

stratum: A System Infrastructure for Massive Agent-Centric ML Workloads

Arnab Phani
BIFOLD & TU Berlin
arnab.phani@tu-berlin.de

Elias Strauss
BIFOLD & TU Berlin
elias.strauss@tu-berlin.de

Sebastian Schelter
BIFOLD & TU Berlin
schelter@tu-berlin.de

ABSTRACT

Recent advances in large language models (LLMs) transform how machine learning (ML) pipelines are developed and evaluated. LLMs enable a new type of workload, *agentic pipeline search*, in which autonomous or semi-autonomous agents generate, validate, and optimize complete data science pipelines. These agents predominantly operate over popular Python ML libraries and exhibit highly exploratory behavior. This results in thousands of executions for data profiling, pipeline generation, and iterative refinement of pipeline stages and hyperparameters. However, the existing Python-based ML ecosystem is built around libraries such as Pandas and scikit-learn, which are designed for human-centric, interactive, sequential workflows and remain constrained by Python’s interpretive execution model, library-level isolation, and limited runtime support for executing large numbers of pipelines. Meanwhile, many high-performance ML systems proposed by the systems community either target narrow workload classes or require specialized programming models, which limits their integration with the Python ML ecosystem and makes them largely ill-suited for adoption by LLM-based agents. This growing mismatch exposes a fundamental systems challenge in supporting agentic pipeline search at scale.

We therefore propose *stratum*, a unified system infrastructure that decouples pipeline execution from planning and reasoning during agentic pipeline search. Stratum integrates seamlessly with existing Python libraries, compiles batches of agent- or human-generated pipelines into optimized execution graphs, and efficiently executes them across heterogeneous backends, including a novel Rust-based runtime. We present stratum’s architectural vision along with an early prototype, discuss key design decisions, and outline open challenges and research directions. Finally, preliminary experiments show that stratum can significantly speed up large-scale agentic pipeline search up to 16.6x.

PVLDB Reference Format:

Arnab Phani, Elias Strauss, and Sebastian Schelter. stratum: A System Infrastructure for Massive Agent-Centric ML Workloads. PVLDB, 19(11): 3799 - 3806, 2026.
doi:10.14778/3836663.3836726

PVLDB Artifact Availability:

The source code, data, and/or other artifacts have been made available at <https://github.com/deem-data/stratum>.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 19, No. 11 ISSN 2150-8097.
doi:10.14778/3836663.3836726

1 INTRODUCTION

The rapid adoption of ML has a profound impact on many domains. Despite this progress, developing ML pipelines remains a labor-intensive process for data scientists, requiring extensive domain knowledge, data engineering effort, iterative experimentation, and exploratory analysis [71, 83, 100]. Recent research has therefore focused on LLM-backed machine learning engineering (MLE) agents [23, 51, 52, 54, 56, 57], leveraging their code generation and reasoning capabilities. These agents frame ML tasks as code optimization problems, and explore a large space of candidate solutions by generating and executing a vast number of Python pipelines, guided by their predictive performance on held-out data.

Agentic Workloads: Large enterprises are increasingly adopting MLE agents for data science and ML application development [3, 15], making *agentic AI* a prominent research direction in the ML community [41, 61]. Practitioners now employ LLMs across a spectrum of autonomy [106]—from fully autonomous agents that generate and validate complete pipelines [5, 23, 51, 54, 57], to semi-automated frameworks that offload specific pipeline stages [21, 32, 59], to AI-assisted programming, where engineers manually assemble LLM-suggested components using declarative APIs from libraries such as skrub [92] or scikit-learn [64]. Fully autonomous agents typically translate natural language tasks into executable code through iterative cycles of planning, implementation, and evaluation. These cycles begin with data profiling [23, 51] and proceed to targeted refinement of ML pipeline stages [52, 54], seamlessly combining widely used libraries with specialized solutions for both tabular and multimodal datasets [52]. Semi-autonomous frameworks automate specific stages such as error detection [53], entity matching [53], or feature engineering [32], through iterative code generation and validation. Recent work on semantic operators [59, 63] further generalizes this paradigm by dynamically delegating fine-grained subtasks to LLMs. Finally, modern frameworks such as skrub unify multiple pipeline variants with different algorithms under a declarative abstraction, exhibiting similar exploratory behavior even without explicit use of agents.

Challenges of Agentic Pipeline Search: MLE agents generate large numbers of heterogeneous Python code—from metadata discovery to full pipeline executions—often at a rate of thousands per second [47, 55]. These agents explore large search spaces by combining broad *exploration* with targeted *exploitation*, yet lack dedicated runtime support, leading to overlapping executions, out-of-memory failures, and inefficient hardware utilization. While prior work proposed optimizations for Text2SQL workloads [47, 80, 98], these techniques do not transfer to ML workloads. LLM-based query synthesis frameworks primarily improve SQL generation through prompt design [13, 91], decomposition [96], and execution error signals [103],

and rely on mature database engines for efficient execution. In contrast, MLE agents perform guided search over pipeline structures and hyperparameters, repeatedly executing entire pipelines (including model training and evaluation) in a fragmented and inefficient ML ecosystem. As LLM throughput continues to increase, smaller models become more efficient [84, 97], specialized hardware for inference advances [6, 26], and models are explicitly trained for agentic behavior [33], research and adoption of autonomy in ML code generation are likely to accelerate. We argue that this shift demands a new class of data systems explicitly designed for agentic pipeline search tailored to ML workloads, for the following reasons:

Insufficient ML Libraries: MLE agents largely rely on popular Python ML libraries. Over the past decades the Python ML ecosystem has matured significantly, with data scientists heavily depending on high-level libraries such as Pandas and scikit-learn, often combined with a long tail of custom and domain-specific libraries to construct complex pipelines [73]. These libraries are deeply embedded in modern ML practice and are extensively represented in LLM pre-training corpora, making them effectively *here to stay* as the primary interface for agent-generated ML code. However, most ML libraries lack a unifying execution layer that can optimize computation holistically. Instead, they prioritize usability and flexibility over computational efficiency. Moreover, Python itself is not designed for high-performance high-concurrency workloads. As a result, today’s Python-based ML ecosystem remains inadequate for supporting the scale of emerging agentic ML workloads.

Lack of Efficient Systems for End-to-end Data Science: In contrast to mainstream Python libraries, the systems research community made significant advances toward efficient ML execution. First, systems such as SystemML [10], SystemDS [11], OptiML [89], KeystoneML [88], and DAPHNE [16] abandon Python in favor of domain-specific languages (DSLs) to enable full-program compilation, cost-based optimization, and multi-backend runtimes, but applying these techniques to the general Python ML ecosystem remains challenging and requires substantial re-engineering. Weld [60] eases integration but requires manual operator porting and glue code. Despite their technical influence [8, 70], these systems see limited adoption [74, 75], making them difficult for LLMs to target due to their scarcity in training corpora. Second, modern task-specific high-performance systems retain Python interfaces while leveraging optimized native engines. Examples include systems for dataframe processing [48, 65, 95], tree-based learning [14, 39, 72], in-process databases [78], and DNN workloads [1, 62]. PyTorch and TensorFlow also support JIT compilation [4, 49], though these features remain underutilized [2, 102]. Distributed frameworks like Ray [50] and Dask [79] focus on task scheduling rather than holistic optimization, while AutoML frameworks [20, 25, 44] automate model selection but lack compilation and runtime support. However, these optimizations are largely specialized to specific tasks and do not generalize across heterogeneous pipelines without additional research effort. Despite these advances, the ecosystem lacks a unified system that efficiently executes heterogeneous, end-to-end data science pipelines within Python and provides a common abstraction for optimization across pipeline stages. The current landscape remains fragmented—optimized for isolated tasks—leaving

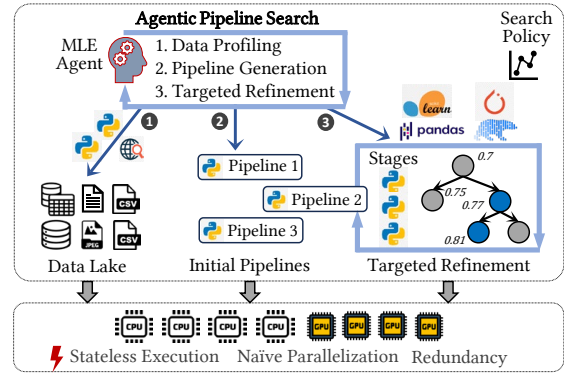


Figure 1: Agentic Pipeline Search

a substantial gap between the usability of Python APIs and the performance demands of large-scale, agentic ML workloads.

Our Vision: We envision a new ML system designed to efficiently execute large-scale agentic pipeline search, addressing the fundamental mismatch between high-level Python APIs and low-level optimizations. To this end, we introduce *stratum*, a system that integrates with MLE agents to accelerate agentic pipeline search. *Stratum*¹ raises the scope of optimization from individual pipelines to the entire search process. It represents batches of agent- or user-generated pipelines—written using popular or custom ML libraries—as lazily evaluated directed acyclic graphs (DAGs), without requiring DSLs or manual operator porting, applies logical rewrites, lowers logical operators to physical implementations, and executes them across heterogeneous backends, including a novel Rust-based runtime. Our detailed contributions are:

- **Agentic Workloads:** We present a representative use case for agentic pipeline search and analyze the execution characteristics of agentic workloads, motivating the design of *stratum* (Section 2).
- **Design Principles:** We discuss our design principles for supporting large-scale agentic pipeline search (Section 3).
- **System Architecture:** We describe *stratum*’s architectural vision, including its logical optimizer, operator selection, Rust backend, parallelization planning, and reuse of intermediates (Section 4). *Stratum* integrates seamlessly with arbitrary ML libraries.
- **Challenges and Key Directions:** We discuss the status of our *early prototype*, open challenges and research directions (Section 5).
- **Preliminary Experiments:** Using a real-world workload generated by the AIDE agent [57], we present preliminary results from both end-to-end and microbenchmark evaluations (Section 6). Even in its early implementation, *stratum* yields significant speedups over the baselines, validating our design decisions.

2 AGENTIC ML WORKLOADS

We introduce a representative enterprise use case for agentic ML to discuss scale and characteristics of this type of workloads.

End-to-end Agentic Pipeline Search Use Case: Consider an enterprise data scientist building a customer churn model from heterogeneous datasets stored in a shared data lake, including CSV and Parquet files (demographics, transaction and usage logs) as well as text and images accumulated over years. The data scientist invokes

¹stratum: <https://github.com/deem-data/stratum>

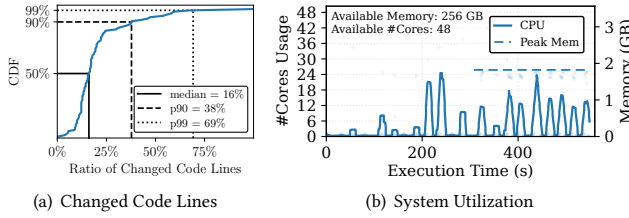


Figure 2: Distribution of code changes and CPU/Memory utilization during agentic search for an example workload.

an MLE agent to generate an end-to-end executable pipeline. As shown in Figure 1, the agent first profiles the data—extracting data characteristics, missing value ratios, cardinalities, vocabulary diversity, and image metadata—often via repeated sampling [23, 51]. Based on the inferred problem type and data characteristics, it retrieves relevant approaches from the web [52] and synthesizes candidate pipelines combining alternative preprocessing strategies, model families, text-derived features, and multimodal variants using specialized libraries. The agent executes and evaluates these pipelines, prunes infeasible candidates, and then shifts from broad *exploration* to targeted refinement (*exploitation*) by iteratively modifying pipeline stages (e.g., feature transformation, imputation, model training), exploring stage-wise variants and hyperparameters [52]. This refinement–evaluation loop repeats across pipeline variants, followed by constructing ensembles, and cross-validation. The data scientist may rerun the agent with modified prompts or launch multiple agents in parallel, each executing pipelines in separate Python processes, to broaden the search.

Execution Characteristics of Agentic ML Workloads: This use case exposes several execution patterns typical of agentic ML: (1) *Redundant Computation:* Dataset discovery triggers many lightweight profiling scripts; for large datasets, repeated data loading often dominates cost. During targeted refinement (exploitation phase), agents execute large numbers of pipelines repeatedly applying similar transformations, resulting in significant redundant computation. Figure 2(a) highlights this on a Kaggle competition [30] executed by the AIDE [57] agent: 50% of iterations modify 16% or fewer lines of code, indicating substantial overlap across pipeline variants. (2) *Limited and Process-based Parallelism:* Modern agents [52, 57] execute Python code in a largely stateless manner, maintaining exploration state (e.g., accuracy feedback and search strategy) outside the runtime. Pipelines are therefore evaluated sequentially or via naive parallelization, spawning many independent Python processes. This incurs substantial hardware contention and serialization overhead. Coarse-grained scale-out further increases cost and energy consumption [94]. (3) *System-agnostic Execution:* Agents operate without awareness of underlying system characteristics such as available memory, CPU cores, GPUs, or distributed backends, leading to out-of-memory (OOM) failures, underutilized hardware, and missed opportunities for holistic data- and task-parallel execution and intermediate reuse. Figure 2(b) shows highly irregular CPU and memory utilization, indicating poor resource efficiency during agentic search. These inefficiencies are most visible in fully autonomous agents but also arise in semi-autonomous settings. Together, they result in poor resource utilization, redundant materialization and computation, and overall inefficient execution.

3 OUR VISION

We motivate the necessity for a new system tailored to agentic pipeline search and outline its key technical requirements.

The Case for a New System: As LLMs are increasingly used for code generation, the popularity of mainstream and specialized Python libraries will continue to grow. Modern MLE agents employ diverse search policies [23, 54] to navigate large search spaces of independently evolving libraries and APIs. These include running multiple agent instances with different prompts [54], exploring sampled datasets [82], naive parallelization across nodes and processes [52, 54], and restarting exploration based on early feedback [82]. However, they primarily optimize search strategies to maximize pipeline quality (e.g., accuracy), while largely overlooking execution latency and throughput. As discussed in Section 1, existing ML systems either target narrow workload classes or rely on DSLs that are ill-suited for LLMs. As a result, the scale of agentic pipeline search cannot be met through incremental extensions to existing systems. We argue that the sheer diversity of ML libraries, coupled with the absence of common abstractions necessitates a new class of systems designed for agentic ML workloads.

The Vision for Stratum: To this end, we envision *stratum*, a system architecture for large-scale agentic pipeline search. Specifically, our design is based on the following key principles:

- *Seamless and unrestricted support* for ML libraries (scikit-learn, Pandas), frameworks (PyTorch, TensorFlow), tabular foundation models [31, 77], and specialized libraries—without requiring operator porting—while remaining extensible to future libraries.
- *A semantic abstraction* built on a minimal set of logical operators that enables rewrites (e.g., common subexpression elimination (CSE)) and lazy evaluation with physical operator independence.
- *A runtime* with efficient operator kernels and scheduling across heterogeneous backends including CPUs, GPUs, and distributed backends to support diverse data characteristics and ML tasks, and cost-based runtime optimizations including reuse of intermediates, memory management, and parallelization planning.

However, realizing this vision raises a range of open challenges across representation, optimization, and execution, and demands novel techniques throughout the system stack.

4 SYSTEM ARCHITECTURE

Here we describe the overall architecture of *stratum* (Figure 3) and its key components. *Stratum* builds on skrub’s [92] operator abstractions. At a high level, *stratum* ingests batches of ML pipelines, fuses them into a unified operator DAG, applies logical optimizations and operator lowering, and selects efficient backends for execution. We implement a Rust-based execution engine along with system-level optimizations such as nested parallelism and intermediate caching. While many of these techniques draw inspiration from prior research, our primary contribution lies in the principled integration of these techniques into a holistic system underneath the Python ML ecosystem, which we argue is both timely and of utmost necessity.

4.1 A Declarative Abstraction

Logical and runtime optimizations require representing ML pipelines as a DAG with well-defined operator semantics, which is

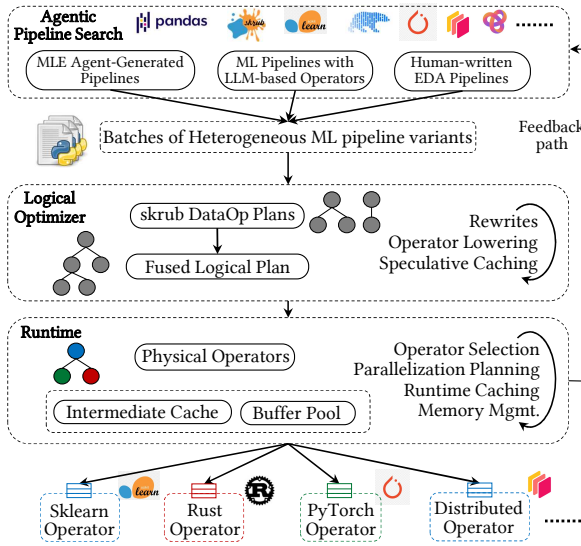


Figure 3: Stratum System Architecture

challenging due to the lack of a universal abstraction for heterogeneous ML code [28]. Translating arbitrary Python ML code into a DAG via instrumentation is also difficult given Python’s dynamic semantics [24, 27, 43]. Stratum adopts *skrub* [92] as its entry point because it converts arbitrary ML pipelines into lazily evaluated DAGs. *skrub* automatically wraps operations—including Pandas, scikit-learn, DNN models, custom library calls, and UDFs—into semantically explicit operators (DataOps [86]), forming a control-flow-free execution DAG. This abstraction unifies data preparation, model training, and cross-validation constructs into a single lazily executable pipeline. Figure 4 shows a simplified *skrub* pipeline, where `make_grid_search` forms and executes the DAG with cross-validation. Although designed for usability rather than performance, *skrub*’s operator semantics provides a foundation for system-level optimization. Stratum extends these abstractions with new operators, iterative lowering, and its own compiler and runtime stack, while remaining API-compatible with *skrub*.

```
prd_c = prd_c.skb.apply(OneHotEncoder())
prd_n = prd_n.skb.apply(StandardScaler())
prd_vec = prd_c.skb.concat([prd_n])
X_jn = X.merge(prd_vec, ...) #join
X_jn.drop(columns=["ID", "basket_ID"])
model = LGBMRegressor(random_state=42)
preds = X_jn.skb.apply(model, y=y)
search = preds.skb.make_grid_search(cv=cv)
```

Figure 4: Example Skrub Code

4.2 Logical Optimizer and Runtime

Agents emit pipeline variants in batches. Stratum fuses each batch into a unified DAG and passes it to the logical optimizer.

Logical Operators and Rewrites: We compile *skrub* DataOps into a common logical operator hierarchy (e.g., SELECT, PROJECT, MAP, TRANSFORMER, and ESTIMATOR). *skrub* treats most operators as black boxes. Stratum performs a metadata collection pass to extract operator-level metadata, such as operator type (e.g., dataframe operation, estimator), source library, structural properties (e.g., selection and projection), and data characteristics (e.g., #rows, #cols, and datatypes). Using this metadata, the optimizer applies logical rewrites that preserve semantic equivalence, including predicate pushdown, read sharing, CSE, and constant folding

to reduce redundant computation and improve data locality. Figure 5 shows a simplified execution graph after fusing four pipelines that combine two preprocessing techniques (in blue) with two models (in red) and applying logical rewrites. Stratum also applies API-aware rewrites, such as reordering Pandas operations to reduce copies under copy-on-write, enabling in-place updates, and replacing non-vectorized loops with vectorized implementations.

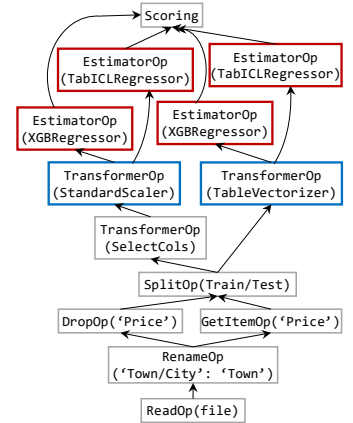


Figure 5: Example DAG

Operator Lowering: Following metadata collection and logical rewrites, stratum lowers logical operators (e.g., TRANSFORMER, ESTIMATOR) into fine-grained physical operators (e.g., OneHotEncoder or ElasticNet) and apply physical-level rewrites. It further decomposes high-level operators such as *skrub*’s TableVectorizer (Figure 5), which encapsulates automatic missing value imputation and feature transformations, into independent operators like *cleaner*, *StringEncoder*, and *OneHotEncoder*. Similarly, higher-level constructs such as cross-validation and hyperparameter search, which repeatedly execute the same subgraph, are unrolled into explicit DAGs. This lowering enables accurate cost estimation, runtime planning, and memory management.

Operator Selection: Stratum maintains multiple implementations for each physical operator with distinct algorithmic and computational characteristics. For example, dimensionality reduction may execute via scikit-learn’s SVD, an approximate alternative such as Frequent Directions [35] during early exploration, or stratum’s native Rust runtime. We maintain a registry of available implementations and select them via late binding. This logical-physical separation enables extensibility across new libraries (e.g., PyTorch, Spark), execution backends (e.g., Dask for out-of-core execution), custom operators, and hardware environments, while naturally supporting both lazily and eagerly evaluated frameworks. Using collected metadata and compute and memory estimates, stratum selects operator implementations that minimize execution time under memory constraints. The optimal choice depends on neighboring operators, data placement, and the availability of intermediates in the cache. For example, if downstream PyTorch operators are placed on GPUs, stratum may prefer cuDF over Pandas or Polars to avoid data movement. For operators that fit in-memory, we prefer efficient native (GIL-releasing) backends such as Rust, Polars, cuDF, or NumPy-backed implementations.

Rust Backend: While efficient frameworks exist for specific ML stages (e.g., Polars, XGBoost, PyTorch), many widely used operators in scikit-learn, and Pandas are implemented in Python. Even when backed by NumPy or Cython, their highly generic implementations—designed to support many hyperparameters and edge cases—incur repeated type conversions, temporary allocations, unnecessary copies, and limited multithreading. Such overheads often dominate end-to-end execution time, diminishing the gains

from optimizations (Amdahl’s Law). To address this, we incrementally develop Rust implementations for frequently used operator configurations [73], exposed via lightweight PyO3 bindings [38]. Rust enables efficient kernels with explicit memory control and zero-copy data access, while fitted model states remain in Rust-managed objects to eliminate data conversions. These kernels release Python’s global interpreter lock (GIL) to enable concurrent kernel execution and exploit native data-parallelism. In addition, the Rust backend lowers operator granularity, enables accurate cost estimates, improves memory management, reduces boundary crossings, and enables advanced optimizations such as operator fusion [12], sparsity exploitation [87], and fine-grained reuse [70].

4.3 Runtime Optimizations

Inter- and Intra-operator Parallelism: A major performance bottleneck in ML libraries stems from Python’s limited support for native parallelism. While multithreading is standard in systems, Python’s GIL restricts concurrency unless native kernels explicitly release it. Multiprocessing-based schedulers (e.g., Dask, Ray, Joblib) achieve parallelism for scikit-learn estimators, but incur high memory and serialization overhead due to process-level data duplication. Moreover, many ML libraries—and stratum’s Rust backend—already employ internal multithreading (e.g., OpenMP, OpenBLAS, Rayon), introducing nested parallelism that risks oversubscription. Although optional GIL removal exists in recent Python versions [76], adoption remains limited. Stratum primarily relies on multithreading. During the exploration phase, agents evaluate many non-overlapping pipelines, increasing opportunities for inter-operator parallelism. Using compute and memory estimates, and available hardware, a cost-based optimizer traverses the DAG, evaluates plans under worst-case memory budgets, and selects plans that minimize execution time subject to memory constraints [69, 104]. In a subsequent pass, we determine the degree of intra- and inter-operator parallelism to avoid oversubscription.

Reuse of Intermediates: To exploit the repetitive nature of the exploitation phase of agentic pipeline search (see Figure 2(a)), we employ a combination of coarse-grained reuse [99]—caching results of top-level operators—with fine-grained reuse [66–68, 70] across shared Rust kernels. The cache maps operator hashes to materialized outputs (e.g., dataframes, NumPy arrays, Rust-backed model states). Operator hashes are derived from input hashes and operator specifications; seed values (e.g., `random_state`) are incorporated to handle non-determinism, while operators without explicit seeds are not cached. Before execution, each operator probes the cache and reuses previously materialized results when available. We allocate a fixed fraction of memory (default 10%) for cached objects. To assist the runtime caching, the optimizer speculatively marks selected operators (e.g., expensive preprocessing) as cache candidates. Cached intermediates are materialized to Parquet and lazily reloaded across iterations. Unlike prior reuse frameworks, designed for iterative human-driven ML workflows or DSL-based pipelines [67, 70, 99], agentic workloads exhibit far larger and more dynamic exploration spaces spanning profiling, heterogeneous pipelines, and DNN workloads. Moreover, exploration-exploitation patterns and multi-tenant execution (see Section 5) make reuse decisions substantially more

challenging [7], motivating specialized caching and reuse strategies for agentic ML workloads.

5 CHALLENGES AND KEY DIRECTIONS

This section first summarizes the current state of the prototype and then discusses major challenges and long-term directions.

Prototype Status: The current prototype establishes stratum’s core architecture, including an initial logical optimizer with meta-data collection and basic rewrites. The Rust backend provides a limited set of execution kernels sufficient to validate key design choices. At present, stratum supports only in-memory operators and relies on heuristics for operator selection and parallelization planning. Finally, we employ a greedy caching strategy that materializes expensive preprocessing operators for reuse. These initial implementations serve as placeholders for more principled cost-based and adaptive strategies, which are under active development.

Open Challenges: While our early prototype shows promising results, several open challenges remain. (1) *Cost Estimates:* Operator selection and runtime optimizations require decent memory and compute estimates, which are difficult to obtain as Pandas and scikit-learn methods often materialize hidden intermediates that inflate memory usage. We are exploring sampling-based cost estimation [18] and adaptive planning. (2) *Cross-library Optimizations:* Heterogeneous pipelines combine operators from diverse libraries (e.g., Pandas, NumPy, PyTorch) with widely different execution granularities, from lightweight dataframe transformations to expensive trainings. These boundaries restrict cross-library fusion and complicate scheduling and inter-operator parallelism. We address this by increasingly relying on homogeneous Rust-based operators, which enable more effective fusion, scheduling, and parallelization. (3) *UDFs:* Custom pipelines components and specialized libraries are often wrapped as UDFs, creating black-box operators that hide semantic information. We are exploring profiling on sampled inputs to estimate costs and adaptive optimizations based on observed costs. For common map-style functions, output dimensions are often preserved, enabling accurate memory estimation. (4) *DNN Workloads:* Supporting DNN and agentic kernel generation requires deeper CUDA and PyTorch integration [58, 101]. We are exploring skorch [93] as well as Rust bindings to PyTorch (e.g., `tch-rs`) to enable fine-grained optimizations such as quantization, weight pruning, input data pipeline reuse, and pipeline parallelism.

Agent-System Co-design: Stratum integrates with existing agents; however, a tighter integration between agents and stratum exposes opportunities for jointly optimizing the agentic search and its execution: (1) *Workload-aware Optimizations:* Instead of executing independent Python scripts, agents annotate pipelines with lightweight metadata (e.g., exploration vs. exploitation stage), enabling stratum to adapt execution strategies such as successive halving [42], reduced training epochs, lower-fidelity operators during exploration, and compiler-assisted caching of operators with high reuse potential during exploitation. (2) *Declarative Pipeline Specification:* Instead of generating full Python scripts, system-aware agents emit abstract pipeline specifications [54], allowing stratum to internally construct execution graphs. This reduces the number of LLM calls and energy consumption, while expanding optimization opportunities. (3) *Overlapping Generation and Execution:* Agents generate

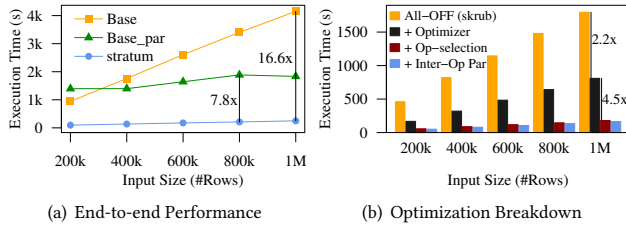


Figure 6: Impact of stratum and its optimizations.

pipelines in batches and wait for feedback. Stratum mitigates idle time by executing subsets of each batch, returning early feedback, and overlapping execution with subsequent pipeline generation. Together, these mechanisms elevate optimization from individual pipeline executions to the entire search process, motivating a new class of agents that jointly optimize pipeline search and execution.

Other Key Directions: Beyond building a robust system implementation and a representative system-aware agent, several longer-term research directions remain: (1) *Multi-tenancy*: We envision stratum evolving into a multi-tenant, cloud-hosted service that interfaces directly with MLE agents, enabling dynamic scheduling, adaptive resource management, and shared intermediate caches across workloads. (2) *Inference engine*: A complementary direction is tailoring the inference stack for agentic workloads through customized inference configurations and shared key-value caches, further reducing latency and cost across agent interactions [105].

6 PRELIMINARY EXPERIMENTS

We evaluate the baseline performance of stratum’s early prototype and its core components to validate its key design principles.

HW Environment and Workload: We run all experiments on a single node with an AMD EPYC 7443P CPU (24 / 48 cores) and 256 GB RAM, using Ubuntu 20.04, Python 3.11, scikit-learn 1.8, and skrub 0.6.2. We use AIDE [57] to generate a representative agentic pipeline search workload with two iterations. The first iteration explores all combinations of two preprocessing strategies and four models. The preprocessing strategies include: (1) missing-value imputation and feature encoding using StringEncoder, a custom target encoder, and StandardScaler; and (2) TableVectorizer, which performs automatic cleaning and applies one-hot encoding and StringEncoder to low- and high-cardinality features. The models include Ridge, XGBoost, LightGBM, and ElasticNet. In the second iteration, we select the best-performing preprocessing strategy and model based on validation accuracy and perform hyperparameter tuning. We use the UK housing dataset from Kaggle [30] and vary dataset sizes to evaluate scalability.

End-to-end Performance: We compare AIDE (which performed best in MLEBench) with stratum. The baselines include: **Base**, representing AIDE with sequential pipeline execution; **Base_par**, where AIDE triggers multiple pipelines concurrently; and stratum with all optimizations enabled. As shown in Figure 6(a), stratum yields a 16.6 $\times$ speedup over Base. This improvement stems from: (i) pipeline fusion in the first iteration, (ii) CSE to deduplicate preprocessing, (iii) operator selection (Polars over Pandas and our Rust kernels over scikit-learn), (iv) intra- and inter-operator parallelism, and (v) reuse of preprocessing results in the second iteration.

While Base_par improves upon Base, its reliance on multiprocessing incurs significant serialization overhead and increases memory consumption by 8 $\times$. Stratum remains 7.8 $\times$ faster than Base_par.

Ablation Study: To study the impact of stratum’s individual optimizations—which vary with data and workload characteristics—we incrementally enable each optimization in isolation. As shown in Figure 6(b), logical optimization alone yields up to a 2.2 $\times$ speedup through CSE and related rewrites. Enabling operator selection provides an additional 4.5 $\times$ improvement by replacing Python-based operators with native implementations, which also enable data-parallelism by releasing the GIL and leveraging multi-threading. Finally, inter-operator parallelism contributes a further 10% speedup. In this workload, the dominant operators are already compute-intensive and fully utilize all available cores, limiting the additional gains achievable through inter-operator parallelism.

Overall, these results validate our design principles, showing that even an early prototype of stratum can significantly accelerate workloads through holistic logical and runtime optimizations.

7 ADDITIONAL RELATED WORK

Beyond the prior work discussed in Section 1, stratum is related to ML systems, accelerating dataframes, and agentic SQL workloads.

Scalable Data Science: Systems for accelerating data science [19, 29, 36, 48, 65, 81, 85, 85] improve dataframe performance by rule-based and dynamic tiling, parallel execution, or by translating Pandas operations to SQL. Our logical rewrites draw inspiration from mlwhatif [27], which builds an operator DAG by instrumenting Python code. In contrast, stratum constructs a lazily evaluated operator DAG directly from arbitrary ML libraries and applies advanced optimizations including rewrites and operator selection.

ML Systems Optimizations: Our compiler and runtime techniques relate to prior work on ML systems [10, 11, 40], pipeline parallelism [22, 34], inter- and intra-operator parallelism [104], task-based execution [9, 50, 69], and coarse- and fine-grained reuse [66, 67, 70, 99]. In contrast, stratum employs operator scheduling, multi-level parallelism, and reuse for agentic pipeline search.

Agentic SQL: Stratum is also related to systems for Text2SQL [46, 47, 98] and optimizing semantic operators [17, 37, 45, 63, 80, 90]. Unlike these systems, stratum targets agentic ML workloads.

8 CONCLUSION

By elevating ML development from manual scripting to autonomous generation and iterative refinement, agentic pipeline search introduces a fundamentally new workload pattern that necessitates a new class of systems tailored to agentic ML workloads. This raising of abstraction in ML development—together with the ML community’s growing appreciation for high-level operator semantics [92]—creates an opportunity to rethink end-to-end ML system design. We introduced stratum, a system infrastructure for agentic pipeline search that enables logical and runtime optimizations while remaining fully compatible with existing Python libraries. In summary, stratum comprises (1) an execution engine supporting lazy evaluation, (2) an efficient Rust backend, (3) cost-based operator selection, parallelization, and caching, and (4) an optimizer that abstracts heterogeneous ML libraries under a unified execution model.

REFERENCES

- [1] Martin Abadi et al. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*. 265–283.
- [2] Zakariya Ba Alawi. 2025. A Comparative Survey of PyTorch vs TensorFlow for Deep Learning: Usability, Performance, and Deployment Trade-offs. arXiv:2508.04035 [cs.LG] <https://arxiv.org/abs/2508.04035>
- [3] Anaconda, Inc. 2025. *State of Data Science 2024: AI and Open Source at Work*. Report. Anaconda, Inc. <https://www.anaconda.com/resources/report/state-of-data-science-report-2024> 7th Annual Report.
- [4] Jason Ansel. 2022. *TorchDynamo*. <https://github.com/pytorch/torchdynamo>
- [5] Eser Aygün et al. 2025. An AI system to help scientists write expert-level empirical software. *CoRR* abs/2509.06503 (2025). <https://doi.org/10.48550/ARXIV.2509.06503>
- [6] Alexander Baumstark and Kai-Uwe Sattler. 2026. Does A Fish Need a Bicycle? The Case for On-Chip NPU in DBMS. In *CIDR*. <https://vldb.org/cidrdb/2026/does-a-fish-need-a-bicycle-the-case-for-on-chip-npus-in-dbms.html>
- [7] Sebastian Baunsgaard et al. 2022. Federated Data Preparation, Learning, and Debugging in Apache SystemDS. In *CIKM*. 4813–4817. <https://doi.org/10.1145/3511808.3557162>
- [8] Sebastian Baunsgaard and Matthias Boehm. 2023. AWARE: Workload-aware, Redundancy-exploiting Linear Algebra. *Proc. ACM Manag. Data* 1, 1 (2023), 2:1–2:28. <https://doi.org/10.1145/3588682>
- [9] Matthias Boehm et al. 2014. Hybrid Parallelization Strategies for Large-Scale Machine Learning in SystemML. *PVLDB* 7, 7 (2014), 553–564. <https://doi.org/10.14778/2732286.2732292>
- [10] Matthias Boehm et al. 2016. SystemML: Declarative Machine Learning on Spark. *PVLDB* 9, 13 (2016), 1425–1436. <https://doi.org/10.14778/3007263.3007279>
- [11] Matthias Boehm et al. 2020. SystemDS: A Declarative Machine Learning System for the End-to-End Data Science Lifecycle. In *CIDR*. [www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p22-boehm-cidr20.pdf](http://cidrdb.org/cidr2020/papers/p22-boehm-cidr20.pdf)
- [12] Matthias Boehm, Berthold Reinwald, Dylan Hutchison, Prithviraj Sen, Alexandre V. Evfimievski, and Niketan Pansare. 2018. On Optimizing Operator Fusion Plans for Large-Scale Machine Learning in SystemML. *PVLDB* 11, 12 (2018), 1755–1768. <https://doi.org/10.14778/3229863.3229865>
- [13] Shuaichen Chang and Eric Fosler-Lussier. 2023. How to Prompt LLMs for Text-to-SQL: A Study in Zero-shot, Single-domain, and Cross-domain Settings. *CoRR* abs/2305.11853 (2023). <https://doi.org/10.48550/ARXIV.2305.11853>
- [14] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *SIGKDD*. 785–794. <https://doi.org/10.1145/2939672.2939785>
- [15] Cloudera, Inc. 2025. 96 Percent of Enterprises are Expanding Use of AI Agents. <https://www.cloudera.com/about/news-and-blogs/press-releases/2025-04-16-96-percent-of-enterprises-are-expanding-use-of-ai-agents-according-to-latest-data-from-cloudera.html> Accessed: January 21, 2026.
- [16] Patrick Damme et al. 2022. DAPHNE: An Open and Extensible System Infrastructure for Integrated Data Analysis Pipelines. In *CIDR*. <https://www.cidrdb.org/cidr2022/papers/p4-damme.pdf>
- [17] Anas Dorbani, Sunny Yasser, Jimmy Lin, and Amine Mhedhbi. 2025. Beyond Quacking: Deep Integration of Language Models and RAG into DuckDB. *PVLDB* 18, 12 (2025), 5415–5418. <https://doi.org/10.14778/3750601.3750685>
- [18] Charles Dubout and François Fleuret. 2011. Boosting with Maximum Adaptive Sampling. In *NIPS*. 1332–1340.
- [19] K. Venkatesh Emani, Avriella Floratou, and Carlo Curino. 2024. PyFroid: Scaling Data Analysis on a Commodity Workstation. In *EDBT*. 61–67. <https://doi.org/10.48786/EDBT.2024.06>
- [20] Nick Erickson et al. 2020. AutoGluon-Tabular: Robust and Accurate AutoML for Structured Data. arXiv:2003.06505 [stat.ML] <https://arxiv.org/abs/2003.06505>
- [21] Meihao Fan, Ju Fan, Nan Tang, Lei Cao, Guoliang Li, and Xiaoyong Du. 2025. AutoPrep: Natural Language Question-Aware Data Preparation with a Multi-Agent Framework. *PVLDB* 18, 10 (2025), 3504–3517. <https://www.vldb.org/pvldb/vol18/p3504-fan.pdf>
- [22] Shiqing Fan et al. 2021. DAPPLE: a pipelined data parallel approach for training large models. In *PPoPP*. 431–445. <https://doi.org/10.1145/3437801.3441593>
- [23] Haoyang Fang et al. 2025. MLZero: A Multi-Agent System for End-to-end Machine Learning Automation. In *Advances in Neural Information Processing Systems*. <https://neurips.cc/virtual/2025/poster/119504>
- [24] Hanxi Fang, Supawit Chockchowwat, Hari Sundaram, and Yongjoo Park. 2025. Enhancing Computational Notebooks with Code+Data Space Versioning. In *CHI*. 154:1–154:17. <https://doi.org/10.1145/3706598.3714141>
- [25] Matthias Feurer et al. 2020. Auto-Sklearn 2.0: The Next Generation. *CoRR* abs/2007.04074 (2020).
- [26] Karl Freund. 2026. Taalas Launches Hardcore Chip With ‘Insane’ AI Inference Performance. <https://www.forbes.com/sites/karlfreund/2026/02/19/taalas-launches-hardcore-chip-with-insane-ai-inference-performance/> Accessed: 2026-02-21.
- [27] Stefan Grafberger, Paul Groth, and Sebastian Schelter. 2023. Automating and Optimizing Data-Centric What-If Analyses on Native Machine Learning Pipelines. *Proc. ACM Manag. Data* 1, 2 (2023), 128:1–128:26. <https://doi.org/10.1145/3589273>
- [28] Stefan Grafberger, Paul Groth, Julia Stoyanovich, and Sebastian Schelter. 2022. Data distribution debugging in machine learning pipelines. *Vldb J.* 31, 5 (2022), 1103–1126. <https://doi.org/10.1007/S00778-021-00726-W>
- [29] Stefan Hagedorn, Steffen Kläbe, and Kai-Uwe Sattler. 2021. Putting Pandas in a Box. In *CIDR*. <https://vldb.org/cidrdb/2021/putting-pandas-in-a-box.html>
- [30] HM Land Registry. 2021. UK Housing Prices Paid. Kaggle. <https://www.kaggle.com/datasets/hm-land-registry/uk-housing-prices-paid> Accessed: 2024-05-20.
- [31] Noah Hollmann, Samuel Müller, Katharina Eggensperger, and Frank Hutter. 2023. TabPFN: A Transformer That Solves Small Tabular Classification Problems in a Second. In *ICLR*. <https://openreview.net/forum?id=cp5Pvcl6w8>
- [32] Noah Hollmann, Samuel Müller, and Frank Hutter. 2023. Large Language Models for Automated Data Science: Introducing CAAFE for Context-Aware Automated Feature Engineering. In *NeurIPS*. http://papers.nips.cc/paper_files/paper/2023/hash/8c2df4c35cdbee764ebb9e9d0acd5197-Abstract-Conference.html
- [33] Mengkang Hu, Pu Zhao, Can Xu, Qingfeng Sun, Jian-Guang Lou, Qingwei Lin, Ping Luo, and Saravan Rajmohan. 2025. AgentGen: Enhancing Planning Abilities for Large Language Model based Agent via Environment and Task Generation. In *SIGKDD*. 496–507. <https://doi.org/10.1145/3690624.3709321>
- [34] Yanping Huang et al. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *NeurIPS*. 103–112.
- [35] Zengfeng Huang. 2019. Near Optimal Frequent Directions for Sketching Dense and Sparse Matrices. *J. Mach. Learn. Res.* 20 (2019), 56:1–56:23. <https://jmlr.org/papers/v20/18-875.html>
- [36] Alekh Jindal et al. 2021. Magpie: Python at Speed and Scale using Cloud Backends. In *CIDR*. <https://vldb.org/cidrdb/2021/magpie-python-at-speed-and-scale-using-cloud-backends.html>
- [37] Saehan Jo and Immanuel Trummer. 2024. ThalamusDB: Approximate Query Processing on Multi-Modal Data. *Proc. ACM Manag. Data* 2, 3 (2024), 186. <https://doi.org/10.1145/3654989>
- [38] Price D. Johnson and Douglas D. Hodson. 2025. PyO3: Building Python Extension Modules in Native Rust with Performance and Safety in Mind. In *Scientific Computing and Bioinformatics and Computational Biology (Communications in Computer and Information Science)*, Vol. 2258. Springer, Cham, 38–51. https://doi.org/10.1007/978-3-031-85902-1_4
- [39] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. LightGBM: A Highly Efficient Gradient Boosting Decision Tree. In *NeurIPS*. 3146–3154. <https://proceedings.neurips.cc/paper/2017/hash/6449f44a102fde848669bdd9eb6b76fa-Abstract.html>
- [40] Andreas Kunft, Asterios Katsifodimos, Sebastian Schelter, Sebastian Breß, Tilman Rabl, and Volker Markl. 2019. An Intermediate Representation for Optimizing Machine Learning Pipelines. *PVLDB* 12, 11 (2019), 1553–1567. <https://doi.org/10.14778/3342263.3342633>
- [41] Adrien Laurent. 2025. *NeurIPS 2025: A Guide to Key Papers, Trends & Stats*. Technical Report. IntuitionLabs. <https://intuitionlabs.ai/pdfs/neurips-2025-a-guide-to-key-papers-trends-stats.pdf> Accessed: 2026-01-18.
- [42] Lisha Li et al. 2017. Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization. *JMLR* 18 (2017), 185:1–185:52. <https://jmlr.org/papers/v18/16-558.html>
- [43] Zhaocheng Li, Supawit Chockchowwat, Areet Sheth, Yongjoo Park, and Ribhav Sahu. 2024. Kishu: Time-Traveling for Computational Notebooks. *PVLDB* 18, 4 (2024), 970–985. <https://doi.org/10.14778/3717755.3717759>
- [44] Edo Liberty et al. 2020. Elastic Machine Learning Algorithms in Amazon SageMaker. In *SIGMOD*. 731–737. <https://doi.org/10.1145/3318464.3386126>
- [45] Chunwei Liu et al. 2025. Palimpzest: Optimizing AI-Powered Analytics with Declarative Query Processing. In *CIDR*. <https://vldb.org/cidrdb/2025/palimpzest-optimizing-ai-powered-analytics-with-declarative-query-processing.html>
- [46] Shu Liu et al. 2025. Optimizing LLM Queries in Relational Data Analytics Workloads. In *MLSys*. <https://openreview.net/forum?id=R7bK9yycHp>
- [47] Shu Liu et al. 2026. Supporting Our AI Overlords: Redesigning Data Systems to be Agent-First. *CIDR* (2026).
- [48] Weizheng Lu, Kaisheng He, Xuye Qin, Chengjie Li, Zhong Wang, Tao Yuan, Xia Liao, Feng Zhang, Yueguo Chen, and Xiaoyong Du. 2024. Xorbits: Automating Operator Tiling for Distributed Data Science. In *ICDE*. 5211–5223. <https://doi.org/10.1109/ICDE60146.2024.00392>
- [49] Dan Moldovan, James M. Decker, Fei Wang, Andrew A. Johnson, Brian K. Lee, Zachary Nado, D. Sculley, Tiark Rompf, and Alexander B. Wiltschko. 2019. AutoGraph: Imperative-style Coding with Graph-based Performance. In *MLSys*. <https://proceedings.mlsys.org/book/272.pdf>
- [50] Philipp Moritz et al. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *OSDI*. 561–577.
- [51] Jaehyun Nam, Jinsung Yoon, Jiefeng Chen, and Tomas Pfister. 2025. DS-STAR: Data Science Agent via Iterative Planning and Verification. *CoRR* abs/2509.21825 (2025). <https://doi.org/10.48550/ARXIV.2509.21825>
- [52] Jaehyun Nam, Jinsung Yoon, Jiefeng Chen, Jinwoo Shin, Sercan Ö. Arik, and Tomas Pfister. 2025. MLE-STAR: Machine Learning Engineering Agent via Search and Targeted Refinement. In *NeurIPS*.

- [53] Avaniika Narayan, Ines Chami, Laurel J. Orr, and Christopher Ré. 2022. Can Foundation Models Wrangle Your Data? *PVLDB* 16, 4 (2022), 738–746. <https://doi.org/10.14778/3574245.3574258>
- [54] Alexander Novikov et al. 2025. AlphaEvolve: A coding agent for scientific and algorithmic discovery. *CoRR* abs/2506.13131 (2025). <https://doi.org/10.48550/ARXIV.2506.13131> arXiv:2506.13131
- [55] NVIDIA Corporation. 2026. *Deep Learning Performance: AI Inference*. NVIDIA Developer. <https://developer.nvidia.com/deep-learning-performance-training-inference/ai-inference> Accessed: 2026-01-18.
- [56] Siyuan Guo others. 2024. DS-Agent: Automated Data Science by Empowering Large Language Models with Case-Based Reasoning. In *ICML*. <https://openreview.net/forum?id=LfJgeBNCFI>
- [57] Zhengyao Jiang others. 2025. AIDE: AI-Driven Exploration in the Space of Code. *CoRR* abs/2502.13138 (2025). <https://doi.org/10.48550/ARXIV.2502.13138>
- [58] Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. 2025. KernelBench: Can LLMs Write Efficient GPU Kernels?. In *ICML*. <https://openreview.net/forum?id=yeoNiiQT1x>
- [59] Olga Ovcharenko, Matthias Boehm, and Sebastian Schelter. 2026. SemPipes – Optimizable Semantic Data Operators for Tabular Machine Learning Pipelines. arXiv:2602.05134 [cs.LG] <https://arxiv.org/abs/2602.05134>
- [60] Shoumik Palkar et al. 2017. Weld: A common runtime for high performance data analytics. *CIDR* (2017).
- [61] Paper Digest Team. 2025. Advances in Agentic AI: Insights from ICLR 2025 Papers. Paper Digest. <https://www.paperdigest.org/report/?id=advances-in-agentic-ai-insights-from-iclr-2025-papers> Accessed: 2026-01-18.
- [62] Adam Paszke et al. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *NeurIPS*. 8024–8035.
- [63] Liana Patel et al. 2025. Semantic Operators and Their Optimization: Enabling LLM-Based Data Processing with Accuracy Guarantees in LOTUS. *PVLDB* 18, 11 (2025), 4171–4184. <https://www.vldb.org/pvldb/vol18/p4171-patel.pdf>
- [64] Fabian Pedregosa et al. 2011. Scikit-learn: Machine Learning in Python. *J. Mach. Learn. Res.* 12 (2011), 2825–2830. <https://doi.org/10.5555/1953048.2078195>
- [65] Devin Petersohn, William W. Ma, Doris Jung Lin Lee, Stephen Macke, Doris Xin, Xiangxi Mo, Joseph Gonzalez, Joseph M. Hellerstein, Anthony D. Joseph, and Aditya G. Parameswaran. 2020. Towards Scalable Dataframe Systems. *PVLDB* 13, 11 (2020), 2033–2046. <http://www.vldb.org/pvldb/vol13/p2033-petersohn.pdf>
- [66] Arnab Phani. 2025. *Fine-grained reuse and feature transformations in machine learning systems*. Ph.D. Dissertation. TU Berlin, Germany. <https://nbn-resolving.org/urn:nbn:de:101:1-2504300159545.425811098571>
- [67] Arnab Phani and Matthias Boehm. 2025. MEMPHIS: Holistic Lineage-based Reuse and Memory Management for Multi-backend ML Systems. In *EDBT*. 255–269. <https://doi.org/10.48786/EDBT.2025.21>
- [68] Arnab Phani and Matthias Boehm. 2026. Lineage-based Reuse and Memory Management for Multi-backend ML Systems. *SIGMOD Rec.* 55, 1 (2026), 85–95. <https://doi.org/10.1145/3810900.3810916>
- [69] Arnab Phani, Lukas Erlbacher, and Matthias Boehm. 2022. UPLIFT: Parallelization Strategies for Feature Transformations in Machine Learning Workloads. *PVLDB* 15, 11 (2022), 2929–2938. <https://doi.org/10.14778/3551793.3551842>
- [70] Arnab Phani, Benjamin Rath, and Matthias Boehm. 2021. LIMA: Fine-grained Lineage Tracing and Reuse in Machine Learning Systems. In *SIGMOD*. 1426–1439. <https://doi.org/10.1145/3448016.3452788>
- [71] Neoklis Polyzotis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. 2018. Data Lifecycle Challenges in Production Machine Learning: A Survey. *SIGMOD Rec.* 47, 2 (2018), 17–28. <https://doi.org/10.1145/3299887.3299891>
- [72] Liudmila Ostroumova Prokhorenkova, Gleb Gusev, Aleksandr Vorobev, Anna Veronika Drogush, and Andrey Gulin. 2018. CatBoost: unbiased boosting with categorical features. In *NeurIPS*. 6639–6649. <https://proceedings.neurips.cc/paper/2018/hash/14491b756b3a51daac41c24863285549-Abstract.html>
- [73] Fotis Psallidas et al. 2022. Data Science Through the Looking Glass: Analysis of Millions of GitHub Notebooks and ML.NET Pipelines. *ACM SIGMOD Record* 51, 2 (2022), 30–37. <https://doi.org/10.1145/3552490.3552496>
- [74] PyPI Stats. 2026. *daphne - PyPI Download Stats*. <https://pypistats.org/packages/daphne-lib>
- [75] PyPI Stats. 2026. *systemds - PyPI Download Stats*. <https://pypistats.org/packages/systemds>
- [76] Python Software Foundation. 2026. *Free-threading CPython How-To*. Python Software Foundation. <https://docs.python.org/3/howto/free-threading-python.html> Accessed: 2026-02-24.
- [77] Jingang Qu, David Holzmüller, Gaël Varoquaux, and Marine Le Morvan. 2025. TabICL: A Tabular Foundation Model for In-Context Learning on Large Data. In *ICML*. <https://proceedings.mlr.press/v267/qu25d.html>
- [78] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *SIGMOD*. 1981–1984. <https://doi.org/10.1145/3299869.3320212>
- [79] Matthew Rocklin. 2015. Dask: Parallel Computation with Blocked algorithms and Task Scheduling. In *SciPy*. 130 – 136.
- [80] Matthew Russo and Tim Kraska. 2026. Deep Research is the New Analytics System: Towards Building the Runtime for AI-Driven Analytics. *CIDR* (2026).
- [81] Maximilian E. Schüle et al. 2023. Blue Elephants Inspecting Pandas: Inspection and Execution of Machine Learning Pipelines in SQL. In *EDBT*. 40–52. <https://doi.org/10.48786/EDBT.2023.04>
- [82] Andrej Schwanke, Lyubomir Ivanov, David Salinas, Fabio Ferreira, Aaron Klein, Frank Hutter, and Arber Zela. 2025. Improving LLM-based Global Optimization with Search Space Partitioning. *CoRR* abs/2505.21372 (2025). <https://doi.org/10.48550/ARXIV.2505.21372>
- [83] D. Sculley et al. 2015. Hidden Technical Debt in Machine Learning Systems. In *NIPS*. 2503–2511.
- [84] Raunak Shah, Zhaoheng Li, and Yongjoo Park. 2025. QStore: Quantization-Aware Compressed Model Storage. *PVLDB* 19, 3 (2025), 388–398. <https://www.vldb.org/pvldb/vol19/p388-li.pdf>
- [85] Bhushan Pal Singh, Priyesh Kumar, Chiranjit Bhattacharya, and S. Sudarshan. 2026. Efficient Dataframe Systems: Lazy Fat Pandas on a Diet. In *EDBT*. 157–169. <https://doi.org/10.48786/EDBT.2026.14>
- [86] Skrub developers. 2026. *Skrub DataOps API Reference*. <https://skrub-data.org/stable/reference/index.html#dataops>
- [87] Johanna Sommer et al. 2019. MNC: Structure-Exploiting Sparsity Estimation for Matrix Expressions. In *SIGMOD*. 1607–1623. <https://doi.org/10.1145/3299869.3319854>
- [88] Evan Randall Sparks et al. 2017. KeystoneML: Optimizing Pipelines for Large-Scale Advanced Analytics. In *ICDE*. 535–546. <https://doi.org/10.1109/ICDE.2017.109>
- [89] Arvind K. Sajeeth et al. 2011. OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning. In *ICML*. 609–616. https://icml.cc/2011/papers/373_icmlpaper.pdf
- [90] Ji Sun, Guoliang Li, Peiyao Zhou, Yihui Ma, Jingzhe Xu, and Yuan Li. 2025. AgenticData: An Agentic Data Analytics System for Heterogeneous Data. arXiv:2508.05002 [cs.DB] <https://arxiv.org/abs/2508.05002>
- [91] Chang-Yu Tai, Ziru Chen, Tianshu Zhang, Xiang Deng, and Huan Sun. 2023. Exploring Chain of Thought Style Prompting for Text-to-SQL. In *EMNLP*. 5376–5393. <https://doi.org/10.18653/V1/2023.EMNLP-MAIN.327>
- [92] The skrub developers. 2025. skrub: Prepping tables for machine learning. <https://skrub-data.org/stable/>. Accessed: 2026-01-15.
- [93] Marian Tietz, Thomas J. Fan, Daniel Nouri, Benjamin Bossan, and skorch Developers. 2017. *skorch: A scikit-learn compatible neural network library that wraps PyTorch*. <https://skorch.readthedocs.io/en/stable/>
- [94] Edan Toledo et al. 2025. AI Research Agents for Machine Learning: Search, Exploration, and Generalization in MLE-bench. *CoRR* abs/2507.02554 (2025). <https://doi.org/10.48550/ARXIV.2507.02554>
- [95] Ritchie Vink and Polars Contributors. 2026. *Polars: Lightning-fast DataFrame library*. <https://polars.rs/>
- [96] Bing Wang et al. 2025. MAC-SQL: A Multi-Agent Collaborative Framework for Text-to-SQL. In *COLING*. 540–557.
- [97] Hongyu Wang et al. 2025. BitNet: 1-bit Pre-training for Large Language Models. *J. Mach. Learn. Res.* 26, 27 (2025), 1–51. <http://jmlr.org/papers/v26/24-2050.html>
- [98] Liangui Weng et al. 2025. BridgeScope: A Universal Toolkit for Bridging Large Language Models and Databases. *CoRR* abs/2508.04031 (2025). <https://doi.org/10.48550/ARXIV.2508.04031>
- [99] Doris Xin et al. 2018. Helix: Holistic Optimization for Accelerating Iterative Machine Learning. *PVLDB* 12, 4 (2018), 446–460. <https://doi.org/10.14778/3297753.3297763>
- [100] Doris Xin, Hui Miao, Aditya G. Parameswaran, and Neoklis Polyzotis. 2021. Production Machine Learning Pipelines: Empirical Analysis and Optimization Opportunities. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). 2639–2652. <https://doi.org/10.1145/3448016.3457566>
- [101] Bing Xu et al. 2026. VibeTensor: System Software for Deep Learning, Fully Generated by AI Agents. arXiv:2601.16238 [cs.SE]
- [102] Kaichao You, Runsheng Bai, Meng Cao, Jianmin Wang, Ion Stoica, and Ming-sheng Long. 2025. depyf: Open the Opaque Box of PyTorch Compiler for Machine Learning Researchers. *J. Mach. Learn. Res.* 26, 25 (2025), 1–18. <https://doi.org/10.48550/arXiv.2403.13839>
- [103] Bohan Zhai, Canwen Xu, Yuxiong He, and Zhewei Yao. 2025. Optimizing Reasoning for Text-to-SQL with Execution Feedback. In *ACL (Findings)*. 19206–19218. <https://aclanthology.org/2025.findings-acl.982/>
- [104] Lianmin Zheng et al. 2022. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. In *OSDI*. 559–578. <https://www.usenix.org/conference/osdi22/presentation/zheng-lianmin>
- [105] Lianmin Zheng et al. 2024. SGLang: Efficient Execution of Structured Language Model Programs. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*. http://papers.nips.cc/paper_files/paper/2024/hash/724be4472168f31ba1c9ac630f15de8-Abstract-Conference.html
- [106] Yizhang Zhu et al. 2025. A Survey of Data Agents: Emerging Paradigm or Overstated Hype? *CoRR* abs/2510.23587 (2025). <https://doi.org/10.48550/ARXIV.2510.23587>