AutoPAC - Automated Plan and Code Synthesis for Continual Machine Learning Ideation

Sarvesh B[1], Aryan Sahu[1], Harshvardhan Mestha[1], Dhruv Shah[1], Gautam Shroff[2], Manasi Patwardhan[2], Lovkesh Vig[2], Ashwin Srinivasan[1], and Tanmay Verlekar[1]

[1] BITS Pilani, Goa, India
[2] TCS Research, Delhi, India

Abstract. The rapid pace of machine learning research presents a significant challenge for practitioners attempting to integrate new ideas into existing projects. While current approaches can retrieve relevant papers, they often fail to articulate how these ideas can be practically applied. This paper introduces AutoPAC (Automated Plan and Code Synthesis for Continual Machine Learning Ideation), an LLM-based pipeline designed to bridge this gap. AutoPAC takes as input a high-level project description, including current implementation details, data used, techniques employed, and associated challenges.It then generates a structured, high-level plan based on these inputs, The system goes beyond plan generation by automatically producing implementation code for the devised techniques, accelerating the integration of new ideas with minimal expert intervention. We demonstrate AutoPAC's effectiveness through experiments involving two datasets (Numerai and a synthetic time series dataset) and four diverse research papers, using two different Large Language Models (LLMs). Our results show that AutoPAC can generate high-quality, running code for most of the relevant papers, with performance varying based on paper relevance and dataset documentation. This work represents a significant step towards automating the research-to-implementation pipeline in machine learning, potentially reducing the barriers to implementing cutting-edge ideas in practical settings.

Keywords: Large Language Models · Machine Learning· Code Generation· Continual Learning.

1 Introduction

With hundreds of Machine Learning (ML) papers published daily across various conferences and journals, ML practitioners face various challenges in their deployed project pipelines on a day-to-day basis. The already existing pipelines can be iterative by augmentation of newer techniques to improve the overall performance and/or efficiency and address specific challenges. One of the best sources of these ML techniques, which can serve as the plausible solution for these issues, can be the ideas developed and elaborated in research papers. However, the fast pace of research in the ML field has made it difficult for machine learning practitioners to keep themselves appraised with the new ideas developed and published in the literature, which can apply to their projects. Current approaches for retrieving relevant scientific articles [19,4,18,15] typically focus on finding papers related to a given query. However, these methods often fall short in explicitly articulating how the ideas presented in a paper can be practically applied to resolve specific issues or improve existing projects. This calls for a need of a platform to cater to ML practitioners, which not only points them to the literature that may be useful for them to address their day-to-day project-related challenges, but also assists them in figuring out how the idea discussed in the literature can be applied to resolve the issues or improve the performance of their projects and further enhances the existing code to integrate the applicable idea in an automated fashion.

As an initial solution to address this need, in this paper, we have designed **AutoPAC**, an LLM (Large Language Model) based pipeline with the aim to assist the practitioners in generating code for newly identified relevant papers publishing on a daily basis. The pipeline expects the ML practitioner to provide a high-level description of the project, including the documentation of its current implementation and data used, the techniques formerly or currently employed, the challenges associated with the project, and the code or pseudo-code (or both) of the current implementation.

AutoPAC identifies if an idea is applicable in silos or a novel technique is elicited by composing ideas from multiple papers which together may resolve a challenge. It further aims at enhancing the existing implementation of the project provided by the practitioner by generating the code for the devised techniques, in an automated manner. Code generation helps in accelerating the process of incorporating the new idea or technique with minimal intervention by a machine learning specialist.

There are existing LLM-based approaches, which try to use the parametric knowledge of LLMs to either find solutions for a specific ML task [28,9] or generate ideas for new research problems [1,5,22] or generates repository level codes for ML problem definitions [2,26,20,27]. As opposed to these prior approaches, **AutoPAC** checks the applicability and leverages ideas from existing research papers or provided by end users to solve challenges faced in existing ML pipelines. It also refines its code to implement the idea in an automated fashion. Our work demonstrates its effectiveness by enabling the generation of high-quality code implementations in a continual setting by taking two datasets and four papers into consideration, using two LLMs, and manually evaluating the outcome, thereby facilitating their seamless integration and application.

AutoPAC facilitates the process of deriving ideas from research papers by creating an intermediate plan that grounds the high-level descriptions of these ideas and subsequently generates the code accordingly. For instance, when implementing the methods based on [16] using the Numerai[[17]] dataset, **AutoPAC** generates a detailed plan that bridges the gap between theory and practical implementation. When users contribute detailed engineering ideas based on new papers, such as [6], **AutoPAC** enhances the existing plan with these specific requirements, ensuring that the code is augmented to meet the updated needs. This approach ensures that implementation remains relevant and up-to-date with the latest advancements. The main contributions of this work are:

- To the best of our knowledge, we are the first to develop an LLM-based pipeline to apply a idea to resolve challenges in an ML pipeline.
- This pipeline models a more realistic setting of incremental development of ML pipelines, resolving the issues in a continual fashion.
- For example, Our pipeline showcases the automatic implementation of concepts such as conservative predictions, drawing from research papers [10] and human insights, like substituting XGBoost with TabPFN [6] on the Numerai dataset [17]. The resulting plans and code exemplify the effectiveness of **AutoPAC**.

2 Background

The rapid advancement of machine learning (ML) research has created a significant challenge for practitioners: staying current with the latest techniques and efficiently integrating them into existing projects. This section provides context for the development of AutoPAC by exploring relevant areas of research and identifying gaps in current approaches.

LLM-based Code Generation Recent advancements in LLMs have demonstrated strong code generation and modification capabilities. These models, trained on vast corpora of natural language and programming code, have shown a remarkable ability to understand and generate complex software implementations. CodeCoT [7] introduced a chain-of-thought prompting approach to enhance code generation, enabling LLMs to break down complex programming tasks into manageable steps. This method significantly improved the quality and correctness of generated code, particularly for challenging problems requiring multi-step reasoning. AgentCoder [8] expanded on this concept by implementing a multi-agent framework for code generation. By simulating different roles in the software development process (e.g., architect, programmer, tester), AgentCoder demonstrated improved code quality and bug detection capabilities compared to single-agent approaches. AceCoder [12] focused on the task of code modification and refactoring. By leveraging existing codebases as context, AceCoder showed proficiency in adapting and improving code to meet new requirements or optimize performance. While these approaches have significantly advanced the field of automated code generation, they primarily focus on generating code from scratch or modifying existing code based on specific instructions. They do not address the challenge of rapidly integrating new research ideas into existing codebases, which is the primary focus of **AutoPAC**.

Continual Learning in Machine Learning Continual learning, also known as lifelong learning or incremental learning, aims to develop systems that can adapt to new information without forgetting previously learned knowledge. This paradigm is crucial for creating AI systems that can evolve and improve over time, similar to human learning [10]. Our work draws inspiration from these continual learning methods to create a system capable of implementing and incorporating new papers without needing model retraining or fine-tuning. By leveraging the generalization capabilities of large language models and structuring the input appropriately, **AutoPAC** aims to achieve a form of continual learning in the domain of code generation and research implementation.

Research Paper Retrieval and Analysis Existing approaches for retrieving relevant scientific articles [19,4,18,15] typically focus on finding papers related to a given query. However, these methods often fall short in explicitly articulating how the ideas presented in a paper can be practically applied to resolve specific issues or improve existing projects. This gap highlights the need for a system that not only identifies relevant literature but also assists in understanding and implementing the ideas within the context of ongoing ML projects.

Automated Machine Learning (AutoML) AutoML systems aim to automate the process of applying machine learning to real-world problems, including tasks such as feature selection, model selection, and hyperparameter optimization [21]. While AutoML has made significant strides in democratizing machine learning, current systems typically focus on optimizing existing pipelines rather than incorporating novel research ideas.

Time Series Datasets Time series data presents unique challenges in machine learning, particularly in the context of continual learning and model updating. Datasets like Numerai exemplify these challenges, requiring frequent model updates and adaptations to changing data distributions. Understanding the nuances of working with such datasets is crucial for developing systems that can effectively integrate new research ideas in real-world scenarios. In this paper, we use two datasets-Numerai Dataset [17] and a TSD (Temporal Sinusoidal Data).

3 Methodology

AutoPAC (Automated Plan and Code Synthesis) is a novel system designed to assist machine learning practitioners in implementing new research ideas and techniques into their existing projects. The system leverages Large Language Models (LLMs) to analyze research papers, generate implementation plans, and produce executable code. **AutoPAC** operates through a two phase process: **Plan Generation** and **Code Generation**. These two phases work in tandem to translate research ideas into implementable code shown in Figure 1, leveraging the power of LLMs at each step. It addresses the challenge of automatically integrating new ML techniques into an existing continual learning pipeline, whether it involves updating models or datasets. Importantly, these phases can be iterated multiple times, allowing for sequentially incorporating ideas from multiple papers. This iterative approach enables continuous refinement and expansion of the implementation based on new research insights. While our experiments focused on two iterations, the process can be repeated indefinitely. In the following sections, we comprehensively explain our Plan Generator and Code Generator, detailing each component thoroughly.

3.1 Plan Generation

The Plan Generation phase forms the conceptual Base of **AutoPAC**, transforming research papers and high-level ideas into structured plans for implementation. This process involves several steps, each designed to distill and organize the essential information from the input sources. The plan generation process involves a multi-step process, similar to how one reads and analyses a paper, inspired from [10]. The first step involves providing the paper as context to an LLM. We then use a prompt to summarise the paper. We provide an idea as context, which includes information about the dataset, model and additional instructions if needed, and ask the LLM to produce a methodology using the paper summary and the high level as context, and finally ask it to refine its

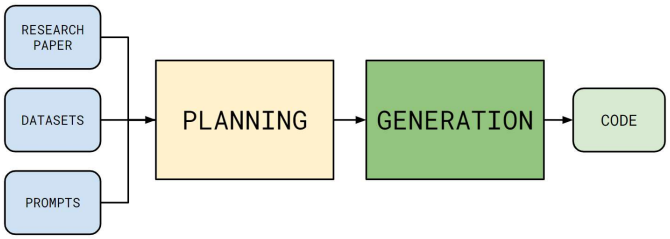

Fig. 1. Overview of **AutoPAC** pipeline

methodology, which gives us the High Level Plan as the output. This is shown in Figure 2, where the Paper and Idea is the input and High Level Plan is the output, with the prompt summaries given in Table 1, and control flow in Algorithm 1.

3.1.1 Providing Context

The user needs to provide two inputs to **AutoPAC**, the Paper(P) and the Idea(I(D,M)) shown in Figure 2 and Algorithm 1.
Paper(P) - A research paper relevant to the problem, ideally containing an algorithmic methodology.
Idea (I(D,M)) - This is the idea or the approach to be given by the user, this involves specifying the details of the dataset(D), such as the presence of missing values, preprocessing, choosing subsets etc. The context of the model(M) also needs to be provided, such as, the abstract of the paper, along with certain limitations in using the model with a dataset, such as handling missing values, size limitations, data type handling, and some additional instructions like doing ensembling for data/feature scaling etc. The model is then instructed to incorporate the dataset(D) and model(M) with the given paper, keeping in mind the specified limitations in the idea (I(D,M)).

3.1.2 Analysing the Paper

Once context is established, **AutoPAC** employs a multi-stage analysis process to extract and synthesize essential information from the provided research paper. This systematic approach ensures that critical methodological insights are captured and integrated into the final plan. The paper is analyzed in various stages. As shown in Figure 2 and Algorithm 1, The LLM first generates a concise summary of the paper by identifying the essential steps and methodology of the paper The LLM then combines the provided Idea (I) with the paper's methodology to create a new approach that leverages the strengths of the paper while accommodating the Dataset (D) and Model (M) requirements finally The LLM verifies if the limitations in the method are addressed, comments on its feasibility, and provides pseudocode for the final output, the High Level Plan (PLN).

3.1.3 High Level Plan

The High Level Plan is the final output of the plan generator, denoted by PLN in Algorithm 2, and shown in Figure 2. It contains all the details of the approach to be implemented. The LLM is made to describe the reasoning behind choosing the particular approach in this High Level Plan, and contains the pseudocode for the approach. This is directly given to the code generation stage for producing code. For a sample High Level Plan, please refer to Appendix A

The *Plan Generation* and *Code Generation* phases of **AutoPAC** work in tandem to translate research ideas into implementable code. The output of the Plan Generation phase, the High Level Plan serves as a crucial input for the Code Generation phase, creating a seamless flow from high-level plans to executable implementations. This transition ensures that the generated code accurately reflects the synthesized ideas and methodologies derived from the research papers. Whenever a new and relevant paper or idea is released, the Plan Generator generates a High Level Plan, to be fed into the Code Generator.

Data: L : an LLM, P : Paper, D : Dataset, M : Model, I : Idea given by user, C : Context/Memory of the LLM, out : Output produced by the LLM
Result: PLN : High Level Plan used for Code Generation
$C \leftarrow \phi$;
append P to C;
$out = summarise(L, P)$;
append out to C;
$out = analyse(L, P, C)$;
append out to C;
$out = generate\_methodology(L, P, C, I(D, M))$;
append out to C;
$PLN = refine\_methodology(L, P, C, I(D, M))$;
return PLN;

Algorithm 1: Plan Generation

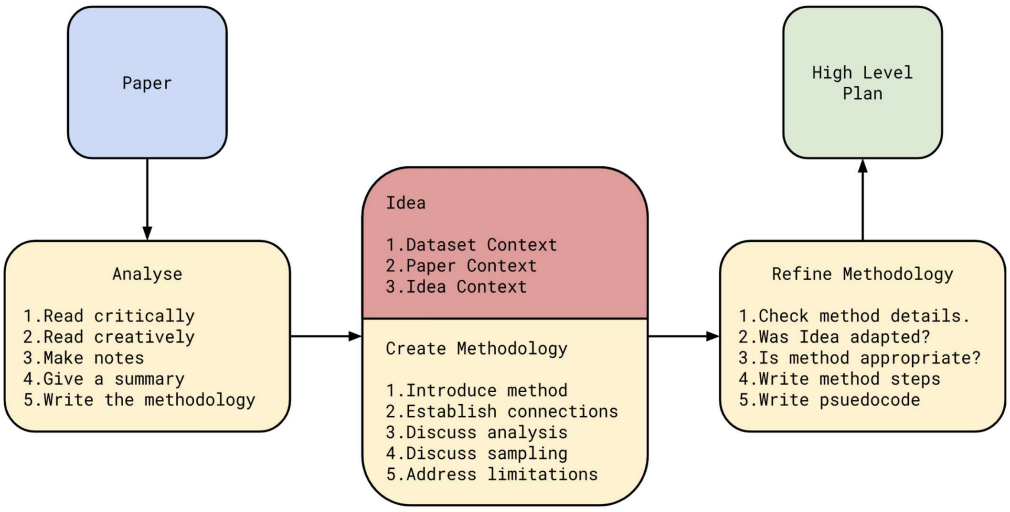

Fig. 2. Overview of the Plan Generation pipeline

3.2 Code Generation

In this section, we describe the structure of the Code Generator along with its components. Figure 3 presents an overview of the Code Generator, our proposed LLM-facilitated pipeline for code generation. The system is comprised of three core components:

Pipeline Generation: This module generates a pipeline or roadmap to be followed for generating the code as shown in Figure 4.**Input**: High-level specification of the task or problem generated from the *Plan Generator* (from Section 3.1).**Output**: Detailed steps and structure for the code generation.

Base Code Generation: This module generates code from the provided High Level Plan, as shown in Figure 5.It aims to capture the core logic and structure of the proposed methodology.The **Input** is the High Level Plan (from Section 3.1), the pipeline from the Pipeline Generation module, and relevant libraries.The **Output** is the Base Code that serves as an initial implementation or sample code for the Iterative Code Generation.

Iterative Code Generator: This module generates code from the modified high-level Plan and uses the generated Base Code from Figure 5 as sample code, and the Iterative Code Generator is shown in Figure C.The **Input** is the Modified High Level Plan(which contains the context of the existing and new pipelines along with dataset context) from 3.1 and the Base Code, along with Sample Model Code specifying model implementation and relevant libraries.The **Output** is the

Step/Prompt	Description
summarise	Giving LLM the paper, either PDF or raw text, without any additional prompt.
analyse	Tell the LLM that it is a researcher and you are doing your literature review, you have been provided with a systematic approach on how to read a paper. Uses Chain of Thought to analyze the methodology of the paper.
generate_methodology	This prompt contains the context of the Dataset and Model on which we want to apply the paper. It also contains the instructions to produce an effective methodology given the context of the paper, dataset, and model.
refine_methodology	This prompt is to refine the methodology produced with generate_methodology and produces a brief methodology and its pseudocode called the High Level Plan which is then directly given as input to the Code Generator.

Table 1. Details of various stages of Plan Generation.

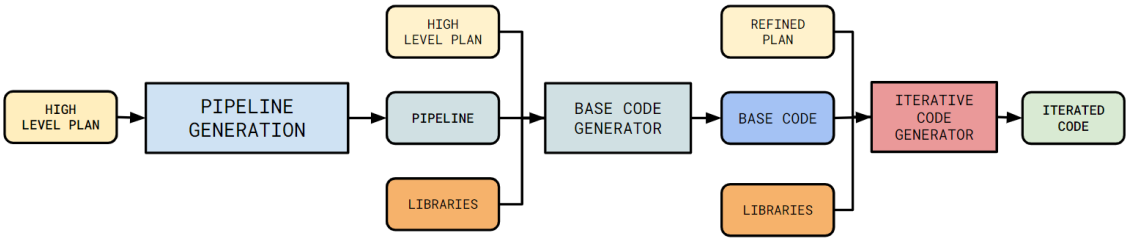

Fig. 3. Overview of the Code Generator

Iterative, refined code that meets the specific requirements of the model/idea which is to be built upon the existing pipelines. The following subsections introduce the details of the pipeline.

3.2.1 Pipeline Generation

This module generates a structured pipeline for code implementation based on the High Level Plan produced by the Plan Generation phase. It breaks down the implementation task into logical steps, creating a road map for the code generation process,as shown in Figure 4.

As elucidated previously, the process commences with a high-level conceptualization, comprising a *methodology* and a *refined methodology*. The refined methodology provides the logical underpinnings for the generated methodology by addressing specific queries. This High Level Plan is supplied as input to a LLMs, either the Gemini 1.0 Pro variant or GPT-4 in this instance. The LLM then generates a set of five distinct pipelines. The prompts employed for the pipeline generation are provided in Appendix B. Subsequently, the LLM is tasked with selecting the most optimal pipeline that exhibits the highest degree of consistency and coherence with the provided High Level Plan. This comprehensive procedure is also known as Self-Consistency in LLMs [23].

3.2.2 Base Code Generator

The Base Code Generator comprises of 2 components, The Code Generator and Debugger. **Code Generator**: This module produces initial code implementations using the generated pipeline and the High Level Plan. The prompts for the Code Generator are appended at the conclusion. These prompts are composed with due consideration given to mitigating hallucinations, circumventing excessive brevity, and adhering stringently to the generated pipeline. This Base Code serves as a starting point, capturing the core logic and structure of the proposed methodology, which may not

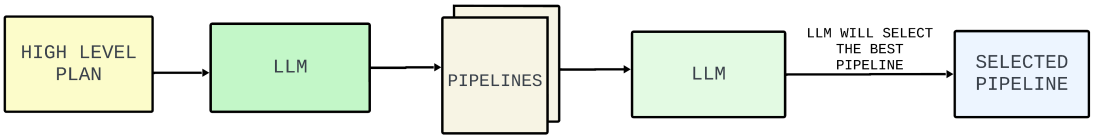

Fig. 4. Pipeline Generator

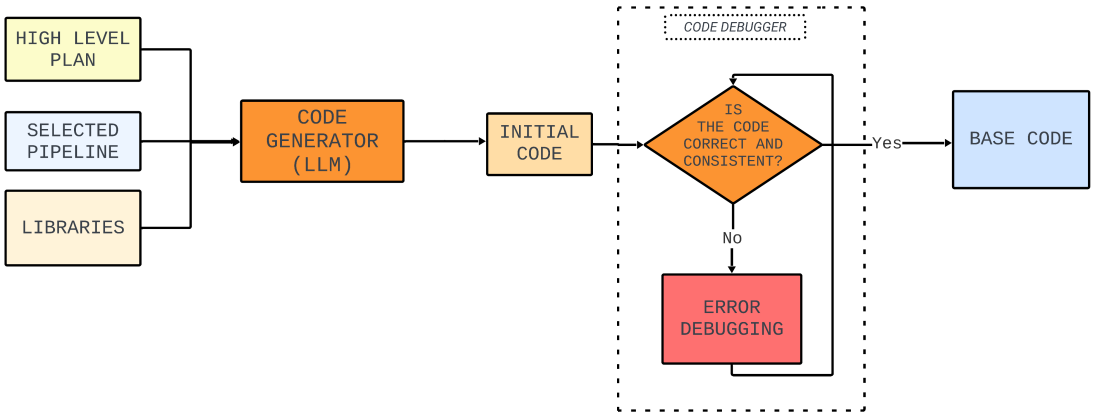

Fig. 5. Overview of the Base Code Generator

be entirely free from errors or exhibit inconsistencies with the High Level Plan. This phenomenon is attributable to the tendency of LLMs to deviate at times, as elucidated in [24].The Base Code generator pipeline is shown in Figure 5. **Code Debugger**: The generated code may have certain inconsistencies due to the inherent behavior of LLMs as illustrated in [13]. Hence, to mitigate such occurrences, we have created an error-debugging module, shown in Figure 5. This debugger is essentially comprised of try-catch-exception blocks. The error, in tandem with the code and pipeline, is iteratively furnished as input to the LLM until an error-free code is generated. Furthermore, we limit the number of iterations to preclude an infinite loop of error correction. The prompts for the code generation is given in Appendix B.

3.2.3 Iterative Code Generator

Data: $L \leftarrow \{\text{gpt-4}, \text{gemini}\}$, PLN: High-Level Plan, $Libs$: Libraries, out: Output produced by the LLM
Result: $debugged\_code$
/* Initialize High-Level Plan and Libraries */
$PLN \leftarrow$ "Plan used for Code Generation";
/* Generate Pipelines from High-Level Plan */
$pipelines \leftarrow L.generate\_pipelines(PLN)$;
/* Select the Optimal Pipeline */
$best\_pipeline \leftarrow L.select\_best\_pipeline(pipelines)$;
/* Generate Initial Code from the Selected Pipeline */
$initial\_code \leftarrow CodeGenerator.generate\_code(best\_pipeline, PLN, Libs)$;
/* Debug the Initial Code */
$debugged\_code \leftarrow Debugger.debug\_code(initial\_code)$;
/* Return the Debugged Code */
return $debugged\_code$;

Algorithm 2: Automated Code Generation and Debugging

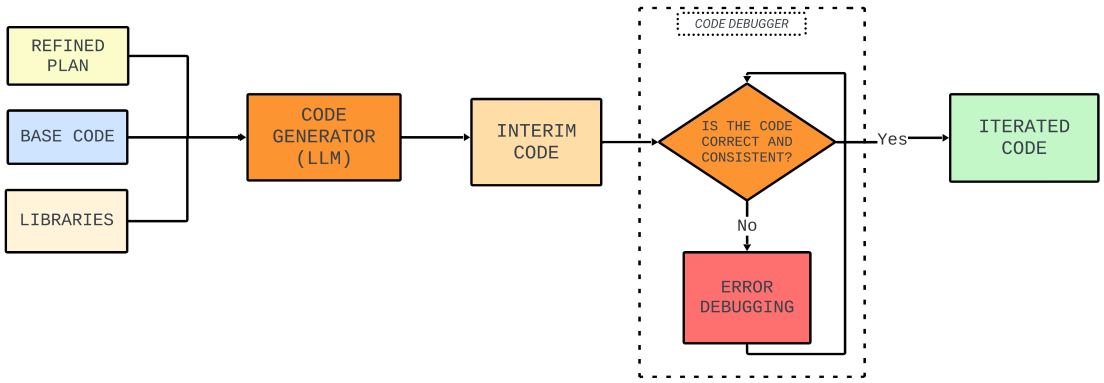

Fig. 6. Overview of the Iterative Code Generator

This module utilizes the Base Code generated in the preceding step. This Base code serves as a source of inspiration for implementing changes in accordance with the requisite model(M). This encapsulates the core principle of Continual Learning [12,25], wherein the Iterative code is built upon the Base code. For generating the Iterative code, a textual description, which is the new High Level Plan (refined plan), is employed. This new High Level Plan not only takes into account the context of the Base paper and dataset but also incorporates the model that is to be utilized. The procedure for generating this new High Level Plan is delineated in Section 3.1.The Iterative Code Generator pipeline is shown in Figure C.

Code Generator: The LLM is furnished with the Base Code Generator, described in Section 3.2.2, along with the new High Level Plan (comprising of the context of the model to be used). Once again, the prompts for the Code Generator are provided in Appendix B.
Code Debugger: This component is the same as the Base Code Debugger, described in Section 3.2.2, but it also incorporates a limited aspect of human feedback for certain intricate ideas.

The output produced by this Iterative Code Generator represents the target result (Figure C) for code generation in a two-hop continual learning approach utilizing LLMs. This can very well be extended to a multi-hop system as well.

4 Experiments

We conducted a comprehensive evaluation of **AutoPAC**'s performance using two distinct datasets and four diverse research papers which we found "interesting" and speculated on their relevance to the experiment. This section details our experimental setup, evaluation metrics, and results. The Iterative model for which the code is to be generated for all the experiments is the TabPFN[6] transformer architecture.This choice allows us to evaluate **AutoPAC**'s ability to generate code for a specific, advanced machine learning model.
We chose the papers referred to as **CONPR**[16] **NNBST**[14] **TSMIX**[3] **VOLTS**[11] in the paper. Each paper's relevance was evaluated based on the following criteria, based on these criteria, we manually evaluated each paper and marked it as either relevant (✓) or not relevant (✗) for our specific experimental setup

Applicability to tabular data: Papers focusing on techniques directly applicable to tabular datasets were considered highly relevant for our experiments, using a procedure similar to [5].
Compatibility with time series analysis: Given our focus on financial datasets (Numerai and TSD), papers addressing time series aspects were deemed more relevant.
Novelty of approach: Papers introducing novel methodologies or improvements to existing techniques for tabular data analysis were prioritized.
Potential for integration: We favored papers whose methods could be feasibly integrated into our existing pipeline and combined with TabPFN.

The Model (**M**) used for the Continual experiment is TabPFN[6]. We selected TabPFN (Tabular Prior-Free Network) as our Iterative model for these experiments due to its unique characteristics and suitability for our task:
Versatility: TabPFN excels in handling tabular data without requiring extensive hyperparameter tuning, making it ideal for our diverse datasets.
Sample efficiency: It performs well with limited training data, which is crucial when integrating new ideas that may not have extensive datasets available.
Adaptability: TabPFN's architecture allows for easy incorporation of new features and methodologies, aligning with our goal of continual learning and integrating novel research ideas.
The datasets chosen are **Numerai** and another tabular dataset similar to Numerai, referred to as **Temporal Sinusoidal Data (TSD)**.

- **Numerai**: This data science competition focuses on predicting the stock market. It features a highly documented dataset with numerous tutorials and guides available. The dataset includes thousands of features and five classes to predict. It evolves continuously, with new data points and features added weekly as new versions of the dataset are released. Due to the need to adapt to frequent dataset changes, Numerai is an optimal dataset for **AutoPAC**.
- **TSD (Temporal Sinusoidal Data)**: This internal dataset is used to test the robustness of our architecture. It originates from synthetic sinusoidal data with various levels of noise added. Temporal features have been computed to convert the time-series data into tabular form, with each row representing a specific time-step and discrete features capturing different aspects of recent temporal behavior. Although it is also time series data, its features and classes differ from those of Numerai. With limited documentation, it presents a greater challenge for an LLM.

4.1 Experimental settings

Plan Generator: The Plan Generator receives the Paper, Dataset, and Model configuration as outlined in section 3.1. The resulting plan is then implemented using the code generator, as described in section 3.2. The LLM utilized for the entire Plan Generation was **`Claude-3-Sonnet`**.
Code Generator: Following the generation of the high-level plan, **`gemini-1.5-pro-latest`** was employed to construct the pipeline. This choice was made to leverage its advanced capabilities in understanding and translating abstract plans into structured workflows. For the subsequent stages of the **AutoPAC** system, the Base Code Generator, Iterative Code Generator, and Code Debugger, we utilized **`gpt-4-turbo`**. The decision to use gpt-4-turbo for these critical components was based on its robust performance in code generation and debugging tasks, ensuring consistency and reliability throughout the latter stages of our automated process The outputs were manually classified into the following **four** categories.:

- **Running, Correct**: The code executes without errors, and the desired logic is fully implemented.
- **Running, Incorrect**: The code executes without errors, but the desired logic is either incomplete or missing.
- **Not Running, Small Errors**: The logic is correctly implemented, but the code has minor issues (e.g., incorrect dataset version, incorrect variable initialization) preventing it from running.
- **Not Running, Incorrect**: The code neither runs without errors nor implements the desired logic correctly.

Evaluation: Regarding the evaluation, the numbers in the cells of Tables represent the proportion of codes that were (Running & Correct, Running & Incorrect, Not Running with Small Errors, Not Running & Incorrect) out of the total 10 codes generated overall.

5 Results

We conducted a comprehensive evaluation of AutoPAC's performance using two distinct datasets and four diverse research papers. This section details our experimental results, highlighting the

system's ability to generate code for both base implementation and iterative improvements incorporating the TabPFN transformer architecture.See Appendix C for some generated code snippets.

5.1 Dataset:Numerai

Paper	Relevant	Running, Correct	Running, Incorrect	Not Running, Small Errors	Not Running, Incorrect
CONPR	✓	8/10	1/10	0/10	1/10
NNBST	✓	6/10	1/10	2/10	1/10
TSMIX	✗	2/10	2/10	0/10	6/10
VOLTS	✗	0/10	1/10	0/10	9/10

Table 2. Performance of the Base Code Generator on different papers.

Paper	Running, Correct	Running, Incorrect	Not Running, Small Errors	Not Running, Incorrect
CONPR + M	7/10	1/10	1/10	1/10
NNBST + M	5/10	1/10	3/10	1/10
TSMIX + M	2/10	1/10	0/10	7/10
VOLTS + M	0/10	1/10	0/10	9/10

Table 3. Performance of the Iterative Code Generator on different papers along with the models.

Tables 2 and 3 in our study present the performance of the Base Code Generator and Iterative Code Generator, respectively, across four different papers. In Table 2, we assess the relevance of each paper and categorize the generated code's performance. CONPR and NNBST are identified as relevant to our dataset, while TSMIX and VOLTS are not. Performance-wise, CONPR demonstrates the strongest results, with 8 out of 10 instances running correctly. In contrast, "VOLTS" shows the weakest performance, with 9 out of 10 instances not running and deemed incorrect. TSMIX yields mixed results, with equal proportions (2 out of 10) running correctly and incorrectly. Table 3 analyses the Iterative Code Generator Figure C with **Model(M)**[6] context, (denoted by "+ M"). Compared to the Base Generator, we observe a slight decrease in performance for CONPR and NNBST, with 7 out of 10 and 5 out of 10 running correctly, respectively. TSMIX and VOLTS maintain similar performance patterns to their Base Generator counterparts, with VOLTS continuing to show poor results. Notably, the "Not Running Small Errors" category exhibits some variation, particularly for NNBST+M, where 3 out of 10 instances fall into this category. These results provide insights into the relative strengths and limitations of our code generation approaches across different paper contexts.

5.2 Dataset:TSD(Temporal Sinusoidal Data)

Paper	Relevant	Running, Correct	Running, Incorrect	Not Running, Small Errors	Not Running, Incorrect
CONPR	✓	6/10	2/10	1/10	1/10
NNBST	✓	4/10	0/10	5/10	1/10
TSMIX	✗	5/10	0/10	4/10	1/10
VOLTS	✓	5/10	1/10	3/10	1/10

Table 4. Performance of the Base Code Generator on different papers using TSD.

Based on the data presented in Tables 4 and 5, we can summarize the performance of the Base Code Generator and Iterative Code Generator across different papers using the TSD dataset.For the Base Code Generator (Table 4), CONPR, NNBST, and VOLTS were deemed relevant to the dataset, while TSMIX was not. CONPR showed the best performance, with 6/10 instances running correctly, followed by TSMIX and VOLTS (5/10 each) and NNBST (4/10). CONPR also had the highest rate of running incorrectly (2/10). NNBST had the most instances of not running due to small errors (5/10). The Iterative Code Generator results (Table 5) show some changes when the **model(M)** context was added (denoted by "+ M"). 'CONPR+M' and 'NNBST+M' both achieved 4/10 instances running correctly, an improvement for NNBST but a slight decline for CONPR. 'TSMIX+M' showed the poorest performance in correct runs (2/10) but had the highest rate of running incorrectly (3/10). 'VOLTS+M' maintained its 4/10 correct run rate. Notably, 'CONPR+M' had the highest rate of not running and being incorrect (3/10), while 'TSMIX+M' had the most instances of not running due to small errors (4/10).

Paper	Running, Correct	Running, Incorrect	Not Running, Small Errors	Not Running, Incorrect
CONPR + M	4/10	1/10	2/10	3/10
NNBST + M	4/10	1/10	3/10	2/10
TSMIX + M	2/10	3/10	4/10	1/10
VOLTS + M	4/10	1/10	3/10	2/10

Table 5. Performance of the Iterative Code Generator on different papers along with the models and using TSD.

5.3 Key Findings

Performance on relevant papers: **AutoPAC** demonstrated high success rates (up to 80%) in generating running and correct code for papers directly relevant to the given dataset and model.
Dataset impact: The system consistently performed better with the Numerai dataset compared to TSD, highlighting the importance of comprehensive documentation in facilitating automated code generation.
Relevance vs. Performance: There was a clear correlation between the relevance of a paper and the quality of generated code, with relevant papers (CONPR and NNBST) consistently outperforming less relevant ones (TSMIX and VOLTS).
Base vs. Iterative Code: The Iterative Code Generator phase, which incorporates model-specific requirements, generally maintained or only slightly decreased upon the performance of the Base Code generation.
Error types: For less relevant papers or the less documented dataset (TSD), the system was more prone to generating code that either didn't run or had incorrect implementations.

6 Discussion

AutoPAC presents a novel approach to automated code generation from research papers using a LLMs and code generation techniques. The results demonstrate the potential of this approach in translating high-level ideas and natural language descriptions into executable code. While the performance varies across different research papers, the Code Generator showcases its ability to generate running and correct code in several instances. It also demonstrates the accuracy and precision of the Plan Generator, which serves as the basis for generating the code. The Base Code Generator performed reasonably well in generating running and correct code for some papers (CONPR and NNBST), with 8/10 and 6/10 instances respectively.Table 2.However, it also generated a significant number of instances where the code was either running but incorrect (CONPR: 1/10, NNBST: 1/10, TSMIX: 2/10, VOLTS: 1/10) or not running with small errors (CONPR: 0/10, NNBST: 2/10, TSMIX: 0/10, VOLTS: 0/10).Subsequently, Table 3 displays the performance of the Iterated Code Generator, while Tables 4 and 5 present the performance for the TSD Dataset.

The poor performance of papers like TSMIX[3] for the TSD Dataset & VOLTS[11] for the Numerai Dataset can be attributed to either the dataset lacking relevance to the paper or there is a requirement of an additional input that is not provided within the paper. Future work should focus on improving performance on less documented datasets and enhancing the system's ability to handle papers of varying relevance.

7 Conclusion

We have built a pipeline using LLMs to continually incorporate new papers into an existing pipeline, by generating code. Our work performs well when combining the relevant papers, with papers that are not relevant our work integrates the paper in a logical manner, albeit with small errors. **AutoPAC** can also reliably handle various datasets. The Plan generation allows the creation of strong plans that account for finer nuances in the problem, and this plan is reliably converted to code with the Code Generator which reliably produces correct code. We hope that our work helps the research community to build working implementations of new ideas, at a rapid pace, making the barriers to implementing new ideas lower than ever before.

References

1. Baek, J., Jauhar, S.K., Cucerzan, S., Hwang, S.J.: Researchagent: Iterative research idea generation over scientific literature with large language models (2024)
2. Bairi, R., Sonwane, A., Kanade, A., C, V.D., Iyer, A., Parthasarathy, S., Rajamani, S., Ashok, B., Shet, S.: Codeplan: Repository-level coding using llms and planning (2023)
3. Chen, S.A., Li, C.L., Yoder, N., Arik, S.O., Pfister, T.: Tsmixer: An all-mlp architecture for time series forecasting (2023)
4. Cohan, A., Feldman, S., Beltagy, I., Downey, D., Weld, D.S.: Specter: Document-level representation learning using citation-informed transformers (2020)
5. Gu, X., Krenn, M.: Generation and human-expert evaluation of interesting research ideas using knowledge graphs and large language models (2024)
6. Hollmann, N., Müller, S., Eggensperger, K., Hutter, F.: Tabpfn: A transformer that solves small tabular classification problems in a second (2023)
7. Huang, D., Bu, Q., Qing, Y., Cui, H.: Codecot: Tackling code syntax errors in cot reasoning for code generation (2024)
8. Huang, D., Zhang, J.M., Luck, M., Bu, Q., Qing, Y., Cui, H.: Agentcoder: Multi-agent-based code generation with iterative testing and optimisation (2024)
9. Huang, Q., Vora, J., Liang, P., Leskovec, J.: Mlagentbench: Evaluating language agents on machine learning experimentation (2024)
10. Jiang, X., Dong, Y., Wang, L., Fang, Z., Shang, Q., Li, G., Jin, Z., Jiao, W.: Self-planning code generation with large language models (2024)
11. Letteri, I.: Volts: A volatility-based trading system to forecast stock markets trend using statistics and machine learning (2023)
12. Li, J., Zhao, Y., Li, Y., Li, G., Jin, Z.: Acecoder: Utilizing existing code to enhance code generation (2023)
13. Liu, F., Liu, Y., Shi, L., Huang, H., Wang, R., Yang, Z., Zhang, L., Li, Z., Ma, Y.: Exploring and evaluating hallucinations in llm-powered code generation (2024)
14. McElfresh, D., Khandagale, S., Valverde, J., C, V.P., Feuer, B., Hegde, C., Ramakrishnan, G., Goldblum, M., White, C.: When do neural nets outperform boosted trees on tabular data? (2023)
15. Mysore, S., Cohan, A., Hope, T.: Multi-vector models with textual guidance for fine-grained scientific document similarity (2022)
16. Nabar, O., Shroff, G.: Conservative predictions on noisy financial data. In: 4th ACM International Conference on AI in Finance. ICAIF '23, ACM (Nov 2023). https://doi.org/10.1145/3604237.3626859, http://dx.doi.org/10.1145/3604237.3626859
17. Numerai: Numerai dataset (1 2024), https://numer.ai/data
18. Ostendorff, M., Rethmeier, N., Augenstein, I., Gipp, B., Rehm, G.: Neighborhood contrastive learning for scientific document representations with citation embeddings (2022)
19. Singh, A., D'Arcy, M., Cohan, A., Downey, D., Feldman, S.: Scirepeval: A multi-format benchmark for scientific document representations (2023)
20. Tang, X., Liu, Y., Cai, Z., Shao, Y., Lu, J., Zhang, Y., Deng, Z., Hu, H., An, K., Huang, R., Si, S., Chen, S., Zhao, H., Chen, L., Wang, Y., Liu, T., Jiang, Z., Chang, B., Fang, Y., Qin, Y., Zhou, W., Zhao, Y., Cohan, A., Gerstein, M.: Ml-bench: Evaluating large language models and agents for machine learning tasks on repository-level code (2024)
21. Tornede, A., Deng, D., Eimer, T., Giovanelli, J., Mohan, A., Ruhkopf, T., Segel, S., Theodorakopoulos, D., Tornede, T., Wachsmuth, H., Lindauer, M.: Automl in the age of large language models: Current challenges, future opportunities and risks (2024)
22. Wang, X., Hu, Z., Lu, P., Zhu, Y., Zhang, J., Subramaniam, S., Loomba, A.R., Zhang, S., Sun, Y., Wang, W.: Scibench: Evaluating college-level scientific problem-solving abilities of large language models (2024)
23. Wang, X., Wei, J., Schuurmans, D., Le, Q., Chi, E., Narang, S., Chowdhery, A., Zhou, D.: Self-consistency improves chain of thought reasoning in language models (2023)
24. Wang, Z., Zhou, Z., Song, D., Huang, Y., Chen, S., Ma, L., Zhang, T.: Where do large language models fail when generating code? (2024)
25. Yadav, P., Sun, Q., Ding, H., Li, X., Zhang, D., Tan, M., Ma, X., Bhatia, P., Nallapati, R., Ramanathan, M.K., Bansal, M., Xiang, B.: Exploring continual learning for code generation models (2023)
26. Zan, D., Yu, A., Liu, W., Chen, D., Shen, B., Li, W., Yao, Y., Gong, Y., Chen, X., Guan, B., Yang, Z., Wang, Y., Wang, Q., Cui, L.: Codes: Natural language to code repository via multi-layer sketch (2024)
27. Zelikman, E., Huang, Q., Poesia, G., Goodman, N.D., Haber, N.: Parsel: Algorithmic reasoning with language models by composing decompositions (2023)
28. Zhang, L., Zhang, Y., Ren, K., Li, D., Yang, Y.: Mlcopilot: Unleashing the power of large language models in solving machine learning tasks (2024)

A High Level Plans Generated

High Level Plan generated for CONPR on the Numerai dataset

```
**Step 1: Data Preprocessing and Feature Engineering**
* Perform data cleaning as described previously.
* Explore feature importance and dimensionality reduction techniques to select a subset of features within TabPFN's limit.
* Consider feature engineering techniques to create new, potentially informative features.
* Analyze the distribution of target classes and implement appropriate class balancing techniques if necessary.
* Split data era-wise, ensuring a balanced distribution of eras and target values across chunks or samples.
**Step 2: Addressing TabPFN Limitations**
* Implement chunking or sampling strategies based on data size and computational resources:
    * **Chunking:** Divide data into chunks with less than 10,000 data points,
    ensuring each chunk contains a representative distribution of eras and target values.
    * **Sampling:** If chunking is not feasible,
    employ stratified sampling to select a subset of eras while maintaining the overall data distribution.
**Step 3: Implementing Conservative Predictions with TabPFN**
* **Model Training:** Train TabPFN models on each data chunk or sample,
incorporating regularization techniques and early stopping to mitigate overfitting.
* **Confidence Estimation:**
    * Calculate Gini impurity, entropy, and maximum predicted probability for each prediction.
   * Evaluate and select the most suitable confidence measure based on its correlation with actual prediction accuracy on a validation set.
* **Thresholding:**
    * Determine the confidence threshold based on the chosen confidence measure and the desired balance between accuracy and support.
    * This could involve analyzing the distribution of confidence scores on a validation set.
* **Cascading:**
    * Implement a multi-stage cascading approach where uncertain predictions from the first model are passed,
    to subsequent models trained on different data chunks or samples.
    * Limit the number of cascading stages to avoid overfitting and diminishing returns.
    * For data points remaining uncertain after all stages, consider assigning them to the majority class,
    using a separate model for uncertain cases, or abstaining from prediction.
**Step 4: Evaluation**
* Evaluate the performance using accuracy, support, utility, DRAR, and Traded Sharpe Ratio.
* Compare the conservative prediction approach with a baseline TabPFN model and potentially other models suitable for tabular data.
* Analyze the characteristics of confident predictions and their alignment with the extremes of the target variable.
**Step 5:  Refined Pseudocode**
```python
def train_conservative_tabpfn(data, confidence_threshold, num_models=1):
    models = []
    for i in range(num_models):
        # Stratified sampling or chunking
        if data_size_exceeds_limit:
            data_chunk = get_stratified_sample_or_chunk(data, chunk_size_or_sample_size)
        else:
            data_chunk = data
        # Train TabPFN model with regularization
        model = TabPFN(regularization=...)
        model.fit(data_chunk)
        models.append(model)
    return models
def predict_conservatively(models, data, confidence_threshold, confidence_measure):
    predictions = []
    for x in data:
        for model in models:
            y_pred_proba = model.predict_proba(x)
            confidence = calculate_confidence(y_pred_proba, confidence_measure)
            if confidence <= confidence_threshold:
                predictions.append(y_pred_proba)
                break
        else:
            # Handle uncertain cases
            predictions.append(handle_uncertain_case(x, models))
    return predictions
# Example usage
models = train_conservative_tabpfn(numerai_data, confidence_threshold=0.5, num_models=3)
predictions = predict_conservatively(models, test_data, confidence_threshold=0.5, confidence_measure="entropy")
```
```

B Prompts used for Code Generation

Prompt for Base Code Generation

```
<role>:"system":
<content>:"You are a Machine Learning engineer.
Your task is to generate comprehensive and detailed
Python code for a complete end-to-end
Machine Learning pipeline. Ensure that no steps
are omitted due to brevity and that all aspects
of the implementation are thoroughly addressed.
Include all necessary data preprocessing, model
training, evaluation, and deployment steps.
Provide detailed explanations. Ensure the code is
robust, efficient,and follows best practices in
software development and machine learning.
Avoid using placeholders or simulating parts of
the code; provide fully functional implementations".
</content>

<role>:"user"
<content>:"idea:{idea} pipeline:{Pipeline}
miscellaneous:{miscellaneous}.Assume the data
is already loaded in train variable. Follow the
instructions clearly. Do all the complicated
implementations. Follow the \textit{Instruction};
\textit{Pipeline} strictly,
Ensure all steps are implemented comprehensively
and efficiently",
```

Prompt for Iterative Code Generator

```
<role>"user"</role>
<content>"Your task is to modify the existing
Python code provided in \n\n{Code}\n\naccording
to {Model_Idea}, ensuring alignment with
{model_refined_methodology}. Maintain the current
code structure while adapting it to meet the
requirements of {Model_Idea}. Avoid altering the
fundamental code layout. Aim for full
implementation without assuming any details.
"</content>

<role>"user"</role>
<content>"Ensure no assumptions are made; employ
various ML techniques creatively to align with
{Model_Idea}. Avoid brevity, placeholders, or
assumptions.Provide a comprehensive implementation."
```

Prompt for Pipeline Generation

```
<content>
You will be provided with an idea and the necessary
libraries. Your task is to outline a detailed
pipeline without giving actual code.

Review the provided context and Idea carefully.
Do not include the code; focus on describing the
pipeline.
Explain each step thoroughly, considering the dataset
characteristics and the proposed approach.
Utilize the provided libraries as necessary.
Aim for clarity and coherence in your response.
Approach this as a Machine Learning Researcher,
providing a step-by-step plan for the analysis.

[Idea]:{Idea}
</content>
```

C Some Code Snippets from Generated Codes

CONPR+M

```
. . . . . .
thresholds = np.percentile(base_impurities, [75])
for level in range(1, num_levels):
    keep_indices = base_impurities < thresholds[-1]
    if len(keep_indices) != len(X_train):
        keep_indices = np.repeat(keep_indices, len(X_train)
          // len(keep_indices) + 1)[:len(X_train)]
    X_train_pruned = X_train[keep_indices]
    y_train_pruned = y_train[keep_indices]
    model, accuracy, impurities, probabilities, predictions = train_and_evaluate(
    X_train_pruned, X_val, y_train_pruned, y_val)
    print(f"Level {level} Model Accuracy: {accuracy}")
    thresholds = np.concatenate((thresholds,
    np.percentile(impurities, [75])))
    models.append(model)
    accuracies.append(accuracy)

return models, accuracies
    . . . . . .
model = TabPFNClassifier(device='cpu',
N_ensemble_configurations=32)
model.fit(X_train, y_train)

# Predict probabilities and classes
val_predictions, probabilities = model.predict(X_val,return_winning_probability=True)

# Gini impurity of predictions
def compute_gini_impurity(probabilities):
    return 1 - np.sum(np.square(probabilities), axis=1)

impurities = compute_gini_impurity(probabilities)
accuracy = accuracy_score(y_val, val_predictions)

return model, accuracy, impurities, probabilities, val_predictions

    . . . . . .
    . . . . . .
```

Implementation of Cascading idea using Model M.

Implementation of Gini impurity

NNBST+M

```
. . . . . .
. . . . . .

def create_tabpfn_datasets(X, y, n_datasets=10, n_samples=1000, n_features=100):
datasets = []
for _ in range(n_datasets):
sample_idx = np.random.choice(X.shape[0], n_samples, replace=False)
      feature_idx = np.random.choice(X.shape[1], n_features,replace=False)
      X_sample = X.iloc[sample_idx, feature_idx]
      y_sample = y.iloc[sample_idx]
      datasets.append((X_sample, y_sample))
return datasets
. . . . . .

predictions = []

for model in best_models:
    if isinstance(model, TabPFNClassifier):
        tabpfn_datasets = create_tabpfn_datasets(test_preprocessed,

        np.zeros(len(test_preprocessed)))
        model_preds = []
        for X_tabpfn, _ in tabpfn_datasets:
            model_preds.append(model.predict_proba(X_tabpfn)[:, 1])
        predictions.append(np.mean(model_preds, axis=0))
    else:
        predictions.append(model.predict_proba(test_preprocessed)[:, 1])

ensemble_predictions = np.mean(predictions, axis=0)
. . . . . .
. . . . . .
```

Data scaling to cater Model M limitations

Model Ensembling to cater Model M limitation