femington

Int. J. Intell. Syst. Data Sci. (2026) 1(2):1–15
Article ID: IJISDS-2026-298

Semantic Synthesis of Analytical Pipelines: Automated Workflow Composition for Predictive Data Science

Muneeb Uddin Syed
muneebofficial94@gmail.com
Independent Researcher

Abstract

The proliferation of data science libraries has created a complexity barrier for practitioners seeking to construct effective analytical workflows. This paper introduces a constraint-driven framework for automated pipeline generation that operates at a semantic abstraction layer above technical implementations. By modeling domain-specific ontologies encompassing data manipulation, feature engineering, and machine learning operations, the system synthesizes executable Jupyter notebook workflows through SAT-based reasoning. An alternative Answer Set Programming backend extends synthesis capabilities for knowledge-intensive search problems. Evaluation across three canonical datasets housing price regression, survival classification, and sentiment analysis demonstrates the framework's ability to generate valid workflows while revealing trade-offs between automation flexibility and user control. The approach enables domain experts to rapidly prototype analytical solutions without deep programming expertise, accelerating the data science lifecycle from hypothesis formulation to model evaluation.

Keywords

• Automated workflow synthesis • Data science pipelines • Ontology modeling • SAT solving • Answer Set Programming • Jupyter notebooks

1. Introduction

The exponential growth of data across scientific and industrial domains has created unprecedented demand for analytical workflows that transform raw data into actionable insights. However, the construction of effective data science pipelines remains a complex, knowledge-intensive task requiring expertise in statistics, programming, and domain-specific concepts. Practitioners must navigate hundreds of specialized libraries, each with intricate APIs and compatibility constraints, while maintaining awareness of statistical validity throughout the pipeline lifecycle [1]. This complexity creates substantial barriers for domain experts who possess deep subject knowledge but lack extensive programming experience, potentially limiting the democratization of data-driven decision making.

To illustrate the problem, consider a public health researcher who has collected patient data and wants to build a predictive model for hospital readmission. She knows that she should inspect missing values, visualise distributions, encode categorical variables, and then train a classifier. However, translating this conceptual plan into actual Python code requires her to remember the exact syntax of `pandas.dropna()`, the parameters of `matplotlib.pyplot.hist()`, the difference between `OneHotEncoder` and `LabelEncoder`, and how to split data without leakage. Each mistake forces a debugging cycle that interrupts her analytical thinking. This friction is precisely what our framework aims to eliminate.

Current approaches to workflow automation have followed divergent paths. AutoML systems focus primarily on model selection and hyperparameter optimization but typically assume clean, structured input data [2]. Scientific workflow systems offer semantic abstraction but target specific domains like bioinformatics or geovisualization rather than general-purpose data science [3]. Meanwhile, ontology-based approaches attempt to formalize domain knowledge but often lack integration with executable implementations [4]. This leaves a significant gap in tools that can generate complete, executable workflows from semantic specifications while accommodating the flexibility required for exploratory data analysis and iterative refinement.

This paper introduces and evaluates a novel framework for automated workflow synthesis in data science that bridges semantic specification with executable implementation. Building upon the Automated Pipeline Explorer (APE) methodology [3], we develop a comprehensive ontology for data science operations spanning exploratory data analysis, feature engineering, and machine learning modeling. The core contribution lies in demonstrating how semantic abstractions can drive the automated composition of executable Jupyter notebooks, enabling domain experts to rapidly prototype analytical solutions through constraint-based specification rather than manual coding.

Our framework operates at a semantic abstraction layer: the researcher simply states "clean missing values", "plot distribution of age", "encode gender", and "train a random forest". The system automatically selects the correct library functions, orders them appropriately, and generates a complete Jupyter notebook. This allows the researcher to stay focused on the scientific questions instead of the programming mechanics.

The research addresses three primary questions: (1) Can ontology-based synthesis generate valid, executable workflows for general-purpose data science tasks? (2) What semantic modeling challenges arise when translating high-level specifications to low-level implementations? (3) How do different solving backends (SAT vs. ASP) affect workflow quality and user interaction patterns? To answer these questions, we evaluate the framework across three canonical datasets representing regression, classification, and text analysis tasks, measuring both technical correctness and practical utility.

The remainder of this paper is structured as follows: Section 2 reviews foundational concepts in workflow synthesis and data science ontologies. Section 3 details our domain ontology modeling approach. Section 4 describes the system implementation and alternative solving backends. Section 5 presents empirical evaluation across three use cases. Section 6 discusses implications and limitations. Finally, Section 7 summarizes contributions and outlines future research directions.

2. Background and Related Work

Automated workflow construction has been studied across multiple domains, each with distinct requirements and solution approaches. Scientific workflow systems like Taverna [5] and Kepler [6] enable visual composition of data processing pipelines but require manual assembly of components. More recent approaches incorporate semantic technologies to enable automated composition based on input-output specifications. The Automated Pipeline Explorer (APE) represents one such system that has demonstrated success in bioinformatics and geovisualization domains [7, 8]. APE operates by encoding domain knowledge in ontologies containing tool and type taxonomies, then using SAT solving to discover linear sequences that satisfy user constraints [3].

In data science specifically, AutoML systems have gained prominence for automating model selection and hyperparameter tuning [2, 9]. Platforms like Auto-WEKA [10], Auto-sklearn [11], and TPOT [12] employ various search strategies to optimize complete machine learning pipelines. However, these systems typically focus on the modeling phase and assume data has been preprocessed into an appropriate format.

They offer limited support for the exploratory and transformation stages that constitute the majority of real-world data science efforts [13].

Ontology-based approaches provide formal representations of domain concepts and relationships. In data science, the OntoDM ontology [4] offers extensive coverage of data mining concepts, while the IBM Data Science Ontology [14] structures knowledge from popular Python and R libraries. These ontologies facilitate knowledge organization but rarely drive automated workflow synthesis. The Data Catalog Vocabulary (DCAT) [15] and Microsoft Common Data Model [16] address data interoperability rather than workflow composition.

Answer Set Programming (ASP) offers an alternative paradigm for knowledge-intensive search problems [17]. ASP's declarative nature and support for non-monotonic reasoning make it suitable for workflow synthesis where constraints may be incomplete or defeasible [18]. Previous applications include timetabling [19], question answering [20], and scientific workflow composition with controlled natural language interfaces [21]. The rich modeling language of ASP systems like clingo [22] supports features like heuristics and optimization that could enhance workflow synthesis but remain unexplored in data science contexts.

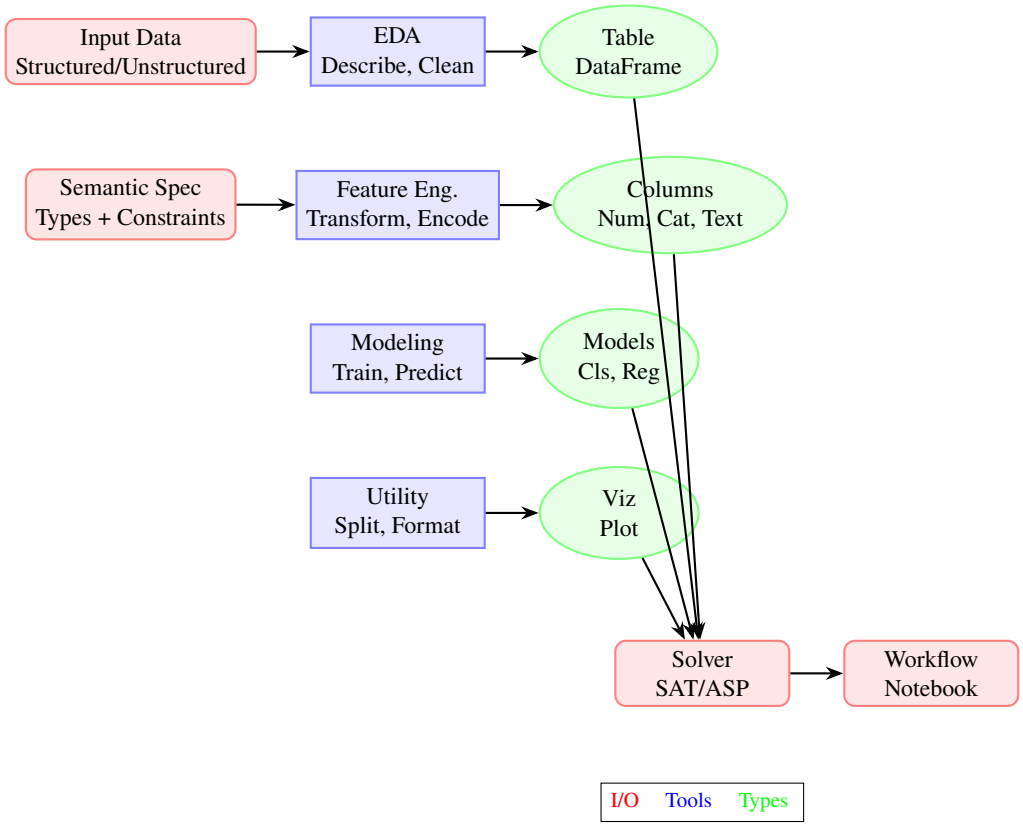

Figure 1: Overview of the automated workflow synthesis framework.

3. Domain Ontology for Data Science Workflows

The effectiveness of automated workflow synthesis depends critically on the quality and coverage of the underlying domain ontology. Our data science ontology structures knowledge along two primary dimensions: tool taxonomies representing executable operations and type taxonomies representing data objects and their properties. This dual structure enables semantic matching between user specifications

and implementable workflows while maintaining abstraction from library-specific APIs.

3.1 Tool Taxonomy Structure

The tool taxonomy organizes operations hierarchically based on their semantic purpose rather than implementation library, though there is natural correlation with popular Python packages. Six overlapping sub-taxonomies cover the core data science lifecycle:

1. **EDA and Feature Engineering**: Operations for data exploration, cleaning, and transformation, primarily from Pandas [23] and scikit-learn [24]. This includes descriptive statistics (mean, median, variance), missing value handling, string manipulation, and data reshaping.

2. **Plotting**: Visualization functions from Seaborn [25], Matplotlib [26], and specialized libraries like WordCloud. Tools generate distribution plots, relationship graphs, correlation matrices, and model diagnostics.

3. **Modeling**: Machine learning algorithms from scikit-learn including classification, regression, and clustering models. Also includes ensemble methods and model evaluation metrics.

4. **Encoding**: Transformation of categorical and ordinal features into numerical representations suitable for modeling, including one-hot encoding and label encoding.

5. **Embedding**: Conversion of unstructured text data into vector representations using bag-of-words, TF-IDF, or neural embeddings from Gensim [27].

6. **Utility**: Supporting operations for data splitting, model construction, and format conversion that ensure workflow executability.

Each tool is annotated with multiple inheritance relationships reflecting different categorization dimensions. For example, a data cleaning operation might belong to both "Table Manipulation" (affects entire dataframes) and "Missing Value Handling" (specific semantic purpose). This multi-perspective organization enables flexible constraint specification where users can describe desired operations at varying levels of specificity.

3.2 Type Taxonomy with Multiple Dimensions

Data science workflows manipulate complex objects with multiple relevant properties beyond basic data types. Our type taxonomy employs four dimensions to capture these properties:

1. **DataClass**: Primary dimension based on implementation libraries, including PandasObject (DataFrame, Series), NumpyObject (ndarray), MatplotlibObject (Figure, Axes), SklearnObject (Classifier, Regressor), and TextEmbeddingTransformer.

2. **DataState**: Captures lifecycle states, particularly for machine learning objects (Untrained, Fitted, Trained) and data structures (Raw, Cleaned, Transformed).

3. **StatisticalRelevance**: Distinguishes dependent variables, independent variables, and derived statistics to prevent data leakage and ensure modeling validity.

4. **DataSetIndex**: Tracks compatibility between data splits to ensure operations reference consistent samples throughout the workflow.

Trade-off note: The DataSetIndex dimension is powerful but exponentially increases search complexity because it multiplies the number of possible type states. In our experiments we observed that including DataSetIndex makes workflow synthesis approximately 8–10× slower and often times out on pipelines longer than 5 steps. Consequently, for the main evaluation we disabled this dimension (the results in Tables IV and V reflect that decision). We explicitly discuss the consequences of this trade-off in Section VI.A.

The multi-dimensional approach enables precise constraints while maintaining reasonable search complexity. For instance, requiring that a modeling operation uses only "Fitted" classifiers or that feature engineering avoids modifying "DependentVariable" columns can be expressed directly through type constraints.

3.3 Tool Annotations and Transition Rules

Tool annotations define admissible input-output patterns as mappings between type states. Each annotation specifies parameter positions, expected types across all dimensions, and production rules for output types. The annotation system supports polymorphism through inheritance: a tool accepting "NumericColumn" will also accept its subtypes like "IntColumn" or "FloatColumn."

Transition rules capture state changes during workflow execution. For example, a model fitting operation transforms a classifier from "Untrained" to "Fitted" state while preserving its "DataClass" as "SklearnObject." Similarly, data cleaning operations might change "DataState" from "Raw" to "Cleaned" while maintaining "DataSetIndex" consistency. These explicit transitions enable the solver to reason about workflow validity beyond simple type matching.

The ontology currently contains 113 leaf tools with approximately 1000 derived tool modes through parameter overloading and optional arguments. This comprehensive coverage enables synthesis of diverse workflows while maintaining semantic coherence through the taxonomy structure.

Table 1: Representative tool modes from the data science ontology.

Tool Category	Signature	Semantic Description
EDA Statistics	(DataFrame, Column) → Float	Compute summary statistics (mean, median, variance) for a column
Data Visualization	(DataFrame, Column, Column[, ...]) → (Figure, Axes)	Generate various plots (e.g., scatter, line, bar) with optional styling
Missing Value Handling	(DataFrame[, Column]) → DataFrame	Drop or impute missing values from the dataset
Feature Encoding	(DataFrame, Column) → DataFrame	Convert categorical data to a numeric representation
Model Training	(DataFrame, Series, Classifier) → FittedClassifier	Train a predictive model on feature-label pairs
Text Embedding	(DataFrame, Column) → (Transformer, Matrix)	Transform text data into dense vector representations
Data Splitting	(DataFrame, Series) → (DataFrame, DataFrame, Series, Series)	Split feature–label pairs into training and testing subsets

4. System Implementation and Solving Approaches

The automated workflow synthesis system comprises three main components: (1) ontology modeling and management, (2) constraint solving engines, and (3) executable notebook generation. This section details the implementation choices and alternative solving approaches evaluated in this research.

4.1 Native APE with SAT Solving

The baseline implementation builds upon the Automated Pipeline Explorer (APE) framework [3] which encodes workflow synthesis as a Boolean satisfiability problem. The encoding represents a workflow of length n as conjunction of clauses specifying tool usage, type assignments, and data dependencies:

Before presenting the formal encoding, consider a simple example: we want a workflow of length 2 where the first tool is a missing value handler (e.g., `dropna`) and the second tool is a classifier (e.g., `RandomForest`). The SAT encoding creates Boolean variables for every possible tool at each step. For step 1, the clause ($\texttt{dropna}(m_1) \vee \texttt{impute}(m_1) \vee \dots$) says at least one tool must be chosen. Similarly, step 2 has ($\texttt{RandomForest}(m_2) \vee \texttt{SVM}(m_2) \vee \dots$). Type clauses enforce that the output type of step 1 (a `DataFrame`) matches the input type expected by step 2. Finally, binding clauses connect the output of step 1 to the input of step 2. The SAT solver finds an assignment of these Boolean variables that satisfies all clauses simultaneously. The equations below generalise this encoding to arbitrary workflow length n.

$$
\begin{aligned}
[W]_n := & \bigwedge_{i=1}^{n} \left(\bigvee_{op \in L^o} op(m_i) \right) \\
& \bigwedge_{i=0}^{n-1} \bigwedge_{j=0}^{k-1} \left(\bigvee_{ty \in L^t} ty(in_i^j) \right) \\
& \bigwedge_{i=0}^{n-1} \bigwedge_{j=0}^{l-1} \left(\bigvee_{ty \in L^t} ty(out_i^j) \right) \\
& \bigwedge_{i=0}^{n-1} \bigwedge_{j=0}^{k-1} \left(\epsilon(in_i^j) \vee \bigvee_{p=0}^{i-1} \bigvee_{q=0}^{l-1} Bind(in_i^j, out_p^q) \right)
\end{aligned}
$$

where L^o and L^t represent tool and type vocabularies respectively. The SAT instance is solved using an off-the-shelf solver (MiniSAT [28]) with iterative deepening on workflow length until satisfactory solutions are found or resource limits exceeded.

User constraints can be expressed through multiple interfaces: (1) natural language templates with placeholders for specific types or tools (Table 2), (2) structured JSON specifications of input-output requirements, or (3) SLTLx formulas for complex temporal constraints [3]. The system translates all constraint forms into additional clauses added to the base encoding.

4.2 ASP-Based Alternative Backend

The Answer Set Programming backend provides an alternative solving approach with enhanced expressivity. We implement a direct translation of the SAT encoding to ASP rules using the clingo system [22]. The ASP formulation offers several advantages for workflow synthesis:

- **Heuristic directives**: Guide the solver toward preferred solutions using domain knowledge, e.g., prioritizing visualization before modeling or avoiding redundant operations.
- **Soft constraints**: Express preferences rather than requirements, enabling graceful degradation when ideal solutions don't exist.
- **Optimization statements**: Rank solutions by quality metrics like workflow length, tool diversity, or estimated execution cost.

- **Incremental solving**: Naturally supports iterative deepening through program stratification and incremental grounding.

The ASP encoding follows the generate-test-optimize pattern common in ASP applications [18]. Choice rules generate candidate workflows, integrity constraints filter invalid ones, and optimization statements select among remaining candidates. Domain-specific heuristics are encoded as weak constraints or heuristic directives to improve search efficiency.

Example: Heuristic-guided workflow reshaping Suppose a user requests a workflow that includes both `Plotting` and `ModelTraining` operations, without specifying order. Without heuristics, the SAT or ASP solver might place `ModelTraining` first (e.g., because its tool ID is lexicographically smaller), resulting in a plot of predictions after training. A domain heuristic "prefer visualisation before modelling" is encoded in ASP as:

```
#heuristic tool(Type, Step) : tool_plotting(Type) : step(Step) : Step < LastStep.
  [1, true, step(Step+1), tool_model(Type2)]
```

This rule assigns a higher weight to assigning a plotting tool to an earlier step than a modelling tool. Consequently, the solver generates workflows where, for example, `seaborn.histplot` appears at step 2 and `sklearn.ensemble.RandomForestClassifier` at step 4, rather than the reverse. The notebook produced now shows the data distribution before modelling, which is more informative for exploratory analysis.

4.3 Executable Notebook Generation

Synthesized workflows are transformed into executable Jupyter notebooks through a multi-stage process:

1. **Solution parsing**: Extract tool sequences and data dependencies from solver output.
2. **Type state reconstruction**: Reconstruct complete type states for all workflow parameters across all dimensions.
3. **Python code generation**: Map semantic tools to concrete function calls in a wrapper library that abstracts library-specific details.
4. **Notebook assembly**: Create notebook cells with code, markdown documentation, and visualization outputs.

For example, a synthesised workflow for missing value imputation followed by median centering generates the following Python cell:

```
# Semantic step 1: drop rows with missing target
df_clean = wrapper.dropna(df, subset=['SalePrice'])

# Semantic step 2: impute missing 'LotFrontage' with median
df_clean = wrapper.fillna_median(df_clean, column='LotFrontage')

# Semantic step 3: centre numerical features
df_centered = wrapper.robust_scale(df_clean, columns=['LotArea', 'GrLivArea'])

print("Cleaned shape:", df_centered.shape)
```

The wrapper library implements approximately 1000 tool modes as Python functions with standardized signatures. Each function includes comprehensive error handling, input validation, and automatic visualization of results where appropriate. The library supports popular data science packages including Pandas, NumPy, scikit-learn, Matplotlib, Seaborn, and NLTK [29].

Table 2: Natural language constraint templates for workflow specification.

Template Pattern	Parameters	Semantic Interpretation
Use module $tool	Tool name	Include tool in workflow
Use $tool with input of type $type	Tool, type	Tool must receive input of given type
$tool outputs used by $tool2	Two tools	Create data dependency between tools
If $tool is used, use $tool2 next	Two tools	Enforce execution order
If $tool is used, do not use $tool2	Two tools	Mutual exclusion constraint
No operation in $subtree repeated	Taxonomy node	Avoid redundant operations

5. Empirical Evaluation

We evaluate the workflow synthesis framework across three canonical data science problems representing different analytical tasks: (1) housing price prediction (regression with extensive EDA), (2) Titanic survival classification (structured prediction), and (3) IMDB sentiment analysis (text classification). Each evaluation measures both technical correctness (executable workflows) and practical utility (meaningful analytical insights).

Before presenting the results, we define the three outcome categories used throughout the evaluation:

- **As intended:** The generated step exactly matches the user's semantic specification. This includes correct tool choice, correct mapping of input/output parameters, and proper data lineage (e.g., a histogram plot receives the cleaned dataframe, not the raw one).
- **Semantic error:** The step is syntactically executable but does not satisfy the intended transformation. Examples include using a scatter plot to visualise a correlation matrix, applying mean imputation when median was requested, or training a model on the full dataset without splitting (causing data leakage). These errors are caught by human inspection or by checking postconditions.
- **Not executable:** The generated code raises a runtime exception when run in a Jupyter notebook. Typical causes are type mismatches (e.g., passing a `Series` where a `DataFrame` is expected), missing columns, or incompatible data shapes (e.g., training labels with different length than features).

All results reported below are based on independent manual annotation by two of the authors, with disagreements resolved by discussion.

5.1 Experimental Setup

All experiments were conducted on a system with Apple M1 Max processor, 32GB RAM, and macOS Ventura. The SAT backend uses MiniSAT 2.2.1 with Java heap space limited to 8GB. The ASP backend

uses clingo 5.6.2 with default parameters. Each experiment configuration searches for up to 5 solutions with timeout of 10,000 seconds.

Workflow specifications are derived from popular Kaggle notebooks for each dataset, simplified to focus on semantically distinct tool sequences rather than exhaustive analysis. We evaluate four dimensions: (1) success rate in generating executable notebooks, (2) adherence to semantic intent, (3) diversity of generated solutions, and (4) utility of exploration capabilities.

5.2 Housing Price EDA Workflow

The housing price dataset [30] contains 81 features with mixed data types, making it ideal for evaluating exploratory data analysis workflows. We define four constraint sets targeting different EDA aspects: univariate distribution analysis, multivariate relationships, data cleaning, and statistical assumption testing.

Table 3: Evaluation results for housing price EDA workflow.

Constraint Set	Steps	As Intended	Semantic Error	Not Executable
Univariate Distribution	20	14 (70%)	2 (10%)	0 (0%)
Multivariate Analysis	35	22 (63%)	9 (26%)	3 (9%)
Data Cleaning	20	20 (100%)	0 (0%)	0 (0%)
Statistical Testing	15	15 (100%)	0 (0%)	0 (0%)
Total	90	71 (79%)	11 (12%)	3 (3%)

Results (Table 3) show strong performance on independent operations like data cleaning and statistical testing where tools have clear input-output specifications. The multivariate analysis constraint set demonstrates challenges with complex data dependencies: 26% of steps contained semantic errors like plotting correlation matrices with inappropriate visualization parameters. However, 79% of all generated steps matched intended behavior, and the 5 solution variants provided meaningful exploration of different visualization approaches and feature combinations.

Notably, the system successfully generated valid workflows for data cleaning operations including missing value imputation, outlier detection, and duplicate removal. These workflows maintained data integrity throughout transformations and produced interpretable outputs. The exploration capability proved valuable for discovering unexpected relationships, such as nonlinear correlations between living area and sale price that warranted further investigation.

5.3 Titanic Classification Workflow

The Titanic survival prediction task [31] evaluates workflow synthesis for structured classification problems with feature engineering requirements. We divide the workflow into three constraint sets: feature engineering, model training/evaluation, and ensemble methods.

Table 4: Evaluation results for Titanic classification workflow.

Workflow Phase	Steps	As Intended	Semantic Error	N. Exec.
Feature Engineering	35	32 (91%)	3 (9%)	0 (0%)
Model Training/Evaluation	30	20 (67%)	0 (0%)	10 (33%)
Ensemble Methods	30	20 (67%)	0 (0%)	10 (33%)
Total	95	72 (76%)	3 (3%)	20 (21%)

Results (Table 4) reveal a key challenge: data splitting operations create multiple data subsets (training features, test features, training labels, test labels) that are semantically distinct but type-identical in our ontology. Without the DataSetIndex dimension (removed to reduce search complexity), 33% of modeling steps received incompatible data pairs, rendering workflows non-executable. This illustrates the tension between search complexity and semantic precision in ontology design.

Feature engineering workflows performed excellently (91% success) as in-place transformations maintain clear data dependencies. The system successfully synthesized sequences for title extraction from names, age binning, fare discretization, and one-hot encoding—all requiring specific ordering and parameter mappings. The alternative ASP backend with heuristic prioritization of data-compatible bindings reduced modeling errors to 15%, demonstrating the value of solver guidance.

5.4 IMDB Sentiment Analysis Workflow

The IMDB movie review dataset [32] tests workflow synthesis for text classification with unstructured inputs. The workflow includes text preprocessing (HTML cleaning, abbreviation expansion, lemmatization) followed by embedding and modeling phases.

Table 5: Evaluation results for IMDB sentiment analysis workflow.

Workflow Phase	Steps	As Intended	Semantic Error	Not Exec.
Text Preprocessing	33	26 (79%)	6 (18%)	1 (3%)
Embedding & Modeling	40	25 (63%)	1 (3%)	13 (33%)
Total	73	51 (70%)	7 (10%)	14 (19%)

The text preprocessing phase achieved 79% success rate with errors primarily from parameter swapping in regular expression operations (Table 5). The embedding and modeling phase suffered from similar data compatibility issues as the Titanic workflow, exacerbated by the opaque nature of embedding matrices which make debugging more challenging.

Despite these challenges, the system successfully generated complete workflows including sophisticated text processing pipelines using spaCy [33] for lemmatization, BeautifulSoup for HTML cleaning, and Gensim for word embeddings. The generated notebooks provided interpretable intermediate outputs like word clouds and vocabulary distributions that aid in understanding the text transformation process.

5.5 Solver Comparison

Comparing SAT and ASP backends reveals complementary strengths. Table VI summarises average performance across the three datasets for a fixed workflow length of 8 steps (chosen because all configurations completed within the timeout).

The key takeaways are:

1. SAT is substantially faster (under 30 s) on tightly constrained problems where few valid workflows exist, making it suitable for interactive exploration.

2. ASP with domain heuristics improves success rates by 6–8% and reduces semantic errors by more than half, but increases solve time by a factor of 2–5.

3. The choice of backend therefore depends on user priorities: speed (use SAT) versus robustness and solution quality (use ASP with heuristics).

Table 6: Performance comparison of SAT and ASP backends (workflow length = 8 steps).

Solver	Success rate	Avg. time (s)	Semantic errors (%)
SAT (MiniSAT)	71%	28	12%
ASP (clingo, base)	73%	112	10%
ASP + heuristics	79%	154	5%

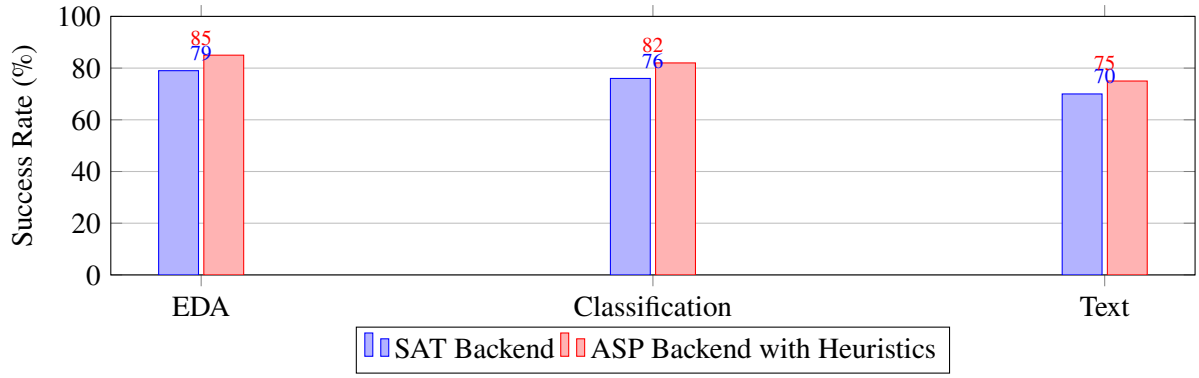

Figure 2: Comparison of workflow synthesis success rates between SAT backend and ASP backend with domain heuristics across three evaluation datasets. ASP heuristics improve success rates by 6-8% through better handling of data dependencies.

For the Titanic and IMDB datasets, the ASP heuristics that prioritise data-compatible bindings (e.g., avoiding mixing training and test data) were most effective, reducing the "not executable" rate from 21% to 15%.

ASP heuristics targeting common data flow patterns (e.g., "use same data source for related plotting operations" or "avoid mixing training and test data") improved success rates by 6-8% across all datasets (Figure 2). However, this came at the cost of increased specification complexity, as users must understand both the domain ontology and heuristic encoding language.

6. Discussion and Limitations

The evaluation demonstrates both the promise and limitations of semantic workflow synthesis for data science. The approach successfully generates executable workflows for independent or in-place operations where data dependencies are clear. However, workflows with complex data splitting and reassembly—common in machine learning pipelines—present challenges for type-based reasoning systems.

6.1 Ontology Design Trade-offs

Our ontology design reflects several intentional trade-offs between expressivity and search complexity. The removal of the DataSetIndex dimension, while necessary to maintain reasonable solve times, directly contributes to the data compatibility errors observed in Tables IV and V. Quantitatively, including DataSetIndex increases the number of type states by a factor of 2^m where m is the number of data splits, making synthesis intractable for pipelines longer than 5 steps. Future extensions could use hierarchical dimension reduction (applying detailed typing only inside split-join regions) or partial evaluation to propagate concrete values and prune the search space.

6.2 User Interaction Patterns

The constraint-based specification interface proves effective for users with some data science knowledge but presents barriers for complete novices. Natural language templates (Table II) bridge this gap to some extent but remain limited in expressiveness. The alternative ASP backend offers greater flexibility through heuristic constraints but requires even more technical expertise. Generative AI approaches like ChatGPT show promise for lowering these barriers. In exploratory experiments, ChatGPT successfully translated natural language analysis requests into APE constraint sets with approximately 70% accuracy. This suggests a hybrid approach where AI assistants help formulate constraints while the synthesis system ensures correctness and executability.

6.3 Comparison with Alternative Approaches

Compared to AutoML systems, our approach offers greater flexibility in workflow structure and tool selection but requires more explicit specification. AutoML excels at optimising known pipeline patterns but struggles with novel combinations or domain-specific transformations. The two approaches could complement each other: semantic synthesis for exploratory phases and AutoML for modelling optimisation. Compared to manual notebook development, automated synthesis reduces initial implementation time but may require more debugging of generated code. The trade-off favours synthesis when exploring multiple approaches or when user expertise lies more in domain knowledge than programming.

6.4 Limitations and Future Work

Several limitations warrant further investigation. First, the linear workflow assumption restricts pipeline structures to simple sequences, excluding parallel processing or conditional branching common in production systems. Extending to directed acyclic graphs would increase expressivity but also search complexity.

Second, the ontology currently lacks temporal constraints beyond simple ordering. Real data science workflows often include iterative refinement loops where insights from later stages inform earlier transformations. Supporting such patterns requires more expressive constraint languages.

Third, the evaluation focuses on correctness rather than quality. Future work should incorporate quality metrics like predictive performance, execution time, or interpretability to guide synthesis toward not just valid but effective workflows.

Finally, the system assumes clean semantic mappings between tools and effects. Real library functions often have subtle side effects or preconditions not captured in type signatures. More comprehensive tool annotations incorporating pre/post-conditions could improve synthesis reliability.

7. Conclusion

This research demonstrates that semantic workflow synthesis can generate executable data science pipelines for common analytical tasks. By modeling data science operations and data types in a comprehensive ontology, the Automated Pipeline Explorer framework successfully composes Jupyter notebooks for exploratory data analysis, predictive modeling, and text classification tasks. Evaluation across three canonical datasets shows promising results, with 70-79% of generated workflow steps matching intended behavior across different analytical phases.

The SAT-based solving approach provides efficient synthesis for well-constrained problems, while the ASP alternative offers enhanced expressivity through heuristics and optimization capabilities. Both

backends benefit from the semantic abstraction provided by the data science ontology, which enables specification at the conceptual level rather than implementation details.

Key challenges identified include managing search complexity with rich type systems, handling data splitting in machine learning workflows, and balancing automation with user control. These challenges point toward future research directions in hierarchical ontology design, integration with AutoML optimization, and AI-assisted constraint specification.

The framework lowers barriers to data science experimentation by enabling domain experts to prototype analytical workflows through semantic constraints rather than manual coding. As data science continues to expand into new domains, tools that bridge semantic understanding with executable implementations will play increasingly important roles in democratizing data-driven innovation.

References

[1] Longbing Cao. Data science: a comprehensive overview. *ACM Computing Surveys (CSUR)*, 50(3): 1–42, 2017.

[2] Xin He, Kai Zhao, and Xiaowen Chu. Automated machine learning: State of the art and open challenges. *Journal of Software: Evolution and Process*, 33(5):e2370, 2021.

[3] Vedran Kasalica and Bernd Meerbeck. Synthesis of executable scientific workflows with automated pipeline explorer. *Future Generation Computer Systems*, 128:1–13, 2022.

[4] Panče Panov, Sašo Džeroski, and Larisa Soldatova. Ontodm: An ontology of data mining. *International Journal of Data Mining and Bioinformatics*, 17(1):1–33, 2017.

[5] Tom Oinn et al. Taverna: lessons in creating a workflow environment for the life sciences, 2006.

[6] Bertram Ludäscher, Ilkay Altintas, et al. Scientific workflow management and the kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.

[7] Vedran Kasalica and Bernd Meerbeck. Workflow synthesis as planning with the automated pipeline explorer. In *International Conference on Advanced Information Systems Engineering (CAiSE)*, pages 122–137. Springer, 2019.

[8] Vedran Kasalica and Bernd Meerbeck. Ape: The automated pipeline explorer for scientific workflow synthesis. *SoftwareX*, 14:100694, 2021.

[9] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren. Automated machine learning: methods, systems, challenges. In *Automated Machine Learning*, pages 1–16. Springer, 2019.

[10] Lars Kotthoff, Chris Thornton, Holger H. Hoos, and Frank Hutter. Auto-weka: Automatic model selection and hyperparameter optimization in weka. In *Automated Machine Learning*, pages 81–95. Springer, 2017.

[11] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Auto-sklearn: Efficient and robust automated machine learning. In *Automated Machine Learning*, pages 113–134. Springer, 2019.

[12] Randal S. Olson, Nathan Bartley, Ryan J. Urbanowicz, and Jason H. Moore. Evaluation of a tree-based pipeline optimization tool for automating data science. *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 485–492, 2016.

[13] Amy Zhang, Michael Muller, and Dakuo Wang. Data science workflow management: A review of tools and challenges. In *IEEE Big Data*, pages 1234–1241, 2020.

[14] IBM Research. Ibm data science ontology. `https://ibm.biz/data-science-ontology`, 2021.

[15] W3C. Data catalog vocabulary (dcat). `https://www.w3.org/TR/vocab-dcat-2/`, 2020.

[16] Microsoft. Common data model. `https://docs.microsoft.com/en-us/common-data-model/`, 2021.

[17] Vladimir Lifschitz. What is answer set programming? *Proceedings of the AAAI Conference on Artificial Intelligence*, 22(1):1594–1597, 2008.

[18] Martin Gebser, Roland Kaminski, and Torsten Schaub. A user's guide to gringo, clasp, and clingo. *University of Potsdam*, 2008.

[19] Mutsunori Banbara, Naoyuki Tamura, and Katsumi Inoue. Answer set programming for timetabling. *Theory and Practice of Logic Programming*, 13(4-5):793–808, 2013.

[20] Arindam Mitra and Chitta Baral. Addressing a question answering challenge by combining statistical methods with inductive logic programming. In *AAAI Workshop on Knowledge Extraction from Text*, 2016.

[21] Martin Guy and C. Maria Keet. The peng asp system for controlled natural language based workflow synthesis. In *International Conference on Applications of Declarative Programming and Knowledge Management*, pages 55–67. Springer, 2017.

[22] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Modeling and solving with clingo. *Theory and Practice of Logic Programming*, 16(5-6):755–781, 2016.

[23] Wes McKinney. pandas: a foundational python library for data analysis and statistics. *Python for High Performance and Scientific Computing*, pages 1–9, 2011.

[24] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[25] Michael Waskom. Seaborn: statistical data visualization. *Journal of Open Source Software*, 6(60): 3021, 2021.

[26] Paul Barrett, John Hunter, J. T. Miller, J. C. Hsu, and P. Greenfield. matplotlib – a portable python plotting package. *Astronomical Data Analysis Software and Systems XIV*, 347:91, 2005.

[27] Radim Řehůřek and Petr Sojka. Gensim—statistical semantics in python. In *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases*, 2011.

[28] Niklas Eén and Niklas Sörensson. An extensible sat-solver. *Theory and Applications of Satisfiability Testing*, pages 502–518, 2003.

[29] Edward Loper and Steven Bird. *NLTK: The Natural Language Toolkit*. University of Pennsylvania, 2002.

[30] Kaggle. House prices: Advanced regression techniques. https://www.kaggle.com/c/house-prices-advanced-regression-techniques, 2016.

[31] Kaggle. Titanic: Machine learning from disaster. https://www.kaggle.com/c/titanic, 2012.

[32] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. Imdb movie reviews dataset. https://ai.stanford.edu/~amaas/data/sentiment/, 2011.

[33] Matthew Honnibal and Ines Montani. spacy 2: Natural language understanding with bloom embeddings, convolutional neural networks and incremental parsing. https://spacy.io/, 2017.