Meaning-Typed Programming: Language-level Abstractions and Runtime for GenAI Applications

Abstract

Software is rapidly evolving from being programmed with traditional logical code, to *neuro-integrated* applications that leverage generative AI and large language models (LLMs) for application functionality. This shift increases the complexity of building applications, as developers now must reasoning about, program, and prompt LLMs. Despite efforts to create tools to assist with prompt engineering, these solutions often introduce additional layers of complexity to the development of neuro-integrated applications.

This paper proposes *meaning-typed programming* (MTP), a novel approach to simplify the creation of neuro-integrated applications by introducing new language-level abstractions that hide the complexities of LLM integration. *Our key insight is that typical conventional code already possesses a high level of semantic richness that can be automatically reasoned about, as it is designed to be readable and maintainable by humans.* Leveraging this insight, we conceptualize LLMs as meaning-typed code constructs and introduce a **by** abstraction at the language level, **MT-IR**, a new meaning-based intermediate representation at the compiler level, and **MT-Runtime**, an automated run-time engine for LLM integration and operations. We implement MTP in a production-grade Python super-set language called **Jac** and perform an extensive evaluation. Our results demonstrate that MTP not only simplifies the development process but also meets or exceeds the efficacy of state-of-the-art manual and tool-assisted prompt engineering techniques in terms of accuracy and usability.

1 Introduction

Software development is rapidly evolving towards *neuro-integrated* applications that leverage generative AI and large language models (LLMs) for critical functionality. This shift represents a convergence of conventional programming with the powerful capabilities of AI, promising to revolutionize how we build software. However, integrating LLMs into traditional programming remains a complex challenge due to the fundamental differences between operating assumptions. In conventional programming, code describes operations performed on explicitly defined objects (Figure 1 left). In contrast, LLMs process natural language text as input and produce text-based outputs. To integrate LLMs into programs, developers must manually construct textual input, a process known as *prompt engineering*.

Prompt engineering, the primary means of LLM programming, can be tedious, complex, and time-consuming. Developers must craft precise instructions, manage textual context, and handle the inherent variability in LLM output. This process lacks the structure, consistency, and debugging capabilities developers are accustomed to in traditional programming environments.

Several efforts have been made to address these challenges by developing infrastructure and tools to facilitate prompt engineering. Frameworks such as LangChain, Language Model Query Language (LMQL) [4], DSPy [16], and SGLang [46] have been introduced to assist developers in generating and managing prompts. However, while these frameworks provide more sophisticated tooling to help bridge the gap between conventional programming and the incorporation of AI models, in practice, they introduce additional layers of complexity for developers:

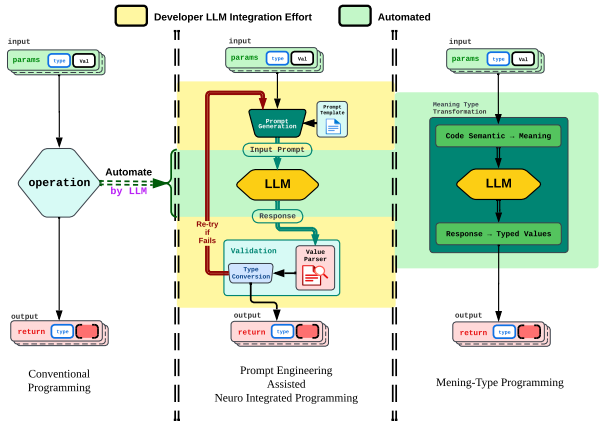

Figure 1. Comparison of LLM as Meaning Typed Code construct abstraction (Right) with the present-day LLM abstraction (Middle) and Symbolic Programming Abstraction (Left) of a programming function.

1. *Prompt design complexity:* Developers remain responsible for the manual construction of prompts, including determining the appropriate prompt language and selecting relevant information to include.
2. *Steep learning curve:* These systems require developers to familiarize themselves with new query languages, frameworks, or specialized syntax, complicating the integration process.
3. *Input/output conversion complexity:* Parsing LLM output and converting them back to objects remains a challenge, especially given the variability in output across different LLMs.

The **key insights** of this work are: *(1) In practice, programs are written to be readable by various developers to comprehend the intentions and semantics conveyed by the code. This semantic richness can be leveraged to automatically translate intent embodied in the code to prompts for the LLM. (2) As state-of-the-art LLMs continue to advance, AI is increasingly feasible to infer code intentions without requiring explicit descriptions or prompt construction by developers. This trend is not only supported by our empirical findings, but we surprisingly find less can be more in prompt engineering.*

Based on this insight, this paper proposes *meaning-typed programming*, a novel approach to simplify the creation of neuro-integrated applications by introducing new language-level abstractions that hide the complexities of LLM integration. There are three key components to enable meaning-typed programs. These include **(1) The 'by' operator** that enables the seamless integration of LLM functionality into

code. It acts as a bridge between traditional and LLM operations, allowing developers to replace function bodies with a simple by `model_name` clause:

```
def calc_age(birth: date, today: date) -> int by gpt4();
```

This declaration, along with the function signature and available semantic information, is sufficient for the LLM to produce the correct return value at runtime, significantly simplifying LLM integration in existing codebases. **(2) MT-IR**, a meaning-based intermediate representation at the compiler level that collects and organizes semantically rich information, preserving the meaning behind variable names, function signatures, and comments. MT-IR is also used dynamically to bind run-time values to semantic representations, enhancing the model's understanding of program context and enabling effective, context-aware LLM integration. **(3) MT-Runtime**, an automated runtime engine within the language's virtual machine, triggered at by call sites. It binds runtime values to MT-IR, manages LLM interactions, generates context-aware prompts, and handles responses and errors. MT-Runtime allows developers to focus on application logic while automatically managing LLM integration, ensuring seamless and contextualized operations.

We implement *meaning-typed programming (MTP)* in a production-grade Python superset language called **Jac** and perform an extensive evaluation. Specifically, we make the following contributions:

- We introduce the concept of **meaning-typed programming**, a novel paradigm that treats LLMs as meaning-typed code constructs, enabling seamless integration of LLMs into conventional programs without explicit prompt engineering.
- We present the design and implementation of the MTP framework: the **'by' operator** for intuitive LLM integration; **MT-IR**, a semantically rich intermediate representation for implied meaning; and **MT-Runtime**, an automated runtime engine for LLM operations
- We **implement MTP in Jac**, a production-grade Python superset language, conduct extensive evaluation and show that MTP significantly reduces development complexity, lines of code, and costs while improving run-time performance and maintaining or exceeding the accuracy of existing approaches.
- We provide an empirical analysis of **LLM evolution trends**, demonstrating how MTP takes advantage of advances in LLM capabilities to simplify neuro-integrated programming over time.
- We present multiple **case studies** demonstrating the application of MTP in complex problem domains, analyzing its impact on code structure, development workflow, and complexity reduction compared to traditional LLM integration methods.

Our results show that MTP simplifies the development process and meets or exceeds the efficacy of state-of-the-art manual and tool-assisted prompt engineering techniques. This approach significantly reduces development complexity and lines of code while improving run-time performance and costs associated with LLM integration.

2 Problem, Background, and Insights

Figure 2 illustrates the challenge of integrating LLMs into traditional programming. The example shows a function

Conventional Prompt Engineering

```
from openai import OpenAI

gpt = OpenAI()

def respond_with_confidence_level(query: str,
    chat_history: dict[str,str],
    user_data:dict) -> tuple[str,float]:

    system_message = (
        "You are a chatbot. Respond to the chat using the available
              information. Additionally generate a confidence level of the
              response as a percentage" )

    prompt = system_message
    prompt += f'\nQuery form the user:\n{query}'
    prompt += '\nChat History:\n'
    for user,content in chat_history:
        prompt += f'{user} : {content} \n'
    prompt += '\nUser Information:\n'
    for key,value in user_data:
        prompt += f'{key} : {value} \n'
    prompt += '\nFollow the format given below for response.'
    prompt += '\nEnd response with newline.'
    prompt += '\nGive confidence level as a float.'
    prompt += '\nResponse:<output>\n'
    prompt += '\nConfidence:<output>\n'

    response = gpt.chat.completions.create(
              messages=[ {"role": "user","content": prompt}],
              model="gpt-4")

    lines = response.strip().split('\n')
    for line in lines:
        if line.startswith("Response:"):
            chatbot_response = line[len("Response:"):].strip()
        elif line.startswith("Confidence:"):
            confidence_level = line[len("Confidence:"):].strip()
            try:
                confidence_level = float(confidence_level)
            except:
                confidence_level = 50.0

    return (chatbot_response, confidence_level)
```

Meaning-Typed Programming

```
from MTP import gpt4

def respond_with_confidence_level(query: str,
       chat_history: dict[str,str],
       user_profile_data:dict) -> tuple[str,float] by gpt4
```

Figure 2. The elevation of abstraction level form generic prompt engineering (top) to neuro-integrated programming (bottom).

`respond_with_confidence_level()`, which generates a chat response based on a user query, previous chat history, and a user profile while also returning the AI's confidence level in that response. The top code snippet demonstrates the current state-of-the-art approach, where a developer manually constructs a prompt for the LLM (lines 5-20) and then interprets the results (lines 25-35). This process introduces several key challenges:

1. **Prompt Design Complexity**: Selecting appropriate string templates, combining them, and constructing prompts to accurately convey the semantic intent and LLM behavior constraints is challenging and time-consuming. Lines 5-20 show the developer manually structuring the prompt, iterating through data structures to add information.
2. **Input/Output Conversion Complexity**: Converting between the program's data structures and the text-based inputs and outputs of LLMs is difficult and error-prone, especially when dealing with complex object types and unpredictable LLM output formats. We illustrate this point relating to object types with class methods in Section 8. Lines 25-35 show the difficulty with type conversions: the developer needs to use a try-catch block to address type mismatch situations.
3. **Maintenance and Adaptability**: When using different LLMs (e.g., GPT, Gemini, Llama), the output utterances can be drastically different and unpredictable,

making using generalized code for output format conversion difficult. This variability complicates code maintenance as models evolve.

In contrast, the bottom snippet in Figure 2 presents our solution for seamless LLM integration using the MTP approach. This approach requires no code changes to the function signature and eliminates the need for manual prompt construction and result interpretation. The only addition is a language construct by `llm()` that tells the compiler to introspect on the function definition using an LLM and generate code corresponding to its intended meaning.

2.1 Limitations of Prompting Frameworks

Several frameworks and tools have been developed to simplify the interaction with LLMs, including LangChain, LMQL, DSPy, and SGLang [4, 10, 16, 46]. While these systems offer valuable capabilities for prompt engineering and LLM integration, they usually incur additional complexity. There is a learning curve for developers to adapt to new query languages, frameworks and specialized syntax, and the prompts that interact with the LLMs still need to be refined by the programmer. While some frameworks like DSPy aim to optimize prompts, they do not hide prompt engineering complexity from developers and incur a run-time cost.

Our approach aims to fully abstract and hide this complexity to minimize manual effort. This enables seamless integration of LLMs into programming languages without requiring developers to learn additional tools.

2.2 The MTP way

Our insight is that the semantic richness of high-level programs can be harnessed to figure out the programmer's intent and to automatically translate between the programming language and the natural language understood by LLMs. Developer intentions are inherent in well-written code, which allows for automatic inference to extract its intended meaning.

Consider the example in Figure 2, the purpose of the function `respond_with_confidence_level()` can be inferred from its name, input variable names, and output. Figure 3 presents three additional illustrative examples showing how meaning can be derived from various code elements:

1. The meaning of the variable dob can be inferred from its name, type, and value. And if it wasn't already clear to the reader, dob is a a commonly used abbreviation for date of birth.
2. The semantics of the `description()` method in the book class can be understood by analyzing the class, variable names, and an example function call.

We show that with the help of a compiler and run-time techniques, LLMs can infer the intended meaning from a program to automatically generate code. As LLMs continue to evolve, the need for explicit prompt engineering or manual meaning extraction diminishes. We present an empirical analysis across generations of LLMs in the evaluation section to demonstrate this trend. We address the following research questions:

1. What is the simplest abstraction to add to a programming language for LLM integration?
2. How can we design this abstraction to work effectively with state-of-the-art generative AI models?

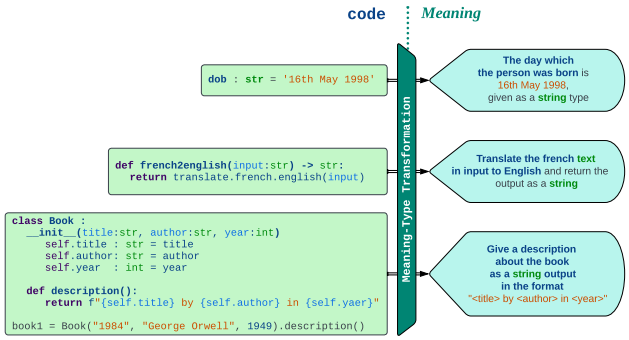

Figure 3. Examples of *meaning* of traditional code constructs in symbolic programs.

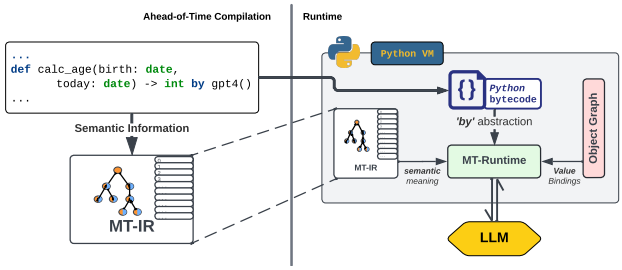

Figure 4. Meaning-Typed Programming system overview

3. How can we automate and abstract away prompt engineering complexity?
4. How does this technique generalize across various application domains?
5. What are the implications for run-time performance and cost when using this approach?

3 Meaning-type Programming: System Overview

The MTP paradigm (Figure 4) consists of three key components that work together to provide a seamless integration of LLMs into programming languages:

1. The **by** operator at the language level
2. Meaning-Typed Intermediate Representation (MT-IR), generated during compilation
3. MT-Runtime, an automated runtime engine that manages the LLM integration

The by operator acts as a bridge between traditional code and LLM operations. Developers can seamlessly integrate LLM functionality into their code simply by adding the keyword **by**. As illustrated by Figure 4's code snippet, developers can integrate an LLM into their code, replacing the function(`calc_age()`)'s implementation with a simple **by** `model_name` clause. At run-time, **by** operator will invoke MT-Runtime to achieve the desired functionality by leveraging the specified LLM. In this example, the system will use gpt4 to calculate the age based on the birthday and today's date. In Section 4, we present more detail on the **by** operator.

MT-IR is an intermediate representation that focuses on meaning. During compilation, MT-IR is created by analyzing and organizing semantically rich information in the program (see Figure 4 (left)). It preserves important aspects like

```
def calculate_age (cur_year: int, dob: str) -> int by llm

age = calculate_age(2024,"March 14, 1879")
```

(a) Function Call

```
class Person :
    name : str
    dob : str

Einstein = Person("Einstein") by llm
```

(b) Object Initialization

```
class Person:
    name : str
    dob : str

    def calculate_age (cur_year: int) -> int by llm(temperature=0.7)

Einstein = Person("Einstein", "March 14, 1879")
Einstein.calculate_age(2024)
```

(c) Member Method Call

Figure 5. Meaning-Typed Programming supports three integration points in symbolic code where LLMs can be leveraged automatically.

variable names, function signatures, object schemas, and comments, to enable better extraction of the the code's original meaning by the LLM at run-time (see Figure 4 (right)). Section 5 presents more details on MT-IR design and compilation pipeline.

MT-Runtime manages the complexities of LLM interaction, through context-aware prompt synthesis using MT-IR, intelligent response parsing, and dynamic error handling. It operates within the language's virtual machine (e.g., Python's VM) and is specifically triggered at **by** call sites. MT-Runtime combines both static information from MT-IR and dynamic values from the language's object graph. By seamlessly integrating with the language's execution environment, MT-Runtime allows developers to focus on their application logic while automating LLM interaction (see Section 6 for more details).

4 Meaning-Type Language Construct: 'by' operator

In this Section, we describe the **by** operator and multiple scenarios where it can be used to integrate a LLM by invoking MT-Runtime. There are three scenarios in modern programming languages for which the **by** *llm* abstraction can be used. These include *function calls*, *object initialization* and *Member method call*.

Function Calls - A function call typically contains a function name, input arguments as well as input and output type annotations. As shown in figure 5a, the intention of the function is quite clear being required to `calculate_age` of a person based on their `dob` (date of birth) and `cur_year` (current year). The output type hint being an integer suggests that the result must be the number of years. When invoked, the semantic meaning of the code is passed to the automated runtime management system and the system will return the output value of the function in the correct type. In the case of the example the age of the person is returned as an integer.

Object Initialization - Figure 5b shows an example of integrating LLMs into object initialization. The **by** `llm` language construct is used in this case, as shown in line 5, to use the values of the partially initialized object (name being "Einstein") to predict and automatically fill in the remaining attributes (dob) accordingly. In this case, MT-runtime is invoked by **by**, then it uses the class name, child attribute names and types from MT-IR and the dynamic value of name ("Einstein"), to generate a prompt to the specified llm and convert the results the correct type (str) to finish initiate the object.

Member Method Call - In contrast to standalone function calls, methods of a class has the access to the class' attributes and such information is needed for the semantic meaning extraction. In the example in Figure 5c, the `calculate_age` method of the class **Person** does not contain the date of birth as an argument, but it has access to the **dob** attribute of the Einstein object. When integrating the **by** `llm` call in line 5, it results the invocation of MT-runtime where additionally to the semantics of the method itself, the semantic meaning of the class name, attribute and their types are also passed in through MT-IR. This allows the LLM to reason better about the intention of the method call in related to the object. Note that in this example, we also illustrate syntax that allows for model hyperparameter configuration (temperature, etc) for more advanced users.

5 Meaning-Typed Compilation and IR

We implement the **by** language abstraction, MT-IR, and our compiler pass through Jac, a production-grade Python superset language [20, 26]. Typically, when a Python program is executed, the source code is compiled into Python bytecode via ahead-of-time compilation. The bytecode is then interpreted or compiled in a just-in-time fashion at runtime. Jac extends Python in the form of a PyPI package [19], providing a CLI command `jac` that replaces the standard Python compiler by inserting a customized jac compilation phase. The resulted python bytecode will then be interpreted using standard `python3`.

The compilation process for MTP in Jac is shown in Figure 6 and involves several key steps:

1. **MTIR Generation**: An additional compiler pass is introduced to produce MT-IR. This pass extracts and organizes semantic elements from the AST, including object names, attributes, variable names, and developer comments.
2. **Analysis Passes**: Multiple analysis passes are performed on the AST to gather and refine information for the MT-IR such as symbol table information.
3. **Lowering to Python AST**: The Jac AST is then lowered to a Python AST. During this process, the **by** code constructs are replaced with calls to the MT-Runtime system (indicated with the red AST nodes in Figure 6).
4. **Type Analysis**: After the Python AST is created, additional Python analyses are run. In particular, we perform type checking on this AST using Python's official MYPY type checker [29]. The resulting type information is then added back to the MT-IR, further enriching its semantic content.
5. **MTIR Registration**: The generated and enriched MT-IR is registered into the MT-Runtime, a part of the Jac library that remains available throughout the execution of the Jac program.
6. **Bytecode Generation**: Finally, the modified Python AST is used to generate standard Python bytecode.

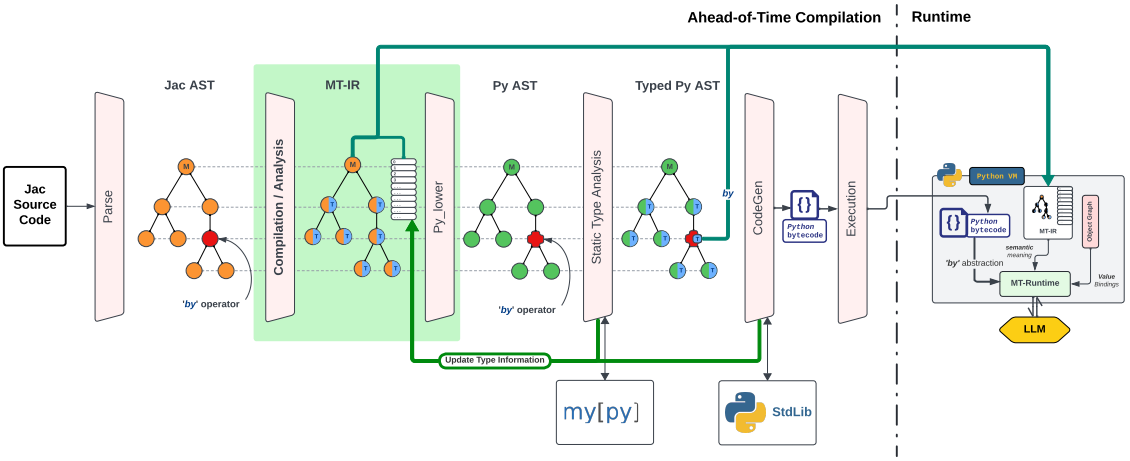

Figure 6. Meaning-Typed Compilation and MT-IR.

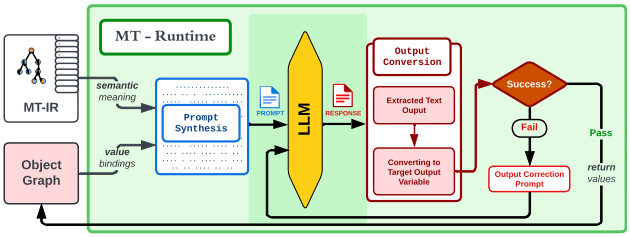

Figure 7. Meaning-typed Runtime (MT-Runtime)

This approach allows us to preserve crucial semantic elements that would typically be lost in standard Python AOT compilation. The MT-IR, persisting within the Python VM during runtime, serves as a repository of semantic information that can be dynamically updated and accessed.

During execution, the MT-Runtime system extracts semantic information via MT-IR as needed for LLM integration, as discussed in Section 3. This enables the seamless integration of the **by** operator and meaning-typed programming constructs within a Python-compatible environment.

Note that while we've implemented MTP using Jac, the core concepts of MT-IR and the **by** abstraction are not Jac-specific. Indeed, through Jac's Python library, we provide a complete set of Python decorators and library calls to allow developers to utilize the **by** abstraction in pure Python. That being said, the Jac-based implementation offers a more integrated solution for meaning-typed programming.

6 Meaning-Typed Runtime System

MT-Runtime is a novel automated runtime engine that resides with the language's virtual machine. This runtime is triggered specifically at the call sites of the **by** abstraction and automatically handles the LLM invocation at runtime. Figure 7 shows an overview of the MT-Runtime engine. In this Section, we describe the design of MT-Runtime, which operates in the following three conceptual stages.

1. Meaning Extraction via MT-IR. Upon the triggering of a **by** call, MT-Runtime first extracts the relevant semantic meanings, in preparation for the synthesis and construction of the input to the LLM. MT-Runtime access the meanings via two avenues. First, it retrieves, from MT-IR, the statically-defined semantic information of the relevant constructs, which includes function signature, type information, code comments and variable names. Second, MT-Runtime leverages the Python object graph to retrieve the dynamic value available during execution runtime and bind them to their corresponding holders in the MT-IR. MT-Runtime curates this set of semantic meanings differently depending on the integration point of the **by** call site (Section 4) and use them for prompt synthesis.

Algorithm 1 Prompt Synthesizer

Require: Task T, Code C, Returns R, MT-IR S
Ensure: Prompt P
Initialize prompt $P \leftarrow \emptyset$
$P \leftarrow T$
for each symbol $s_i \in C$ **do**
  if symbol s_i is a basic type **then**
    $m_i \leftarrow m$ for s_i in S
  else
    Unroll custom type:
    Call `UnrollType`(s_i) :
    while s_i contains custom typed children **do**
      for each child s_j of s_i **do**
        $m_j \leftarrow m$ for s_j in S
        if s_j is a basic type **then**
          Append meaning m_j for s_j to e
        else
          $e \leftarrow$ `UnrollType`(s_j)
        end if
      end for
    end while
    Append explanation e to P
  end if
  Append m_i to P
end for
Append "Return type and format:" to P
for each symbol $r_i \in R$ **do**
  if symbol r_i is a basic type **then**
    $m_i \leftarrow m$ for r_i in S
  else
    Call `UnrollType`(r_i) :
  end if
  $m_i \leftarrow m_i$ to P
end for
return P

2. Prompt Synthesis from Meanings. With the relevant set of semantic meanings at the ready, MT-Runtime synthesize the input to the LLM. Algorithm 1 shows a pseudo-code of the prompt synthesis algorithm. Depending on the location of the **by** call site, MT-Runtime constructs a prompt that includes the static semantics and dynamic values of variables contextually relevant to the **by** call site, and the expected type of the output variable. For example, for a **by** call defined for a function call (Figure 5a), MT-Runtime will synthesize a prompt including information on the name and type of the function parameters and the function return type.

In the case of complex variables with nested custom-typed attributes, MT-Runtime recursively performs a type unrolling operation to capture detailed attributes of the variables in the prompt. With each unrolling step, MT-Runtime accesses from MT-IR the semantic information of the attributes. This unrolling step is described by line 6 - 21 in Algorithm 1.

3. Conversion from LLM Output to Target Output Variable. MT-Runtime queries the LLM with the dynamically synthesized input prompt and receive an output. We need to parse the LLM textual output and convert it to the target output variable. There exists two key challenges.

First, converting the LLM output to a target output variable is of non-primitive custom type requires special handling, as direct type casting will not be sufficient. MT-Runtime addressees this by instructing the model to generate output following the python object schema and leverage Python's built-in $ast.literal_eval()$ to evaluate the output text and construct the variable instance. For example, in Figure 5b, the variable *Einstein* is of the custom type *Person* with attribute $name : str$ and $dob : str$. In this case, MT-Runtime instructs the LLM to generate an output such as $Person(name = \text{"}Albert\ Einstein\text{"}, dob = \text{"}03/14/1879\text{"})$ which will evaluate to a valid *Person*-typed object.

Second, because of the probabilistic nature of LLM inference, it is not guaranteed that the LLM output will conform to the expected output type. In the case where MT-Runtime is not able to convert the LLM output to the target output variable, MT-Runtime constructs a revised prompt including details of the current output and the details of the expected type. MT-Runtime queries the LLM with this revised input and attempt to parse its output again. With this approach, the second revised prompt is mostly short and concise since it only aims to correct the type mis-match. This results in less overall LLM token usage, lower cost and faster inference speed compared to other retry mechanism used in other frameworks (e.g. DSPy).

7 Evaluation

In this section, we present the evaluation of MTP against the state-of-the-art. Our evaluation include the following aspects.

1. **Lines of code (LOC)**. We first evaluate MTP for its effectiveness in reducing programming complexity through a quantitative study measuring lines of code across a comprehensive suite of benchmarks (Section 7.2).
2. **Accuracy and Trend across LLM evolution**. We evaluate the program accuracy when using MTP versus prior work across 10 various LLMs (Section 7.3).
3. **Token Usage, Cost and Runtime** We further measured those performance metrics for MTP compared to DSPy.
4. **User study** In addition to the LOC study, we conducted a user study to evaluate MTP's effectiveness in reducing complexity and increasing developer's productivity.
5. **Case study** Lastly we present case studies of leveraging MT-LLM to integrate LLMs in more complex applications.

Table 1. Benchmark Descriptions

Benchmarks	Task Description
Math Problem	Given a grade school math question an integer answer is returned. [16]
Translation	Translates a given text from English to French.[4]
Essay Reviewer	Reviews an essay and assigns a letter grade based on the quality of the essay. [46]
Joke Generator	Generates jokes based on a given topic, e.g., jokes with punchline. [4]
Expert Answer	Given a question, predicts the expert profession best suited to answer and generates an answer. [46]
Odd Word Out	Identifies the odd word out in a list of words along with a reason for why it is the odd word out. [4]
MCQ Reasoning	Answers multiple-choice questions by providing a reason for the chosen answer.[4]
Personality Finder	Given a public figure's name, predicts their personality type.
Template	Given a template object and a set of variables, fills in the template with the variables.[4]
Text to Type	Given a text input and a structured data type, extracts the structured data into the structured data type.[4]
Taskman	Given a list of tasks, assigns priority ranks and estimates completion times for each task.
Level Generator	Generates new game-level maps based on previous level playthroughs for a role playing game.
Wikipedia	Given a question and an external tool (e.g., Wikipedia), retrieves the relevant answer using ReAct methodology. [4]

7.1 Experimental Setup

We use OpenAI APIs for experiments with GPT models. We host llama models on a local server with a Nvidia 3090 GPU with 24GB VRAM and 64GB RAM. We instrument the source code to profile token usage. Python cProfile[35] is used to measure program runtime. Our benchmark suite consists of problems representative of applications with LLM integration [4, 16, 46]. To construct the suite, we use the entire set of benchmarks and examples from prior work [4], supplemented with all relevant benchmarks in other work [16, 46] (Table 1).

7.2 Complexity - Line of Code (LOC)

We evaluate MTP for its effectiveness in reducing programming complexity through a quantitative study measuring lines of code across our benchmark suite. Each benchmark was implemented using all three frameworks, LMQL, DSPy and MTP. Figure 8 presents the lines of code for LLM integration across benchmarks. LOC for LLM integration counts the amount of lines needed to be added or modified when a developer is integration a LLM into traditional code. For example, in Figure 2, the LOC for integration using the conventional approach is 41 lines vs. MTP's 6 lines. Figure 8 shows that minimum code changes are required to integrate a LLM into traditional programming when using MTP, while DSPy and LMQL require much larger LOC.

Table 2 summarizes the LOC reduction achieved by MTP, on average, 5.3x and 4.0x reduction compared to DSPy and LMQL respectively. *This large reduction is because DSPy requires tedious type conversions and LMQL requires manual*

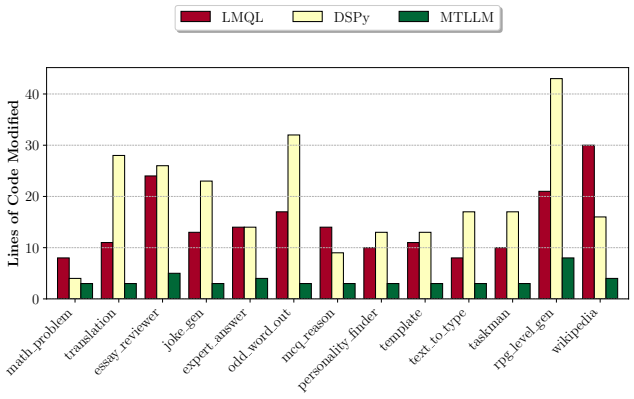

Figure 8. Lines of Code for LLM integration across benchmark programs.

programmatic prompt engineering. In contrast, MTP requires minimum code change for LLM integration by simply leveraging the **by** *operator and hiding all complexity.* Our case study 8 presents an example of the code changes required in DSPy versus MTP.

7.3 Program accuracy

The previous section has shown that MTP can significantly reduce the amount of code changes and programming complexity. In this section, we aim to evaluate if MTP's complexity reduction comes at a cost of accuracy loss. In addition, we aim to investigate the impact of LLM evolution on program accuracy when using MTP compared to previous work.

We first use the benchmark suite described in section 7.1 for our evaluation using GPT-4o. Since 11 out of 12 benchmarks have text outputs with no reference output, we manually evaluate the output for its correctness. MTP achieved the highest accuracy by producing correct output on 12 out of the 12 benchmarks, while DSPy only achieved correctness 5 out of the 12. We observed that DSPy's common issues are *"incorrectly formatted outputs"* and *"incorrectly typed outputs"*. These issues were not observed in MTP, demonstrating the robustness of MTP's automatic prompt generation and type checking capabilities. LMQL is able to produce correct output on all 12 benchmarks. However, LMQL requires developers to manually craft prompts as opposed to the automated prompt generation of MTP and DSPy, resulting in a high complexity and lower developer productivity, as further demonstrated in our user study (Section 7.5).

7.3.1 Accuracy for GSM8k and Trends Across Model Evolution. In addition, we follow the same methodology as in previous work [16] for the evaluation of accuracy by focusing on the mathematical problem benchmark using a standard dataset, GSM8K [8], with expected answers. This dataset allows us to perform an objective evaluation of accuracy across a variety of inputs. We sample 300 question-answer pairs and evaluate the accuracy of the programs implemented using MTP and DSPy. Furthermore, we conduct the accuracy evaluation using 10 LLMs across generations throughout the model evolution to understand the accuracy trend as LLMs become more intelligent.

MTP vs. DSPy. As shown in Figure 9, for the latest GPT models (GPT-4-turbo and GPT-4o) DSPy and MTP perform similarly while MTP outperform slightly for the most recent gpt-4o model. This shows that although MTP hides all of the prompt engineering and type-conversion complexity, it achieves similar and sometimes even better performance accuracy through high quality automatic prompt synthesis and output interpretation.

Even more interestingly, we observe that *as the models continue to evolve and improve, the accuracy of programs implemented using MTP improves significantly.* while DSPy's accuracy plateaued and also degraded a bit. *This trend indicates that as models get better, the need for complex prompt engineering actually is largely reduced.* LLM's increasing capability to understand the traditional code, extract semantic meanings from the code and generate desirable output enabled MTP's high-accurate automatic prompt synthesis and output interpretation.

MTP vs. DSPy (Compiled). It is also interesting to note that DSPy supports a compilation mode that requires additional training examples. It then searches across a set of prompt templates to optimize the prompt to improve accuracy, which could be time-consuming. We observe that MTP (with no training) outperforms DSPy (compiled with training) on all LLMs with the exception of gpt-4 (Figure 9). These results further highlight the effectiveness of MTP in automatically generating robust prompts using the semantic elements available in MT-IR.

Llama models. Figure 10 shows a similar trend: MTP accuracy improves as the Llama model gets better, indicating the decreasing need for complex prompt engineering for more intelligent models.

Figure 11 presents the failure / success breakdowns between the two approaches. It shows that for DSPy, the inaccuracy when using GPT models is often due to "type mismatch": e.g. the traditional code is expecting an integer output, yet the automatic type conversion in DSPy failed. For llama, the degradation of DSPy's accuracy at the latest model is mostly due to longer execution time of the LLMs (timeout set to be 120s). In the next section, we dive deeper into the token usage of each approach to further evaluate the runtime and latency of the systems.

Table 2. Lines of Code comparison between DSPy, LMQL and MTP across multiple benchmark applications. LOC on the left and LOC for LLM integration on the right.

Problem	LoC for LLM Integration with MTP	
	vs. LMQL	vs. DSPy
Math Problem	↓ 2.7×	↓ 1.3×
Translation	↓ 3.7×	↓ 9.3×
Essay Reviewer	↓ 4.8×	↓ 5.2×
Joke Generator	↓ 4.3×	↓ 7.7×
Expert Answer	↓ 3.5×	↓ 3.5×
Odd Word Out	↓ 5.7×	↓ 10.7×
MCQ Reasoning	↓ 4.7×	↓ 3.0×
Personality Finder	↓ 3.3×	↓ 4.3×
Template	↓ 3.7×	↓ 4.3×
Text to Type	↓ 2.7×	↓ 5.7×
Taskman	↓ 3.3×	↓ 5.7×
RPG Level Generator	↓ 2.3×	↓ 4.8×
Wikipedia (ReAct)	↓ 7.5×	↓ 4.0×
Average	**↓ 4.0×**	**↓ 5.3×**

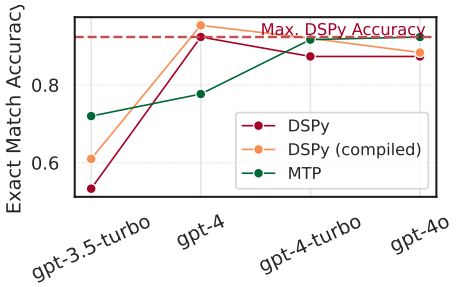

Figure 9. Accuracy on GSM8K across OpenAI models.

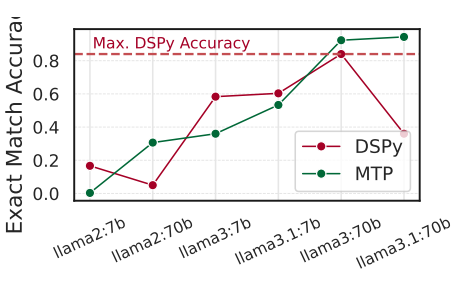

Figure 10. Accuracy on GSM8K across Llama models.

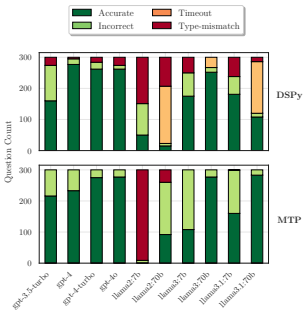

Figure 11. Results breakdown for DSPy and MTP on GSM8K. Timeout set at 120s.

7.4 Token Usage, Cost and Runtime

Token usage is an important metric since a higher token count leads to longer runtime (LLM inference time) and increased expenditure. Figure 12 presents the token usage comparison between DSPy (not-compiled) and MTP across the benchmark suite both for prompt (input) and completion (output) token usage. Across the benchmark suite, we observe that MTP consistently uses fewer tokens compared to DSPy, including the math problem using GSM8K dataset (first cluster of bars). This trend also holds true for all GPTs we have tested from gpt-3.5-turbo to gpt-4o both in with and without compiled mode. Figure 13 presents the runtime improvement and cost reduction achieved by MTP over DSPy. For each benchmark, the first bar shows the DSPy's cost relative to MTP (left y-axis) and the second bar shows the MTP's runtime improvement over DSPy (right-axis). The inference cost with the gpt-4o model is calculated using the OpenAI's pricing formula [32]. The runtime is measured using cprofile. Figure 13 demonstrates that MTP achieves significant runtime improvement and cost reduction due to its more efficient token usage.

7.5 Complexity - User study

In addition to the quantitative LOC study, we conducted a user study consisting of 20 software developers to gain further insight into the effectiveness of MTP in reducing programming complexity and improving developer productivity, compared to prior work.

7.5.1 User Study Protocol. We issued a call soliciting developers with solid software engineering background to participate in the study. During the study, 20 Participants were first given one day to familiarize themselves with MTP, DSPy, and LMQL, through standard documentations, tutorials and available online resources. Participants were then tasked with implementing three progressively challenging programming tasks utilizing all three frameworks to integrate LLMs in traditional programming. These tasks were selected from our benchmark suite including: Essay Evaluator (Easy),*Task Manager (Medium)* and *Game Level Generator (Hard)*. Each participant was allocated 90 minutes per task. After completing coding, participants filled out a questionnaire and provided feedback on the framework.

7.5.2 Success Rate. Figure 14 presents the success rates across three programming tasks using DSPy, LMQL, and MTP. Success is defined as when the implementation a participate developed generates accurate outputs across test inputs. Overall, implementation utilizing MTP achieved the highest success rate in 2 out of the 3 tasks. It also performs the most consistently across all tasks. These results demonstrate MTP's effectiveness in enabling programmers to leverage LLM capabilities intuitively. In contrast, we observe that participants found using LMQL to be the most challenging, with the lowest success rates. Qualitative feedback from participants indicated that this is because LMQL requires developers to manually craft prompts while DSPy and MTP automate the process. For the most challenging problem, *Game Level Generator (Hard)*, developers achieved zero success case, highlighting the difficulty to manually craft prompt and then manually interpret output to convert to a complex object type. This problem is further discussed in case study 8.

Table 3. Questions asked in the user study questionnaire, grouped under five criteria: Ease of Use, Clarity of Documentation, Learning Curve, Efficiency of Problem-Solving, and Overall Satisfaction.

Category	Question #	Statement
Ease of use	Q1	How easy was it to set up and start using the framework?
	Q2	How intuitive do you find the syntax and structure of the framework?
	Q3	How would you rate the ease of performing common tasks with the framework?
	Q4	How quickly will be able to integrate the framework into your existing projects?
Clarity of Documentation	Q5	How clear and understandable do you find the official documentation?
	Q6	How well is the documentation structured to find information quickly?
	Q7	How helpful are the provided examples and tutorials in understanding the framework?
	Q8	How complete and detailed are the code examples in the documentation?
Learning Curve	Q9	How long did it take you to feel comfortable with the basic features of the framework?
	Q10	How easy was it to learn and implement advanced features of the framework?
	Q11	How well does the framework support users in learning and utilizing advanced concepts?
Efficiency of Problem-Solving	Q12	How efficient is the framework in solving problems compared to others you have used?
	Q13	How easy was it to learn and implement advanced features of the framework?
Overall Satisfaction	Q14	Overall, how satisfied are you with the framework?
	Q15	Would you recommend this framework to others?
	Q16	How likely are you to continue using this framework in your future projects?
	Q17	What are the main reasons for your rating in the previous question?

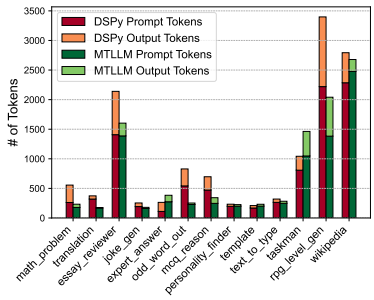

Figure 12. Token usage comparison between MTP and DSPy.

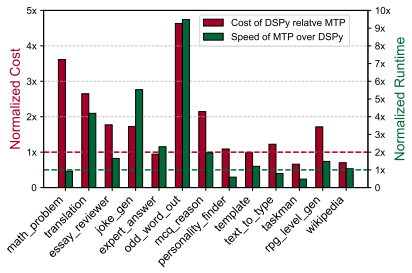

Figure 13. Cost and inference speed comparison between MTP and DSPy.

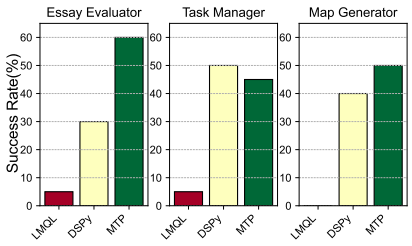

Figure 14. User study success rates across LMQL, DSPy, and MTP.

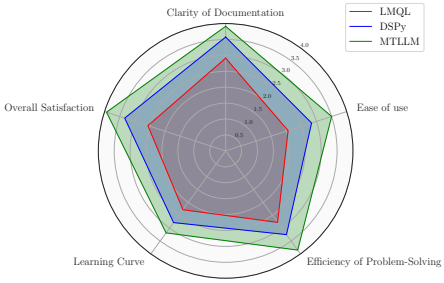

Figure 15. User evaluation of LMQL, DSPy and MTP on five usability criteria

7.5.3 User Study Feedback. Our questionnaire consists of 17 questions grouped under five criteria: Ease of Use, Clarity of Documentation, Learning Curve, Efficiency of Problem-Solving, and Overall Satisfaction (Table 3). The average user scores for each framework across the five criteria are shown in Figure 15. MTP was consistently rated the better framework compared to LMQL and DSPy. DSPy generally scored in the mid-range, while LMQL scored the lowest across all criteria, suggesting it introduced the most complexity for programmers.

The Violin plot in figure 16 shows the distribution of scores given by participants for each framework. Across all criteria, MTP exhibits higher density around the upper rating regions (higher scores) compared to LMQL and DSPy, indicating a generally more favorable rating. Notably, for categories such as Ease of use, efficiency of problem solving, MTP achieved higher density peaks near the higher scores, reflecting higher perceived developer productivity and lower complexity of MTP compared to the other frameworks.

Open-ended participant feedback further highlighted MTP's strengths, with comments such as *"Learning LMQL was challenging, and it ultimately proved inadequate for accomplishing the required tasks."*, *"MTP provided a much simpler and more precise solution, making it the most efficient tool for the tasks at hand."* and *"MTP code is shorter than the other two frameworks... My favorite feature is the automatic filling of object attributes using by llm.' conversions between data types can be done easily."*.

8 Case Studies

In this section, we present two case studies, where we leverage MTP to develop real-world use cases.

8.1 CASE 1: Dynamic Level Generation in Video Game Development

With this case study, we demonstrate the efficacy of MTP in three key aspects:

1. MTP enables the LLM integration into a real-world complex application to create interesting functionalities;
2. MTP reduces development complexity for the LLM integration in this non-straightforward use case.
3. MTP supports automatic conversion of LLM outputs to variables of nested and complex custom types

The application in study is a fully-playable video game developed as a pygame tutorial[37]. In this game, the player controls the main character in a 2D map containing enemies and obstacles. The original game implementation has a set of maps at various difficulty levels generated by designers manually. Figure 18a shows the baseline implementation of the application where *random_map_generator*() selects a new map from the set of pre-designed maps.

This study aims to integrate LLMs to automatically generate new game levels. In fact, using AI for game level generation is an emerging area in game development [18].

MTP enables LLM integration into conventional programs - With MTP, we are able to use LLM to dynamically and automatically generate a new game level and the corresponding map. Our implementation using MTP (Figure 18b), takes previous levels and desired difficulty as input, is able to consistently generate playable levels and new maps that are increasingly more challenging during our testing. Figure 17 shows one real sequence of three game maps with increasing difficulty levels, generated by our implementation using MTP.

MTP reduces development complexity - With MTP, we are able to integrate LLM into the application with zero changes on the existing conventional code in Figure 18a and only minimum additional code (~ 12 lines), as shown in Figure 18b.

For comparison, we implemented the LLM integration using DSPy shown in Figure 18c. We found that DSPy requires significant modification of the existing conventional code. First, DSPy requires *Pydantic* [34] for type conversion for LLM inputs and outputs. Any classes that are to be used in the LLM have to inherit from the *BaseModel* class from *Pydantic* (line 7 - 30). Second, we are required to add natural language descriptions to each attribute of the building block classes (line 7 - 30). In addition, we need to define two new

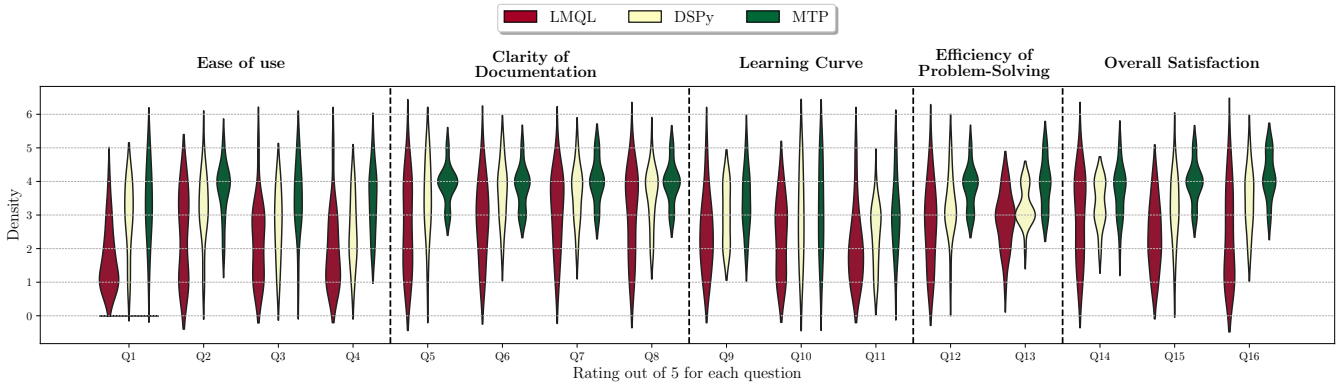

Figure 16. Violin plot showing the distribution of scores given by participants for each framework across the five criteria: Ease of Use, Clarity of Documentation, Learning Curve, Efficiency of Problem-Solving, and Overall Satisfaction.

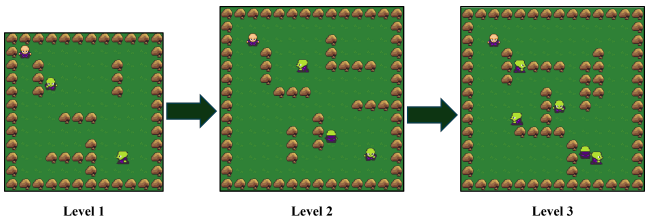

Figure 17. A real sequence of three game levels dynamically generated by the MTP implementation at runtime. The LLM uses details of the previous level as input and output the configuration for a more challenging new level.

classes to represent the LLM operations (*CreateNewLevel* and *CreateMap* at line 32 - 46). Lastly, we need to invoke the *TypedPredictor* module of DSPy to perform the LLM inference (line 49 - 56). In contrast, MTP automatically integrates LLMs and does not require modification to existing conventional code beyond using the **by** operator.

MTP supports generation of variables of complex custom types - The game map is represented in the code as a complex custom class *Maps*, which then includes a nested custom class *Level*, and other attributes *Wall* and *Position* that are of custom non-primitive types (Fig. 18a). With MTP, we are able to use a LLM to automatically generate a new *Level*, and then use a LLM to instantiate a *Map* variable, generating the map configurations and populate these custom-typed attributes. The MT-runtime engine in MTP automates and removes the need to parse the LLM output and convert it to a custom-typed variable, saving developer effort and reducing potential coding errors.

8.2 CASE 2: Using Multi-Modal Models to solve vision problems

There has been significant advancement recently in Multi-modal Generative AI models [38]. These models, such as LLaVa [24], QWEN [42], OpenAI's GPT-4o, Anthropic's Claude 3 and Google's Gemini can process both text and images and perform visual tasks such as object detection and visual reasoning. In this case study, we demonstrate how to use MTP to leverage multi-modal generative models to build neuro-symobilc programs to solve visual problems.

We select 5 vision benchmarks from prior work [23]. This set of problems represents a wide range of multi-modal tasks such as visual reasoning and visual question-answering, as shown in Figure 19. Using MTP, we are able to implement fully working solutions for the 5 vision benchmarks. Figure 20 shows our implementation of one of the problems, which involves reasoning about an input image and providing an answer to a user's question.

9 Related Work

There has been several recent efforts to help developers programming with LLMs. DSPy [16] and LMQL [4] and SGLang [46] introduce new libraries to help facilitating the integration of LLM into applications. While these frameworks provide additional tooling for LLM integration, they also introduce additional development complexity and new learning curves. LangChain [21] and LlamaIndex [25] focus on simplifying building LLM application while integrating with external data sources and can be leveraged together with our work. With this approach, the developer can retain the novel language-level abstraction that automates away much of complexity while benefit from the additional functionality introduced in these libraries.

There has been extensive studies in specialized compilation approach for AI and ML models [3, 6, 9, 13, 22, 33, 36, 40, 45]. In addition, recent works have also proposed novel runtime techniques and software system to accelerate training and inference of large scale models including large language models [1, 2, 5, 7, 12, 14, 15, 17, 27, 28, 30, 31, 39, 41, 43, 44]. Furthermore, researchers have explored architectural support for ML/AI models [11, 47]. These work focus on accelerate training and inference of machine learning models. Our work focus on simplifying the development required to integrate LLM models in applications and can benefit from improved model performance.

10 Conclusion

Software is rapidly evolving from traditional logical code to neuro-integrated applications that leverage generative AI and large language models (LLMs) for application functionality. However, leveraging GenAI models in those applications is complicated and requires significant expertise and efforts. This paper presents meaning-typed programming (MTP), a

```python
class Position:
    x: int
    y: int

class Wall:
    start_pos: Position
    end_pos: Position

class Level:
    name: str
    difficulty: int
    width: int
    height: int
    num_wall: int
    num_enemies: int

class Map:
    level: Level
    walls: list[Wall]
    small_obstacles: List[Position]
    enemies: list[Position]
    player_pos: Position

# Map generation function call
new_level_map = random_map_generator()
```

(a) Conventional Implementation

```
import:py from jaclang.core.llms, OpenAI;

glob llm = OpenAI(model_name="gpt-4o");

...
can create_next_level(last_levels: list[Level], difficulty: int,
      level_width: int, level_height: int)
    -> Level by llm();

...
new_level = create_next_level(prev_levels, current_difficulty, 20, 20);

new_level_map = Map(level=new_level by llm());
```

(b) MTP Implementation

```python
import dspy
from pydantic import BaseModel, Field

llm = dspy.OpenAI(model="gpt-4o", max_tokens=1024)
dspy.settings.configure(lm=llm)

class Position(BaseModel):
    x: int = Field(description="X Coordinate")
    y: int = Field(description="Y Coordinate")

class Wall(BaseModel):
    start_pos: Position = Field(description="Start Position of the Wall")
    end_pos: Position = Field(description="End Position of the Wall")

class Level(BaseModel):
    name: str = Field(description="Name of the Level")
    difficulty: int = Field(description="Difficulty of the Level")
    width: int = Field(description="Width of the Map")
    height: int = Field(description="Height of the Map")
    num_wall: int = Field(description="Number of Walls in the Map")
    num_enemies: int = Field(description="Number of Enemies in the Map")
    time_countdown: int = Field(description="Time Countdown of the Level")
    n_retries_allowed: int = Field(description="Number of Retries Allowed")

class Map(BaseModel):
    level: Level = Field(description="Level of the Map")
    walls: list[Wall] = Field(description="Walls in the Map")
    small_obstacles: List[Position] = Field(description="Obstacles")
    enemies: list[Position] = Field(description="Enemies in the Map")
    player_pos: Position = Field(description="Player Position in the Map")

class CreateNextLevel(dspy.Signature):
    """Create Next Level"""

    last_levels: list[Level] = dspy.InputField(desc="Last Played Levels")
    difficulty: int = dspy.InputField(desc="Difficulty of the New Level")
    level_width: int = dspy.InputField(desc="Width of the Level")
    level_height: int = dspy.InputField(desc="Height of the Level")
    next_level: Level = dspy.OutputField(desc="Next Level")


class CreateMap(dspy.Signature):
    """Create Map for the Level"""

    level: Level = dspy.InputField(desc="Level")
    map: Map = dspy.OutputField(desc="Map")

...
new_level = dspy.TypedPredictor(CreateNextLevel)(
    last_levels=prev_levels,
    difficulty=current_difficulty,
    level_width=20,
    level_height=20,).next_level

...
new_level_map = dspy.TypedPredictor(CreateMap)(level=new_level).map
```

(c) DSPy Implementation

Figure 18. Three implementations of a video game where the game level is generated dynamically at runtime. (a) shows the conventional code implementation. (b) and (c) shows implementations that use LLMs to generate the game level via MTP and DSPy, respectively.

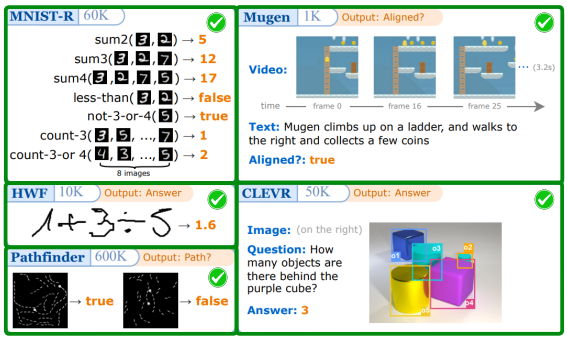

Figure 19. We successfully use MTP to implement solutions that leverage multi-modal models to achieve this suite of visual tasks.

```
import:py from mtllm.llms, OpenAI;
import:py from mtllm, Image;

glob llm = OpenAI(model_name="gpt-4o");

can get_answer(img: Image, question: str) -> str by llm();

with entry {
    question:str = 'How many objects behind the purple cube?';
    print(get_answer(Image('image.png'), question));
}
```

Figure 20. Using MTP and multi-modal model for vision.

novel approach to simplify the creation of neuro-integrated applications by introducing new language-level abstractions that hide the complexities of LLM integration. MTP automatically extracts the semantic meaning embedded in the traditional code, combined with dynamic information to dynamically synthesize prompts for LLMs and then convert the LLM output to the types that traditional code expects. We implemented MTP in a super-setted python language and demonstrate its effectiveness in reducing development complexity, improving runtime performance while maintaining high accuracy.

References

[1] Sohaib Ahmad, Hui Guan, Brian D Friedman, Thomas Williams, Ramesh K Sitaraman, and Thomas Woo. Proteus: A high-throughput inference-serving system with accuracy scaling. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 318–334, 2024.

[2] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, et al. Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2022.

[3] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, et al. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings*

of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, pages 929–947, 2024.

[4] Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. Prompting is programming: A query language for large language models. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1946–1969, 2023.

[5] Renze Chen, Zijian Ding, Size Zheng, Chengrui Zhang, Jingwen Leng, Xuanzhe Liu, and Yun Liang. Magis: Memory optimization via coordinated graph transformation and scheduling for dnn. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 607–621, 2024.

[6] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: An automated end-to-end optimizing compiler for deep learning, 2018.

[7] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. Serving heterogeneous machine learning models on Multi-GPU servers with Spatio-Temporal sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 199–216, Carlsbad, CA, July 2022. USENIX Association.

[8] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

[9] Michael Davies, Ian McDougall, Selvaraj Anandaraj, Deep Machchhar, Rithik Jain, and Karthikeyan Sankaralingam. A journey of a 1,000 kernels begins with a single step: A retrospective of deep learning on gpus. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 20–36, 2024.

[10] Asbjørn Følstad and Marita Skjuve. Chatbots for customer service: user experience and motivation. In *Proceedings of the 1st international conference on conversational user interfaces*, pages 1–9, 2019.

[11] Soroush Ghodrati, Sean Kinzer, Hanyang Xu, Rohan Mahapatra, Yoonsung Kim, Byung Hoon Ahn, Dong Kai Wang, Lavanya Karthikeyan, Amir Yazdanbakhsh, Jongse Park, et al. Tandem processor: Grappling with emerging operators in neural networks. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 1165–1182, 2024.

[12] Cong Guo, Jiaming Tang, Weiming Hu, Jingwen Leng, Chen Zhang, Fan Yang, Yunxin Liu, Minyi Guo, and Yuhao Zhu. Olive: Accelerating large language models via hardware-friendly outlier-victim pair quantization. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–15, 2023.

[13] Muyan Hu, Ashwin Venkatram, Shreyashri Biswas, Balamurugan Marimuthu, Bohan Hou, Gabriele Oliaro, Haojie Wang, Liyan Zheng, Xupeng Miao, Jidong Zhai, et al. Optimal kernel orchestration for tensor programs with korch. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 755–769, 2024.

[14] Qijing Huang, Minwoo Kang, Grace Dinh, Thomas Norell, Aravind Kalaiah, James Demmel, John Wawrzynek, and Yakun Sophia Shao. Cosa: Scheduling by constrained optimization for spatial accelerators. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 554–566. IEEE, 2021.

[15] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 463–479. USENIX Association, November 2020.

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

[17] Seah Kim, Hyoukjun Kwon, Jinook Song, Jihyuck Jo, Yu-Hsin Chen, Liangzhen Lai, and Vikas Chandra. Dream: A dynamic scheduler for dynamic real-time multi-model ml workloads. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, pages 73–86, 2023.

[18] Vikram Kumaran, Bradford Mott, and James Lester. Generating game levels for multiple distinct games with a common latent space. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 15, pages 102–108, 2019.

[19] Jaseci Labs. Jaclang pypi package, 2024. Accessed: 2024-10-18.

[20] Jaseci Labs. Jaseci github repo, 2024. Accessed: 2024-10-18.

[21] Langchain. Langchain, 2024. Accessed: 2024-10-18.

[22] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14. IEEE, 2021.

[23] Ziyang Li, Jiani Huang, and Mayur Naik. Scallop: A language for neurosymbolic programming. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1463–1487, 2023.

[24] Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. Visual instruction tuning. In *NeurIPS*, 2023.

[25] llama$_i ndex. llama_i ndex$, 2024. $Accessed$: 2024 − 10 − 18.

[26] Jason Mars, Yiping Kang, Roland Daynauth, Baichuan Li, Ashish Mahendra, Krisztian Flautner, and Lingjia Tang. The jaseci programming paradigm and runtime stack: Building scale-out production applications easy and fast. *IEEE Computer Architecture Letters*, 22(2):101–104, 2023.

[27] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, et al. Specinfer: Accelerating large language model serving with tree-based speculative inference and verification. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 932–949, 2024.

[28] Xupeng Miao, Chunan Shi, Jiangfei Duan, Xiaoli Xi, Dahua Lin, Bin Cui, and Zhihao Jia. Spotserve: Serving generative large language models on preemptible instances. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 1112–1127, 2024.

[29] mypy. mypy - optional static typing for python, 2024. Accessed: 2024-10-18.

[30] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. Dnnfusion: accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 883–898, 2021.

[31] Hyungjun Oh, Kihong Kim, Jaemin Kim, Sungkyun Kim, Junyeol Lee, Duseong Chang, and Jiwon Seo. Exegpt: Constraint-aware resource scheduling for llm inference. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 369–384, 2024.

[32] OpenAI. Openai api pricing, 2024. Accessed: 2024-10-18.

[33] Zaifeng Pan, Zhen Zheng, Feng Zhang, Ruofan Wu, Hao Liang, Dalin Wang, Xiafei Qiu, Junjie Bai, Wei Lin, and Xiaoyong Du. Recom: A compiler approach to accelerating recommendation model inference with massive embedding columns. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, pages 268–286, 2023.

[34] Pydantic Team. *Pydantic Documentation*, 2024. Accessed: 2024-10-19.

[35] Python Software Foundation. *The Python Standard Library: cProfile âĂŤ Profile Module*, 2024. Accessed: 2024-10-19.

[36] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, et al. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907*, 2018.

[37] ShawCode. Pygame rpg tutorial. YouTube, 2021. Accessed: 2024-09-24.

[38] Shakti N Wadekar, Abhishek Chaurasia, Aman Chadha, and Eugenio Culurciello. The evolution of multimodal model architectures. *arXiv preprint arXiv:2405.17927*, 2024.

[39] Zheng Wang, Yuke Wang, Jiaqi Deng, Da Zheng, Ang Li, and Yufei Ding. Rap: Resource-aware automated gpu sharing for multi-gpu recommendation model training and input preprocessing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 964–979, 2024.

[40] Chunwei Xia, Jiacheng Zhao, Qianqi Sun, Zheng Wang, Yuan Wen, Teng Yu, Xiaobing Feng, and Huimin Cui. Optimizing deep learning inference via global analysis and tensor expressions. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 286–301, 2024.

[41] Daliang Xu, Mengwei Xu, Chiheng Lou, Li Zhang, Gang Huang, Xin Jin, and Xuanzhe Liu. Socflow: Efficient and scalable dnn training on soc-clustered edge servers. In *Proceedings of the 29th ACM International Conference on*

Architectural Support for Programming Languages and Operating Systems, Volume 1, pages 368–385, 2024.

[42] An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Haoran Wei, Huan Lin, Jialong Tang, Jialin Wang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Ma, Jianxin Yang, Jin Xu, Jingren Zhou, Jinze Bai, Jinzheng He, Junyang Lin, Kai Dang, Keming Lu, Keqin Chen, Kexin Yang, Mei Li, Mingfeng Xue, Na Ni, Pei Zhang, Peng Wang, Ru Peng, Rui Men, Ruize Gao, Runji Lin, Shijie Wang, Shuai Bai, Sinan Tan, Tianhang Zhu, Tianhao Li, Tianyu Liu, Wenbin Ge, Xiaodong Deng, Xiaohuan Zhou, Xingzhang Ren, Xinyu Zhang, Xipin Wei, Xuancheng Ren, Xuejing Liu, Yang Fan, Yang Yao, Yichang Zhang, Yu Wan, Yunfei Chu, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, Zhifang Guo, and Zhihao Fan. Qwen2 technical report, 2024.

[43] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, Carlsbad, CA, July 2022. USENIX Association.

[44] Chenguang Zheng, Hongzhi Chen, Yuxuan Cheng, Zhezheng Song, Yifan Wu, Changji Li, James Cheng, Hao Yang, and Shuai Zhang. Bytegnn: efficient graph neural network training at large scale. *Proceedings of the VLDB Endowment*, 15(6):1228–1242, 2022.

[45] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: Generating High-Performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 863–879. USENIX Association, November 2020.

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

[47] Kai Zhong, Zhenhua Zhu, Guohao Dai, Hongyi Wang, Xinhao Yang, Haoyu Zhang, Jin Si, Qiuli Mao, Shulin Zeng, Ke Hong, et al. Feasta: A flexible and efficient accelerator for sparse tensor algebra in machine learning. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 349–366, 2024.