first group it matches. The function returns a single pandas DataFrame with an added column labeling the search group to which each report was first matched. This query was used to benchmark the grouped search performance, reported in the last table in Appendix 3.

While custom boolean logic and the grouped search method enable a user to comprehensively specify their search criteria, any search of MAUDE is prone to both missed and off-target reports because of inconsistent field use and overlapping device names. Because a missed report cannot be recovered in later data processing, we encourage users to favor recall when constructing initial queries and to remove off-target results after review or via subsequent filtering steps. For example, in the October 21, 2026 database, a search for inferior vena cava (IVC) filters that required "vena cava" or "IVC" alongside "filter" using search_by_device_names(...) yielded 23,927 reports. A user might instead search by the IVC filter product code, "DTK," but, as previously mentioned, product codes are inconsistently applied and thus cannot be substituted for name searches [4]. Even so, the DTK reports provide a useful reference for what a too-restrictive name search misses: of 38,385 reports coded DTK, the name search captured 23,639 (61.6%) and missed 14,746 (38.4%). Broadening the search to "filter" alone missed only 376 (1.0%) of the DTK reports, but also returned 11,389 reports under other product codes, which likely contain relevant reports but would need to be reviewed to exclude unrelated devices. To maximize recall while still achieving a specific cohort, users should begin with a broad search term such as "filter" and then prune the result set. Tabulating the distinct brand names, generic names, and product codes returned identifies off-target devices for exclusion, which can then be removed with standard pandas operations on the returned DataFrame, for example by excluding rows whose DEVICE_NAME_CONCAT, BRAND_NAME, GENERIC_NAME, or DEVICE_REPORT_PRODUCT_CODE values match the identified off-target devices.

For queries not expressible through the two structured functions, query(...) accepts arbitrary SQL and executes it against the underlying DuckDB connection. This additionally provides query access to tables other than device.

Search results are returned as pandas DataFrames [17] and may be enriched with patient outcome fields, device problem codes, patient problem codes, and free-text narratives, then filtered by any of these. After enrichment, a result set might contain multiple rows for a given MDR_REPORT_KEY, accounting for multiple patients, problem codes, or narratives matching that report. Therefore, report-level analysis should take care to count unique MDR_REPORT_KEY values rather than rows in enriched results. Appendix 2 describes the enrichment and filtering functions in detail.

Table *2* lists each PyMAUDE function by purpose, and the table in Appendix 2 provides a complete function

reference.

Purpose	Functions	Description
Database construction	MaudeDatabase, add_years, update, info, close	Download source data; build, refresh, and inspect the local database
Archiving	archive, MaudeDatabase.from_archive, pymaude.verify_archive	Export, restore, and verify date-stamped full-database snapshots
Device search	query_device, search_by_device_names	Exact-match, substring, and compound substring device queries
Custom queries	query	Run arbitrary SQL against the local database
Enrichment	enrich_with_patient_data, enrich_with_device_problems, enrich_with_patient_problems, get_narratives	Attach patient outcome, patient- and device-problem codes, and narrative data
Filtering	filter_by_outcome, filter_by_patient, filter_by_device_problem, filter_by_patient_problem, filter_by_narrative	Restrict results by enriched fields
Summary	get_trends_by_year	Row counts by year

TABLE 2: Brief listing of PyMAUDE functions by purpose

PyMAUDE functions listed by category. Unless prefixed, functions are called on a database object created with MaudeDatabase(…), in the form db.function_name(…), where db is the MaudeDatabase object representing the local database. The table in Appendix 2 provides an in-depth function reference, including arguments and return type of each, while Figure 1 and Figure 2 provide usage examples with context. Search, enrichment, and filtering functions return pandas DataFrames. SQL: Structured Query Language.

Validation

Data retrieval was validated at three levels: 1) comparison of the set of MDR_REPORT_KEY values in the complete PyMAUDE database to those present in the openFDA device event bulk export; 2) comparison of query results returned from PyMAUDE to those returned from the openFDA API; and 3) internal comparison of substring and compound query results to those returned by equivalent SQL queries run against the locally constructed database, as openFDA provides no equivalent query for comparison.

To compare MDR coverage, the complete openFDA device event export [9] was downloaded on October 21, 2026. MDR report keys were extracted and compared against those in a PyMAUDE database built from source files retrieved on October 21, 2026. Of 26,558,073 unique report keys across both sources, 26,557,462 (99.998%) were present in both, 299 (<0.01%) were present only in PyMAUDE, and 312 (<0.01%) were present only in the openFDA export. All 299 records present only in PyMAUDE were confirmed to be absent from the openFDA bulk export raw JSON files, and additionally each was noted to not resolve to any report when queried against the openFDA search API. Furthermore, of these 299 PyMAUDE-only reports, 298 were related to toothbrushes of a single manufacturer reporter ("BRAUN GMBH WERK MARKTHEIDENFELD", ID: 3000302531) in years 2021-2023. The remaining one record present only in PyMAUDE (report key: 21641462) did not appear to be related to the other 298 PyMAUDE-only reports. In the other direction, all 312 records present only in the openFDA export were confirmed to be absent from the MDR Data Files for the master table, with 281 of these having records in child tables in the MDR Data Files without a parent record. Overall, all discrepancies in report coverage, in both directions, reflected true differences between the FDA's data distribution channels rather than errors in PyMAUDE's processing. The first table of Appendix 3 further specifies agreement by report year.

To validate query accuracy, five PyMAUDE queries were compared to queries via the openFDA API, with PyMAUDE queries using query_device(...) for exact field matching. Query cases included "product code alone", "product code with date restriction," "brand name alone," "generic name alone," and the "combination of both manufacturer name and product code". The second table in Appendix 3 reports specific queries tested and their results. Query results ranged in size from 774 to 129,149 reports. Returned sets were identical in two of five queries ("product code with date restriction" and "brand name alone"). The remaining three queries differed in result size by less than 1.5% in each case, with all differences explainable by case-sensitivity differences between PyMAUDE's and openFDA's text-field matching, incomplete product code and generic name fields in openFDA's own records for a small number of older reports, or true differences in the reports present in the MDR Data Files versus the openFDA records. The specifics of each query result difference are discussed in subsequent paragraphs below.

For the "generic name alone" query, PyMAUDE returned 11 additional reports (785 total) versus the openFDA API (774 total). All 11 PyMAUDE-only reports have their "GENERIC_NAME" field recorded in MAUDE in mixed case ("Biliary Stent") rather than the uppercase convention seen in most records; PyMAUDE's case-

insensitive matching discovers them during its query, while openFDA's case-sensitive exact-field match does not, even though all 11 records are present in both systems.

For "product code alone", PyMAUDE returned 129,149 reports and the openFDA API returned 129,148 reports; however, the PyMAUDE results included three reports not in the openFDA results, while the openFDA results included two reports not in the PyMAUDE results, leading to a net difference in count of only one report. The three PyMAUDE-only reports (report keys 672200, 678169, and 737040) were all reported in 2005 and appeared in openFDA's data with a placeholder ("---") product code and a blank generic name; all three appeared in the MDR Data Files with product code NIQ and non-placeholder generic names, and so were correctly included in PyMAUDE's results. Although the two openFDA-only reports (report keys 25854490 and 2067908) have records in other tables of the MDR Data Files, neither has a device record, so neither was returned by the PyMAUDE device-level query.

For the "combination of manufacturer name and product code", PyMAUDE returned 10,673 reports and the openFDA API returned 10,672; the single PyMAUDE-only report is report key 672200, the same record described above for the "product code alone" query, which was not discovered by openFDA API because of a placeholder product code.

openFDA provides no equivalent to PyMAUDE's search_by_device_names(...), which performs case-insensitive substring matching across device name fields, so this function was validated internally. Results for four search term sets were compared against direct DuckDB queries (while bypassing PyMAUDE); these direct queries applied the same conditions independently to brand name, generic name, and manufacturer name. Returned sets were identical in each of the four cases. The third table in Appendix 3 reports specific terms tested and their results.

Performance

All measures of performance were obtained on a 2020 MacBook Air (Apple M1, 8 GB RAM, macOS 14.5) running Python 3.12.3, DuckDB 1.5.2, and PyMAUDE 0.3.2, with data written to and read from the built-in solid-state drive. The build comprised all six tables for their available report years 1991 through 2026 as of October 21, 2026, totaling 70 source files, 26.6 million reports, and over 204 million rows.

Downloading and uncompressing the source files (7.76 GB compressed, 49.6 GB uncompressed, 1 GB = 10^9 bytes) completed in 6 minutes and 21 seconds (estimated as total build time minus median load-only time) on a home internet connection with the machine described above. Download time was not replicated, as it depends on network conditions rather than on PyMAUDE itself. Local construction of the database from previously downloaded source files took a median of 9 minutes 14 seconds (range 9 minutes 10 seconds to 9 minutes 19 seconds; n=5). 4,012 malformed records (3,802 from master and 210 from device) were repaired and included upon parsing source data. Five additional malformed records (three from master and two from patient) were unrepairable and therefore dropped. DuckDB's internal memory limit was set to 4.0 GB, and peak memory usage was 4.76 GB across all replicates; memory usage likely exceeded the set limit because it was measured as peak memory of the entire Python process, which included allocations outside of DuckDB's managed limit. Once built, the database occupied 57.7 GB on disk. Archiving the complete database completed in a median of 3 minutes 19 seconds (range 3 minutes 13 seconds to 3 minutes 36 seconds; n=5), used a peak memory of 2.96 GB under the same limit, and produced an archive of 13.7 GB (the Parquet exports, checksum manifest, and a tar bundle of the raw source files). For each replicate of both benchmarks, a new Python process was executed with a new DuckDB connection.

The query functions query_device(...) and search_by_device_names(...) were benchmarked against representative queries, each in five replicates (n=5), specified in the second and third tables in Appendix 3. Per-query timing results are reported in the last table of Appendix 3. Across the tested cases and their replicates, query_device(...) completed in 3.1 to 6.9 seconds and search_by_device_names(...) in 3.7 to 5.2 seconds. Additionally, a grouped compound search representative of a real cohort definition (Figure *3*) completed in a median of 28.0 seconds (range 27.7 to 36.6 seconds; n=5).

Discussion

PyMAUDE provides device safety researchers with a combination of query flexibility and reproducibility that we have not identified in existing tools. The FDA's routes of access to MAUDE, ranging from the online tool to the API, the flat file release, and now AEMS, achieve public availability but fall short of meeting the needs of investigators performing large-scale analyses across many report inclusion criteria, and additionally lack functionality to archive MAUDE for later replication of analysis. PyMAUDE fills both gaps by allowing users to define custom queries, either through the provided query functions or with further flexibility through direct SQL access, while also enabling a user to archive MAUDE in the state in which it was queried for later re-execution against the same data.

Validation showed close agreement of PyMAUDE with openFDA at both the API and bulk export levels. Report coverage matched the bulk export for 99.998% of the unique report keys present in either source, and exact-match queries returned results identical to or differing by less than 1.5% versus analogous API calls, with every difference traced to a specific difference in the source data. Additionally, a subset of query result differences between PyMAUDE and openFDA (the 11 PyMAUDE-only "generic name alone" results) indicates

that PyMAUDE can, in some cases, achieve superior report discovery via its case-insensitive search that openFDA API lacks.

It should, however, be noted that openFDA does not constitute an independent standard against which PyMAUDE can be validated, as both the openFDA dataset and PyMAUDE likely derive from the same primary source, the FDA flat files. Agreement between them indicates that two independently written pipelines arrive at the same records, which is meaningful evidence of correct data ingestion and querying but is not verification against an external, independent reference. The flat files are the FDA's native distribution of MAUDE, and we were not able to identify an independent representation of them against which either pipeline could be checked. To our knowledge, this is also the first systematic comparison of the MDR Data Files against openFDA's own bulk export and API, and it surfaces genuine discrepancies between FDA's distribution channels. These include the aforementioned 298-report single-manufacturer batch missing from openFDA's export, as well as the 312 reports not present in the MAUDE master table, which each may be of independent interest to researchers relying on either source. Furthermore, substring and compound queries, for which the API offers no equivalent, were instead compared against direct SQL queries applied to the same local database. Validation of these queries therefore indicated correctness of the query construction layer but not of data ingestion; however, this separately remains supported by the aforementioned two other validation methods.

Performance measures indicate that PyMAUDE operates at practical speeds on ordinary hardware. Benchmarks were run on a single machine, deliberately chosen as a consumer laptop with modest specifications rather than a research computing environment, to reflect resources available to most investigators. On that machine, downloading the source files and constructing the local database of over 26 million records completed in less than 20 minutes. Most queries completed in under 10 seconds, with the grouped search case completing in under 30 seconds. At these speeds, neither database construction nor execution time is likely to be an impediment to analysis.

Relative to existing tools, PyMAUDE delivers a distinct solution. MAUDEMetrics offers a no-code interface that automates interaction with the openFDA API, maximizing accessibility for investigators without programming experience, but in doing so forgoes the query complexity and pipeline integration made possible by a programming interface. MAUDE-Dash shares PyMAUDE’s use of the FDA flat files as source data for the creation of a local DuckDB database, but it presents the data via a dashboard that also lacks advanced programmatic query features. Neither MAUDEMetrics nor MAUDE-Dash produces a versioned copy of the data for reproducible analysis. To our knowledge, PyMAUDE is the first tool to pair programmatic access to the complete MAUDE database with checksummed, date-stamped, citable data snapshots.

PyMAUDE originated in a MAUDE analysis of venous thrombectomy devices conducted by several of the present authors (unpublished manuscript by Almoussa M, Schwartz JH, Ahmed O, Makary MS). That analysis used purpose-built code that was not released, against source files whose state at the time of retrieval cannot now be reconstructed. That study is therefore subject to the same reproducibility limitations described above. Creating a Python library and archivable data snapshots in place of that single-use code was motivated by the recognition that this situation is not unique to our group and that other researchers might benefit from a solution.

PyMAUDE does have several limitations that require consideration. First, PyMAUDE does not circumvent the intrinsic limitation of MAUDE that only reported events are included in the database, without any estimation of the relative number of unreported events or device use counts at baseline. Therefore, just as MAUDE data obtained via other methods cannot be used to estimate frequency of any given outcome or device failure, neither can data retrieved with PyMAUDE.

Using PyMAUDE requires writing and executing Python code, which may be a barrier for some users, though the requirement of computer programming is also imposed by working directly with the flat files or with the API. Additionally, maintaining updated local copies of MAUDE with PyMAUDE requires re-downloading each source file to compute its checksum because the FDA does not provide file-level checksums or other release metadata. This design avoids relying on change indicators supplied by the FDA, albeit at the cost of redundant downloading. Also, PyMAUDE’s built-in query functions do not cover every column in MAUDE, and are only built to query the device table. This was a deliberate design choice intended to minimize complexity of the codebase while supplying search functions users are most likely to need, all while preserving maximum customizability and ability to query the other tables via direct SQL access.

Lastly, PyMAUDE is released during the FDA’s ongoing transition of adverse event data into AEMS but does not currently support AEMS as a source. The FDA’s new focus on AEMS raises the possibility that the flat files might one day no longer be accessible or maintained by the FDA. If this were to become the case, PyMAUDE would provide the advantage of preserving over three and a half decades of medical device safety data via the Zenodo archive released with this manuscript. Additionally, because PyMAUDE separates the data-ingestion and database construction layers from the local query layer, support for AEMS-supplied data could be added in the future while preserving access to data previously sourced from MAUDE. This would allow analyses written against MAUDE-era data to be re-executed against AEMS data, and would permit both integration of and comparison between the two representations of adverse event data.

Conclusions

PyMAUDE is a Python library that enables medical device safety researchers to construct local DuckDB copies of MAUDE, to query those data with greater flexibility and power than existing methods allow, and to archive the underlying data of specific analyses so that it can be cited and re-executed by others. Validation against openFDA showed cross-pipeline consistency in both report coverage and query results, and benchmarks demonstrated acceptable runtimes on a consumer laptop. By combining programmatic access with citable data snapshots, PyMAUDE addresses two long-standing barriers to reproducible post-market device safety research.

Appendices

Appendix 1: Data sources and ingestion

PyMAUDE constructs its database from the flat text files published on the FDA MDR Data Files webpage [1]. These files vary in naming, partitioning, temporal coverage, and column structure, both across tables and across years within a single table. This appendix describes those variations and how PyMAUDE resolves them.

The source files follow two organizational conventions. The device and narrative text tables are distributed as yearly ZIP archives, each containing a single flat text file for one calendar year, except for their earliest years, which are bundled into one cumulative legacy file per table. The master, patient, device_problem, and patient_problem tables are each distributed as a single file covering all years through the prior calendar year, plus a separate file each for the current year, with minor variation in file naming patterns across tables. Table *3* describes each source file and its usage in PyMAUDE in more detail. add_years(...) resolves differing naming schemes automatically, so a PyMAUDE user need not know these subtle naming conventions when using the library for an arbitrary range of years.

PyMAUDE table	Description	FDA file name pattern	Distribution pattern	First available file year
master	Contains base records (1 MDR report key per event report)	mdrfoithru[prior year].txt, mdrfoi.txt	Cumulative	1991
device	Contains information about each device associated with an event report	1999 and earlier: foidevthru1997.txt, foidev1998.txt, foidev1999.txt; 2000 and later: DEVICE[year].txt, DEVICE.txt	Cumulative through 1997, Yearly since 1998	1991
text	Contains narrative descriptions associated with each event report	1995 and earlier: foitextthru1995.txt; 1996 and later: foitext[year].txt; current year: foitext.txt	Cumulative through 1995, Yearly since 1996	1991
patient	Contains patient information and outcomes associated with each event report	patientthru[prior year].txt, patient.txt	Cumulative	1991
device_problem	Contains device problem code(s) describing the identified device problems associated with each event report	foidevproblem_thru[prior year].txt, foidevproblem.txt	Cumulative	1993
patient_problem	Contains patient problem code(s) describing the identified patient problems associated with each event report	patientproblemcode_thru[prior year].txt, patientproblemcode.txt	Cumulative	1993

TABLE 3: MAUDE database source files and their handling in PyMAUDE

Summary of the uncompressed FDA MDR Data Files used as source files for PyMAUDE, including FDA-defined file-naming patterns, distribution pattern, and first available valid file year for each database table. Files are initially downloaded as ".zip" archives, each containing the ".txt" files of the same base name. File years are defined by dates that reports were received by the FDA, not the date of the events themselves that the reports describe. Where file name patterns contain brackets, these indicate placeholders for year, and the brackets are not in the corresponding file names; e.g. "mdrfoithru[prior year].txt" would indicate "mdrfoithru2025.txt" when the current year is 2026, and "DEVICE[year].txt" would indicate "DEVICE2020.txt" for the year 2020. "Cumulative" distribution pattern indicates that there is a single historical file containing all records of years prior to the current calendar year, up to the year before the current calendar year, plus a second file for the current calendar year. "Yearly" distribution pattern indicates that data are distributed in one file per calendar year for each year up to and including the current calendar year. device and text data changed distribution format over time; both source types were initially distributed cumulatively and later transitioned to yearly files (device in 1998 and text in 1996). device_problem and patient_problem tables contain coded descriptions of device- and patient-related problems, corresponding to MedWatch Form Sections F10 and H6 of each report. FDA: US Food and Drug Administration, MAUDE: Manufacturer and User Facility Device Experience, MDR: Medical Device Report.

While the earliest MAUDE data date back to 1991, temporal coverage (in terms of the corresponding DATE_RECEIVED field in master) differs by table, so PyMAUDE queries have table-specific date-range limits.

The master, patient, device, and text tables contain reports received from 1991 onward, although only a handful are present in 1991, with volume rising through the mid-1990s. The device_problem and patient_problem tables begin in 1993 and are also sparse in their early years. Note that master also contains a DATE_OF_EVENT field that can pre-date 1991, including likely placeholder values such as 1900 and 1901.

Additionally, the source files themselves contain certain inconsistencies even within single tables. Device source files before 2000 carry 45 columns, while source files from 2000-2012 use 31 columns, with the exception of the 2001 file, which has 34 columns, and source files from 2013 onward contain 34 columns as well. PyMAUDE detects and incorporates such schema differences automatically, retaining the union of columns (51 for device) encountered over each table's history and leaving columns null where that column did not exist for the year of an individual record.

Separately, a small fraction of records are split across two or more physical lines by a raw newline embedded within an unquoted text field (e.g. a multi-line address). These malformed lines were detected only in tables master and device but are rare (4,012 of ~53.2 million rows in these two tables; <0.01%). They are additionally heavily concentrated in recent years, with 3,970 of 4,012 (99.0%) occurring 2022-2026 (2026 partial). add_years(...) detects this pattern and reconstructs the original record by concatenating the affected lines' raw text before parsing, recovering all 4,012 affected records. Additionally, three further records in master as well as two records in patient each contained an extraneous delimiter; these lines were excluded from the final database because columns could not be resolved.

Appendix 2: Enrichment and filtering functions

Search results returned by query_device(...) and search_by_device_names(...) can be enriched with corresponding patient, device_problem, patient_problem, and narrative text data, and subsequently filtered by any of these.

Functions enrich_with_patient_data(...), enrich_with_device_problems(...), and enrich_with_patient_problems(...) attach patient outcome fields, device problem codes, and patient problem codes, respectively, to an existing result set, returning an updated DataFrame. Reports lacking enrichment data are retained, with null values in the added columns.

get_narratives(...) instead takes a set of report keys and returns the associated free-text narratives as a separate DataFrame. A single report may correspond to more than one narrative, for example in cases where multiple parties provided descriptions for an event, or because a narrative was subsequently revised (which is often entered as a new narrative to MAUDE). Therefore, the output of get_narratives(...) contains one row per narrative rather than one row per report.

Functions filter_by_outcome(...) and filter_by_patient(...) operate on a previously patient-enriched result set, filtering it by user-specified MAUDE patient outcome codes [1] or by a specified age range and/or sex, respectively. filter_by_device_problem(...) and filter_by_patient_problem(...) operate on device problem- and patient problem-enriched result sets, respectively, filtering by user-specified problem codes. filter_by_narrative(...) filters a result set by case-insensitive substring matching against corresponding narrative text entries; unlike the other filter functions, it queries the text table directly and requires no prior result enrichment.

A full function reference for PyMAUDE, including enrichment and filtering functions, is reported in Table *4*.

Function	Purpose	Key arguments	Returns
MaudeDatabase	Opens or creates a local DuckDB database	db_path, data_dir, memory_limit, verbose	MaudeDatabase object
add_years	Downloads and loads MAUDE source files for a range of years	years, tables, download, force_download, force_reload, force_partial	None
update	Re-downloads loaded source files and reloads those that have a changed checksum	download, force_download	None
info	Prints row counts and year coverage for each table in the local database	None	None (prints summary)
close	Closes the local database connection	None	None
archive	Exports the database as Parquet files with a checksum manifest and a tar bundle of the raw source files	output_dir, include_raw, compression_level, resume, overwrite	String, the path to manifest.json
MaudeDatabase.from_archive	Builds a DuckDB database locally from an archive	archive_dir, db_path, data_dir, verify,	MaudeDatabase object

		restore_raw, memory_limit	
pymaude.verify_archive	Computes checksums of files in an archive and checks if they match the checksum manifest in that archive	archive_dir	List, empty if verified or containing strings describing problems with verification
query_device	Exact case-insensitive search by one or more device fields, combined with AND	brand_name, generic_name, manufacturer_name, product_code, start_date, end_date	DataFrame, joined master and device records of query results
search_by_device_names	Case-insensitive substring search across device-name fields (brand, generic, and manufacturer names), with optional AND/OR logic or grouped searches	criteria, start_date, end_date, group_column	DataFrame, joined master and device records of query results
query	Executes arbitrary SQL against the local database	sql, params	DataFrame of query results
enrich_with_patient_data	Attaches patient outcome and demographic information to a search result DataFrame	results_df	DataFrame, result set with patient data columns added
enrich_with_device_problems	Attaches device problem codes to a search result DataFrame	results_df	DataFrame, result set with device problem columns added
enrich_with_patient_problems	Attaches patient problem codes to a search result DataFrame	results_df	DataFrame, result set with patient problem columns added
get_narratives	Retrieves report narrative text for a set of report keys	mdr_report_keys	DataFrame, narratives and their report key, one row per narrative
filter_by_outcome	Restricts a patient-enriched result set to specified outcome codes	results_df, outcome	Filtered DataFrame
filter_by_patient	Restricts a patient-enriched result set to a specified age range and/or sex	results_df, age_min, age_max, sex	Filtered DataFrame
filter_by_device_problem	Restricts a device problem-enriched result set to specified device problem codes	results_df, problem_code	Filtered DataFrame
filter_by_patient_problem	Restricts a patient problem-enriched result set to specified patient problem codes	results_df, problem_code	Filtered DataFrame
filter_by_narrative	Restricts a result set to reports that have narratives containing a specific term	results_df, term	Filtered DataFrame
get_trends_by_year	Counts input DataFrame rows by year and by search group, if in use	results_df	DataFrame of year and count

TABLE 4: PyMAUDE function reference

Reference for functions provided by PyMAUDE. Unless prefixed, functions are called on a database object created with MaudeDatabase(…), in the form db.function_name(…), where db is the MaudeDatabase object representing the local database. “Key arguments” lists the mandatory and most-likely used inputs by a user of PyMAUDE; documentation of defaults and optional parameters is provided in source code comments. “Returns” describes the function output, if any. All search, enrichment, and filtering functions return pandas DataFrames. Enrichment and filtering functions operate on DataFrames, for example after being returned by the search functions. Figure 1 and Figure 2 provide examples for use, as do Jupyter notebook examples in the source code. SQL: Structured Query Language. MAUDE: Manufacturer and User Facility Device Experience.

Appendix 3: Supplemental validation and performance tables

This appendix contains supplemental tables reporting validation and benchmark metrics. Table *5* reports report key coverage between PyMAUDE and the openFDA bulk export by year. Table *6* reports query agreement between PyMAUDE and equivalent openFDA API queries. Table *7* reports results of the internal validation of search_by_device_names(...) via direct SQL queries of the local database. Table *8* reports performance measures of query_device(...) and search_by_device_names(...).

Year received	In both	PyMAUDE only	openFDA only	Disagreement (%)
1991	15	0	0	-
1992	3,098	0	1	0.03
1993	4,409	0	1	0.02
1994	11,272	0	0	-
1995	9,758	0	0	-
1996	32,789	0	0	-
1997	77,691	0	0	-
1998	61,652	0	0	-
1999	52,909	0	0	-
2000	52,569	0	0	-
2001	58,391	0	0	-
2002	69,596	0	2	<0.01
2003	77,009	0	4	<0.01
2004	81,805	1	11	0.01
2005	98,939	0	12	0.01
2006	119,640	2	28	0.03
2007	171,340	2	33	0.02
2008	194,475	0	6	<0.01
2009	241,972	0	6	<0.01
2010	303,302	0	12	<0.01
2011	445,371	0	20	<0.01
2012	486,300	0	26	<0.01
2013	679,725	0	15	<0.01
2014	862,112	0	4	<0.01
2015	861,798	0	3	<0.01
2016	867,439	0	8	<0.01
2017	936,901	0	0	-
2018	1,048,080	0	2	<0.01
2019	1,328,954	0	1	<0.01
2020	1,564,999	0	0	-
2021	2,028,307	5	3	<0.01
2022	2,945,367	281	18	0.01
2023	2,339,790	12	16	<0.01
2024	2,627,111	0	40	<0.01
2025	2,887,986	1	17	<0.01
2026	2,924,586	0	28	<0.01

TABLE 5: External (cross-pipeline) report coverage agreement between PyMAUDE and openFDA, by year

Comparison of annual report coverage between PyMAUDE and openFDA bulk export using the unique MDR report key as the unit of comparison. PyMAUDE (v0.3.2) source data and openFDA bulk data were both downloaded on October 21, 2026. Reports are grouped according to the year received. “In both” counts the intersection of report keys identified in both PyMAUDE and openFDA; “PyMAUDE only” counts report keys identified in PyMAUDE but

not openFDA; “openFDA only” counts report keys identified in openFDA but not PyMAUDE. Five reports appear in both PyMAUDE and the openFDA export but with different received dates; each is therefore counted as "PyMAUDE only" in one year and "openFDA only" in another year, and column totals differ from counts reported in the main text by five. Percentage disagreement was calculated as: (PyMAUDE only+openFDA only)/In both×100%. Percentage disagreement values greater than 0% but less than 0.01% are reported as <0.01%. Agreement was assessed only by determining whether the same unique report key existed in both data sources, without assessment of individual data fields, narrative content, or other report contents. MDR: Medical Device Report, FDA: US Food and Drug Administration.

Query	Search criteria	In both	PyMAUDE only	openFDA only	Disagreement (%)
product code alone	Matches all reports with device product code NIQ, which corresponds to coronary drug-eluting stents	129,146	3	2	<0.01
product code + date range	Matches all reports with device product code NIQ, within the date range 01-01-2015 – 12-31-2023	53,027	0	0	-
brand name alone	Matches all reports with brand name RESOLUTE ONYX RX	5,839	0	0	-
generic name alone	Matches all reports with generic name BILIARY STENT	774	11	0	1.42
manufacturer + product code	Matches all reports with manufacturer name BOSTON SCIENTIFIC and device product code NIQ	10,672	1	0	<0.01

TABLE 6: External (cross-pipeline) query-level agreement between PyMAUDE and openFDA

Evaluation of agreement between PyMAUDE and openFDA for representative device queries. PyMAUDE (v0.3.2) queries were run against MAUDE data retrieved on October 21, 2026, and openFDA queries were run against the openFDA API on the same date. “Search criteria” describes the specific field value or combination of field values used to define each query, including product codes, date restrictions, exact device names, and manufacturer name. “In both” represents report keys returned by both systems, whereas “PyMAUDE only” and “openFDA only” represent report keys returned by only one of the two systems. Percentage disagreement was calculated as: (PyMAUDE only+openFDA only)/In both×100%. Percentage disagreement values greater than 0% but less than 0.01% are reported as <0.01%. Agreement was assessed only on the basis of the presence of the same unique report key in both result sets; the equality of individual data fields or complete report contents within matched records was not evaluated. MDR: Medical Device Report, FDA: US Food and Drug Administration.

Test case	Search criteria	In both	Identical (Y/N)
single term	“argon”	5,438	Y
OR of two terms	“argon” OR “cleaner”	8,408	Y
AND of two terms	“argon” AND “cleaner”	52	Y
AND-within-OR (compound)	(“argon” AND “cleaner”) OR “angiojet”	2,121	Y

TABLE 7: Internal validation of PyMAUDE substring and compound device-name searches

Internal validation of PyMAUDE substring compound queries with function search_by_device_names(…) against corresponding direct SQL queries against the local database. PyMAUDE (v0.3.2) queries were run against MAUDE data retrieved on October 21, 2026. Validation includes a single search term, Boolean OR logic, Boolean AND logic, and a case of combination AND/OR logic. search_by_device_names(…) performs queries against a concatenated field that combines generic, brand, and manufacturer names in the device table, enabling multi-device name field queries without concern for inconsistent use of these three fields by reporters. It matches substrings of the concatenated field rather than exact matching; combined with AND/OR logic, users can develop queries that capture alternative spellings, abbreviations, or other device naming variations in their search. Boolean OR logic retrieves records matching either specified term, whereas Boolean AND logic requires the specified terms to occur together according to the defined search criteria. Compound searches allow multiple Boolean conditions to be combined. “In both” represents the number of reports returned by both the PyMAUDE search and the corresponding direct SQL query. “Identical” indicates whether the two approaches produced the same result set for the specified test; Y and N denote yes and no, respectively. SQL: Structured Query Language.

Test case	Function	Result size	Median time (s)	Range (s)
product code alone	query_device	129,364	6.50	6.22-6.93
product code + date range	query_device	53,055	3.75	3.67-3.92
brand name alone	query_device	5,839	3.87	3.82-4.04
generic name alone	query_device	786	3.53	3.51-3.58
manufacturer + product code	query_device	10,731	3.14	3.11-3.26
single term	search_by_device_names	5,540	4.26	4.24-4.29
OR of two terms	search_by_device_names	8,513	5.20	5.19-5.21
AND of two terms	search_by_device_names	52	3.73	3.72-3.74
AND-within-OR compound	search_by_device_names	2,130	4.91	4.89-5.02
venous vs. arterial vs. ambiguous thrombectomy cohorts	search_by_device_names	6,088	27.97	27.71-36.56

TABLE 8: PyMAUDE query performance across representative search types

Summary of PyMAUDE's execution performance across representative exact-match, date-restricted, substring, and compound device searches. "Test case" identifies the type of search performed; "Function" identifies the PyMAUDE function used to execute the search; "Result Size" gives the number of device rows returned, and "Median Time" and "Range" report observed execution times in seconds across five replicates (n = 5). Exact searches based on fields such as product code, brand name, generic name, manufacturer name, and date were performed using query_device. Substring and compound device-name searches were performed using search_by_device_names, which searches across the concatenated device-name field and supports combinations of search terms. The venous-versus-arterial thrombectomy cohort search represents the grouped compound cohort-definition example illustrated in Figure 3, which was designed to demonstrate performance for a complex, clinically relevant cohort. Performance was measured on October 21, 2026 on an Apple Macbook M1 (2020) using PyMAUDE 0.3.2, DuckDB 1.5.2, Python 3.12.3, and source data which was retrieved on the same day.

Additional Information

Author Contributions

All authors have reviewed the final version to be published and agreed to be accountable for all aspects of the work.

Concept and design: Jacob H. Schwartz, Mina S. Makary, Maya M. Almoussa

Acquisition, analysis, or interpretation of data: Jacob H. Schwartz, Nicole Blattman

Drafting of the manuscript: Jacob H. Schwartz, Maya M. Almoussa, Nicole Blattman

Critical review of the manuscript for important intellectual content: Jacob H. Schwartz, Mina S. Makary, Maya M. Almoussa, Nicole Blattman

Supervision: Mina S. Makary

Disclosures

Human subjects: All authors have confirmed that this study did not involve human participants or tissue. **Animal subjects:** All authors have confirmed that this study did not involve animal subjects or tissue. **Conflicts of interest:** In compliance with the ICMJE uniform disclosure form, all authors declare the following: **Payment/services info:** All authors have declared that no financial support was received from any organization for the submitted work. **Financial relationships:** All authors have declared that they have no financial relationships at present or within the previous three years with any organizations that might have an interest in the submitted work. **Other relationships:** All authors have declared that there are no other relationships or activities that could appear to have influenced the submitted work.

Acknowledgements

The PyMAUDE source code is available at https://github.com/jhschwartz/PyMAUDE and on PyPI at https://pypi.org/project/pymaude under the MIT license. A complete archived copy of the MAUDE database constructed on October 21, 2026, including the Parquet exports, checksum manifest, and raw FDA source files, is available at https://doi.org/10.5281/zenodo.23537517. All underlying source data are published by the US Food and Drug Administration and are publicly available at https://www.fda.gov/medical-devices/medical-device-reporting-mdr-how-report-medical-device-problems/mdr-data-files. The scripts used to produce the validation and performance results reported here are included in the repository and archived release. Development of PyMAUDE was assisted by AI code generation (Claude, Anthropic, San

Francisco, CA, USA). All AI-generated code was reviewed and validated by the corresponding author prior to inclusion. Claude was additionally used to assist with editing portions of this manuscript; all text was reviewed and approved by the authors.

References

1. MDR data files. (2026). Accessed: October 12, 2026: https://archive.ph/OPYt6.
2. About Manufacturer and User Facility Device Experience (MAUDE) database . (2025). Accessed: October 12, 2026: https://archive.ph/rHpu1.
3. Device adverse event overview (openFDA). Accessed: October 12, 2026: https://archive.ph/ZgAA2.
4. Ensign LG, Cohen KB: A primer to the structure, content and linkage of the FDA's Manufacturer and User Facility Device Experience (MAUDE) files. EGEMS (Wash DC). 2017, 5:12. 10.5334/egems.221
5. Alemzadeh H, Raman J, Leveson N, Kalbarczyk Z, Iyer RK: Adverse events in robotic surgery: a retrospective study of 14 years of FDA data. PLoS One. 2016, 11:e0151470. 10.1371/journal.pone.0151470
6. Manufacturer and User Facility Device Experience (MAUDE) database . Accessed: October 12, 2026: https://archive.ph/7aFOB.
7. Li X, Feng Y, Gong Y, Chen Y: Assessing the reproducibility of research based on the Food and Drug Administration Manufacturer and User Facility Device Experience data. J Patient Saf. 2024, 20:e45-58. 10.1097/PTS.0000000000001220
8. Paging (openFDA). Accessed: October 12, 2026: https://archive.ph/34sfC.
9. Device adverse events - download the database (openFDA) . Accessed: October 12, 2026: https://archive.ph/8lOCV.
10. FDA launches new adverse event look-up tool . (2026). Accessed: October 12, 2026: https://archive.ph/cwsFH.
11. Marouf M, Hawas Y, Alkhawaldeh IM, Ghazou A, Awad A, Negida A: MAUDEmetrics: An open-source framework for automated FDA MAUDE data extraction and reproducible device safety research [Preprint]. Res Sq. 2026, 10.21203/rs.3.rs-9990811/v1
12. Porwal MH: Manufacturer and User Facility Device Experience (MAUDE)-Dash: an open-source, interactive dashboard for real-time post-market surveillance of medical devices. Surg Neurol Int. 2026, 17:289. 10.25259/SNI_201_2026
13. autonlab/fda_maude. (2023). Accessed: August 31, 2026: https://archive.softwareheritage.org/swh:1:snp:686b78c5cffc36f81e80378d208dbccbecdb6f98;origin=https://github.com/aut
14. tklovett/MaudeMiner. (2013). Accessed: August 31, 2026: https://archive.softwareheritage.org/swh:1:snp:df65c9d9c9c11a252e37515195eb25f1735121dd;origin=https://github.com/tkl
15. HazyResearch/icij-maude. (2019). Accessed: August 31, 2026: https://archive.softwareheritage.org/swh:1:snp:edaaf818e70483d30d43ae3401aa14408fc56cd3;origin=https://github.com/Ha
16. Raasveldt M, Mühleisen H: DuckDB: an embeddable analytical database. Proceedings of the 2019 International Conference on Management of Data. Association for Computing Machinery, New York; 2019. 1981-4. 10.1145/3299869.3320212
17. McKinney W: Data structures for statistical computing in Python. Proceedings of the 9th Python in Science Conference. Python in Science Conference (SciPy), Austin; 2010. 56-61. 10.25080/Majora-92bf1922-00a
18. Schwartz J: jhschwartz/PyMAUDE: 0.3.2 (Version v0.3.2). Zenodo. 2026, 10.5281/zenodo.23539288
19. Schwartz J: PyMAUDE complete archive October 21, 2026. Zenodo. 2026, 10.5281/zenodo.23537517