SHORT-PAPER

PalimpChat: Declarative and Interactive AI analytics

CHUNWEI LIU, Massachusetts Institute of Technology, Cambridge, MA, United States

GERARDO VITAGLIANO, Massachusetts Institute of Technology, Cambridge, MA, United States

BRANDON ROSE

MATTHEW PRINTZ

DAVID ANDREW SAMSON

MICHAEL JOHN CAFARELLA, Massachusetts Institute of Technology, Cambridge, MA, United States

Open Access Support provided by:

Massachusetts Institute of Technology

Published: 22 June 2025

SIGMOD/PODS '25: International Conference on Management of Data
June 22 - 27, 2025
Berlin, Germany

Conference Sponsors:
SIGMOD

PalimpChat: Declarative and Interactive AI Analytics

Chunwei Liu*
MIT CSAIL
Cambridge, MA, USA
chunwei@mit.edu

Gerardo Vitagliano*
MIT CSAIL
Cambridge, MA, USA
gerarvit@mit.edu

Brandon Rose
Jataware
Washington, DC, USA
brandon@jataware.com

Matthew Printz
Jataware
Washington, DC, USA
matt@jataware.com

David Andrew Samson
Jataware
Washington, DC, USA
david@jataware.com

Michael Cafarella
MIT CSAIL
Cambridge, MA, USA
michjc@csail.mit.edu

Abstract

Thanks to the advances in generative architectures and large language models, data scientists can now code pipelines of AI operations to process large collections of unstructured data. Recent progress has seen the rise of declarative AI frameworks (e.g., Palimpzest, Lotus, and DocETL) to build optimized and increasingly complex pipelines, but these systems often remain accessible only to expert programmers. In this demonstration, we present PalimpChat, a chat-based interface to Palimpzest that bridges this gap by letting users create and run sophisticated AI pipelines through natural language alone. By integrating Archytas, a ReAct-based reasoning agent, and Palimpzest's suite of relational and LLM-based operators, PalimpChat provides a practical illustration of how a chat interface can make declarative AI frameworks truly accessible to non-experts.

Our demo system is publicly available online. At SIGMOD'25, participants can explore three real-world scenarios—scientific discovery, legal discovery, and real estate search—or apply PalimpChat to their own datasets. In this paper, we focus on how PalimpChat, supported by the Palimpzest optimizer, simplifies complex AI workflows such as extracting and analyzing biomedical data.

CCS Concepts

• **Information systems** → **Data analytics**; • **Computing methodologies** → **Natural language processing**.

Keywords

Compound AI, LLMs, AI programming, Chat-box

ACM Reference Format:
Chunwei Liu*, Gerardo Vitagliano*, Brandon Rose, Matthew Printz, David Andrew Samson, and Michael Cafarella. 2025. PalimpChat: Declarative and Interactive AI Analytics. In *Companion of the 2025 International Conference on Management of Data (SIGMOD-Companion '25), June 22–27, 2025, Berlin, Germany*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3722212.3725122

* Denotes equal first author contribution.

SIGMOD-Companion '25, Berlin, Germany

ACM ISBN 979-8-4007-1564-8/2025/06
https://doi.org/10.1145/3722212.3725122

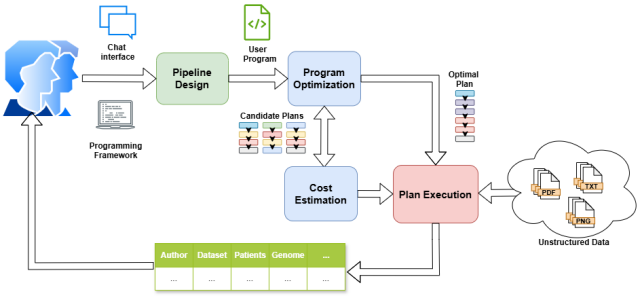

Figure 1: An overview of data processing using PalimpChat and Palimpzest

1 Introduction

Generative AI has transformed our interactions with data and computing by introducing Large Language Models (LLMs) capable of complex tasks such as scientific discovery, multimodal extraction, reasoning, and code synthesis [2, 5, 11]. Yet, implementing such functionality often requires coordinating multiple software stacks—vector databases [4], relational operators [7], and novel programming practices like prompt engineering [3]. As these AI systems grow in scope, users face major challenges around runtime cost, resource consumption, and the coordination of diverse tools.

Palimpzest is one of several new AI systems (e.g., Lotus [8], DocETL [9], Caesura [10]) that adopt a declarative approach for building AI pipelines over unstructured datasets. These systems greatly simplify this task for proficient programmers by allowing them to write high-level *logical* specifications - and let automated optimization produce efficient *physical* implementations of these pipelines [6]. However, this programming model remains inaccessible to less technical users. To bridge that gap, we propose PalimpChat, a chat-based interface that integrates Palimpzest with Archytas, a framework to design reasoning agents. Our system lets users specify data processing pipelines in natural language, while still benefiting from the underlying automated optimization provided by declarative AI frameworks.

In this demonstration, we showcase how both developers and non-experts can collaboratively harness our system. All users, regardless of their coding expertise, can design and execute pipelines entirely through PalimpChat. Furthermore, expert users can either further iterate on the code produced using the chat interface, or program their pipelines directly within Palimpzest. To illustrate the versatility of this approach, we present a real-world use case

in scientific discovery, where medical researchers want to identify relevant genomic and proteomic datasets in cancer-related papers. Conference attendees can similarly explore additional cases or upload their own unstructured data for processing.

PalimpChat is an early, but promising, prototype, and we welcome suggestions for further optimizations and features. We encourage readers to consult our extended technical description in [6], experiment with our open-source Palimpzest code [1], interact with the PalimpChat demo online[2] or locally [3], or view the video[4].

The remainder of this paper is structured as follows: section 2 describes the core architecture of Palimpzest and the reasoning framework driving PalimpChat. Section 3 demonstrates a concrete usage scenario through the chat-based interface, and Section 4 briefly concludes the paper.

2 System Architecture

The PalimpChat system is composed of two interconnected systems: the Palimpzest framework, a declarative system to build and automatically optimize data processing pipelines; and the Archytas framework, a system to code reasoning agents with access to external tools. We briefly describe both individually, and then detail how they integrate to form PalimpChat.

2.1 Palimpzest

Palimpzest is a declarative system designed to build and optimize pipelines for unstructured data processing, harnessing both LLM-based and conventional operations. At its core, Palimpzest programs can be viewed as collections of relational operators. First, users must specify one or more input datasets and their corresponding "Schema." A schema consists of the attribute names, types, and descriptions used to process the dataset. Given that a dataset can contain unstructured data, Palimpzest automatically extracts attribute values from each data record using either LLM-based or conventional methods. Subsequently, users specify a series of transformations over these data records to form the final output.

Although Palimpzest implements most relational algebra operators, our demo emphasizes two special operators: (1) *Convert*, which transforms an object of schema A into an object of schema B by computing the fields in B that do not explicitly exist in A, and (2) *Filter*, which applies a natural language predicate or UDF on a dataset's records and returns the subset that satisfies the predicate. All other operations (e.g., *Aggregation*) follow conventional database semantics.

A Palimpzest plan is a sequence of these operators over a dataset. By design, users write *logical* plans only; the choice of the physical implementation is deferred until runtime. For each logical operator, multiple equivalent physical implementations may be available. For instance, a filter operation might be performed via different LLM models, each representing a distinct physical method. More details about logical and physical implementations are available in [6].

After the user specifies their pipelines, Palimpzest creates a search space of all possible physical plans that implement such plan, which are effectively logically equivalent but may yield outputs of different quality, with a different cost, or with a different runtime.

In a subsequent optimization phase, Palimpzest automatically ranks physical plans and selects the most optimal one that meets user-defined preferences. Users can specify whether they are interested in quality, runtime, or cost of executing their pipelines. They may instruct the system to narrow its optimization on one of these dimensions (e.g., to minimize the cost no matter the quality), or specify a meaningful combination of them (e.g., maximize the output quality while being under a certain latency).

```python
@tool()
async def create_schema(
    self, schema_name: str, schema_description: str,
    field_names: list, field_descriptions: list,
    agent: AgentRef) -> str:
    """
    This tool should be used to generate a new extraction schema.
    The inputs are a schema name and a set of fields. For example,
    let's say the user is interested in extracting author
    information from a paper. In this case the schema name might
    be 'Author' and the fields may be 'name', 'email', 'affiliation'.
    You should provide a short description for each field.
    Field names cannot have spaces or special characters.
    """

    class_name = "{{ schema_name }}"
    schema = {"__doc__": "{{ schema_description }}"}
    for idx, field in enumerate({{ field_names }}):
        desc = {{ field_descriptions }}[idx]
        schema[field] = pz.Field(desc=desc)
    new_schema = type(class_name, (pz.Schema,), schema)
    return new_schema
```

Figure 2: An example Archytas tool used to generate an extraction schema with Palimpzest. Docstrings are important to provide context for the reasoning agent. Inputs variables are injected with the template syntax `{{variable}}`.

2.2 Archytas

Archytas is a toolbox for enabling LLM agents to interact with various tools in order to solve tasks more effectively, following the ReAct (Reason & Action) paradigm [11]. It is similar in functionality to existing solutions like LangChain [1], but focuses on providing a streamlined interface for tools. By implementing ReAct, an agent can decompose a user request into smaller steps, decide which tools to invoke for each step, provide corresponding input to those tools, and iterate until the task is complete. This flexible interface is well-suited for both purely textual tasks and more complex operations (e.g., looking up a dataset, calling AI-based filtering routines, etc.).

2.3 PalimpChat

The PalimpChat interface integrates with Palimpzest with Archytas by exposing a series of tools that the LLM-based agent can leverage. Essentially, these tools correspond to templated code snippets that can 1. perform fundamental Palimpzest operations (e.g., registering a dataset, generating schemas, filtering records) and 2. orchestrate entire pipelines of transformations. Figure 2 provides an example of a tool implementation.

The Archytas agent will read tool code as natural language, and consider its doc-string and input/output parameters in order to decide whether to use it to satisfy the user requests. All tools adhere to a similar pattern in terms of input and output. The general

[1] www.palimpzest.org
[2] www.palimpzest.org/chat
[3] https://github.com/mitdbg/PalimpChat
[4] https://people.csail.mit.edu/chunwei/demo/palimpchat.mp4

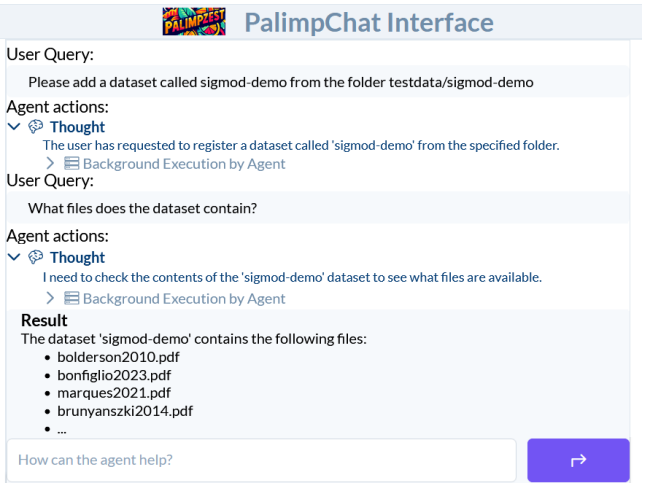

Figure 3: Setting an input dataset through PalimpChat

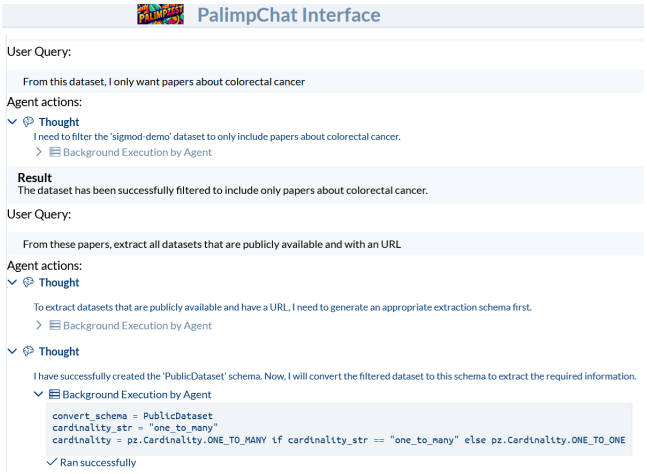

Figure 4: Building a pipeline through natural language. As seen in the last call, the agent reasons and may decide to decompose a user question into several tasks before execution.

docstring of a tool summarizes what each tool accomplishes and when it is appropriate to use. The `Args` section of the docstring can be used to describe the input and output arguments expected for each tool. Providing a few examples of usage within the docstring proved to be the most efficient solution to improve the quality of the reasoning agent.

The code of each tool is a Python function with the `@tool()` annotation, and a Jinja-based templated syntax can be used to inject run-time variables. Within the tool code, if a variable is expressed in round brackets as `{{variable}}`, the Archytas agent will fill the variable with a variable available at run-time in the Python execution environment. When parsing a single user request, the reasoning Archytas agent can decide to chain several tool invocations if it deems it necessary to fulfill the desired request (See Section 3 for an example).

We host our PalimpChat system within a hybrid notebook-chat environment, called Beaker. Beaker is an implementation of computational notebooks that integrates AI capabilities into the interactive coding environment. Building on Project Jupyter's foundation, Beaker incorporates an AI agent that facilitates code generation and execution while maintaining awareness of the complete notebook state. This enables context-aware coding, automated debugging, and dependency management, along with comprehensive state management that allows users to restore previous notebook states.

The combination of notebook as well as chat-based features allows users to interact with code, data, and LLM features all from one seamless interface.

3 Demonstration

In this demonstration scenario, we will present a real-world use case leveraging PalimpChat to build an end-to-end pipeline for scientific discovery.

Use case: Consider medical researchers interested in performing a literature survey on *colorectal cancer*. They are interested in finding studies that report on the correlation between gene mutation and tumor cells, providing or using publicly available datasets of genomic data, and fetching these datasets to run their own analysis.

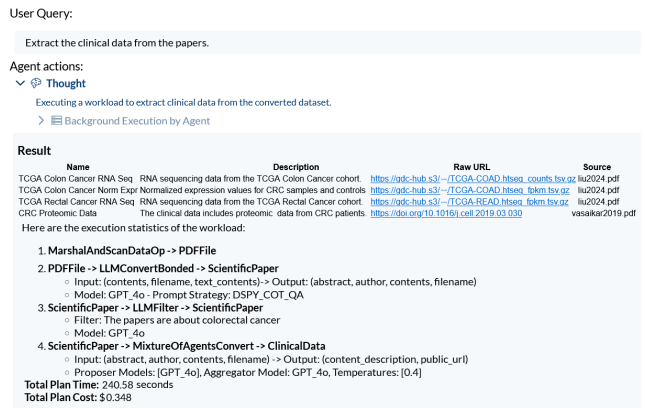

Figure 5: The output of the execution of the scientific discovery use case. Users can visualize both output records, as well as summary information about the plan execution such as the operators chosen and the total pipeline cost and runtime.

Assume they have access to a digital library of scientific papers, but this collection is potentially large, containing unrelated papers, and it is not annotated with metadata about the data sources.

To automate this task, they can build an AI data-processing pipeline using Palimpzest through the PalimpChat interface. At a high level, the medical researcher has to upload a collection of papers, filter for those about colorectal cancer, and then extract any reference of publicly available datasets.

At the core of Palimpzest, there are *datasets*: collections of input records. The first step when building a pipeline is to define an input dataset - this could either be a local folder, for which every file will constitute an individual record; or an iterable object in memory, for which every item will be a record. Additionally, more experienced users can define any custom logic to marshal arbitrary objects or paths into input datasets. For our demonstration, the user instructs PalimpChat to load an input dataset from PDFs of scientific papers contained in a local folder (See Figure 3).

The core PalimpChat system includes a native PDFFile schema, which is automatically chosen to parse the files in this dataset given their extension. However, this schema only represents the filename and the raw textual content extracted for a given paper.

Next, the user informs PalimpChat that they are interested in papers that are about colorectal cancer, and for these papers, that they would like to extract whatever public dataset has been used by the study in the paper. From these requests, the reasoning agent creates a pipeline of Palimpzest operations: it first uses the `filter()` function to select papers that satisfy the desired condition. Then, it dynamically generates an extraction schema with the fields `dataset_name`, `description`, and URL. Finally, it applies the extraction schema to the filtered dataset with a `convert()` operation. Figure 4 highlights these steps. In the background, PalimpChat decomposes the user's request into sub-steps for data loading, filtering, conversion, and execution. Before running the full pipeline to process the inputs, the users can specify their optimization goal: minimum cost, maximum quality, or minimum runtime. Palimpzest will automatically determine the appropriate physical plan that implements the logical plan built so far. The final code generated can be seen in Figure 6.

When the users prompt the chat to run the pipeline, they will visualize the extracted dataset names and URLs. Moreover, users can gain insights into the pipeline execution by asking the system to provide statistics such as how much runtime was needed to produce the output, and how much the LLM invocations cost. A sample of this output can be seen in Figure 5. For the scientific discovery use case, out of an input dataset of 11 papers, the pipeline managed to extract 6 publicly available datasets related to colorectal cancers, together with the associated URLs. We manually verified the validity of these URLs. The execution statistics of the pipeline show that the pipeline was executed in about 240s and with a cost of about 0.35 USD.

Finally, after building a pipeline, users may continue to iterate on the code produced either through the chat interface or by downloading a Jupyter notebook that contains all inputs and generated snippets of code.

```python
#Set input dataset
schema = PDFFile
dataset = pz.Dataset(source="sigmod-demo", schema=schema)

#Filter dataset
dataset = dataset.filter("The papers are about colorectal cancer")

#Create new schema
class_name = "ClinicalData"
schema = {"__doc__": "A schema for extracting clinical data datasets from papers."}
field_names = ['name', 'description', 'url']
field_descriptions = ['The name of the clinical data dataset',
    'A short description of the content of the dataset',
    'The public URL where the dataset can be accessed']
for idx, field enumerate(field_names):
    desc = field_descriptions[idx]
    attributes[name] = pz.Field(desc=desc)
new_class = type(class_name, (pz.Schema,), attributes)

#Perform conversion
convert_schema = ClinicalData
cardinality = pz.Cardinality.ONE_TO_MANY
dataset = dataset.convert(convert_schema, desc=ClinicalData.__doc__, cardinality=cardinality)

#Execute workload
output = dataset
policy = pz.MaxQuality()
records, execution_stats = Execute(output, policy=policy)
```

Figure 6: The final Palimpzest pipeline built iteratively using the chat interface for the scientific discovery page.

4 Conclusion

We demonstrate the PalimpChat system to interactively build AI pipelines using Palimpzest, Archytas and Beaker. Although our demo did not extensively cover physical optimization aspects, more details can be found in [6]. The PalimpChat interface offers a convenient tool for data practitioners to build complex data processing pipelines with little effort and a soft learning curve. Our vision is that on the one hand, the future of data engineering will include more and more sophisticated frameworks to build complex applications that mix LLMs and traditional data processing. On the other hand, tools like PalimpChat will assist developers and make it easier to adopt new technologies and programming paradigms.

Acknowledgments

We gratefully acknowledge the support of the DARPA ASKEM project (Award No. HR00112220042 and HR00112490514), and the ARPA-H Biomedical Data Fabric project.

References

[1] LangChain Contributors. 2024. LangChain: Open Source Framework for Building Language Models. https://github.com/LangChain/langchain.
[2] 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).
[3] 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).
[4] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.
[5] Chunwei Liu, Enrique Noriega-Atala, Adarsh Pyarelal, Clayton T Morrison, and Mike Cafarella. 2025. Variable Extraction for Model Recovery in Scientific Literature. In *AISD @ NAACL 2025*.
[6] Chunwei Liu, Matthew Russo, Michael Cafarella, Lei Cao, Peter Baile Chen, Zui Chen, Michael Franklin, Tim Kraska, Samuel Madden, Rana Shahout, et al. 2025. Palimpzest: Optimizing AI-Powered Analytics with Declarative Query Processing. In *CIDR 2025*.
[7] 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).
[8] 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).
[9] Shreya Shankar, Aditya G Parameswaran, and Eugene Wu. 2024. DocETL: Agentic Query Rewriting and Evaluation for Complex Document Processing. *arXiv preprint arXiv:2410.12189* (2024).
[10] Matthias Urban and Carsten Binnig. 2024. Demonstrating CAESURA: Language Models as Multi-Modal Query Planners. In *Companion of the 2024 International Conference on Management of Data*. 472–475.
[11] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629* (2022).