MTP: A Meaning-Typed Language Abstraction for AI-Integrated Programming

JAYANAKA L. DANTANARAYANA, University of Michigan, USA
YIPING KANG, University of Michigan, USA
KUGESAN SIVASOTHYNATHAN, Jaseci Labs, USA
CHRISTOPHER CLARKE, University of Michigan, USA
BAICHUAN LI, University of Michigan, USA
SAVINI KASHMIRA, University of Michigan, USA
KRISZTIAN FLAUTNER, University of Michigan, USA
LINGJIA TANG, University of Michigan, USA
JASON MARS, University of Michigan, USA

Software development is shifting from traditional programming to *AI-integrated* applications that leverage generative AI and large language models (LLMs) during runtime. However, integrating LLMs remains complex, requiring developers to manually craft prompts and process outputs. Existing tools attempt to assist with prompt engineering, but often introduce additional complexity.

This paper presents **Meaning-Typed Programming (MTP)**, a novel paradigm that abstracts LLM integration through intuitive language-level constructs. By leveraging the inherent semantic richness of code, MTP automates prompt generation and response handling without additional developer effort. We introduce the **(1) by** operator for seamless LLM invocation, **(2) MT-IR**, a meaning-based intermediate representation for semantic extraction, and **(3) MT-Runtime**, an automated system for managing LLM interactions. We implement MTP in **Jac**, a programming language that supersets Python, and find that MTP significantly reduces coding complexity while maintaining accuracy and efficiency. MTP significantly reduces development complexity, lines of code modifications needed, and costs while improving run-time performance and maintaining or exceeding the accuracy of existing approaches. Our user study shows that developers using MTP completed tasks 3.2× faster with 45% fewer lines of code compared to existing frameworks. Moreover, MTP demonstrates resilience even when up to 50% of naming conventions are degraded, demonstrating robustness to suboptimal code. MTP is developed as part of the Jaseci open-source project, and is available under the module **byLLM**.

CCS Concepts: • **Computing methodologies → Artificial intelligence**.

Additional Key Words and Phrases: programming languages for AI, prompt engineering, generative AI

Authors' Contact Information: Jayanaka L. Dantanarayana, University of Michigan, Ann Arbor, USA, jayanaka@umich.edu; Yiping Kang, University of Michigan, Ann Arbor, USA, ypkang@umich.edu; Kugesan Sivasothynathan, Jaseci Labs, Ann Arbor, USA, kugesan.sivasothynathan@jaseci.org; Christopher Clarke, University of Michigan, Ann Arbor, USA, csclarke@umich.edu; Baichuan Li, University of Michigan, Ann Arbor, USA, patrli@umich.edu; Savini Kashmira, University of Michigan, Ann Arbor, USA, savinik@umich.edu; Krisztian Flautner, University of Michigan, Ann Arbor, USA, manowar@umich.edu; Lingjia Tang, University of Michigan, Ann Arbor, USA, lingjia@umich.edu; Jason Mars, University of Michigan, Ann Arbor, USA, profmars@umich.edu.
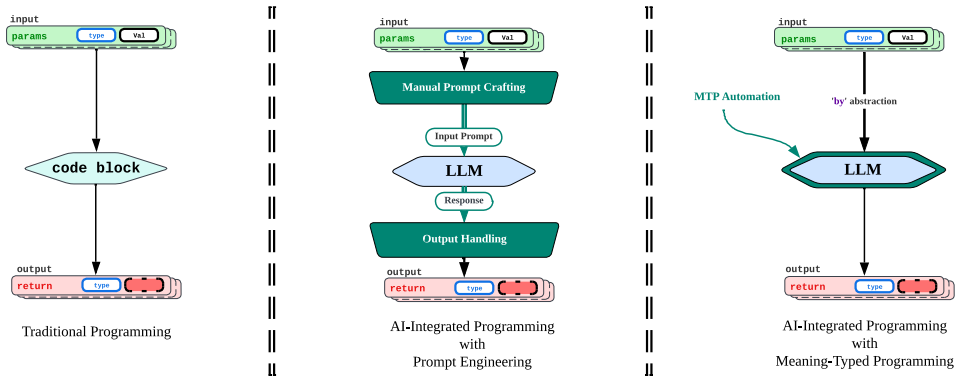
Fig. 1. High level view of what traditional program functionality consists of (Left) vs. AI-integrated programming using prompt engineering (Middle) and Meaning-Typed Programming(MTP) paradigm (Right).

1 INTRODUCTION

With the advent of generative AI and Large Language Models (LLMs), the way we develop software is rapidly evolving. While much work has focused on using LLMs in *code generation* [23, 43], an emerging trend is the integration of LLMs as components that provide essential functionality [4, 24, 40, 45]. These *AI-integrated applications* leverage generative AI at runtime to perform critical program functionality, merging conventional programming with AI-driven capabilities. This shift fundamentally transforms software development, redefining how applications are designed and built.

Building AI-integrated applications remains a complex challenge due to the fundamental differences between operating assumptions. In conventional programming, code describes precise operations performed on explicitly defined variables such as 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 is the primary method of realizing AI-integrated programming and can be tedious, complex, and imprecise. Developers must craft descriptive instructions, manage textual context, and handle the inherent variability in LLM output. This process includes a significant amount of trial-and-error, making it time-consuming and difficult to maintain.

Several efforts have been made to address these challenges by developing frameworks and tools to facilitate prompt engineering. Frameworks such as LangChain [21], Language Model Query Language (LMQL) [4], DSPy [17], and SGLang [54] 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:

(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.

In this paper, we introduce *meaning-typed programming* (MTP), a novel approach to simplify the creation of AI-integrated applications by (i) introducing new high level language abstractions that

hide the complexities of *AI-integration*, (ii) a novel technique for automating prompt generation using inherent code semantics, and (iii) a runtime system that automates LLM inference and output handling.

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 becomes 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 perhaps be more in prompt engineering (see § 5).*

There are four primary **challenges** that must be overcome in the design and implementation of MTP. Firstly, the introduced language abstraction should be simple, intuitive, and flexible. Secondly, developing an automated approach that provides LLMs with sufficient semantic information for accurate task performance is challenging. Thirdly, since AI-integrated applications rely on dynamic values, prompts must be generated at runtime, requiring a lightweight system to handle prompt generation and LLM inference efficiently. Lastly, because LLMs generate non-deterministic outputs, the outputs must be carefully processed to ensure correct type conversion. To address these challenges, we introduce three key components to enable the seamless creation of AI-integrated 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.

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

As shown later in our evaluation, meticulously extracted relevant semantic information is sufficient for the LLM to produce the accurate result, significantly simplifying LLM integration into 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 types, during runtime.

(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 dynamically, manages LLM interactions, generates context-aware prompts, and handles responses and errors. Through these three components, MTP 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 [20, 28, 29]. The **by** operator was introduced as a language primitive in Jac, and the MT-IR construction is implemented as a compilation pass in Jac's ahead-of-time compiler. The MT-Runtime system is implemented as a language plugin which can be separately installed as a PyPi package. This end-to-end framework has been used to develop real-world applications and benchmarks utilized in our extensive evaluation.

Specifically, we make the following contributions:

- We introduce **meaning-typed programming**, a novel paradigm that leverages the inherent semantic richness of code to automate LLM integration into applications without requiring explicit prompt engineering.
- We introduce the **by operator**, a simple yet powerful language abstraction that provides both intuitive and flexible integration of LLM functionality into code. It enables developers

to seamlessly incorporate language models for multiple code constructs including functions, methods, and object initialization while maintaining familiar programming patterns.

- We develop supporting compiler and runtime infrastructure: **MT-IR**, a semantically rich intermediate representation that extracts and organizes code meaning, and **MT-Runtime**, an automated execution engine that dynamically converts semantic information into prompts, manages LLM interactions, and handles type-safe response processing.
- 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 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.

We perform extensive evaluation to investigate the implications of MTP on code complexity, AI-integration accuracy, inference token usage, runtime performance, and cost compared to prior work. We also assess how sensitive MTP is to poor coding practices, as our system is built on leveraging semantically rich code. Our evaluations show that MTP reduces lines of code by factors of 2.3-7.5× compared to LMQL and 1.3-10.7× compared to DSPy, demonstrating significant code complexity reduction. In terms of accuracy, MTP performs on par with or better than DSPy, while surpassing LMQL on average by 8% across all benchmarks when evaluated with gpt-4o model. For accuracy on math problems from the GSM8k [8] dataset, MTP outperformed other frameworks with newer OpenAI and Llama models, achieving accuracies approaching 90%. Regarding inference costs, MTP demonstrated savings over DSPy on most benchmarks, with reductions up to 4.5×. Moreover, MTP enables program speedups of up to 4.75× over DSPy, indicating minimal runtime overhead. Our sensitivity studies confirm that MTP maintains effectiveness even when 50% of naming conventions are suboptimal. These results establish MTP as a practical, efficient solution for streamlining AI-integrated application development.

```
1   class Position:
2       x: int
3       y: int
4
5   class Wall:
6       start_pos: Position
7       end_pos: Position
8
9   class Map:
10      walls: list[Wall]
11      enemies: list[Position]
12      player_pos: Position
13
14  class Level:
15      level_id: int
16      difficulty: int
17      width: int
18      height: int
19      num_wall: int
20      num_enemies: int
21      map: Map
```

Fig. 2. A typical implementation for a video game level development usecase. Code shows how the data-structure definitions for the video game application.

```
1   def get_next_level(prev_level: Level) -> Level
        :
2       ...
3       ...
4       ## code for generating/fetching the next
            level
5       ...
6       ...
7       return next_level
8
9   new_level = get_next_level(curr_level)
```

Fig. 3. Traditional implementation of game levels for the non AI-integrated usecase.

2 MOTIVATING EXAMPLE

This section presents a motivating example to illustrate the concept of *AI-integrated* programming and to highlight the complexities involved in the current implementation technique, i.e., prompt engineering. The example application used here is a video game implementation, where dynamically generating game levels is achieved through AI-integration with an LLM.

The section is organized as follows. We first introduce the example application and the code sketch of traditional implementation without an LLM (Figure 3). We then demonstrate AI-integrated implementation using the industry-standard approach of prompt engineering (Figure 4), highlighting the significant overhead it imposes on developers. Lastly, we present implementation using our

proposed MTP approach (Figure 5), demonstrating MTP's capabilities in removing complexities associated with prompt engineering.

2.1 Traditional Programming

```
1  import openai
2  import json
3
4  def get_next_level(prev_level: Level) -> Level:
5      prev_level_json = json.dumps(prev_level.__dict__,
                default=lambda o: o.__dict__)
6
7      prompt = f"""
8      You are an expert game level generator. Given the
                current level details, generate the next level
                with increased difficulty.
9
10     ### Current Level:
11     {prev_level_json}
12
13     ### Rules for Next Level:
14     - Increase difficulty
15     - Increase width and height slightly (max 10% each)
16     - Increase the number of walls and enemies based on
                difficulty progression
17     - Generate a valid map with walls and enemy positions
18     - Ensure `player_pos` is within bounds and not
                colliding with walls
19
20     ### Output Format:
21     Provide the next level in JSON format:
22     {{
23         'level_id': <int>,
24         'difficulty': <int>,
25         'width': <int>,
26         'height': <int>,
27         'num_wall': <int>,
28         'num_enemies': <int>,
29         'map': {{
30             'walls': [{{'start_pos': {{'x': <int>, 'y': <int>}},
                    'end_pos': {{'x': <int>, 'y': <int>}}}}],
31             'enemies': [{{'x': <int>, 'y': <int>}}],
32             'player_pos': {{'x': <int>, 'y': <int>}}
33         }}
34     }}
35     """
36
37     response = openai.ChatCompletion.create(
38         model='gpt-4',
39         messages=[{"role": "system", "content": "You are an
                expert in procedural game level generation."}
                ,
40             {"role": "user", "content": prompt}],
41         temperature=0.7,
42         max_tokens=500
43     )
44
45     level_dict = json.loads(response['choices'][0]['message
                ']['content'])
46
47     def parse_position(data: dict) -> Position:
48         return Position(x=data["x"], y=data["y"])
49
50     def parse_wall(data: dict) -> Wall:
51         return Wall(
52             start_pos=parse_position(data["start_pos"]),
53             end_pos=parse_position(data["end_pos"])
54         )
55
56     def parse_map(data: dict) -> Map:
57         walls = [parse_wall(w) for w in data.get("walls",
                    [])]
58         enemies = [parse_position(e) for e in data.get("
                    enemies", [])]
59         player_pos = parse_position(data["player_pos"])
60         return Map(walls=walls, enemies=enemies, player_pos=
                    player_pos)
61
62     next_level = Level(
63         level_id=level_dict["level_id"],
64         difficulty=level_dict["difficulty"],
65         width=level_dict["width"],
66         height=level_dict["height"],
67         num_wall=level_dict["num_wall"],
68         num_enemies=level_dict["num_enemies"],
69         map=parse_map(level_dict["map"])
70     )
71
72     return next_level
```

Fig. 4. Implementation of the video game level generation using prompt-engineering

The video game implementation used here is a real program developed based on the pygame [30] tutorials. In this video game, the player must complete the current game level to proceed to the next. An example of the video game level progression is shown in § 5 (Figure 12), where each level includes a set of configurations (maps, enemies, difficulties, etc). Figure 2 presents the basic data structures used in the video game implementation, including the Level object (lines 14–21), which encapsulates the configurations and a nested Map object (lines 9–12).

get_next_level The level progression logic is achieved through get_next_level function, which uses the previous level as input and returns the next level. Figure 3 shows the code sketch, including a placeholder (lines 2–6) for the function body. In traditional programming, developers must implement complex algorithms and logic to generate the next level based on the current one, often leveraging procedural generation techniques, which can be both challenging and time-consuming. However, by integrating an LLM, a level can be generated automatically without developers' explicit coding.

2.2 State-of-the-Art AI-integration Approach

Code synthesis vs. AI-integration. Before presenting the current approach to AI-integration, it is important to highlight the difference between AI-integration and the use of LLMs for code synthesis [43]. LLM-enabled code synthesis uses LLMs to generate code that is executed at runtime. In contrast, *AI-integrated* code [45] leverages an LLM to dynamically perform tasks within a program. During runtime, the LLM inference generates output values based on given inputs to the LLM. In this example, *AI-integrated* approach uses the input (previous level object) to query an LLM at runtime. The output of the LLM inference will be used as the next level configuration.

Current approaches to AI-integration primarily rely on prompt engineering as the industry standard, which involves crafting input prompts to guide LLMs in generating desired responses. It involves structuring inputs, optimizing the wording and context of the prompt to maximize the accuracy and relevance of the model's output. Figure 4 presents the prompt engineering-based implementation of get_next_level functionality, where the input of the previous level object is described in a manually-craft prompt to an LLM, and the output of the LLM (in a text format) is used to construct the next level. The LLM innovation is in lines 37-42, which eliminates the need to manually implement the level generation algorithms. However, this example illustrates the significant complexity surrounding the LLM invocation/integration.

Manual Prompt Crafting. The example includes a manually crafted prompt template (lines 7-35). As shown here, to effectively improve LLM accuracy, the prompt consists of several key components: a problem description(line 8), an input object (including a description of the input and the prev_level object in JSON format), and a set of rules (lines 14-18) to guide the LLM's generation. In addition, developers need to explicitly specify the output format(lines 20-34), which in this case follows a JSON schema for the Level object. Since the Level object contains deeply nested objects, structuring the output format is tedious and labor-intensive. This prompt engineering process is time-consuming and often requires an iterative trial-and-error process.

Output Type Conversion. In addition to prompt crafting, developers must also convert the LLM's output into a return-typed object compatible with traditional programming. In this example, LLM's output follows the JSON object format, as specified in the prompt, which then must be converted into a Level object that is understandable by the rest of the application(lines 47-70). This includes mapping the JSON output to the corresponding attributes of the Level class (lines 62-70). To handle deeply nested objects, three additional helper functions are defined on lines 47, 50, and 56. This added complexity makes AI-integration with prompt engineering more cumbersome and less intuitive.

Error Handling and Retries. Due to the non-deterministic nature of LLMs, the generated output may not always be consistent with the expected format, leading to program runtime errors that developers must anticipate and handle. In addition to error-handling, a retry mechanism may be needed to reattempt the LLM invocation process until a correctly formatted output is produced. Although the example code snippet does not include error handling and retries due to space constraints, incorporating such mechanisms adds additional complexity and requires development effort to ensure robustness and reliability.

In conclusion, this example demonstrates that while AI-integration can leverage LLMs to improve traditional programming, the complexities associated with prompt engineering may introduce significant cognitive and implementation burden. Several frameworks and tools have been developed to simplify the interaction with LLMs. While these systems offer valuable capabilities for prompt engineering and LLM integration, they usually incur additional complexity. LMQL [4] requires developers to learn a new query language, while DSPy [17] requires the developers to annotate all essential information in code, replacing prompt-engineering complexities with annotation overhead. The comparison to prior work is presented later in § 5.

2.3 MTP Vision for AI-integrated Programming

To address the challenges in the current AI-integration approach, our research aims to *design a language abstraction and system to streamline the process of integrating LLMs into traditional programming by abstracting and automating prompt generation, output type conversions, and error handling.*

Figure 5 illustrates our envisioned approach, where adding only a few lines of code enables LLM integration, eliminating the need for prompt engineering and its associated complexities. Moreover, this approach is significantly more intuitive than prior works. The code snippet provides a preview of what we introduce in this paper as *Meaning-Typed Programming*. Line 5 of the code snippet shows how AI-integration is achieved. MTP hides prompt-engineering using a language abstraction, automating the prompt generation, LLM inference and output handling. We will describe details on our language abstraction in § 4.2.

When hiding prompt engineering, the information required for accurate LLM inference should be available to generate prompts. As shown in Figure 5, the intent of the `get_next_level()` function on line 5 is evident from the code's semantics. This observation leads to the key insight of our paper: *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.*

```
1  from mtp.core.llms import OpenAI
2
3  llm = OpenAI(model_name="gpt-4o")
4
5  def get_next_level(prev_levels: list[Level]) -> Level by
       llm
6
7  new_level = get_next_level(prev_levels)
```

Fig. 5. Implementation of AI-integration as envisioned in this paper, with Meaning-Typed programming.
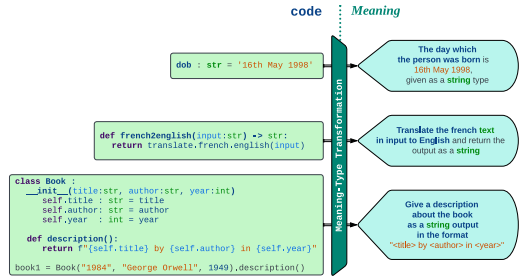

Fig. 6. Examples of *meaning* embedded in code semantics.

Figure 6 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. Here dob is a commonly used abbreviation for date of birth which still retains adequate semantic meaning.

(2) The meaning of the `french2english()` function can be inferred from its name, input parameter, and return type. These semantics indicate the function takes French text as input and returns an English translation of that text.

(3) The semantics of the `description()` method in the book class can be understood by analyzing the class, variable names, and an example function call.

With the help of a compiler and run-time techniques, LLMs can infer the intended meaning from a program to perform AI-integration tasks as shown in our evaluations, later in the §5.

3 PROBLEM, OBJECTIVES AND CHALLENGES

Our goal is to leverage semantically rich, human-readable code expressions to automate prompt-engineering. Instead of requiring developers to manually craft detailed prompts or annotations, we harness the inherent structure and semantics of the code itself to generate meaningful LLM inputs.

To achieve this goal, we define ***three key objectives***:

O$_1$ - Design simple, intuitive and flexible language abstractions for developers while hiding prompt engineering and other complexities.

O$_2$ - Design techniques to extract relevant and sufficient semantic information from code for automatic prompt synthesis.

O$_3$ - Design a system that combines semantic information and dynamic information to manage prompt synthesis and output interpretation.
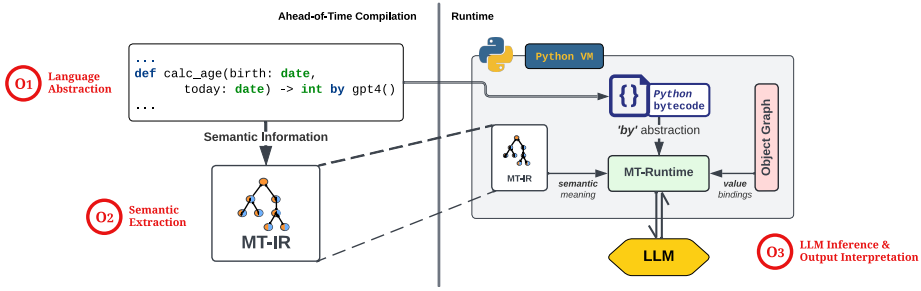
Fig. 7. Meaning-Typed Programming system overview.

When designing a solution that achieves these objectives, ***four unique challenges arise***:

C_1 - How do we design a language abstraction that supports multiple AI-integration methods while remaining easy to learn and use, minimizing the developer learning curve?

C_2 - What semantic information must be extracted to generate an LLM prompt that ensures program accuracy without incurring cost and runtime overhead?

C_3 - What type of runtime system design is appropriate to combine static semantic information with dynamic information to manage LLM input?

C_4 - How do we design a system that handle output type conversion and error handling without increasing token cost and runtime cost?

C_1 involves designing a simple and intuitive language abstraction, minimizing the learning curve and cognitive load. At the same time, it must be flexible enough to seamlessly integrate across different code locations. Designing a system that is both intuitive and flexible is challenging, as the abstraction must be simple enough to reduce the learning curve while also being usable orthogonally at multiple AI-integration points.

C_2 involves extracting only the relevant semantics for AI-integration points. A naive approach would be to include the entire source code in the prompt, but this would lead to high token costs, or even context window overflow. To optimize token usage, we must selectively extract only the relevant semantics. But this is particularly challenging because these semantics are often scattered across different code locations, requiring sophisticated codebase-wide analysis to identify and extract only what is necessary for accurate LLM prompting.

C_3 involves combining extracted code semantics with dynamic values to create effective prompts. This requires accessing variable data at runtime, which can vary in complexity depending on where the integration happens. For functions, it is straightforward since values are passed as parameters. However, for object methods, it becomes more complex because the system must also access relevant object attributes at runtime. To handle this, a sophisticated runtime system is needed to capture the execution context and combine dynamic values into the prompts.

C_4 involves correctly interpreting LLM-generated responses and converting them to appropriate types with robust error handling. This is particularly difficult because LLMs generate outputs non-deterministically and often deviate from expected formats. As demonstrated in the prompt-engineering example in §2.2, extracting correct results requires careful parsing, but inconsistencies in output structure make this process complex. Implementing a retry mechanism to correct such inconsistencies requires additional LLM inferences, increasing token usage and cost. Therefore, the system design must balance effective error handling with minimal impact on token consumption and runtime cost.

4 MEANING TYPED PROGRAMMING

To address the problem of automating and streamlining AI-integrated application development discussed in §3, we introduce *Meaning Typed Programming (MTP)*. The MTP paradigm (Figure 7) comprises three key components that achieves each objective defined in §3, enabling seamless AI-integration into programming languages:

(1) The **by** operator as the language level abstraction. (To achieve O_1)
(2) Meaning-Typed Intermediate Representation (MT-IR), generated during compilation. (To achieve O_2)
(3) MT-Runtime, an automated runtime engine that manages the LLM integration. (To achieve O_3)

This section presents an overview of MTP as well as an in depth view into each component introduced in this paper with respect to the challenges mentioned in §3.

4.1 MTP Overview

The by operator acts as a bridge between traditional code and LLM operations, allowing developers to seamlessly integrate LLM functionality into their code by simply adding the **by** keyword. As shown in Figure 7, developers can replace a function's implementation, such as calc_age(), with a straightforward **by** <model_name> clause. At runtime, the **by** operator invokes MT-Runtime to execute the desired functionality using the specified LLM, achieving O_1. In this example, the system leverages GPT-4 to calculate age based on a given birthday and the current date. § 4.2 provides further details on the **by** operator.

MT-IR is an intermediate representation designed to capture and preserve the meaning of the program to be used at runtime upon invocation of the **by** keyword. During compilation, MT-IR is generated by analyzing and organizing semantically rich information in the code (see Figure 7 (left)). It preserves key elements such as variable names, function signatures, and object schema relevant to a given **by** call-site to provide access to the code's original meaning by the LLM at run-time (see Figure 7 (right)) achieving O_2. (see § 4.3 for more details).

MT-Runtime manages the complexities of LLM interaction, through prompt synthesis using MT-IR. It enhances execution by intelligent response parsing, and dynamic error handling, achieving O_3. 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 § 4.4 for more details).

4.2 Meaning-Type Language Construct: 'by' operator

Hiding prompt engineering within *a language abstraction that is simple, intuitive and flexible is challenging* as discussed in §3 under C_1. To address this we introduce the **by** operator as the core language abstraction of MTP. The base syntax of the **by** operator is,

$$\textit{<code construct>} \textbf{ by } \textit{llm\_ref} (\textit{ model\_hyperparameters })$$

Here, the workload defined by the code construct on the left-hand side of the **by** operator is AI-integrated using the LLM specified on the right. This syntax is simple and intuitive, making it easy to learn. Additionally, this syntax provides flexibility by enabling developers to integrate LLMs at multiple points within standard code constructs. Furthermore, additional hyperparameters can be provided as arguments to the LLM object for further customization.

In this paper, we introduce three code construct locations to use the **by** operator, allowing developers to use MTP flexibly while maintaining simplicity and minimal overhead. These locations include *function definitions*, *object initialization* and *member method definitions*.

To formally express the semantic representation of **by** operator usage across various integration points and their respective call sites, we introduce a formal semantics that captures the operational behavior of **by** in a principled way. We define the following notation:

- $\mathcal{M}$: the set of available language models.
- $eval(\cdot)$: the semantic evaluation function.

These definitions serve as the foundation for describing how **by** transforms traditional code constructs into AI-integrated operations.

4.2.1 **by** *at Function Definitions*. Consider a standalone function f with parameters $p_1, p_2, \ldots, p_n$ of types $T_1, T_2, \ldots, T_n$ respectively, and having output type T_r. Using the **by** operator, this function can be integrated with an LLM $m \in \mathcal{M}$ with hyperparameters θ as follows:

$$\text{def } f(p_1 : T_1, \ldots, p_n : T_n) \rightarrow T_r \text{ \textbf{by} } m(\theta) \tag{1}$$

At runtime, invoking this function with actual arguments $v_1, \ldots, v_n$ to return v_r of type T_r, is semantically equivalent to:

$$v_r = eval(f(v_1, \ldots, v_n)) = \text{invoke-model}(m, \theta, f, [v_1, \ldots, v_n], [(T_1, \ldots, T_n), T_r]) \tag{2}$$

Here, the model (m) receives the function signature f, the actual arguments ($v_1, \ldots, v_n$) with their corresponding types ($T_1, \ldots, T_n$), and the expected output type T_r. By leveraging this rich semantic context, the model can reason about the intended behavior of the function and generate an appropriate output conforming to T_r. An illustrative example is provided in Figure 8a, where the function calculate_age is defined to return an integer output by computing a person's age given their date of birth (dob) and the current year (cur_year).

4.2.2 **by** *at Object Initialization*. The **by** operator can also be used during object initialization to let an LLM fill in missing attribute values. Consider a class C with attributes $a_1, \ldots, a_n$, each having types $T_1, \ldots, T_n$. If only the first k attributes are provided during initialization ($k < n$), the remaining attributes can be completed by $m \in \mathcal{M}$ with θ hyperparameters, as shown below:

$$C(a_1, \ldots, a_k) \text{ \textbf{by} } m(\theta) \tag{3}$$

At runtime, when such an object is created, the system calls the model with the class name C, the values of the attributes that were provided ($v_1, \ldots, v_k$), their types ($T_1, \ldots, T_k$) and the types of the remaining attributes ($T_{k+1}, \ldots, T_n$). The model then uses this information to infer and generate appropriate values for the missing attributes to fully instantiate an object obj of type C.

$$obj = eval(C(v_1, \ldots, v_k) \text{ \textbf{by} } m(\theta)) = \text{invoke-model}(m, \theta, C, [v_1, \ldots, v_k], [(T_1, \ldots, T_k), (T_{k+1}, \ldots, T_n)]) \tag{4}$$

As per the example shown in Figure 8b, the **by** call uses the partially initialized object (name being "Einstein") to predict and fill in the remaining attributes (dob). MTP generates a model prompt using the class name, attribute names, types, and the provided values, and ensures that the model's output is converted into the correct types using MT-Runtime. This allows for clean and semantically guided object completion even when only partial input is given.

4.2.3 **by** *at Member Method Definition*. The **by** operator can also be used to define methods within a class. Consider a method *mth* within a class C, which takes parameters $p_1, \ldots, p_n$ with types $T_1, \ldots, T_n$, and returns a value of type T_r. This method can be implemented using $m \in \mathcal{M}$, configured with hyperparameters θ, as follows.

$$class\ C: def\ mth(p_1 : T_1, \ldots, p_n : T_n) \rightarrow T_r\ \textbf{by}\ m(\theta) \qquad (5)$$

At runtime, when this method is called on an object instance obj with input values $v_1, \ldots, v_n$, evaluation to return v_r of type T_r, proceeds as:

$$v_r = eval(obj.mth(v_1, \ldots, v_n)) = \text{invoke-model}(m, \theta, C.mth, obj, [v_1, \ldots, v_n], [C, (T_1, \ldots, T_n), T_r]) \qquad (6)$$

Unlike standalone function calls, class methods have access to the instance's internal state, which provides important context for the model. During evaluation, the model is given the method signature $C.mth$, the object instance obj, the input values and their types, and the expected return type. The object includes all class attributes along with their current values and type information, allowing the model to use this internal context in its reasoning. For example, in Figure 8c, the calculate_age method of the Person class derives the dob attribute directly from the einstein object rather than requiring it as an argument.

The **by** operator provides specific behavioral guarantees through MT-IR (§ 4.3) and MT-Runtime (§ 4.4):

(1) **Type Safety**: Output type conformance to T_r is enforced for all LLM-generated results. When model output cannot be converted to the expected type T_r, the system raises a type error.

```
1  def calculate_age (cur_year: int, dob: str) -> int
       by llm
2
3  age = calculate_age(2024,"March 14, 1879")
```

(a) Function Call

```
1  class Person :
2      name : str
3      dob : str
4
5  einstein = Person("Einstein") by llm
```

(b) Object Initialization

```
1  class Person:
2      name : str
3      dob : str
4
5      def calculate_age (cur_year: int) -> int by llm
           (temperature=0.7)
6
7  einstein = Person("Einstein", "March 14, 1879")
8  einstein.calculate_age(2024)
```

(c) Member Method Call

Fig. 8. Meaning-Typed Programming supports three AI-integration points.

(2) **Temporal Behavior**: The **by** operator exhibits distinct execution patterns based on integration point. For functions and methods, definition time tags the construct for MT-IR processing, while call time triggers MT-Runtime execution. For object initialization, the **by** operator executes immediately during instantiation.

4.3 Meaning-Typed Compilation and IR (MT-IR)

When a **by** call site is encountered, we perform static analysis to extract relevant code semantics, which are used later at runtime to automatically generate prompts for the LLM inference. These relevant semantics, such as object types and function/argument signatures, are available at the source code level and might be lost at the bytecode level. Therefore, they must be captured ahead of time to enable accurate inference during the runtime. As discussed in §3 $\mathbf{C}_2$, this task is challenging because the relevant semantics are often distributed across multiple source files. Figure 9a illustrates this challenge using the video game example from §2. In this case, the Level object referenced in the **by** call site within game.py is defined in a separate file, level.py. Furthermore, Level itself depends on additional nested types declared in yet another file, primitives.py. Such modular code organization, while common for improving code maintainability, complicates the task of statically gathering all necessary semantic context.

To address these complexities, we introduce MT-IR, a data structure that captures relevant code semantics at each **by** call site for prompt generation. MT-IR serves as a bridge between source-level semantics and the prompts for LLM inference. Just as traditional IR is used to generate machine code, MT-IR is used to generate prompts for the model at runtime, where the MT-Runtime relies

solely on MT-IR without needing access to the original source code. Figure **??** presents an example of MT-IR, capturing code semantics for the source code shown in Figure 9a. The MT-IR is then used to generate the prompt shown in Figure 9d.

The compilation pipeline for generating the MT-IR for each **by** call-site involves several key steps:

1. Semantic Registry Generation. During the compilation phase of the language, we construct a *semantic registry* using the Abstract Syntax Tree (AST) generated for each code module. This registry serves as a comprehensive repository of all semantic information present in the code, including variable names, function names, and type information. As seen in Figure 9a, semantic information from multiple modules (game.py, level.py, and primitives.py) is integrated into a unified data structure. The resulting semantic registry, shown in Figure 9b, is a super set of the symbol-table of the language as it contains the definitions as well as the usages of objects, linking usages with definitions.

2. by Call-site Semantic Extraction. The created semantic registry is used to traverse and find semantics relevant to each **by** call-site. The process begins by extracting semantics from **by** call-site. For example, consider the **by** call-site in the game.py module, shown in Figure 9a for a the function call. As defined in equation 2, the **by** operator calls the MT-Runtime using $(f, [v_1, \ldots, v_n], [(T_1, \ldots, T_n), T_r], m, \theta)$ as input information. These are the critical set of semantics relevant to the **by** call-site, which needs to be extracted first from the semantic registry as shown in Figure **??** according to Algorithm 1 line 3-9. Next, the input and output types are checked to determine whether they are primitive types (e.g., integers, floats). If the types are non-primitive, we need to extract their definitions by recursively traversing the semantic registry. Using this traversal, we need to extract the required semantics of the Level object, as illustrated in Figure **??**, which depends on Map. In turn, Map relies on Wall and Position. The use-definition links within
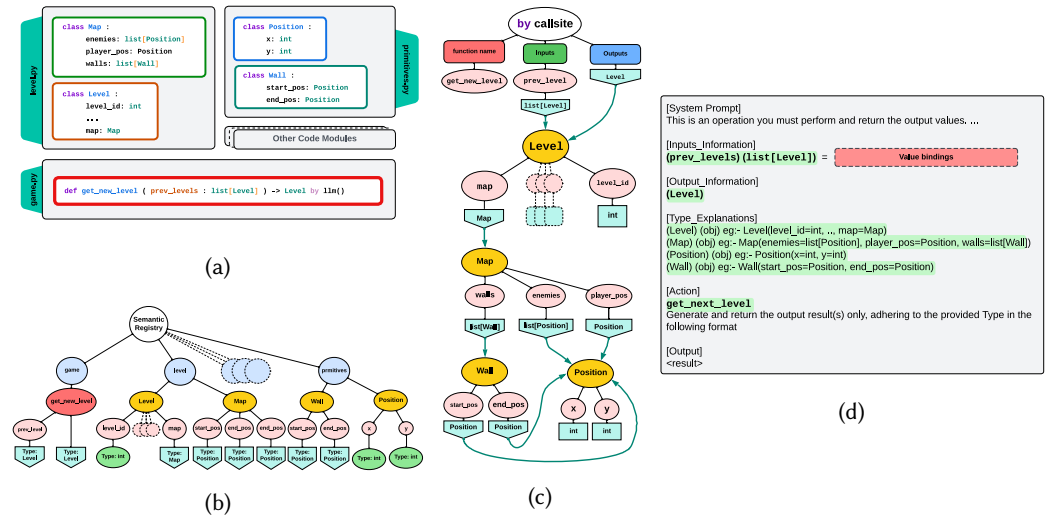
Fig. 9. Representation of how the MTIR is generated and used for the video game example in §2. (a) Code layout in modules. (b) Corresponding semantic registry containing all semantic information in the codebase. (c) MTIR relevant for the get_next_level **by** call-site. (d) Generated prompt using the MTIR (semantic information extracted using MTIR are highlighted in green).
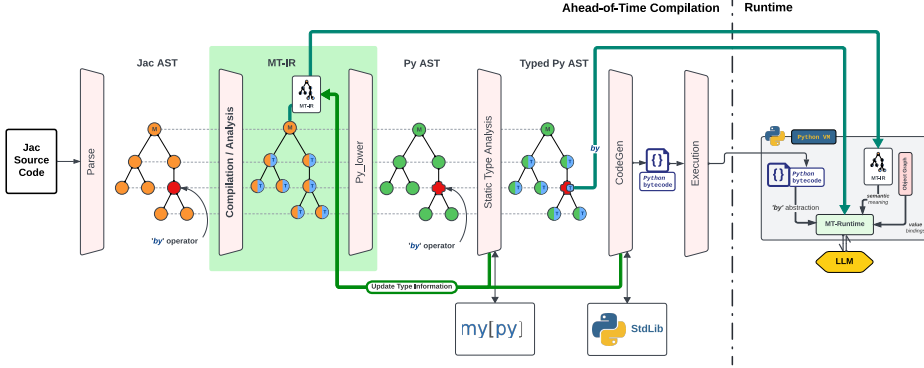
Fig. 10. The Jac compiler workflow on which Meaning-Typed compilation is implementation in the Jac programming language.

the semantic registry enable this traversal, ensuring that all types are resolved down to primitive types. This recursive type resolution is signified with "ExtractTypeDefinition" function on Algorithm 1. This approach effectively captures type information for nested types and addressing (C_2). Algorithm 1 presents the complete MT-IR construction process for reproducibility.

The extracted MT-IR can be used to fill in the semantic information in the prompt as shown in Figure 9d, where the highlighted text of the prompt represents all the semantics available in the MT-IR for the given **by** call-site.

Implementation of MT-IR. We implement the **by** language abstraction, MT-IR, and our compiler pass through *Jac*, a production-grade Python superset language [20, 27–29]. 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 as in Figure 10. The resulted python bytecode will then be interpreted using standard `python3`.

Algorithm 1 MT-IR Constructor Algorithm

Require: Code Base C, Semantic Registry S
Ensure: MT-IR Map M
1: Initialize MT-IR $M \leftarrow \emptyset$
2: **for** each $\mathbf{by}_i$ call-site in C **do**
3:     **if** $\mathbf{by}_i$ is a function-call with name f_i **then**
4:         Initialize $M[\mathbf{by}_i] \leftarrow \langle f_i, (T_1, \dots, T_n) \rightarrow T_r, m, \theta \rangle$ {From equation 1}
5:     **else if** $\mathbf{by}_i$ is a method-call mth_i in object of class C_j **then**
6:         Initialize $M[\mathbf{by}_i] \leftarrow \langle mth_i, C_j, (T_1, \dots, T_n) \rightarrow T_r, m, \theta \rangle$ {From equation 5}
7:     **else if** $\mathbf{by}_i$ is an object-initialization of class C_j **then**
8:         Initialize $M[\mathbf{by}_i] \leftarrow \langle C_j, (T_1, \dots, T_k), (T_{k+1}, \dots, T_n), m, \theta \rangle$ {From equation 4}
9:     **end if**
10:    % Extract Parameter Type Semantics:
11:    **for** each type T_j in input types of $M[\mathbf{by}_i]$ **do**
12:        $type\_def_j \leftarrow \textsc{ExtractTypeDefinition}(T_j, S)$
13:        Add $type\_def_j$ to $M[\mathbf{by}_i]$
14:    **end for**
15:    % Extract Return Type Semantics:
16:    **for** each type T_j in output/return types of $M[\mathbf{by}_i]$ **do**
17:        $type\_def_j \leftarrow \textsc{ExtractTypeDefinition}(T_j, S)$
18:        Add $type\_def_j$ to $M[\mathbf{by}_i]$
19:    **end for**
20: **end for**
21: **return** M

The compilation process for MTP in Jac involves six key steps through the Jac compiler as shown in Figure 10. Initially, an additional compiler pass generates MT-IR by extracting semantic elements from the AST. This information is then refined through multiple analysis passes. Next, the Jac AST is lowered to a Python AST, where specific code constructs are replaced with MT-Runtime calls (depicted as red nodes). Next, MyPy type checking is performed on the Python AST with resulting type information fed back into the MT-IR to enhance its semantic content. This enriched

MT-IR is then registered into the MT-Runtime library, which remains available throughout the program execution. Finally, standard Python bytecode is generated from the modified Python AST, preserving crucial semantic elements in the runtime that would otherwise be lost in conventional compilation in the MT-IR.

4.4 Meaning-Typed Runtime System

For AI-integrated programs, LLM inference is done during the runtime as the prompt requires the semantic information as well as the dynamic values available when the **by** is called. This demands a dynamic system which can combine the MT-IR and dynamic value bindings to generate the final prompt for LLM inference. To this end, we introduce MT-Runtime, a novel automated runtime

Fig. 11. Meaning-typed Runtime System (MT-Runtime).

engine integrated into the language's VM to realize O_3. It is specifically triggered at the **by** call site abstractions, automatically managing the LLM invocation at runtime. MT-Runtime engine effectively addresses the final two challenges mentioned in §3 (C_3 and C_4).

In this Section, we describe the design of MT-Runtime, which functions through the following conceptual stages.

1. Prompt Synthesis. Upon the triggering of a **by** call, MT-Runtime first retrieves the MT-IR which is relevant to that particular **by** call-site and uses its information to fill a generalized prompt template. Based on the location of the **by** call-site, MT-Runtime constructs a prompt that incorporates static semantics, dynamic variable values relevant to the call-site, and the expected output type. As the MT-Runtime resides on the language VM as shown in Figure 10, it has access to the variable values at runtime. In Figure 11, this can be seen where the semantic information is fetched from the MT-IR while the variable values are fetched from the languages object graph. The synthesized prompt can be seen in Figure 9d. Here, the function name determines the 'Action' in the prompt, representing the intended task, while type information and nested types are included under 'Type_Explanations' in the prompt. Additionally, the inclusion of the output type and other schema information in the prompt forces the LLM to generate the output as per the defined schema, making it easier to convert in the type conversion stage. This stage address the challenge of generating meaningful but concise prompts discussed in §3 as C_3.

2. Conversion from LLM Output to Target Output Variable. MT-Runtime queries the LLM with the dynamically synthesized input prompt and receives an output. The next step is to parse the LLM's textual output and convert it into the target output variable addressing C_4.

During the prompt synthesis phase, the prompt instructs the model to generate output following the predefined Python object schema. MT-Runtime leverages Python's built-in `ast.literal_eval()` to evaluate the output text and construct the corresponding variable instance. For example, as in Figure 8b, the variable `einstein` is of the custom type `Person` with attributes `name: str` and `dob: str`. In this case, MT-Runtime instructs the LLM to generate an output as `Person(name="Albert Einstein", dob="03/14/1879")`, which can be directly evaluated into a valid `Person`-typed object.

When MT-Runtime encounters an output that cannot be converted into the target variable type, it constructs a revised prompt that attempts to rectify the error as illustrated in Figure 11. This
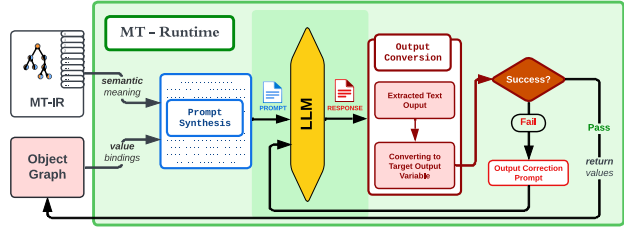
Table 1. Benchmark Suite Descriptions and Correctness Criteria.

Benchmarks	Task Description	Correctness Criteria
Math Problem	Given a grade school math question an integer answer is returned. [17]	Answer matches the correct integer solution.
Translation	Translates a given text from English to French. [4]	Translation preserves meaning and fluency.
Essay Reviewer	Reviews an essay and assigns a letter grade based on the quality of the essay. [54]	Grade is appropriate for essay content and structure.
Joke Generator	Generates jokes based on a given topic, e.g., jokes with punchline. [4]	Joke is coherent, on-topic, and has a valid punchline.
Expert Answer	Given a question, predicts the expert profession best suited to answer and generates an answer. [54]	Expert domain is appropriate and the answer is relevant and accurate.
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]	Correct word identified with valid justification.
MCQ Reasoning	Answers multiple-choice questions by providing a reason for the chosen answer. [4]	Correct choice selected with logically sound reasoning.
Personality Finder	Given a public figure's name, predicts their personality type.	Predicted personality matches known or inferred type.
Template	Given a template object and a set of variables, fills in the template with the variables. [4]	All template fields filled correctly with appropriate values.
Text to Type	Given a text input and a structured data type, extracts the structured data into the structured data type. [4]	Extracted output matches the target structured format exactly.
Taskman	Given a list of tasks, assigns priority ranks and estimates completion times for each task.	Task rankings and estimates are reasonable and well justified.
Level Generator	Generates new video game-levels based on previous levels.	Two playable levels are generated, with the second level being harder than the first based on complexity and enemy count.
Wikipedia	Given a question and an external tool (e.g., Wikipedia), retrieves the relevant answer using ReAct methodology. [4]	Answer is factually correct and supported by retrieved evidence.

revised prompt includes details of the current incorrect output and the expected type, guiding the model toward the correct format. MT-Runtime queries the LLM with this revised input and attempts to parse its output again. These retry attempts continue until either the type conversion succeeds or the maximum number of retries specified by the developer is reached. If the LLM fails to generate a correct output at the end, the MT-Runtime raises an exception with a type error. While errors are handled elegantly through this approach, the revised prompt is mostly short and concise since it only aims to correct the type mismatches. This offers an efficient solution for C_4, handling errors gracefully.

5 EVALUATION

In this section, we present our evaluation of MTP in comparison to prior work (LMQL [4] and DSPy [17]) aimed at reducing the complexities associated with prompt engineering. With reduced developer effort, we strive to ensure that MTP maintains accuracy without increasing token usage with respective to prior works. Additionally, our lightweight runtime system is designed to minimize both cost and runtime overhead. Furthermore, we assess the robustness of MTP in scenarios where poor coding practices diminish the semantic richness of the code. Our evaluation aims to answer the following research questions.

RQ_1 - *How effectively does MTP reduce development complexity for AI-integration?*
RQ_2 - *How does the accuracy of the MTP compare to prior work?*
RQ_3 - *What are the trade-offs for token usage, cost, and runtime in MTP?*
RQ_4 - *How resilient is MTP to sub-optimal coding practices?*

To evaluate these research questions, we introduce, (i) two case-studies, (ii) a user study and (iii) a suite of benchmarks.

(1.) Case Studies (RQ_1). We first qualitatively evaluate the reduction of complexities for AI-integration using MTP (§ 5.1).

(2.) Lines of code (LOC) (RQ_1). In addition to the user study, we evaluate MTP for its effectiveness in reducing programming complexity through a quantitative study measuring lines of code across a comprehensive suite of benchmarks (§ 5.2).

(3.) User study (RQ$_1$). We conducted a user study to evaluate MTP's effectiveness in reducing complexity and increasing developer's productivity (§ 5.3).

(4.) Accuracy and Trend across LLM evolution (RQ$_2$). We evaluate the program accuracy when using MTP versus prior work across 10 various LLMs (§ 5.4).

(5.) Token Usage, Cost and Runtime (RQ$_3$). We measured these performance metrics for MTP compared to DSPy (§ 5.5).

(6.) Sensitivity To Coding Practices (RQ$_4$). Here we analyze how the MTP paradigm can be affected through bad coding practices using the game level generation benchmark. (§ 5.6)

Experimental Setup. We implemented MTP using Jaseci [20, 28, 29], an open-source research infrastructure that offers full control over language semantics, compilation processes, and runtime execution, all built as a superset of Python.

We use OpenAI APIs for experiments with GPT models. We host llama models on a local server with Nvidia 3090 GPU with 24GB VRAM and 64GB RAM. We instrument the source code to profile token usage. Python cProfile[38] is used to measure program runtime. Our benchmark suite consists of problems representative of applications with AI-integration. 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 [17, 54] (Table 1).

5.1 A Case Study : Dynamic Level Generation in Video Game Development

§2 presented an example use-case of AI-integration for automatic level generation of a video game. This video game is a real program developed using an online tutorial [41] using the pygame [30] library. Figure 12 represents three consecutive levels generated by an LLM through AI-integration.

Fig. 12. A sequence of three game levels dynamically generated by an LLM at runtime.

First we examine how this can be achieved using prior work, LMQL [4] and DSPy [17] which aims to simplify prompt engineering. Figure 13 includes the two code snippets for implementation from each framework.

LMQL [4]. Language Model Query Language introduces a SQL-like query language for LLM prompting with a focus on constraining model outputs. This framework enables developers to enforce structured responses through predefined rules and output schemas. As shown in Figure 13a, LMQL provides a structured approach to our video game level progression use case. While the core prompt (lines 8-34) contains similar instructions to traditional prompt engineering methods in Figure 4, LMQL's key advantage lies in its explicit output format definition (lines 22-33), which specifies expected data types and formats for each attribute. During inference, LMQL operates iteratively, generating one output field at a time using uppercase text placeholders to mark generation points. Each successfully generated value is injected back into the prompt, repeating until all required values are obtained. While LMQL improves control over LLM-generated outputs, it does not fully abstract away prompt-engineering, as developers still need to construct base prompts manually. This limits its effectiveness in simplifying AI-integrated application development.

DSPy [17]. This developer-friendly framework offers a more structured approach than LMQL for AI-integration. For simple tasks like question answering, developers use *dspy.Signature* to define structured inputs and outputs. However, complex tasks requiring typed outputs still necessitate manual code modifications. Figure 13b shows our DSPy implementation where lines 30–34 define a
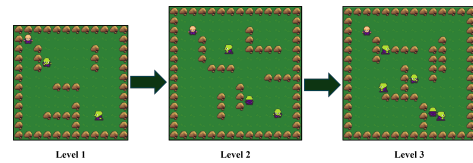
```
1  import json
2  import lmql
3
4  @lmql.query
5  def generate_next_level(prev_level_json: str):
6      '''lmql
7      argmax
8      "You are an expert game level generator. Given the
              current level details, generate the next level
              with increased difficulty.

9
10     ### Current Level:
11     {prev_level_json}
12
13     ### Rules for Next Level:
14     - Increase difficulty by at least 1.
15     - Increase width and height slightly (max 10% each).
16     - Increase the number of walls and enemies based on
              difficulty progression.
17     - Generate a valid map with walls and enemy positions.
18     - Ensure `player_pos` is within bounds and not
              colliding with walls.
19
20     ### Output Format:
21     Provide the next level in JSON format:
22     {'
23         'level_id': [LEVEL_ID:int],
24         'difficulty': [DIFFICULTY:int],
25         'width': [WIDTH:int if WIDTH > current_width and
                  WIDTH <= current_width * 1.1],
26         'height': [HEIGHT:int if HEIGHT > current_height
                  and HEIGHT <= current_height * 1.1],
27         'num_wall': [NUM_WALL:int if NUM_WALL >=
                  current_num_wall],
28         'num_enemies': [NUM_ENEMIES:int if NUM_ENEMIES >=
                  current_num_enemies],
29         'map': {
30             'walls': [WALLS:list of {'start_pos': {'x': [X1
                      :int], 'y': [Y1:int]}, 'end_pos': {'x':
                      [X2:int], 'y': [Y2:int]}}],
31             'enemies': [ENEMIES:list of {'x': [EX:int], 'y
                      ': [EY:int]}],
32             'player_pos': {'x': [PLAYER_X:int if PLAYER_X
                      in (0, WIDTH-1)], 'y': [PLAYER_Y:int if
                      PLAYER_Y in (0, HEIGHT-1)]}
33         }
34     }'"
35     '''
36
37  def get_next_level(prev_level: Level -> Level:
38      prev_level_json = json.dumps(prev_level.__dict__,
              default=lambda o: o.__dict__)
39      level_dict = json.loads(generate_next_level(
              prev_level_json=prev_level_json))
40
41      def parse_position(data: dict) -> Position:
42          ...
43
44      def parse_wall(data: dict) -> Wall:
45          ...
46
47      def parse_map(data: dict) -> Map:
48          ...
49
50      return Level(
51          level_id=level_dict["level_id"],
52          difficulty=level_dict["difficulty"],
53          width=level_dict["width"],
54          height=level_dict["height"],
55          num_wall=level_dict["num_wall"],
56          num_enemies=level_dict["num_enemies"],
57          map=parse_map(level_dict["map"])
58      )
```

(a) AI-integrated Game Levels : LMQL

```
1   import dspy
2   from pydantic import BaseModel, Field
3
4   llm = dspy.OpenAI(model="gpt-4o", max_tokens=1024)
5   dspy.settings.configure(lm=llm)
6
7   class Position(BaseModel):
8       x: int = Field(description="X Coordinate")
9       y: int = Field(description="Y Coordinate")
10
11  class Wall(BaseModel):
12      start_pos: Position = Field(description="Start
              Position of the Wall")
13      end_pos: Position = Field(description="End Position of
              the Wall")
14
15  class Map(BaseModel):
16      walls: list[Wall] = Field(description="Walls in the
              Map")
17      small_obstacles: List[Position] = Field(description="
              Obstacles")
18      enemies: list[Position] = Field(description="Enemies
              in the Map")
19      player_pos: Position = Field(description="Player
              Position in the Map")
20
21  class Level(BaseModel):
22      level_id: int = Field(description="Level No")
23      difficulty: int = Field(description="Difficulty of the
              Level")
24      width: int = Field(description="Width of the Map")
25      height: int = Field(description="Height of the Map")
26      num_wall: int = Field(description="Number of Walls in
              the Map")
27      num_enemies: int = Field(description="Number of
              Enemies in the Map")
28      map: Map = Field(description="Layout of the Map")
29
30  class get_next_level(dspy.Signature):
31      """Create Next Level"""
32
33      prev_levels: list[Level] = dspy.InputField(desc="Last
              Played Levels")
34      level: Level = dspy.OutputField(desc="Level")
35
36  new_level_map = dspy.TypedPredictor(get_next_level)(level
          =prev_levels).level
```

(b) AI-integrated Game Levels : DSPy

```
1   from mtp.core.llms import OpenAI
2
3   llm = OpenAI(model_name="gpt-4o")
4
5   def get_next_level(prev_levels: list[Level]) -> Level by
          llm
6
7   new_level = get_next_level(prev_levels)
```

(c) AI-integrated Game Levels : MTP

Fig. 13. Implementation of the video game usecase using (a) LMQL and (b) DSPY frameworks. (c) This gives a preview how the same task can be achieved using the MTP paradigm introduced in the paper.

`dspy.Signature` with task description and explicit input/output fields. A key challenge is DSPy's inability to infer what a `Level` object represents without parsing the full codebase. As a workaround, developers must explicitly annotate class definitions to inherit from Pydantic's `BaseModel` (lines 7, 11, 15, and 21), with attributes defined as `Field` objects containing descriptions. While DSPy improves usability over generic prompt engineering and LMQL, its Pydantic reliance introduces a steep learning curve and forces developers to modify existing code accordingly.

In contrast, our MTP implementation introduces a simple abstraction that reduces complexity for developers. MTP achieves the same functionality with just a few lines of code additions and
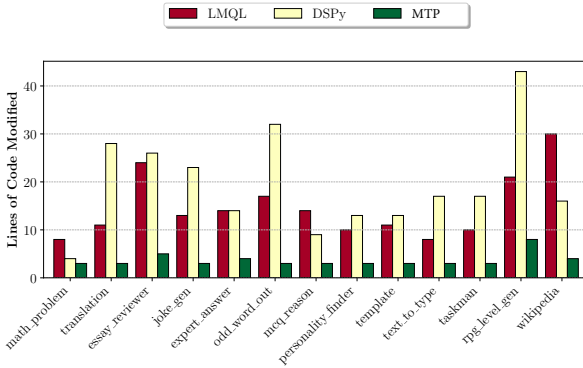
Fig. 14. Lines of Code Modifications required for LLM integration into benchmark programs with for implementing a new feature.

Table 2. Lines of Code Modified reduction achieved by MTP Compared to DSPy and LMQL for LLM integration.

Problem	LoC Modified for 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×**

modifications (Figure 13c), compared to the 58 lines of code modifications required for LMQL and 36 lines for DSPy integration (here *LoC modified* metric signifies the developer complexity of integrating an LLM through the respective framework when adding a new functionality, further elaborated in § 5.2). MTP maintains the same function signature and leverages the semantics of this signature along with the data structures in Figure 2 to automate prompt generation. MT-runtime engine described in §4.4 automates LLM output parsing and conversion to custom-typed variables, eliminating additional developer effort.

When comparing all three implementations, only LMQL includes map generation rules and instructions in its prompts, whereas DSPy and MTP do not, as they do not require manual prompt engineering. However, despite the absence of manual prompts, MTP consistently maintains higher accuracy than LMQL, as we show later in Figure 18 and § 5.4. We hypothesize that this is due to the fact that MTP uses code semantics to infer the nature of the task. In this case, the source code semantics, such as the variable names and values associated to previous_levels and difficulty, enable the LLM to infer the need for increasing difficulty across game levels, achieving high accuracy. It is worth noting that additional specific rules such as "width/levels increase by 10%" can also be passed as arguments to the **by** call-site, allowing MTP to consistently apply them during level generation.

5.2 Lines of Code (LOC) Modified Comparison

We evaluate MTP for its effectiveness in reducing programming complexity when integrating an LLM into traditional programs that do not use any LLM features. We conduct a quantitative study measuring *lines of code (LOC) modified* across our benchmark suite (Table 1), where each benchmark was implemented using all three frameworks: LMQL, DSPy, and MTP. **LOC modified for LLM integration counts the number of lines that need to be added or modified when a developer integrates a new LLM feature into an existing codebase.** LOC-based metrics have also been used in prior work to assess integration effort in language model programming frameworks [4]. Figure 14 presents the LOC modifications required for AI-integration across benchmarks, starting from a non-LLM integrated baseline. The results show that MTP requires the minimum code modifications to integrate an LLM into traditional programming. DSPy and LMQL require much

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?

larger LOC across all benchmarks. This difference is due to the simple yet intuitive abstraction introduced in MTP.

Table 2 presents the *LOC Modified* reduction factors for MTP, illustrating how it simplifies LLM feature integration with reduced implementation effort relative to LMQL and DSPy. It should be noted that these factors were derived form the LoC modified represented on Figure 14. This large reduction is because DSPy requires tedious type annotations and LMQL requires manual programmatic prompt engineering. In contrast, MTP requires minimum code change for LLM integration by simply leveraging the **by** operator and hiding all complexity.

5.3 User Study Evaluating MTP against LMQL and DSPy

To quantify the degree of reduction in developer complexity and effort achieved through MTP, compared to prior work, we conducted a user study consisting of 20 software developers.

User Study Protocol. We recruited 20 undergraduate students from a Natural Language Processing class who had Python coding experience but minimal LLM-integrated development background (limited to basic prompt engineering). This allowed us to evaluate developer complexity from a newcomer's perspective, as participants lacked prior experience with the baseline frameworks or MTP. Participants were given one day to familiarize themselves with MTP, DSPy, and LMQL through standard documentation, tutorials, and online resources, then tasked with implementing three progressively challenging problems using all three frameworks: *Essay Evaluator (Easy)*, *Task Manager (Medium)*, and *Game Level*

Fig. 15. User study success rates across LMQL, DSPy, and MTP.

Generator (Hard). Each task was selected from our benchmark suite to represent varying complexity levels, with 90 minutes allocated per implementation. Upon completion, participants provided feedback on each framework's usability and effectiveness through a comprehensive questionnaire.
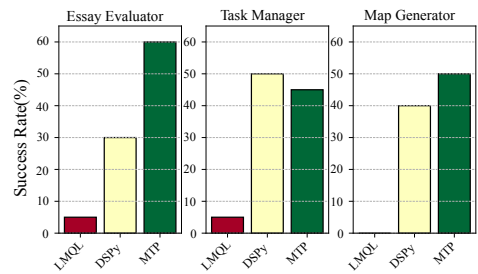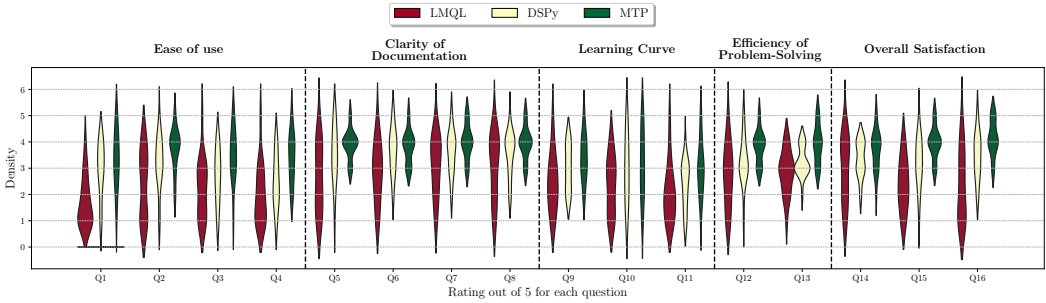
Fig. 16. Violin plot showing the distribution of scores given by participants for each framework across the five criteria: Ease of Use, Clarity of Documentation, Learning Curve, Efficiency of Problem-Solving, and Overall Satisfaction.

To ensure the integrity of the study, responses were collected anonymously, participants were compensated for their time, and the sole objective disclosed to them was the evaluation of three different LLM integration frameworks.

Success Rate. Figure 15 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 with LMQL, highlighting the difficulty to manually craft prompt and than manually interpret output to convert to a complex object type.

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 17. 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.

Fig. 17. User evaluation of LMQL, DSPy and MTP on five usability criteria

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.
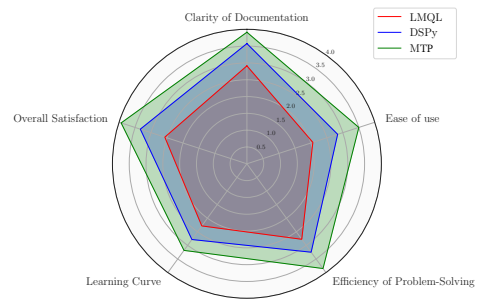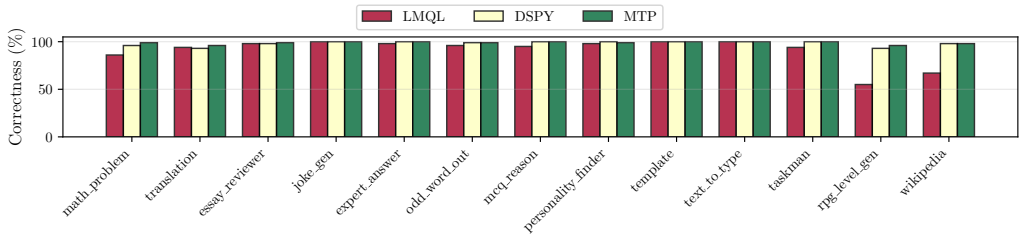
Fig. 18. Accuracy of each benchmark relative to the correctness criteria defined for each benchmark, for 100 trails.

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."*.

Using the results of the case study, user study and LoC reduction evaluation we can conclude that the abstraction offered through MTP reduces complexity for developers streamlining model-integration compared to other available solutions.

5.4 Program accuracy

In this section, we evaluate whether MTP's complexity reduction comes at a cost of accuracy loss and investigate the impact of LLM evolution on program accuracy when using MTP compared to previous work.

Correctness across benchmark suite. We evaluate our system using the benchmark suite outlined in Table 1, employing GPT-4o. Each benchmark includes task-specific correctness criteria, also detailed in Table 1. Our evaluation consists of 100 runs per benchmark, derived from 20 distinct test inputs, each evaluated 5 times. As many benchmarks involve generative tasks without a single ground truth answer, we adopt different evaluation methodologies based on task type. These, summarized in Table 4, include GPT4Score [11] and human evaluation for open-ended tasks. Across all methods, we apply a binary correctness standard: only fully correct outputs are marked as correct, with partially correct answers considered incorrect. For each framework, we report accuracy as the percentage of correct outputs across the 100 trials for each benchmark.

Table 4. Evaluation methodologies for benchmark correctness assessment

Method	Description	Applied Benchmarks
ExactMatch	Direct equality with ground truth	text_to_type, mcq_reason, math_problem
GPTScore [11]	GPT model as judge [11, 52] evaluating compliance with criteria	essay_reviewer, expert_answer, wikipedia, template, taskman, translation, personality_finder, joke_gen, odd_word_out
Human Evaluation	Human evaluation based on defined criteria, conducted by members of the lab cohort.	rpg_level_gen

Figure 18 shows that all frameworks achieve above 50% correctness across all benchmarks. It should be noted that the correctness of LMQL is directly tied to how well developers manually craft prompts, unlike the automated prompt generation in MTP and DSPy. This manual effort increases complexity and reduces developer productivity, as further demonstrated in our user study (§ 5.3). Therefore, LMQL implementations are used as provided in the source material when available, or otherwise faithfully implemented. LMQL demonstrates lower accuracy than DSPy and MTP in 10 out of 13 benchmarks. DSPy and MTP show nearly identical performance, with MTP outperforming
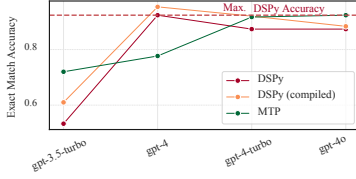
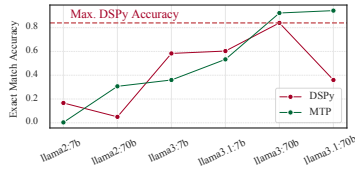Fig. 19. Accuracy on GSM8K across OpenAI models.

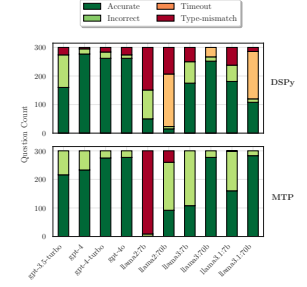Fig. 20. Accuracy on GSM8K across Llama models.

Fig. 21. Results breakdown for DSPy and MTP on GSM8K. (Timeout set at 120s).

DSPy in 9 out of 13 benchmarks. The average correctness across benchmarks is 90.85% for LMQL, 98.23% for DSPy, and 98.92% for MTP, indicating that MTP maintains competitive correctness while significantly reducing developer complexities as shown in § 5.1, § 5.2 and § 5.3.

Accuracy for GSM8k and Trends Across Model Evolution. In addition, we follow the same methodology as in previous work [17] 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 19, 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 19). These results further highlight the effectiveness of MTP in automatically generating robust prompts using the semantic elements available in MT-IR.

We performed the same experiments with recent Llama models. Figure 20 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 21 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).
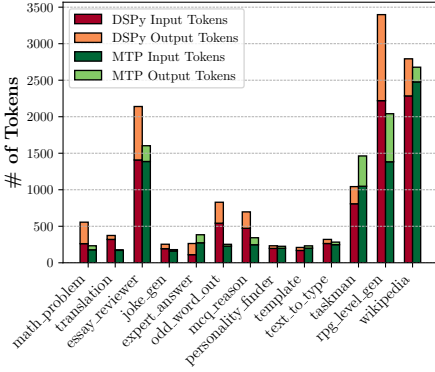
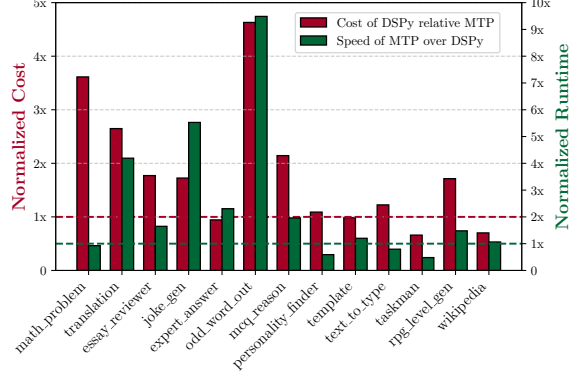Fig. 22. Token usage comparison between MTP and DSPy.

Fig. 23. Cost and runtime speed comparison between MTP and DSPy.

5.5 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 22 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). Figure 23 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 [36]. The runtime is measured using cprofile. Figure 23 demonstrates that MTP achieves significant runtime improvement and cost reduction due to its more efficient token usage on majority of the benchmarks. As an example, for the math_problem benchmark, we can see more that 3.5x cost savings while maintaining same speed as DSPy. On the other hand, for the odd_word_out benchmark we can observe more than 4x cost reduction as well as runtime improvement over DSPy.

5.6 Sensitivity to Coding Practices

MTP leverages semantics, enabling automated prompt generation for AI-integrated applications. Here, we evaluate how sensitive MTP is to poor coding practices, particularly when semantic richness is reduced.

To study this, we used the same video game level generation program from § 2. Starting with the original implementation, we systematically introduced poor coding practices and measured the probability of generating a correct output out of 100 consecutive runs. Correctness of the output is defined in Table 1.

Figure 24 shows a gradual decline in correctness of output as semantic quality deteriorates. We first examined the effects of shortened but still meaningful names, a common practice for maintainability. This had no impact on correctness. Next, we evaluated progressively worse
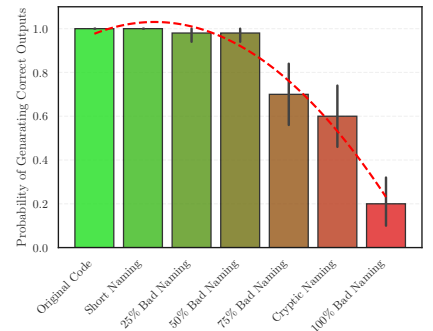

Fig. 24. Sensitivity of MTP to poor coding practices. The correct output probability declines slowly for first three experiments showing the resilience of MTP.

naming conventions by replacing meaningful identifiers with increasingly abbreviated or single letter alternatives. We tested four levels of degradation: renaming 25%, 50%, 75%, and 100% of identifiers.

At 25% and 50% renaming, MTP remained highly effective, with a correct output probability of 98%. However, as renaming increased to 70%, where nearly all data structures had poorly chosen names, correctness dropped to 70%. When renaming reached 100%, including function signatures, correctness declined sharply to 20%, highlighting the limits of MTP's resilience.

Additionally, we tested cryptic naming conventions, where identifiers retained internal consistency but lost intuitive meaning (e.g., renaming "Level" to "Boss" and "Wall" to "Chasm"). As shown in Figure 24, this resulted in a 60% success rate, having a greater negative impact than 75% abbreviation but still performing better than complete semantic loss.

These results demonstrate MTP's robustness, as the LLM can infer missing context, but excessive semantic degradation reduces accuracy.
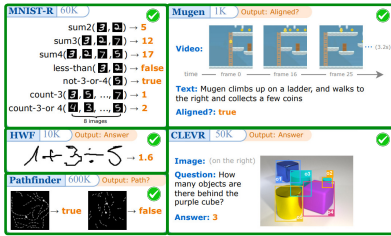
(a) Multi-modal solutions from Scallop [24].

```
1   # CLEVR benchmark problem implemented in MTP
2
3   from mtp.core.llms import OpenAI
4   from mtp.core.types import Image
5
6   llm = OpenAI(model_name="gpt-4o")
7
8   can get_answer(img: Image, question: str) -> str by llm
        ()
9
10  question:str = 'How many objects behind the purple cube
        ?'
11  print(get_answer(Image('image.png'), question))
```

(b) Example CLEVR code snippet.

Fig. 25. Implemented multi-modal solutions for visual tasks from Scallop (Left) and CLEVR code snippet (Right).

5.7 Using Multi-Modal LLMs to solve vision problems

There has been significant advancement recently in Multi-modal Generative AI models [42]. These models, such as LLaVa [25], QWEN [49], 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. Multi-modal LLMs achieve this by using specialized adapters that map different information modalities (text, images, video) into a unified vector space, enabling seamless reasoning across diverse input types.

MTP leverages a similar approach by providing custom type interfaces that serve as adapters for different modalities. Just as multi-modal LLMs use internal adapters to handle various data types, developers can define custom interfaces such as Image or Video classes that seamlessly integrate with underlying system to interact with multi-modal LLMs. In this example, we demonstrate how MTP generalizes leveraging such multi-modal generative models to build AI-integrated programs using the same abstraction, reducing the developer effort at implementing such applications.

We select 5 vision benchmarks from related work Scallop [24]. This set of problems represents a wide range of multi-modal tasks such as visual reasoning and visual question-answering, as shown in Figure 25a. Using MTP, we are able to implement fully working solutions for these 5 vision problems using the same abstraction. The code snippet in Figure 25b shows our implementation of one of the benchmark problems (CLEVR), which involves reasoning about an input image and providing an answer to a user's question. As MTP treats every problem using semantics and types, using images as inputs just comes down to defining the Image type used on line 4 in Figure 25b. This adapter-like interface abstracts away the complexity of multi-modal model

integration, demonstrating the wide range of use cases for MTP while further reducing the need for developers to learn different frameworks for various AI-integration problems.

Our evaluation confirms that MTP simplifies LLM integration by reducing developer effort (**RQ**₁) while maintaining accuracy (**RQ**₂). By minimizing code complexity and improving usability, MTP outperforms DSPy and LMQL without increasing token usage and runtime costs (**RQ**₃). Additionally, MTP shows a good degree of resilience to poor coding practices (**RQ**₄).

6 RELATED WORK

Recent efforts to integrate Large Language Models (LLMs) into applications have emerged along two primary dimensions discussed in Section 2: AI-code generation and AI-integration. Works such as Pythoness [23] directly address AI-code generation, while we focus on AI-integration in this paper.

Several frameworks have been developed to facilitate LLM integration into applications. DSPy [17], LMQL [4], and SGLang [54] 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.

In addition to these frameworks there are tools built to streamline some parts of the prompt engineering process such as Outlines [46] and TypeChat [33]. These tools focus on structured output generation using schemas and are introduced as complementary tools on top of prompt engineering which can improve accuracy, not as standalone frameworks that reduce prompt engineering effort. LangChain [21] and LlamaIndex [26] 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, 10, 14, 22, 37, 39, 47, 53]. 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, 13, 15, 16, 18, 31, 32, 34, 35, 44, 48, 50, 51]. Furthermore, researchers have explored architectural support for ML/AI models [12, 55]. 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.

7 CONCLUSION

Software is rapidly evolving from traditional logical code to AI-integrated applications that leverage generative AI and 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 novel approach to simplify the creation of AI-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.

Acknowledgments

Data-Availability Statement

The artifact accompanying this paper, which includes the implementation, benchmarks, and evaluation scripts, is available on github.com/Jayanaka-98/mtllm-oopsla2025 [9].

References

[1] Sohaib Ahmad, Hui Guan, Brian D. Friedman, Thomas Williams, Ramesh K. Sitaraman, and Thomas Woo. 2024. 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* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 318–334. doi:10.1145/3617232.3624849

[2] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, and Yuxiong He. 2022. DeepSpeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Dallas, Texas) *(SC '22)*. IEEE Press, Article 46, 15 pages.

[3] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith Chintala. 2024. 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* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 929–947. doi:10.1145/3620665.3640366

[4] Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. 2023. Prompting Is Programming: A Query Language for Large Language Models. *Proc. ACM Program. Lang.* 7, PLDI, Article 186 (June 2023), 24 pages. doi:10.1145/3591300

[5] Renze Chen, Zijian Ding, Size Zheng, Chengrui Zhang, Jingwen Leng, Xuanzhe Liu, and Yun Liang. 2024. 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*. 607–621.

[6] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. https://api.semanticscholar.org/CorpusID:52939079

[7] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. 2022. Serving Heterogeneous Machine Learning Models on Multi-GPU Servers with Spatio-Temporal Sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 199–216. https://www.usenix.org/conference/atc22/presentation/choi-seungbeom

[8] Karl Cobbe, Vineet Kosaraju, Mo Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training Verifiers to Solve Math Word Problems. *ArXiv* abs/2110.14168 (2021). https://api.semanticscholar.org/CorpusID:239998651

[9] Jayanaka Lakindu Dantanarayana, Yiping Kang, Kugesan Sivasothynathan, Christopher Clarke, Baichuan Li, Savini Kashmira, Krisztian Flautner, Lingjia Tang, and Jason Mars. 2025. *Jayanaka-98/mtllm-oopsla2025: OOPSLA 2025 Artifact.* doi:10.5281/zenodo.16929189

[10] Michael Davies, Ian McDougall, Selvaraj Anandaraj, Deep Machchhar, Rithik Jain, and Karthikeyan Sankaralingam. 2024. 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* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 20–36. doi:10.1145/3620665.3640367

[11] Jinlan Fu, See-Kiong Ng, Zhengbao Jiang, and Pengfei Liu. 2024. GPTScore: Evaluate as You Desire. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, Kevin Duh, Helena Gomez, and Steven Bethard (Eds.). Association for Computational Linguistics, Mexico City, Mexico, 6556–6576. doi:10.18653/v1/2024.naacl-long.365

[12] Soroush Ghodrati, Sean Kinzer, Hanyang Xu, Rohan Mahapatra, Yoonsung Kim, Byung Hoon Ahn, Dong Kai Wang, Lavanya Karthikeyan, Amir Yazdanbakhsh, Jongse Park, Nam Sung Kim, and Hadi Esmaeilzadeh. 2024. 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* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 1165–1182. doi:10.1145/3620665.3640365

[13] Cong Guo, Jiaming Tang, Weiming Hu, Jingwen Leng, Chen Zhang, Fan Yang, Yunxin Liu, Minyi Guo, and Yuhao Zhu. 2023. OliVe: Accelerating Large Language Models via Hardware-friendly Outlier-Victim Pair Quantization. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 3, 15 pages. doi:10.1145/3579371.3589038

[14] Muyan Hu, Ashwin Venkatram, Shreyashri Biswas, Balamurugan Marimuthu, Bohan Hou, Gabriele Oliaro, Haojie Wang, Liyan Zheng, Xupeng Miao, Jidong Zhai, and Zhihao Jia. 2024. 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*. ACM, 755–769. doi:10.1145/3620666.3651383

[15] Qijing Huang, Minwoo Kang, Grace Dinh, Thomas Norell, Aravind Kalaiah, James Demmel, John Wawrzynek, and Yakun Sophia Shao. 2021. CoSA: Scheduling by Constrained Optimization for Spatial Accelerators. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 554–566. doi:10.1109/ISCA52012.2021.00050

[16] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. 2020. 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)*. USENIX Association, 463–479. https://www.usenix.org/conference/osdi20/presentation/jiang

[17] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan A, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. 2024. DSPy: Compiling Declarative Language Model Calls into State-of-the-Art Pipelines. (2024). https://openreview.net/forum?id=sY5N0zY5Od

[18] Seah Kim, Hyoukjun Kwon, Jinook Song, Jihyuck Jo, Yu-Hsin Chen, Liangzhen Lai, and Vikas Chandra. 2024. 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* (Vancouver, BC, Canada) *(ASPLOS '23)*. Association for Computing Machinery, New York, NY, USA, 73–86. doi:10.1145/3623278.3624753

[19] Jaseci Labs. 2024. Jaclang Pypi Package. https://pypi.org/project/jaclang/ Accessed: 2024-10-18.

[20] Jaseci Labs. 2024. Jaseci Github Repo. https://github.com/Jaseci-Labs/jaseci Accessed: 2024-10-18.

[21] Langchain. 2024. Langchain. https://github.com/langchain-ai/langchain 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. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. doi:10.1109/CGO51591.2021.9370308

[23] Kyla H. Levin, Kyle Gwilt, Emery D. Berger, and Stephen N. Freund. 2025. Effective LLM-Driven Code Generation with Pythoness. arXiv:2501.02138 [cs.PL] https://arxiv.org/abs/2501.02138

[24] Ziyang Li, Jiani Huang, and Mayur Naik. 2023. Scallop: A Language for Neurosymbolic Programming. *Proc. ACM Program. Lang.* 7, PLDI, Article 166 (June 2023), 25 pages. doi:10.1145/3591280

[25] Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. 2023. Visual Instruction Tuning. In *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (Eds.), Vol. 36. Curran Associates, Inc., 34892–34916. https://proceedings.neurips.cc/paper_files/paper/2023/file/6dcf277ea32ce3288914faf369fe6de0-Paper-Conference.pdf

[26] Jerry Liu. 2022. *LlamaIndex*. doi:10.5281/zenodo.1234

[27] Jason Mars. 2025. Extending Object Spatial Semantics for Scale Agnostic Programming. arXiv:2504.03109 [cs.PL] https://arxiv.org/abs/2504.03109

[28] Jason Mars. 2025. Object-Spatial Programming. arXiv:2503.15812 [cs.PL] https://arxiv.org/abs/2503.15812

[29] Jason Mars, Yiping Kang, Roland Daynauth, Baichuan Li, Ashish Mahendra, Krisztian Flautner, and Lingjia Tang. 2023. The Jaseci Programming Paradigm and Runtime Stack: Building Scale-Out Production Applications Easy and Fast. *IEEE Comput. Archit. Lett.* 22, 2 (July 2023), 101–104. doi:10.1109/LCA.2023.3274038

[30] Will McGugan. 2007. *Beginning game development with Python and Pygame: from novice to professional*. Apress.

[31] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, et al. 2024. 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*. 932–949.

[32] Xupeng Miao, Chunan Shi, Jiangfei Duan, Xiaoli Xi, Dahua Lin, Bin Cui, and Zhihao Jia. 2024. 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*. 1112–1127.

[33] Microsoft. [n. d.]. TypeChat: Helps get well-typed responses from language models to build pragmatic natural language interfaces. https://microsoft.github.io/TypeChat. Accessed: 2025-08-21.

[34] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. 2021. 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 (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 883–898. doi:10.1145/3453483.3454083

[35] Hyungjun Oh, Kihong Kim, Jaemin Kim, Sungkyun Kim, Junyeol Lee, Du-seong Chang, and Jiwon Seo. 2024. 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*. 369–384.

[36] OpenAI. 2024. OpenAI API Pricing. https://openai.com/api/pricing/ Accessed: 2024-10-18.

[37] Zaifeng Pan, Zhen Zheng, Feng Zhang, Ruofan Wu, Hao Liang, Dalin Wang, Xiafei Qiu, Junjie Bai, Wei Lin, and Xiaoyong Du. 2024. 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* (Vancouver, BC, Canada) *(ASPLOS '23)*. Association for Computing Machinery, New York, NY, USA, 268–286. doi:10.1145/3623278.3624761

[38] Python Software Foundation. 2024. *The Python Standard Library: cProfile — Profile Module.* https://docs.python.org/3/library/profile.html#module-cProfile Accessed: 2024-10-19.

[39] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, Jack Montgomery, Bert Maher, Satish Nadathur, Jakob Olesen, Jongsoo Park, Artem Rakhov, Misha Smelyanskiy, and Man Wang. 2019. Glow: Graph Lowering Compiler Techniques for Neural Networks. (2019). arXiv:1805.00907 [cs.PL] https://arxiv.org/abs/1805.00907

[40] Reshabh K Sharma, Jonathan De Halleux, Shraddha Barke, and Benjamin Zorn. 2025. PromptPex: Automatic Test Generation for Language Model Prompts. (2025). arXiv:2503.05070 [cs.SE] https://arxiv.org/abs/2503.05070

[41] ShawCode. 2021. Pygame RPG Tutorial. YouTube. https://www.youtube.com/playlist?list=PLkkm3wcQHjT7gn81Wn-e78cAyhwBW3FIc Accessed: 2024-09-24.

[42] Shakti N. Wadekar, Abhishek Chaurasia, Aman Chadha, and Eugenio Culurciello. 2024. The Evolution of Multimodal Model Architectures. (2024). arXiv:2405.17927 [cs.AI] https://arxiv.org/abs/2405.17927

[43] Jianxun Wang and Yixiang Chen. 2023. A Review on Code Generation with LLMs: Application and Evaluation. In *2023 IEEE International Conference on Medical Artificial Intelligence (MedAI)*. 284–289. doi:10.1109/MedAI59581.2023.00044

[44] Zheng Wang, Yuke Wang, Jiaqi Deng, Da Zheng, Ang Li, and Yufei Ding. 2024. 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* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 964–979. doi:10.1145/3620665.3640406

[45] Irene Weber. 2024. Large Language Models as Software Components: A Taxonomy for LLM-Integrated Applications. (2024). arXiv:2406.10300 [cs.SE] https://arxiv.org/abs/2406.10300

[46] Brandon T. Willard and Rémi Louf. 2023. Efficient Guided Generation for Large Language Models. (2023). arXiv:2307.09702 [cs.CL] https://arxiv.org/abs/2307.09702

[47] Chunwei Xia, Jiacheng Zhao, Qianqi Sun, Zheng Wang, Yuan Wen, Teng Yu, Xiaobing Feng, and Huimin Cui. 2024. 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* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 286–301. doi:10.1145/3617232.3624858

[48] Daliang Xu, Mengwei Xu, Chiheng Lou, Li Zhang, Gang Huang, Xin Jin, and Xuanzhe Liu. 2024. 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* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 368–385. doi:10.1145/3617232.3624847

[49] 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. 2024. Qwen2 Technical Report. arXiv:2407.10671 [cs.CL] https://arxiv.org/abs/2407.10671

[50] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A Distributed Serving System for Transformer-Based Generative Models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 521–538. https://www.usenix.org/conference/osdi22/presentation/yu

[51] Chenguang Zheng, Hongzhi Chen, Yuxuan Cheng, Zhezheng Song, Yifan Wu, Changji Li, James Cheng, Hao Yang, and Shuai Zhang. 2022. ByteGNN: efficient graph neural network training at large scale. *Proc. VLDB Endow.* 15, 6 (Feb.

2022), 1228–1242. doi:10.14778/3514061.3514069

[52] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging LLM-as-a-judge with MT-bench and Chatbot Arena. , Article 2020 (2023), 29 pages.

[53] 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. 2020. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 863–879. https://www.usenix.org/conference/osdi20/presentation/zheng

[54] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2025. SGLang: efficient execution of structured language model programs. , Article 2000 (2025), 27 pages.

[55] Kai Zhong, Zhenhua Zhu, Guohao Dai, Hongyi Wang, Xinhao Yang, Haoyu Zhang, Jin Si, Qiuli Mao, Shulin Zeng, Ke Hong, Genghan Zhang, Huazhong Yang, and Yu Wang. 2024. 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* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 349–366. doi:10.1145/3620666.3651336