Palimpzest: Optimizing AI-Powered Analytics with Declarative Query Processing

Chunwei Liu[*], Matthew Russo[*], Michael Cafarella,
Lei Cao[†], Peter Baile Chen, Zui Chen, Michael Franklin[‡],
Tim Kraska, Samuel Madden, Rana Shahout[¦], Gerardo Vitagliano
[*]MIT, [†] University of Arizona, [‡] University of Chicago, [¦] Harvard University
chunwei,mdrusso,michjc,madden,gerarvit@csail.mit.edu, peterbc,chenz429,kraska@mit.edu,
caolei@arizona.edu, mjfranklin@uchicago.edu, rana@seas.harvard.edu

ABSTRACT

A long-standing goal of data management systems has been to build systems which can compute quantitative insights over large collections of unstructured data in a cost-effective manner. Until recently, it was difficult and expensive to extract facts from company documents, data from scientific papers, or metrics from image and video corpora. Today's models can accomplish these tasks with high accuracy. However, a programmer who wants to answer a substantive AI-powered query must orchestrate large numbers of models, prompts, and data operations. In this paper, we present PALIMPZEST, a system that enables programmers to pose AI-powered analytical queries over arbitrary collections of unstructured data in a simple declarative language. The system uses a cost optimization framework — which explores the search space of AI models, prompting techniques, and related foundation model optimizations. PALIMPZEST implements the query while navigating the trade-offs between runtime, financial cost, and output data quality. We introduce a novel language for AI-powered analytics tasks, the optimization methods that PALIMPZEST uses, and the prototype system itself. We evaluate PALIMPZEST on a real-world workload. Our system produces plans that are up to 3.3x faster and 2.9x cheaper than a baseline method when using a single-thread setup, while also achieving superior F1-scores. PALIMPZEST applies its optimizations automatically, requiring no additional work from the user.

1 INTRODUCTION

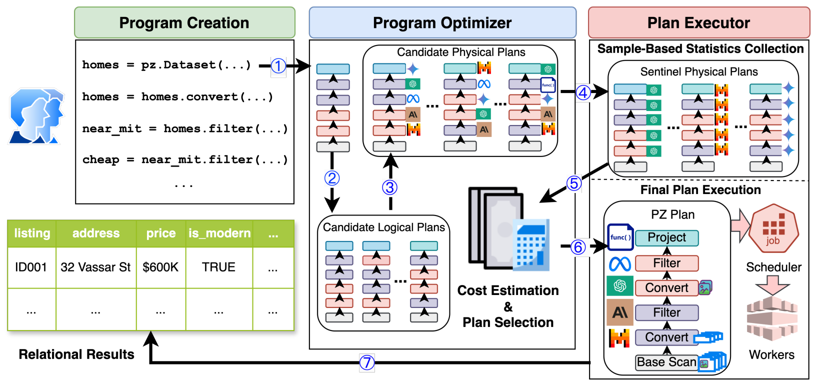
Advances in AI models have driven progress in applications such as question answering [11, 25], chatbots [5], autonomous agents [17, 18], and code synthesis [7, 9]. In many cases, these systems have evolved far beyond posing a simple question to a chat model: they are compound AI systems [24] that combine elements of data processing, such as Retrieval Augmented Generation (RAG); ensembles of different models; multi-step chain-of-thought reasoning; and in many cases, cloud-based modules. It is easy for the runtime, cost, and complexity of these AI systems to escalate quickly, particularly when applied to large collections of documents.

The performance gap between traditional data processing components and AI-powered components is profound. Naively scaling AI systems to process workloads with thousands or millions of inputs requires spending a large amount of runtime and money executing high-end AI models. For instance, a high-quality open source LLM on a modern GPU processes about 100-125 tokens per second, yielding a throughput of *less than 1 KB per second*, assuming that each token averages 5 bytes. OpenAI's GPT-4o mini model costs 15 cents for 1M input tokens, equivalent to processing 5MB of data. These numbers are many orders of magnitude worse than any other component of the modern data processing stack, such as storage, network bandwidth, SQL query processing time, and so on. Thus, optimizing the use of AI components is crucial. Meanwhile, current AI infrastructure is in tremendous technical flux. New models and implementation techniques are published weekly, while model costs and runtimes change frequently. Harnessing the latest advances in model runtime, cost, and quality is complex, error-prone, and requires developers of AI applications to constantly rewrite and retune their systems.

AI engineers face a variety of technical decisions, including optimizing prompt wording and strategy (e.g. chain-of-thought, ReAct [23]), selecting the best model for each subtask while balancing time, cost and quality, and deciding on the best implementation approach for each subtask (e.g. using a foundation model query, synthesized code, locally trained model, or a mixture of agents [20]). Additionally, engineers must manage GPU cache and memory utilization, handle LLM context limits, design efficient execution plans for scaling to larger datasets, and integrate parallelized components for optimal system efficiency. Decisions on integration with external data systems also require careful parameter selection to achieve the best trade-offs between speed, cost, and quality.

The space of possible decisions is vast, and choosing wisely depends on low-level details of the exact task. Moreover, the definition of "best" can change over time: a developer might prefer "cheap and low-quality" execution when quickly testing initial proof-of-concept ideas, then switch to "costly but high-quality" for customer deployment. Finally, the changing technical landscape means that the optimization choices made today might be obsolete tomorrow.

Our Goal: The key insight is that machines, not human engineers, should decide how best to optimize AI applications. Engineers should be able to write AI programs at a high level of abstraction and rely on an optimizer to find a physical implementation that best fits their use case. This concept echoes the circumstances that led to the creation of the relational database query optimizer in the 1970s — a period marked by the need for performance enhancements amid significant technological shifts. While today's technical challenges differ, the principle of declarative program optimization remains profoundly relevant.



```
homes = pz.Dataset(...)
homes = homes.convert(...)
near_mit = homes.filter(...)
cheap = near_mit.filter(...)
...
```

listing	address	price	is_modern	...
ID001	32 Vassar St	$600K	TRUE	...
...	...	...	...	...

Algorithm 1 Optimized Plan Selection

```
Require: code, policy                                          # Step ①
 1: logicalPlans = getLogicalPlans(code)                       # Step ②
 2: sentinels = getSentinelPlans(logicalPlans)                 # Step ③
 3:
 4: planStats = {}
 5: for 0...NUM_SAMPLES do
 6:     input = getSampledInput()
 7:     stats = runAndProfile(sentinels, input)                # Step ④
 8:     planStats.update(stats)                                # Step ⑤
 9: end for
10:
11: physicalPlans = getPhysPlans(logicalPlans, planStats)
12: reducedPlans = naivePrune(physicalPlans)
13: frontier = scoreAndPrunePlans(reducedPlans, planStats)
14: return chooseBestPlan(frontier, policy)                    # Step ⑥
```

Figure 1: Overview of the PALIMPZEST system. Users write their program(s) in a declarative language which undergoes compilation ①, logical plan generation ②, and physical plan generation ③. Subsequent steps involve profiling sample plans ④ and analyzing performance statistics to estimate costs ⑤. The optimal plan, tailored to user-specified preferences (e.g. to maximize quality at fixed cost), is selected ⑥ and executed, delivering relational results to the user ⑦.

In this paper, we propose PALIMPZEST[1], a system that enables engineers to write succinct, declarative code that can be compiled into optimized programs. PALIMPZEST is designed to optimize a broad class of data-intensive AI workloads we term Semantic Analytics Applications (defined in Section 2), which includes large-scale information extraction, data integration, discovery from scientific papers, image understanding tasks, and multimodal analytics. As shown in Figure 1, when running an input user program, PALIMPZEST considers a range of logical and physical optimizations, then yields a set of possible concrete executable programs. PALIMPZEST estimates the cost, time, and quality of each one, then chooses a program based on runtime user preferences. The system is designed to be extensible so that new optimizations can be easily added in the future. Just as the RDBMS allowed users to write database queries more quickly and correctly than they could by writing traditional code, PALIMPZEST will allow engineers to write better AI programs more quickly than they could unaided.

Our Approach: A core challenge in building PALIMPZEST is creating an optimizer that can marshal many optimizations to meet a user's cost, runtime, and quality goals. By using a high-level, type-focused, and declarative language, PALIMPZEST can exploit many optimizations that are not otherwise available. Another key challenge involves designing a programming interface that simultaneously enables engineers to express the broadest possible set of AI programs while imposing structure on their programs that the optimizer can exploit. To this end, we created a Python library that implements a thin abstraction over an underlying relational algebra. The core intellectual difference between PALIMPZEST and previous database-style systems is the addition of the relational **convert** operator, which transforms an object of one user-defined schema to another. This operator, implemented using various methods, often based on foundation models — allows the programmer to implement many AI tasks in a relational and optimizable style.

Contributions: In this paper we:

- Introduce Semantic Analytics Applications (SAPPs), a prominent yet demanding class of data-intensive AI workloads that can benefit from many traditional ideas in data management. Addressing them requires a range of solutions and abstractions. (Section 2.)
- Discuss the architecture of PALIMPZEST, including the **convert** operator and the optimization module, and explore how it is designed to tackle the challenges associated with SAPPs. (Section 3.)
- Describe a set of physical and logical optimizations implemented and evaluated in our prototype. (Section 4.)
- Present experimental results demonstrating that with these optimizations, PALIMPZEST can execute SAPP workloads in both single and parallel modes, while offering a range of trade-offs that are more favorable than those of a baseline approach. (Section 5.)

2 WORKLOADS

Before we describe the details of the PALIMPZEST system, it is useful to discuss the workloads PALIMPZEST aims to support, in particular, the *semantic analytics applications* — or, **SAPPs.**

SAPPs (1) combine traditional data processing and AI elements, (2) potentially process large amounts of data, and (3) can be decomposed into a tree of operations over sets of data objects. As a running example, consider a Real Estate Search task. In this task, the user wants to search all of the real estate listings near Cambridge, MA to find a house that is (a) modern and attractive, (b) within two miles of MIT, and (c) within a certain price range.

This task meets the first criterion as it likely requires a vision or text model to analyze textual listings and images to determine if a house is "modern and attractive," and to extract price and location data from text. The second criterion is met with Zillow currently displaying 1,327 home listings in the Cambridge and Boston areas, each with a text description and typically over 10 images. Finally, the task can be decomposed into a tree of operations over the listing text and images that compute fields (e.g. address, price, attractiveness) and apply selection predicates.

Optimization Challenges. The ideal implementation of an AI system for a SAPP workload will jointly optimize its AI- and conventional data processing elements. For example, in Real Estate Search, an extremely naive implementation might waste time and money processing images of apartments to test whether they are

[1]Like an ancient palimpsest, our system entails constant revision and rethinking, in our case by the optimizer. Only zestier! Please check the open-source code https://github.com/mitdbg/palimpzest

```
1   import palimpzest as pz
2
3   class Email(pz.TextFile):
4     """Represents an email, subclass of text file"""
5     sender = pz.StringField(desc="The email address of the
      ↪  sender", required=True)
6     subject = pz.StringField(desc="The subject of the email",
      ↪  required=True)
7
8   # define logical plan
9   emails = pz.Dataset(source="enron-emails", schema=Email)
10  emails = emails.filter("The email is not quoting from a news
    ↪  article or an article ...")
11  emails = emails.filter("The email refers to a fraudulent scheme
    ↪  (i.e., 'Raptor', ...")
12
13  # user specified policy and plan execution
14  policy = pz.MinimizeCostAtFixedQuality(min_quality=0.8)
15  results = pz.Execute(emails, policy=policy)
```

Figure 2: PALIMPZEST code for an email processing task. The user program constructs a sequence of logical operations (e.g., scan, convert, filter) which define an initial logical plan. The user also specifies a policy, i.e., a preference for where to operate on the Pareto frontier of physical plans. The plan and policy are fed into pz.Execute, which compiles the initial logical plan into a set of physical plans, chooses the physical plan that is Pareto-optimal for the given policy, and executes that plan.

"modern and attractive" — only to discard them when they fail to meet text-based constraints. A slightly more sophisticated implementation would reorder the plan so that it applies the cheapest and most restrictive constraints first, thereby avoiding the time and expense of invoking a vision model on candidates that will later be discarded. The decisions around which optimizations to employ are unique and specific for each new task and dataset.

Optimizing an AI system for SAPP workloads involves accurately predicting runtime, cost, and quality for each data processing step, which is particularly challenging for semantic tasks. For instance, predicting the performance of a vision model requires understanding the input/output tokens per record and the total processed records. Additionally, assessing output quality without labeled data often relies on error-prone heuristics or costly comparisons with a superior model. The system must also forecast how various physical optimizations, like using multiple models or reducing image resolution, will affect these metrics. We discuss how PALIMPZEST performs cost estimation and searches the space of physical plans in Section 3.

3 OVERVIEW

PALIMPZEST treats its programs primarily as a form of computing *relational views*: the user specifies a (set of) input relation(s) (called `Datasets`) and a target output relation to be computed. Each relation has a corresponding `Schema`. The user also describes a series of operations to be applied that transform the input(s) into the output. Figure 2 shows a short example program.

Unlike SQL, PALIMPZEST is intended mainly to be used as a library in a host language: the current implementation is in Python although there is no reliance on any language-specific features.

The logical relational operators supported by PALIMPZEST are shown in Table 1. Some of the operators, such as groupby, aggregation, and limit, are not showcased in Figure 2. These operators currently follow their standard definitions from the data management literature, but in the future groupby and aggregation could also be implemented using AI-based operations. We emphasize that *users don't need to specify the implementation of a particular operation*. Instead, it is the system's job to implement and perform the operation. By employing a range of different AI models and generation techniques, PALIMPZEST can automatically compute an implementation for the data-flow corresponding to a user's program.

Table 1: PALIMPZEST's full relational algebra includes operators which produce multiple relations (e.g., *Group by*).

Project : π(rel., cols)	Group by : Γ(rel., group_cond., agg.)
Select : σ(rel., predicate)	Convert : χ(rel., schema_a, schema_b)
Limit : L(rel., limit)	Aggregate : α(rel., func)

Convert: The *Convert* operator transforms a typed data object with schema A into a new object with a different schema B, by computing the set of fields in schema B which do not explicitly exist in schema A. One physical implementation of convert uses an LLM: the fields of schema A will be marshaled into a prompt as key-value pairs along with the user-provided field descriptions, and the LLM will be asked to produce the output field(s) for schema B.

The correct behavior of a convert operation is often implied only by the user's specification of the input and output `Schemas`. For example, in Figure 2, the convert operator (implied by the call to **pz.Dataset** with the specified schema) extracts the `sender` and `subject` fields from raw text to produce an `Email`.

Cost Optimization Framework: At its core, PALIMPZEST enables users to define and execute logical *plans*, which are sequences of relational operations on datasets. The declarative nature of these plans often leaves the execution details underspecified. PALIMPZEST's cost optimizer plays a crucial role in identifying (near) Pareto-optimal physical implementations of these plans that meet user preferences. The process from program implementation to the selection and execution of the most cost-effective plan is illustrated in Figure 1 and algorithmically detailed in Algorithm 1.

Developers begin by writing declarative programs, such as the one shown in Figure 2, which lazily constructs a sequence of data-loading and processing logical operations. On line 14, the developer specifies a `policy` which determines how the system should choose among multiple Pareto-optimal implementations of the logical plan. (In this case, the plan with the lowest expected financial cost, subject to a lower bound on quality, is preferred.) The current policies are predefined to focus on runtime, cost, or quality, with customizable parameters set by users. Upon executing `Execute()` on line 15, this chain is compiled into an initial logical plan by the **Program Optimizer** (step ①; Algorithm 1, line 1). This plan undergoes logical optimization to generate functionally equivalent plans with varying cost, runtime, and quality trade-offs (step ②; Algorithm 1, line 2). For our prototype implementation, the only logical optimization we consider is filter pushdown, but more optimizations (e.g. join re-ordering) could also be considered in the future.

For each logical plan, the Program Optimizer generates a larger set of candidate physical plans (step ③). This stage includes decisions specific to AI systems, such as model choice and prompt generation. For instance, combining multiple subtasks into a single LLM query to reduce token duplication is an optimization similar to FrugalGPT's *query concatenation* method [4]. In the worst case, this would require enumerating an exponentially large number of physical plans and estimating the performance of each one. However, in practice, we make the simplifying assumption that operators are independent. This simplifies our problem of estimating the runtime, cost, and quality of each operator, which we can then compose to estimate the runtime, cost, and quality of a much larger space of plans. Our assumption of operator independence is strong, however it makes estimating plan costs for an exponentially large space of plans tractable and still results in the optimizer selecting plans which perform well in practice (as we will show in Section 5).

To efficiently estimate per-operator statistics, PALIMPZEST executes a set of *sentinel plans* on a small set of validation examples, where each plan uses a different model for its LLM-enabled operations. In our prototype, the validation examples were the first N records of the workload (for small N relative to the workload size). For each operator, we directly observe a distribution of runtimes and per-record cost of execution. We also infer a distribution of the quality of each operator by comparing its performance to that of the operator which uses the "champion model" (e.g. GPT-4 in our experiments, but generally the most expensive and/or highest quality model). While there is no guarantee that the champion model's output is correct, computing quality in this manner guarantees that (at worst) we can identify operators and plans which provide similar output quality to the champion model at a fraction of the cost. (We may also validate operator quality against groundtruth labels, although we have left this to future work). Given these per-operator estimates of runtime, cost, and quality, PALIMPZEST estimates the quality of each physical plan by composing its per-operator estimates. Specifically, it sums the runtime and cost of each operator and takes the product of their qualities. For quality metrics such as accuracy, taking the product of qualities is consistent with our operator independence assumption. The Program Optimizer then generates a potentially large set of programs consistent with the user's input, varying across the optimization space of runtime, financial cost, and quality. However, PALIMPZEST still needs to compute estimates of these metrics for each physical plan (steps ④ and ⑤; Algorithm 1, lines 4-9). The **Plan Executor** executes a small set of **sentinel plans** to gather sample data on plan execution statistics. The quality of plan outputs is then evaluated against the output from a "champion" plan, at the granularity of an individual operator. (We currently test against the plan which uses GPT-4 for every operation).

Finally, the Program Optimizer selects the best physical plan based on its plan estimates and the policy provided by the user (step ⑥; Algorithm 1, line 14). This choice is then executed by the Plan Executor, which utilizes computation and financial resources, potentially drawing on many external models and data service providers (step ⑦).

This optimization framework helps PALIMPZEST hypothesize, select, and execute plans better optimized for user preferences.

4 PROGRAM OPTIMIZATION

Managing and exploiting a large space of useful optimizations is PALIMPZEST's core feature. In this section, we describe the key logical and physical optimizations that can be used by PALIMPZEST.

Logical Optimizations: PALIMPZEST employs logical optimizations to refine the logical plan derived from a user's program, aiming to enhance runtime efficiency and reduce costs. These optimizations generate logically equivalent plans that may differ in execution cost and efficiency due to varying operation costs and filter selectivities.

The implemented optimizations include: (1) *Filter Reordering*, permuting the order of selection filters in a logical plan. By evaluating all possible permutations, PALIMPZEST can potentially minimize the number of records processed, even though initial selectivities are unknown and estimated later during execution. (2) *Convert Reordering*, rearranging convert operations within the plan. It is particularly effective when expensive convert operations can be postponed until after selective filters that do not depend on their outputs, thus saving computation. Dependencies between operations are considered to ensure plan equivalence. These optimizations ensure that the logical plans remain semantically equivalent while potentially differing in their physical execution, which is crucial for maintaining the integrity of the results as described below.

Physical Optimizations: PALIMPZEST unlocks numerous low-level optimizations that are unavailable through standard LLM API services, thanks to our declarative programming framework. We implement and evaluate these optimizations to demonstrate their effectiveness:

(1) *Model Selection*, i.e., choosing different models and LLM services to perform different operations. PALIMPZEST can decompose high-level programs into smaller operations and choose the most appropriate model for each. It might be fine to use a cheap, small, fast model for easy operations, and only use the expensive model for harder ones. The idea is simple, but implementing it is not: because (1) a single program can comprise many operations, (2) the exact operation decomposition can change depending on other optimizations, (3) the difficulty of an operation can change over time, and (4) model quality can fluctuate as LLM services make updates. This optimization can easily become burdensome without PALIMPZEST's help.

(2) *Code Synthesis*, i.e., dynamically generating synthesized code to handle specific operations where deep semantic understanding is not crucial. By substituting LLM calls with synthesized functions based on a set of sample inputs, PALIMPZEST significantly reduces runtime and costs. PALIMPZEST uses an LLM to analyze sample inputs and generate functions for conversions, optimizing performance and resource utilization.

(3) *Multi-data Prompt Marshaling*, i.e., determining the optimal way to map user operations to LLM prompts, such as processing data in a row-centric or column-centric manner. Row-centric processing involves a single LLM call for multiple outputs from one input record, while column-centric processing may enhance accuracy by focusing on one field at a time. The choice between these approaches depends on various runtime factors, and PALIMPZEST dynamically profiles and optimizes based on these considerations and user-specified policies.

(4) *Input Token Reduction*, i.e., minimizing the input data required for specific operations, which enhances both cost-efficiency and execution speed. For instance, in document processing, PALIMPZEST can determine essential input regions for converting a PDF to a `ScientificPaper`, significantly reducing token usage. This process, similar to selecting key excerpts in a "micro-RAG" task, is facilitated by PALIMPZEST's ability to learn operator properties at the schema level, optimizing input dynamically during program execution. This

method's applicability is limited in scenarios without clear schemas, such as in the naive chat-processing use case.

PALIMPZEST is actively exploring a range of additional optimizations. These include reducing output tokens and strategically scheduling and batching requests that share similar characteristics to improve memory and cache utilization, reduce wait times, and increase throughput, all while maintaining data quality with minimal compromise.

5 EVALUATION

Our Prototype: We have implemented an early PALIMPZEST prototype in about 9,200 lines of Python code. It implements the operators in Table 1, and can run many programs, although in this paper we primarily report detailed results for the Real Estate Search due to its multi-modality, diverse operators, and challenging optimization space. A comprehensive workload evaluation is available in our technical report [12]. We have implemented the optimizations described in Section 4: model selection, code synthesis, multi-data prompt marshaling, and input token reduction. We currently test the system using the `gpt-3.5-turbo-0125`, `gpt-4-0125-preview`, and `gpt-4-vision-preview` OpenAI models [14] and the `Mixtral-8x7B-Instruct-v0.1` model served by the Together.ai API [2]. We also use the Modal online service [1] for bulk non-AI function execution, such as parallel PDF processing and equation image extraction and conversion.

PALIMPZEST is implemented using the iterator model. Thus, plan execution proceeds one record at a time with each operator blocking until it receives the necessary input record(s) from its source operator(s). For clarity's sake, most of our experiments report simple single-threaded execution time, so we can better show the work saved by system optimizations. However, many operations — including the convert operator — have parallel implementations that take advantage of parallelism offered by the underlying service or hardware, and we report some experiments with parallelism enabled.

Evaluation Workload: We evaluated PALIMPZEST using the Real Estate Search workload, which consists of 100 manually scraped real estate listings from Boston and Cambridge, MA. Each listing included descriptions in natural language and three images. We manually labeled each listing based on whether it was in a specific geographic area, within a given price range, and "modern and attractive with lots of natural sunlight." Of the listings, 23 met all criteria.

Optimization Trade-offs: Our first experimental claim is that PALIMPZEST can use its three optimization strategies to create a set of physical plans that offer appealing trade-offs regardless of the user policy. To evaluate this claim, we ran PALIMPZEST up to (but not including) the final step in Algorithm 1. We then took the set of frontier plans, three baseline plans, and the top-*k* plans from the *reducedCandidates* that were closest to the approximated Pareto frontier, such that we ultimately executed 20 plans in total. Our three baselines were the naive physical plans for each workload which used only GPT-4, GPT-3.5, or Mixtral-8x7B, respectively. We chose these plans as baselines because they represent the performance one might expect if they naively implemented the system using a single model without tuning or optimizing individual operators. The optimization process took 13.1s.

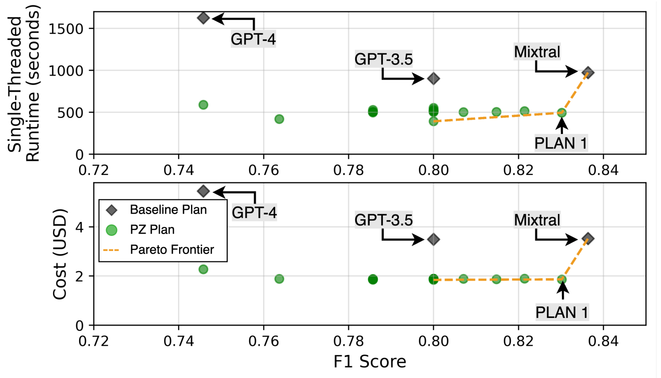


Figure 3: Performance of different plans (bottom-right is better). Black diamonds are naive plans and green circles are optimized PALIMPZEST plans. PALIMPZEST plans are consistently on the Pareto frontiers of runtime vs. quality and cost vs. quality.

Figure 3 shows the runtime, cost, and quality observed from the execution of all the aforementioned plans. Plans closer to the bottom-right are better. During sample-based statistics collection, we used 5% of the workload's total input size to run our sentinel plans, which gathered data to help estimate the cost of all plans.

We found that PALIMPZEST is capable of creating useful plans at a number of different points in the trade-off space. PALIMPZEST demonstrated its ability to generate physical plans that offer significant improvements over baseline plans. Specifically, PALIMPZEST was able to produce physical plans (e.g. PLAN 1) that obtained 3.3x lower runtime, 2.9x lower cost, and up to 1.1x better F1-score than the GPT-4 baseline. These performance improvements are especially impressive considering the majority of the cost and runtime on this workload are dominated by calls to the vision model — which cannot be optimized away using the methods in our current prototype. In the future, we will explore options for visual processing optimizations.

The physical plan for PLAN 1 achieved improved performance relative to the GPT-4 baseline through the combination of two optimizations. First, the plan re-ordered the execution of the convert and filter operations such that the text-based operators were executed before the image-based operators. This provided significant runtime and cost savings by avoiding calls to the GPT-4 vision model altogether. Second, the plan used input token reduction to trim the real-estate listing text by 50%. This technique was particularly effective, as the home address and listing price regularly appear at the top of the text for the listing.

It is worth mentioning that the GPT-4 baseline plan occasionally failed to format its output(s) correctly, which contributed to its lower F1-score. Implementing a regular expression to enforce formatted output, as used in SGLang [26], could be a potential solution. However, users rarely know ahead of time whether a model will succeed for a given prompt and input. A key benefit of using PALIMPZEST is that it can detect bad plans for the user at runtime, and select better ones instead.

We have shown that PALIMPZEST can generate plans like PLAN 1, which provides users with compelling performance trade-offs. However, this does not automatically confirm that PALIMPZEST's optimizer will choose these plans during runtime. In the following

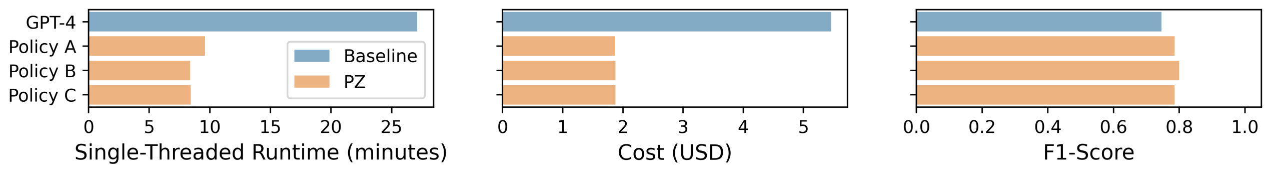


Figure 4: PALIMPZEST selects plans with better costs, better runtimes, and comparable F1-scores relative to a naive GPT-4 baseline.

Table 2: PALIMPZEST selects plans which satisfy (or nearly satisfy) the given policy constraint. Constraints were chosen to be feasible but non-trivial to satisfy.

Policy	Constraint	Cost	Runtime (s)	F1-score
Max F1	Cost< \$3	**\$1.87**	577	0.79
Max F1	Time< 600 s	\$1.88	**502**	0.80
Min Cost	F1> 0.80	\$1.87	505	0.79

section, we demonstrate that for a variety of policies, PALIMPZEST's cost optimizer does indeed identify and select such plans.

Cost Optimizer and Performance Gains: Our second experimental claim is that PALIMPZEST can identify plans that have better end-to-end runtime, cost, and quality than a naive plan that uses the same state-of-the-art language model for each operation. To test this, we executed PALIMPZEST as outlined in Algorithm 1, including the crucial final step of selecting the optimal plan for a given policy. We implemented three specific policies: **Policy A** aimed to maximize quality while keeping costs below \$3. **Policy B** sought to maximize quality with a runtime constraint of less than 600s. **Policy C** focused on minimizing cost while ensuring an F1-score > 0.8. These thresholds were designed to be challenging yet achievable based on our findings in the previous experiments and detailed in Table 2.

Figure 4 presents our results across three performance metrics, with each metric displayed in a separate column. The results are further divided by the physical plan selected by the optimizer, with each plan occupying a separate row. We compared the plans chosen by PALIMPZEST to a baseline plan, which employs GPT-4 for all conversion and filtering operations.

Overall, we found that PALIMPZEST identifies plans in the space of physical candidates which (1) offer significant performance improvements over the GPT-4 baseline, (2) generally satisfy policy constraints, and (3) have speedups and cost savings which outweigh the optimization overhead. The plans chosen by PALIMPZEST achieved (on average) 67.5% lower runtime, 65.7% lower cost, and 6% better F1-score than the baseline GPT-4 plan.

We also evaluated PALIMPZEST using parallel implementations of the convert and filter operations, achieving substantial runtime reductions while maintaining competitive costs and F1-scores compared to a single-threaded baseline. Detailed results are available in our technical report [12]. PALIMPZEST's straightforward parallel processing abstractions significantly enhance performance without additional user effort.

6 RELATED WORK

The literature on programming frameworks for LLMs and foundation models is extensive, focusing on prompt management and task-specific functionalities. LangChain [6] and LlamaIndex [13] provide libraries for managing prompt templates and user-provided examples, yet they lack higher-level task representations. DSPy [8] enhances prompt quality by translating high-level LLM goals into concrete prompts, differing from PALIMPZEST in its focus and absence of performance and cost considerations. SGLang [26] merges prompt templating and output constraints with performance enhancements like parallel execution, potentially complementing PALIMPZEST in future integrations. However, the structured generation language remains low-level, requiring developers to make numerous manual decisions.

SkyPilot [22] optimizes the deployment of ML tasks across cloud platforms focusing on cost efficiency, a goal shared with PALIMPZEST, though PALIMPZEST also seeks efficient program-specific implementations. FrugalGPT [4] optimizes atop LLM platforms [14], aligning with PALIMPZEST's efficiency objectives, but lacks clarity in task description and broader AI program integration. AutoGen [21] envisions LLM applications as agent conversations, differing significantly from PALIMPZEST's data processing model. This vision of LLM applications is driven by a fundamentally different model from our own, which is based more in the data processing tradition.

In query planning, Caesura [19] generates executable query plans from natural language descriptions for multimodal data, but lacks optimization capabilities, a gap PALIMPZEST addresses with explicit programmer input for logic construction. ZenDB [10] focuses on structural optimizations for document queries, sharing objectives with PALIMPZEST but limited to text data and logical optimizations. Evaporate [3] examines cost-quality trade-offs in LLM workloads using static prompts and code generation, similar to PALIMPZEST's strategies but more narrowly focused on information extraction. Lotus [16] offers extensive semantic operations with specialized optimizations. However, its approach to query-level optimization relies on user specifications and lacks automated logical and physical optimization processes. Recent work [15] employs LLMs as digital crowd workers to refine prompt engineering, replacing human tasks to enhance efficiency and manage trade-offs in LLM operations. Unlike crowd-based systems, PALIMPZEST leverages LLMs to offer broader flexibility (e.g., rapid code generation), addresses unique challenges like token optimization, and avoids issues with inconsistent human labor.

7 CONCLUSION

PALIMPZEST allows users to program AI workloads using a declarative language and optimizes them efficiently, enabling focus on application logic rather than AI model intricacies. It integrates logical and physical planning for optimal execution and uses sample data to balance runtime, cost, and quality. This makes PALIMPZEST valuable for developers and organizations leveraging modern AI efficiently and affordably, particularly in developing SAPPs.

8 ACKNOWLEDGMENTS

We are grateful for the support from the DARPA ASKEM Award HR00112220042, the ARPA-H Biomedical Data Fabric project, NSF DBI 2327954, a grant from Liberty Mutual, and the Amazon Research Award. Additionally, our work has been supported by contributions from Amazon, Google, and Intel as part of the MIT Data Systems and AI Lab (DSAIL) at MIT, along with NSF IIS 1900933. This research was sponsored by the United States Air Force Research Laboratory and the Department of the Air Force Artificial Intelligence Accelerator and was accomplished under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Department of the Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

REFERENCES

[1] 2024. Modal.com API. https://modal.com. Accessed: 2024-04-30.

[2] 2024. Together.ai. https://together.ai. Accessed: 2024-04-30.

[3] Simran Arora, Brandon Yang, Sabri Eyuboglu, Avanika Narayan, Andrew Hojel, Immanuel Trummer, and Christopher Ré. 2023. Language models enable simple systems for generating structured views of heterogeneous data lakes. *arXiv preprint arXiv:2304.09433* (2023).

[4] Lingjiao Chen, Matei Zaharia, and James Zou. 2023. Frugalgpt: How to use large language models while reducing cost and improving performance. *arXiv preprint arXiv:2305.05176* (2023).

[5] Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E Gonzalez, et al. 2023. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality. *See https://vicuna. lmsys. org (accessed 14 April 2023)* 2, 3 (2023), 6.

[6] LangChain Contributors. 2024. LangChain: Open Source Framework for Building Language Models. https://github.com/LangChain/langchain. Accessed: 2024-04-18.

[7] Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, et al. 2023. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352* (2023).

[8] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T Joshi, Hanna Moazam, et al. 2023. Dspy: Compiling declarative language model calls into self-improving pipelines. *arXiv preprint arXiv:2310.03714* (2023).

[9] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with AlphaCode. *Science* 378, 6624 (Dec. 2022), 1092–1097. https://doi.org/10.1126/science.abq1158

[10] Yiming Lin, Madelon Hulsebos, Ruiying Ma, Shreya Shankar, Sepanta Zeigham, Aditya G Parameswaran, and Eugene Wu. 2024. Towards Accurate and Efficient Document Analytics with Large Language Models. *arXiv preprint arXiv:2405.04674* (2024).

[11] Chunwei Liu, Enrique Noriega-Atala, Adarsh Pyarelal, Clayton T Morrison, and Mike Cafarella. 2024. Variable Extraction for Model Recovery in Scientific Literature. *arXiv preprint arXiv:2411.14569* (2024).

[12] Chunwei Liu, Matthew Russo, Michael Cafarella, Lei Cao, Peter Baille Chen, Zui Chen, Michael Franklin, Tim Kraska, Samuel Madden, and Gerardo Vitagliano. 2024. A Declarative System for Optimizing AI Workloads. *arXiv preprint arXiv:2405.14696* (2024).

[13] Jerry Liu. 2022. LlamaIndex. https://doi.org/10.5281/zenodo.1234

[14] OpenAI. 2024. OpenAI API. https://platform.openai.com. Accessed: 2024-04-30.

[15] Aditya G Parameswaran, Shreya Shankar, Parth Asawa, Naman Jain, and Yujie Wang. 2023. Revisiting prompt engineering via declarative crowdsourcing. *arXiv preprint arXiv:2308.03854* (2023).

[16] Liana Patel, Siddharth Jha, Carlos Guestrin, and Matei Zaharia. 2024. LOTUS: Enabling Semantic Queries with LLMs Over Tables of Unstructured and Structured Data. *arXiv preprint arXiv:2407.11418* (2024).

[17] Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. 2023. Gorilla: Large language model connected with massive apis. *arXiv preprint arXiv:2305.15334* (2023).

[18] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2024. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems* 36 (2024).

[19] Matthias Urban and Carsten Binnig. 2023. CAESURA: Language Models as Multi-Modal Query Planners. *arXiv preprint arXiv:2308.03424* (2023).

[20] Junlin Wang, Jue Wang, Ben Athiwaratkun, Ce Zhang, and James Zou. 2024. Mixture-of-Agents Enhances Large Language Model Capabilities. *arXiv preprint arXiv:2406.04692* (2024).

[21] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. 2023. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. *arXiv preprint arXiv:2308.08155* (2023).

[22] Zongheng Yang, Zhanghao Wu, Michael Luo, Wei-Lin Chiang, Romil Bhardwaj, Woosuk Kwon, Siyuan Zhuang, Frank Sifei Luan, Gautam Mittal, Scott Shenker, and Ion Stoica. 2023. SkyPilot: An Intercloud Broker for Sky Computing. In *Symposium on Networked Systems Design and Implementation*. https://api.semanticscholar.org/CorpusID:258559424

[23] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *International Conference on Learning Representations (ICLR)*.

[24] Matei Zaharia, Omar Khattab, Lingjiao Chen, Jared Quincy Davis, Heather Miller, Chris Potts, James Zou, Michael Carbin, Jonathan Frankle, Naveen Rao, and Ali Ghodsi. 2024. The Shift from Models to Compound AI Systems. https://bair.berkeley.edu/blog/2024/02/18/compound-ai-systems/.

[25] Jiahao Zhang, Haiyang Zhang, Dongmei Zhang, Yong Liu, and Shen Huang. 2024. End-to-End Beam Retrieval for Multi-Hop Question Answering. In *2024 Annual Conference of the North American Chapter of the Association for Computational Linguistics*. https://arxiv.org/abs/2308.08973

[26] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. 2023. Efficiently programming large language models using sglang. *arXiv preprint arXiv:2312.07104* (2023).