Agentic Partial Evaluation on Pandas Programs

Xavier Adettu
University of Wisconsin - Milwaukee
Milwaukee, WI 53211
xadettu@uwm.edu

Tian Zhao
University of Wisconsin - Milwaukee
Milwaukee, WI 53211
tzhao@uwm.edu

***Abstract*—Pandas is essential for Python data processing, but real-world pipelines often suffer from Python-level overhead due to row-wise callbacks, small control loops, and repetitive assignments. While manual refactoring can help, it requires expertise and risks errors. We introduce a policy-guided system that optimizes Pandas programs via source-to-source transformation while preserving behavior. The system combines program and schema analysis with a policy agent that returns conservative decisions. Deterministic rewrites and online partial evaluation (PE) then specialize the program using recovered static information. Transformed programs are accepted only after verification: both original and transformed programs execute on identical inputs and outputs must match exactly. Failures revert to the original code. We evaluate five policy conditions and compare against direct LLM code generation. Results show that our policy agent provides verified and program-controlled speedups and it is more adaptable than rule-based optimization. While direct code generation from LLM can have good performance, it is unreliable across repeated trials.**

***Index Terms*—Pandas, Software Engineering Tools, Program Transformation, Partial Evaluation, Data Processing**

I. Introduction

Pandas is a cornerstone of Python data preparation, offering an expressive DataFrame abstraction [1], [2]. However, many real-world pipelines suffer performance bottlenecks from Python-level patterns such as row-wise `DataFrame.apply` callbacks, compact control loops updating derived columns, and recomputed schema-dependent expressions. While these patterns aid rapid development, they often dominate end-to-end preprocessing costs.

Automating Pandas optimization is challenging. High-impact transformations are localized, but manually rewriting row-wise logic into vectorized form while preserving semantics such as missing-value behavior requires expertise [3], [4]. Python's dynamic typing and library calls limit static compiler techniques [5], [6]. Engine-level solutions (Dask [7], Modin [8], Weld [9], Polars [10]) impose adoption costs and may miss Python-level anti-patterns like UDFs (user-defined functions) in `apply`. Accelerated loading [11] and heterogeneous execution [12] address orthogonal bottlenecks. Direct LLM code generation can hallucinate incorrect rewrites, mishandle edge cases, or drop columns. Consequently, most practitioners rely on time-consuming, error-prone manual refactoring.

This paper presents a policy-guided specialization approach that optimizes existing Pandas scripts via source-to-source transformation while preserving behavior. Our system integrates lightweight program and schema analysis with a policy agent that returns conservative decisions. It applies deterministic rewrites followed by online partial evaluation (PE) [13], [14], specializing programs using automatically recovered static information. To ensure correctness, both the original and the transformed programs execute on identical inputs so that optimizations are accepted only when output DataFrames match exactly. Verification failures safely revert to the original code.

```python
import pandas as pd
df = pd.read_csv(path)

if "C" in df.columns:
    df["flag"] = df.apply(lambda r:(r["ratio"] > 0.7) and\
                                   (r["C"] in {"x", "y"})\
                          ,axis=1)
for t in (0.1, 0.2, 0.3):
    df[f"gt_{t}"] = df["ratio"] > t
```

Fig. 1. Motivating example: schema check,row-wise callback and small loop.

```python
import pandas as pd
df = pd.read_csv(path)

df["flag"] = (df["ratio"] > 0.7) & \
             (df["C"].isin({"x", "y"}))

df["gt_0.1"] = df["ratio"] > 0.1
df["gt_0.2"] = df["ratio"] > 0.2
df["gt_0.3"] = df["ratio"] > 0.3
```

Fig. 2. Residual program after specialization: the schema check is discharged, `apply` vectorized, and loop unrolled into straight-line assignments.

Figure 1 shows a representative feature-engineering pattern with a schema dependent guard, a row-wise callback and small loop that dominate runtime on large datasets. Figures 2 show the residual programs our system produces. This example reveals three recurring optimization opportunities. First, row-wise `apply` calls can often be rewritten as columnar expressions (boolean masks and `isin`) when schema details permit. Second, static constants (thresholds and column names) and schema properties enable specialization that eliminates redundant checks. Third, short loops over fixed constants can be unrolled to reduce interpretation overhead.

In this paper, we make the following contributions. (1) An end-to-end pipeline that analyzes Pandas programs, selects high-impact rewrite units, applies policy-guided deterministic rewrites, and emits specialized residual code. (2) A policy agent that returns only conservative decisions (bounded unrolling budgets and unit-level enable/disable actions). (3) An

online PE engine that specializes Pandas code using recovered static information and produces readable residual code with safe fallback. (4) Empirical evaluation with an ablation study.

II. System Overview

In this section, we present an end-to-end policy-guided specialization pipeline. The input is a Python program that constructs Pandas DataFrames and produces a final output DataFrame. A LLM serves as a policy agent, returning conservative decisions without generating code. All rewriting and specialization are performed by deterministic transformation passes and an online partial evaluator. The system produces an optionally rewritten source program, a specialized residual program, and structured reports capturing analysis results, policy decisions, verification outcomes, and timing.

A. Inputs, Outputs, and Artifacts

The primary input is a Python program using Pandas, with a configured output variable (or user-provided selector) identifying which DataFrame to compare during verification. The system generates three output artifacts. An optional rewritten program (`rewritten.py`) results from policy-enabled deterministic rewrite rules. A specialized residual program (`residual.py`) is emitted by the partial evaluator. Structured reports (`policy.json`, `snippets_report.json`, `report.json`) capture extracted features, selected rewrite units, policy decisions, verification results, and timing data, supporting reproducibility and debugging.

B. Pipeline Stages

Figure 3 summarizes the main stages. Analysis first parses the input program into an AST and extracts coarse features (counts of `apply` operations, loops, conditionals, group-by/merge) to guide rewrite selection and inform policy decisions. When the program reads from a discoverable source (e.g., `read_csv`), the system samples a bounded number of rows to infer a schema slice: column names, data types, and nullability. This slice serves as advisory context for policy decisions and enables schema-static treatment during specialization. The system sends a compact program summary, schema slice (if available), and a bounded set of rewrite units to the policy agent. The agent returns a policy file specifying safe transformations: whether bounded loop unrolling is allowed, the maximum unroll budget, and unit-level simplifications. Invalid responses fall back to conservative defaults.

Rewrite-unit selection identifies high-leverage localized regions matching known costly patterns: row-wise callbacks and short loops that update DataFrame columns. Each unit is annotated with referenced/defined columns and required helper definitions, then scored and capped to bound analysis cost. Guided by the validated policy, the program optimizer applies deterministic rewrite rules (e.g., vectorizing row-wise patterns) to selected units, producing an optional rewritten source program with no LLM-generated code.

The partial evaluator then specializes the program using static constants and schema facts, performing constant propagation and bounded loop unrolling when allowed, while residualizing dynamic control flow. The result is a standard Pandas program. The candidate residual program is executed and its output DataFrame is compared to the baseline using a comparator like `df.equals`. Failed candidate is rejected. Finally, the system benchmarks the verified candidate and returns the candidate if it is faster than the original program.

III. Rewrite-Unit Selection

To maintain locality and auditability [3], our system identifies a small set of high-leverage rewrite units: localized code regions matching known high-cost Pandas patterns [2], [4]. These units serve two purposes: they provide structured context for the policy agent's conservative decisions and they define source spans where deterministic rewrite rules may be applied when enabled.

A. Definition

A rewrite unit is a contiguous source region (a single AST node or small group of adjacent nodes) containing a pattern that incurs high Python-level overhead. Each unit has a precise source span, a kind (e.g., row-wise apply or loop-generated columns), referenced and defined DataFrame columns, and required definitions (helper functions or constants needed for interpretation). The selector enforces two constraints. First, transformations stay within the unit boundary, ensuring locality. Second, all helper definitions referenced by the unit remain available and unchanged.

B. Target Patterns

The selector prioritizes several high-cost patterns from feature-engineering pipelines [1], [2]. Row-wise callbacks include `DataFrame.apply(axis=1)`, `df.apply` with lambdas using `axis=1`, and `Series.apply` with UDFs or lambdas. These execute Python code per row and often dominate runtime [3], [4]. Short-loops that produce derived-columns are *for*-loops over small literal lists, tuples, or ranges that assign to DataFrame columns. Even with vectorized bodies, repeated Python dispatch and dynamic column-name construction are costly. Repeated derived-column chains are sequences of DataFrame column assignments referencing the same intermediates, becoming candidates for simplification when they contain static subterms.

C. Extraction and Metadata

AST traversal extracts units deterministically. For each candidate, the selector derives metadata: defined/referenced columns (from subscript expressions like `df["col"]` and attribute access) and required definitions for locally referenced functions or constants. Each unit gets a compact signature: kind, source span, defined/referenced columns, and a normalized surface form.

D. Scoring and Budgeting

To bound analysis cost, the system caps selected units via `max_units`. Each unit receives a priority score based on heuristics: pattern weight (row-wise `apply` highest), estimated dynamic cost (units inside loops favored), column

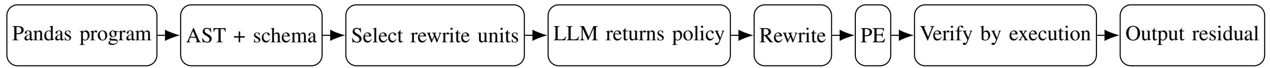

Fig. 3. Policy-guided specialization pipeline. LLM returns decisions only; Optimizer performs rewriting and PE.

TABLE I
POLICY JSON SCHEMA: FIELDS, TYPES, VALID VALUES, AND DEFAULTS.

Field	Type	Valid Values	Default
`strategy.allow_unroll`	bool	`true`, `false`	`false`
`strategy.max_unroll`	int	1 – 20	5
`unit_actions.<id>`	string	`true`, `false`	`false`
`notes`	string	any	`""`

impact (columns used later), and feasibility (resolvable references). The top-ranked units proceed downstream.

IV. POLICY AGENT

This section describes how our system employs a language model such as Gemini as a policy agent [3], [15]. The language model never proposes code edits, patches, or rewritten snippets. Instead, it returns conservative policy decisions that control which deterministic transformations are allowed and how aggressive they may be.

A. Policy Contract and Inputs

Given a set of rewrite units, program-level features from AST analysis, and a schema slice when available, the policy agent returns a JSON object describing the optimization strategy. Global strategy includes configuration values such as whether unrolling is allowed and the maximum unroll budget. Unit-level actions provide per-unit decisions enabling or disabling specific deterministic rewrite rules on that unit. An optional backend recommendation offers conservative suggestions such as pandas execution, which are recorded but do not affect correctness.

The policy agent receives structured inputs comprising a program summary (coarse-grained AST features), rewrite units (snippet text, kind, source span, referenced/defined columns), an optional schema slice, and a safety rules checklist emphasizing conservative decisions and prohibiting code generation.

B. Structured Output Format

To support deterministic parsing and logging, the policy agent returns JSON with specific fields. The strategy field contains global settings including unrolling allowance and maximum unroll budget. The unit_actions field maps unit identifiers to actions. The notes field offers brief justification for decisions. Table I enumerates selected policy fields.

C. Validation and Application

Policy outputs undergo validation before influencing program optimization. JSON validation ensures required keys and types are present, with defaults applied otherwise. Action validation confirms that unit actions match known unit identifiers and allowed actions. Strategy validation verifies that numeric parameters such as max_unroll fall within safe bounds. After validation, the program optimizer applies deterministic rewrite rules enabled by the policy and then runs PE. Correctness does not depend on the policy agent, as the final acceptance gate remains whole-program verification.

V. PARTIAL EVALUATION AND SPECIALIZATION

After policy-guided rewriting, the system runs a partial evaluator to further simplify the program. PE specializes a program with respect to information known before full dataset processing, avoiding overhead from recomputed constants, repeated string construction, or small fixed loops.

A. Code Representation

The tool parses Python source into an AST using the standard parser, providing a structured view of assignments, control flow, and function calls for identifying Pandas patterns and applying deterministic rewrites.

To simplify later passes, we normalize surface forms into canonical patterns. Column reads/writes treat `df["col"]` and `df["col"] = expr` as explicit operations. Calls and method calls become uniform call nodes. Control flow (`if`, `for`) remains explicit, letting the specializer decide whether to execute early or residualize.

The system focuses on a subset of Python (assignments, expressions, `if`, `for`, function defs/calls) sufficient for typical Pandas programs. Unsupported constructs are left unchanged.

B. Binding-Time Classification

We employ a simple binding-time view [14], [16] distinguishing *static* values (known during specialization) from *dynamic* values (depending on runtime data).

Static values include literals, constant lists/tuples, and derived constants (e.g., formatted column names). The optional schema slice provides advisory metadata for safe transformations. Dynamic values encompass DataFrame/Series contents, data-dependent boolean masks, and control flow conditions. Binding-time propagates conservatively: any dynamic subexpression makes the whole expression dynamic.

C. PE Strategy

We implement an online, interpreter-style partial evaluator [14]. During a single AST traversal, the specializer maintains an environment mapping static variables to values and emits residual code for runtime computations [13].

The specialization follows three rules. First, static expressions are evaluated immediately via constant folding and propagation. Second, for-loops over static iterables are unrolled when the iteration count $\leq$ `max_unroll`; otherwise emitted unchanged. Third, dynamic constructs are residualized without early execution.

This approach is practical for Pandas because many performance issues stem from repeated constant work and small fixed loops around vectorized operations. Specialization always terminates, where data-dependent code is not executed, unrolling is bounded, and unsupported constructs are left unchanged.

D. Core Transformations

The partial evaluator performs several safe simplifications. Constant propagation computes static expressions immediately, eliminating repeated work. Bounded loop unrolling expands short fixed loops into straight-line code within the policy budget. Function specialization creates specialized versions when functions are called with static arguments, substituting constants into function bodies. Residualization emits unchanged Python for dynamic or unsupported constructs, serving as the primary safety mechanism.

VI. Verification and Safe Fallback

All optimizations in our system are treated as candidates. We accept a transformed program only when it passes verification checks, returning the original program unchanged when any check fails. This rule constitutes the primary safety mechanism that guarantees safe optimization and prevents silent behavior changes.

A. Equivalence Check

For each candidate transformed program, the system executes both the original baseline program and the candidate program, then compares the configured output DataFrame. A candidate is considered verified only when the output variable exists in both runs, the output DataFrame schemas match with respect to column names and order and data types when reliably comparable, and the output values match under a DataFrame comparator such as df.equals. When any condition fails, the candidate is rejected.

B. Correctness Properties of Deterministic Rewrites

The deterministic rewrite rules and partial evaluator provide correctness properties that are independent of the verification layer. Each rewrite rule is applied only when a syntactic precondition is satisfied that guarantees behavioral equivalence for the targeted pattern. For example, the vectorization rewrite of `df.apply(lambda r: ..., axis=1)` in Figure 1 is triggered only when the lambda body contains no side effects, no non-local variable writes, and no operations outside a safe subset. Under these conditions, the replacement using a boolean mask and `isin` in Figure 2 is semantically equivalent for all input DataFrames that satisfy the column preconditions checked during schema sampling.

The partial evaluator provides a weaker but still meaningful guarantee: because it never executes data-dependent code during specialization and only unfolds statically-bounded loops [13], the residual program is a syntactic specialization of the original that cannot introduce new data-dependent behavior. Verification therefore remains the acceptance gate. The risk of accepting an incorrect transformation is bounded by the coverage of the verification dataset.

TABLE II
Benchmark descriptions and dataset statistics. Baseline times are mean ± standard deviation in seconds under cached I/O.

Benchmark	Rows	Cols	Apply Ops	Baseline (s)
`synth`	500,000	8	7	1.0684 ± 0.0045
`nyc_taxi`	1,000,000	18	1	1.2949 ± 0.0097
`homecredit`	307,511	120+	2	21.2291 ± 0.3462

VII. Evaluation

The prototype is implemented in Python as a set of deterministic components coordinated by an end-to-end driver.

A. Benchmarks

We evaluate the performance of three Pandas preprocessing pipelines. Table II summarizes their key characteristics and average runtime. Experiments were conducted on an Azure instance running Linux (Ubuntu 24.04), which was equipped with 40 vCPUs and 320 GB RAM

The `synth` benchmark is a synthetic feature-engineering workload designed to stress Python-level overhead, containing numerous element-wise transformations and repeated `Series.apply` calls. The `nyc_taxi` benchmark implements a feature construction pipeline on NYC Taxi trip data [17], containing a single apply-style UDF. The `homecredit` benchmark is a large real-world feature engineering workflow from the Home Credit Default Risk dataset [18], involving multiple CSV reads, joins, merges, and `groupby` aggregations.

B. Evaluation and Ablation Study

To assess whether the LLM as policy agent design is justified, we evaluate five optimization policies in Table III.

1) **Baseline**: No rewrite or loop-unrolling. Only applies constant folding and dead code elimination.
2) **Partial**: Baseline plus limited form of rewriting such as vectorizing UDFs with a simple *if* expression and limited loop unrolling (5 at most).
3) **PE+rewrite**: all rewrite rules, max loop-unrolling (up to 20), and vectorizing all UDFs that fits heuristic patterns.
4) **LLM+PE**: LLM provides policy advice for PE including unrolling but no rewrites.
5) **LLM+PE+rewrite**: the full system. LLM provides policy advice for both PE and rewrites.

Table IV separately reports the performance of direct LLM-generated code, which often fails verification checks.

a) LLM-guided optimization.: The results in Table III show that LLM+PE+rewrite has the best performance on two of the three benchmarks. PE+rewrite also provided speedups on all benchmarks though it does not perform as well on the `synth` benchmark. The reason is that `synth` includes a UDF with a rarely taken but expensive branch:

TABLE III
EVALUATION AND ABLATION: TIMES ARE MEAN ± STANDARD DEVIATION IN SECONDS UNDER CACHED I/O AND ALL RESIDUALS ARE VERIFIED.

Benchmark	Policy	Residual (s)	Speedup
synth	Baseline	1.1957 ± 0.0051	0.894×
synth	Partial	1.2043 ± 0.0063	0.887×
synth	PE+rewrite	0.6439 ± 0.0129	1.659×
synth	LLM+PE	1.2019 ± 0.0061	0.889×
synth	LLM+PE+rewrite	0.3935 ± 0.0049	**2.715**×
nyc_taxi	Baseline	1.3155 ± 0.0108	0.984×
nyc_taxi	Partial	1.3288 ± 0.0074	0.974×
nyc_taxi	PE+rewrite	0.8623 ± 0.0039	**1.502**×
nyc_taxi	LLM+PE	1.3236 ± 0.0103	0.978×
nyc_taxi	LLM+PE+rewrite	0.8826 ± 0.0071	1.467×
homecredit	Baseline	20.6739 ± 0.1074	1.027×
homecredit	Partial	18.0266 ± 0.0659	1.178×
homecredit	PE+rewrite	17.5720 ± 0.0615	1.208×
homecredit	LLM+PE	20.7054 ± 0.0836	1.025×
homecredit	LLM+PE+rewrite	17.1809 ± 0.0866	**1.236**×

```python
def rare_heavy_branch(v):
    if v < -999999.0:
        return (
            v * 1.0001 + v * 1.0002 +
            v * 1.0003 + v * 1.0004 +
            # many similar arithmetic terms
            v * 1.0600
        )
    return 0.0

df["rare_heavy"] = df["x"].apply(rare_heavy_branch)
```

The aggressive PE+rewrite policy enables every supported rewrite and therefore rewrites this unit into a vectorized `np.where` expression:

```python
df["rare_heavy"] = np.where(
    df["x"] < -999999.0,
    df["x"] * 1.0001 + df["x"] * 1.0002 +
    df["x"] * 1.0003 + df["x"] * 1.0004 +
    # many similar vectorized terms
    df["x"] * 1.0600,
    0)
```

Although this rewrite is semantically valid, it is performance unprofitable on this benchmark because the expensive vectorized branch is evaluated over the full column. In contrast, LLM+PE+rewrite keeps this unit unchanged while still vectorizing the profitable UDFs:

```python
df["rare_heavy"] = df["x"].apply(rare_heavy_branch)
```

b) Direct LLM: The low pass-rates in Table IV show that the direct LLM generation is not reliably semantics-preserving. The directly generated code can have good performance when it passes verification. For example, `synth` optimized through Gemini has an algebraic simplification outside our current deterministic rewrite set. Specifically, it collapses the long `rare_heavy_branch` arithmetic expression into `df["x"] * 618.03`. Our optimizer performs constant propagation, but it does not perform this kind of algebraic factoring over dynamic expressions. Our approach restricts LLM to policy selection and keeps residual code generation deterministic.

TABLE IV
DIRECT LLM CODE GENERATION. TIMES ARE MEAN ± STANDARD DEVIATION IN SECONDS. THE *verified* COLUMN SHOWS THE PASS/ATTEMPT COUNTS. ONLY VERIFIED CODE IS EVALUATED.

Benchmark	Tool	Verified	Residual (s)	Speedup
synth2	Gemini	2/5	0.225 ± 0.002	4.744×
synth2	Claude	0/5	invalid	invalid
synth2	ChatGPT	3/5	0.336 ± 0.025	3.178×
nyc_taxi	Gemini	1/5	0.73 ± 0	1.782×
nyc_taxi	Claude	1/5	0.94 ± 0	1.380×
nyc_taxi	ChatGPT	1/5	0.69 ± 0	1.872×
homecredit	Gemini	3/5	16.76 ± 1.16	1.271×
homecredit	Claude	5/5	17.43 ± 0.21	1.218×
homecredit	ChatGPT	3/5	16.67 ± 0.35	1.274×

C. Threats to Validity

Performance variability due system load may remain despite repeated runs. The verification mechanism using `df.equals` may reject acceptable numerical differences. Schema sampling may produce incomplete metadata, leading to conservative policy decisions.

Our benchmark suite includes only three pipelines, which do not capture the full diversity of real-world Pandas usage. The policy agent's decisions may not generalize across different language models.

End-to-end runtime speedup conflates computational improvements with I/O behavior. Our I/O caching focuses on computation, representing an upper bound on achievable improvements.

The policy agent's decisions depend on an external language model API, which may evolve over time, affecting the baseline and the optimized performance.

VIII. Related Work

Pandas documentation emphasizes vectorized operations over row-wise callbacks and loops [1], [2]. Empirical comparisons show performance differences often stem from execution model choices [4].

Several systems accelerate DataFrame workloads by modifying the execution engine. Dask and Modin parallelize Pandas programs across cores [7], [8]. Weld optimizes data-parallel operators via a common IR [9]. Arrow focuses on columnar formats for interoperability [19], [20]. Polars and DataFusion offer alternative designs that can outperform Pandas on some workloads [10], [21]. Our work complements these by keeping programs in ordinary Python while reducing Python-level overhead on baseline Pandas.

FORTE [15] targets query plan optimization rather than source transformation. ConnectorX [11] and Maximus [12] address data loading and heterogeneous execution, orthogonal to our focus.

JIT compilation and tracing can accelerate numeric kernels but struggle with full Pandas programs due to dynamic typing

and side effects [5], [6], [22]. Our work targets a different layer: source-to-source simplification that reduces Python overhead while producing standard Pandas code.

Dias [3] is the most related work, which rewrites code patterns such as `series.apply()`, `series.fillna()`, and `df.sort_values().head(n)` in Pandas programs. We ran Dias on `synth`, `nyc_taxi`, and `homecredit` using the same cached-I/O harness. The resulting speedups were modest: 1.124× on `synth`, 1.008× on `nyc_taxi`, and 1.023× on `homecredit`. Inspection of the generated Dias residuals shows that the key apply-style sites remain unchanged. The `synth` benchmark still contains named-function calls and `homecredit` still contains lambda/map and apply patterns inside function bodies.

PE separates early computations from runtime execution, producing faster residual programs [13]. Tempo studies controlled specialization in real languages [23], [24]. Online PE specializes during interpretation using available static values [14]. Hybrid approaches combine static and online reasoning [16]. Multi-stage programming systems generate residual code while preserving readability [25], [26], [27]. Our work adapts these techniques to Pandas, combining a lightweight binding-time view with Pandas-aware rewrites and verification.

IX. Conclusion

We have presented a policy-guided optimization system for Pandas programs that combines deterministic rewrites, PE, and verification. The system employs a language model solely as a policy agent that returns conservative decisions. The optimizer applies local rewrite rules and then specializes programs by folding constants, unrolling small fixed loops, and residualizing dynamic parts. All transformed programs are accepted only when verification confirms output DataFrame equivalence. An ablation study shows that PE and rewrites are the primary performance driver, while policy control prevents regressions. The core architecture generalizes beyond Pandas. The AST analysis, binding-time analysis, PE engine, and verification harness are library-agnostic; extending the system to NumPy, Polars, or Spark would require adding new pattern matchers and rewrite rules. Future work will explore broader coverage of rewrite patterns, stronger equivalence checks, and larger benchmark suites.

The source code of this paper is available at https://github.com/uwm-se/pe-agent.

References

[1] W. McKinney, "Data structures for statistical computing in python," in *Proceedings of the 9th Python in Science Conference (SciPy)*, 2010, pp. 56–61.

[2] Pandas Developers, "Pandas user guide: Enhancing performance," https://pandas.pydata.org/docs/user_guide/enhancingperf.html, 2023, accessed: 2025-09-11.

[3] S. Baziotis, D. Kang, and C. Mendis, "Dias: Dynamic rewriting of pandas code," *Proceedings of the ACM on Management of Data (SIGMOD)*, vol. 2, no. 1, pp. 1–27, 2024, published February 2024.

[4] A. Mozzillo, L. Zecchini, L. Gagliardelli, A. Aslam, S. Bergamaschi, and G. Simonini, "Evaluation of dataframe libraries for data preparation on a single machine," arXiv:2312.11122, 2023.

[5] S. K. Lam, A. Pitrou, and S. Seibert, "Numba: A llvm-based python jit compiler," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, 2015.

[6] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo, "Tracing the meta-level: Pypy's tracing jit compiler," in *Proceedings of the 4th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, 2009, pp. 18–25.

[7] M. Rocklin, "Dask: Parallel computation with blocked algorithms and task scheduling," in *Proceedings of the 14th Python in Science Conference (SciPy)*, 2015, pp. 130–136.

[8] D. Petersohn, S. Macke, D. Xin, W. Ma, D. Lee, X. Mo, J. E. Gonzalez, J. M. Hellerstein, A. D. Joseph, and A. Parameswaran, "Towards scalable dataframe systems," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 2033–2046, 2020.

[9] S. Palkar, J. Thomas, V. Shanbhag, H. Pirk, M. Schwarzkopf, S. Amarasinghe, and M. Zaharia, "Weld: A common runtime for high performance data analytics," in *CIDR*, 2017.

[10] Polars Developers, "Polars: Fast dataframe library written in rust," https://www.pola.rs/, 2022.

[11] X. Wang *et al.*, "ConnectorX: Accelerating data loading from databases to dataframes," in *Proc. VLDB Endowment*, vol. 15, no. 11, 2022.

[12] M. Kabić, S. Chandran, and G. Alonso, "Maximus: A modular accelerated query engine," *Proc. ACM Mgmt. Data*, vol. 3, no. 3, 2025.

[13] N. D. Jones, C. K. Gomard, and P. Sestoft, *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993, freely available author edition.

[14] W. R. Cook and R. Lämmel, "Tutorial on online partial evaluation," in *IFIP Working Conference on Domain-Specific Languages (EPTCS 66)*, 2011, pp. 81–104.

[15] Y. Choi *et al.*, "FORTE: Online dataframe query optimizer," in *IEEE/ACM CGO*, 2026.

[16] A. Shali and W. R. Cook, "Hybrid partial evaluation," in *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 375–390. [Online]. Available: https://doi.org/10.1145/2048066.2048098

[17] NYC Taxi and Limousine Commission, "Tlc trip record data," 2023. [Online]. Available: https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page

[18] Home Credit Group, "Home credit default risk," Kaggle, 2018. [Online]. Available: https://www.kaggle.com/c/home-credit-default-risk

[19] Apache Software Foundation, "Apache arrow: A cross-language development platform for in-memory data," https://arrow.apache.org/, 2017.

[20] T. Ahmad, Z. Al-Ars, and H. P. Hofstee, "Benchmarking apache arrow flight – a wire-speed protocol for data transfer, querying and microservices," in *PPoPP '22: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2022. [Online]. Available: https://arxiv.org/abs/2204.03032

[21] A. Lamb, Y. Shen, D. Heres, J. Chakraborty, M. O. Kabak, C. Sun, L.-C. Hsieh *et al.*, "Apache arrow datafusion: A fast, embeddable, modular analytic query engine," in *Proceedings of the 2024 ACM SIGMOD International Conference on Management of Data*, 2024.

[22] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith, "Cython: The best of both worlds," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 31–39, 2011.

[23] C. Consel *et al.*, "Tempo: Specializing systems applications and beyond," *ACM Computing Surveys*, vol. 30, no. 3es, p. 19es, 1998.

[24] C. Consel, J. L. Lawall, and G. Muller, "A tour of tempo: A program specializer for the c language," *Science of Computer Programming*, vol. 52, no. 1–3, pp. 341–370, 2004.

[25] W. Taha and T. Sheard, "Metaml and multi-stage programming with explicit annotations," *Theoretical Computer Science*, vol. 248, no. 1–2, pp. 211–242, 2000.

[26] T. Rompf and M. Odersky, "Lightweight modular staging: A pragmatic approach to runtime code generation and compilation," in *Proceedings of the 9th ACM SIGPLAN International Conference on Generative Programming and Component Engineering (GPCE)*, 2010, pp. 127–136.

[27] R. Leißa, K. Boesche, S. Hack, A. Pérard-Gayot, R. Membarth, P. Slusallek, A. Müller, and B. Schmidt, "Anydsl: a partial evaluation framework for programming high-performance libraries," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, Oct. 2018. [Online]. Available: https://doi.org/10.1145/3276489