Structure is All You Need

Arnav Wadehra
Charlottesville, USA
txt3kr@virginia.edu

Jeremy Suh
Charlottesville, USA
bes9ub@virginia.edu

ABSTRACT

This paper introduces an approach to integrating Large Language Models (LLMs) into application-layer software by leveraging structured outputs. We demonstrate that while LLMs excel in generating rich, natural language responses, traditional programs struggle to parse these responses due to their lack of structure. We propose a solution that involves guiding LLMs to produce structured outputs, such as JSON or TypeScript types, which are more compatible with existing software. This approach, which we term "Type Engineering", bridges the gap between natural language intelligence and application logic, making LLMs backwards compatible with code. We argue that this structured approach not only simplifies the integration of LLMs into applications, but also enhances their utility by making their outputs more predictable and manageable. This structured approach to LLM output paves the way for more dynamic and interactive user interfaces, enabling LLMs to easily "plug into" and provide an intelligence layer in the software systems that already exist today.

INTRODUCTION

In the realm of artificial intelligence, large language models (LLMs) have emerged as a powerful tool for processing and understanding vast amounts of natural language data. These models, built on neural network architectures (specifically the transformer architecture) are capable of understanding complex language patterns and reasoning across large amounts of unstructured data. Despite their potential, LLMs behave differently from a programming language, which can make it difficult to use and integrate them effectively into existing software systems.

To address these challenges, the concept of structured outputs from LLMs can be used. Structured outputs refer to the use of specific formats or schemas to organize the data returned by a program or function. In the context of data-driven programming, structured outputs can be the new data structures created by functions that take in data as input. By having LLMs generate only structured outputs, we can make LLMs backwards compatible with existing code, thereby creating a structured interface between natural language and application logic. This approach not only simplifies the integration of LLMs into applications but also enhances their practicality for everyday use.

ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

STRUCTURED OUTPUTS IN PROGRAMMING

In traditional software development, structured outputs play a crucial role in ensuring that data is organized and predictable. This concept is especially pertinent in the realm of application-layer software, where data consistency and integrity are paramount. For instance, JSON (JavaScript Object Notation) has emerged as a popular data format due to its lightweight nature and easy readability by humans and machines alike (Crockford, 2006). Similarly, XML (eXtensible Markup Language) is widely used for its ability to define custom data structures, making it a versatile choice for many applications (Bray et al., 2008). These structured formats enable programmers to represent complex data in a way that is both standardized and flexible, facilitating data interchange between different software systems.

The importance of structured outputs extends to the concept of type-hints in programming languages like Python, TypeScript, and others. Type-hinting involves specifying the expected data types of function arguments, return values, and variables within a program. This practice not only improves code readability but also assists in error detection during the development process (van Rossum et al., 2021). For example, TypeScript, a superset of JavaScript, introduces static typing to a traditionally dynamically-typed language, allowing developers to catch errors early in the development cycle and ensure that the types of data being passed around in their applications are consistent (Bierman et al., 2014).

Moreover, structured outputs and type-hints play a significant role in API (Application Programming Interface) design. APIs act as intermediaries, allowing different software systems to communicate and exchange data. Structured outputs like JSON or XML ensure that the data exchanged via APIs adheres to a predefined format, which is critical for the seamless integration of various software components (Fielding, 2000). Additionally, RESTful (Representational State Transfer) APIs, which are a common architectural style for networked applications, heavily rely on structured data formats for sending and receiving data across the web (Fielding Taylor, 2002).

In the context of application-layer software, these structured outputs and type-hints contribute significantly to software maintainability and scalability. By enforcing data structure and type consistency, developers can more easily understand, debug, and extend the software. This aspect is particularly important in large-scale projects or in enterprise environments where software systems need to be robust, reliable, and easy

to maintain (Gamma et al., 1994). The use of structured outputs and type-hints thus forms a foundational aspect of software engineering, enhancing code quality and facilitating the integration of complex software systems.

TYPE ENGINEERING

The recent wave of integration of Large Language Models (LLMs) into existing software systems represents a significant leap in the field of artificial intelligence and its application in software development. LLMs have shown extraordinary ability in generating human-like text, making them invaluable across various applications, from content generation to conversational agents (Brown et al., 2020). However, integrating these models into traditional software poses challenges, primarily due to the unstructured nature of their outputs. This lack of structure necessitates additional parsing and processing, which can be both resource-intensive and prone to errors.

To mitigate these challenges, we introduce the concept of "Type Engineering". This involves guiding LLMs to produce structured outputs, like JSON or TypeScript types, thereby streamlining their integration into traditional software systems. This method effectively combines the strengths of conventional programming with advanced AI capabilities. For example, by generating outputs in JSON format, LLMs can seamlessly integrate their data within existing software frameworks, significantly reducing the complexity and overhead associated with integrating AI-generated content (Huang et al., 2020).

An essential advancement in Type Engineering is the use of "function calling" as a means to direct LLMs towards producing structured outputs. Function calling is a method where the model is given a set of function parameters to generate responses in a structured format, akin to the return values of a function call in programming.

Through function calling, Type Engineering involves the use of type-hints and schemas to guide the generation of structured outputs. By defining clear types and structures, developers can prompt the LLM to produce outputs that are immediately usable within the application's logic. This not only simplifies the integration process but also minimizes the potential for errors due to misinterpretation or parsing errors of unstructured text. Robust systems such as TypeScript and Pydantic are ideal for defining these types and schemas, further enhancing the utility of LLMs in software applications (Lambert, 2020).

The integration of LLMs into application-layer software through structured outputs, particularly via function calling, opens up possibilities for dynamic and interactive user interfaces. Enabling LLMs to "plug into" existing systems allows developers to create applications that leverage the advanced natural language processing capabilities of LLMs while maintaining the structural and reliability standards of traditional software. Such integration heralds a new era in software interaction, making applications more intuitive and responsive to human language (LeCun et al., 2015).

While there other methods of LLM structured output, such as "guided generations" using grammars or regex, they are beyond the scope of this paper. The focus here is on the efficacy of function calling and Type Engineering in producing structured outputs that are both efficient and backwards compatible with existing software paradigms.

STRUCTURED LLM OUTPUTS IN PRACTICE

TypeScript types and Pydantic models, in particular, offer significant advantages in guiding LLM responses. TypeScript, an extension of JavaScript, provides static typing, which brings clarity and predictability to the data structures used in a program (Bierman et al., 2014). This clarity is vital when integrating LLM outputs, as it reduces ambiguity and ensures that the data conforms to the expected format. Similarly, Pydantic, a Python library, leverages Python type hints to validate and manage data structures (Lambert, 2020). By defining types and schemas, developers can guide LLMs to produce outputs that are not only structured but also conform to the specific requirements of the application, thereby reducing the risk of errors and misinterpretations.

Additionally, developer experience is greatly enhanced by the use of structured outputs. Tools like TypeScript and Pydantic offer features like autocomplete and syntax highlighting in Integrated Development Environments (IDEs), which significantly improve and accelerate the coding experience (Pierce, 2002). These features help developers write code more efficiently by providing real-time feedback and suggestions, reducing the cognitive load and the likelihood of errors. When dealing with complex LLM responses, these tools can be instrumental in managing and interpreting the data effectively.

Moreover, structured outputs simplify the debugging and maintenance of software that integrates LLMs. By having a well-defined structure, it becomes easier to trace issues and understand the flow of data within the application. This simplicity is particularly crucial in large-scale or enterprise-level applications, where the complexity and scale of the software make maintainability a key concern.

Using structured outputs to guide LLM responses also aligns with the best practices in software engineering of more modular and decoupled architectures. By providing a clear interface between the LLM and the rest of the application, structured outputs facilitate the development of modular components that can be independently developed, tested, and deployed. This modularity is key to building resilient and adaptable software systems that can rapidly evolve in response to changing requirements or technological advancements (Fielding & Taylor, 2002).

DYNAMIC USER INTERFACES

The transformative potential of integrating structured outputs from LLMs into application-layer software primarily manifests in dynamic user interfaces (UIs). This approach transcends traditional static data presentation, enabling interfaces that are interactive, responsive, and tailored to individual user experiences. By utilizing structured formats like JSON or TypeScript types, LLMs can output a predictable and standardized data format that can be intricately mapped to UI components, fostering a more intuitive and engaging interaction for users (Brown et al., 2020).

Dynamic UIs significantly benefit from the structured data provided by LLMs in personalizing user experiences. In sectors like e-commerce, LLMs can analyze user interactions to generate bespoke product suggestions. The structured output ensures that these recommendations are not only accurately reflected in the UI but also allow for dynamic updates, enhancing user engagement significantly compared to static interfaces (Huang et al., 2020).

Another pivotal aspect is the role of structured outputs in enabling reasoning engines within UIs. These engines can interpret the data from LLMs, apply contextual logic or user preferences, and subsequently generate UI components that represent this refined information. For example, in financial applications, LLMs could analyze market data to provide investment advice, with the structured output then tailored by the reasoning engine to align with the user's specific investment profile (LeCun et al., 2015).

The predictability and consistency offered by structured outputs are crucial in maintaining the integrity and reliability of dynamic UIs. With predefined data formats, UI components can be designed to anticipate specific data types, thus minimizing errors and enhancing the user experience. This is especially critical in multifaceted applications, like data analytics dashboards, where the UI must handle diverse data types and structures efficiently and accurately (Gamma et al., 1994).

The integration of structured outputs from LLMs into dynamic UIs is synergistic with contemporary web development trends that prioritize modularity and reusability. A consistent structured data format permits the design of UI components that are not only reusable across various application segments but also across different applications. This approach accelerates development processes and ensures uniformity in user experience across diverse interfaces, embodying the principles of modern, modular web architecture (Fielding & Taylor, 2002).

CONCLUSION

In conclusion, this paper has demonstrated the significant potential of integrating Large Language Models (LLMs) with application-layer software through the use of structured outputs. By guiding LLMs to produce data in formats like JSON or TypeScript types, a bridge is created between the rich, natural language capabilities of LLMs and the structured, logical world of traditional programming. This "Type Engineering" approach not only simplifies the integration of LLMs into existing software systems but also enhances their utility and applicability. The predictability and manageability of structured outputs ensure that LLMs can be seamlessly incorporated into various software architectures, improving both developer experience and application performance. This integration paves the way for more intelligent, dynamic, and interactive user interfaces, thereby expanding the capabilities and reach of modern software applications.

Furthermore, the structured approach advocated in this paper is not just a technical enhancement; it represents a paradigm shift in how we perceive and interact with software systems. By enabling LLMs to communicate effectively with traditional programming structures, we open up new avenues for innovation in software development. The dynamic user interfaces that emerge from this integration are not only more engaging and responsive but also more aligned with the evolving needs and expectations of users. As we continue to explore and refine the integration of LLMs into application-layer software, we stand on the cusp of a new era in software development, where artificial intelligence and traditional programming converge to create more powerful, intuitive, and adaptable software solutions.

ACKNOWLEDGMENTS

We thank Professor Seongkook Heo and his lab for providing valuable input on this project. We are also grateful to YCombinator for funding us in the upcoming batch to continue pushing the limits of how we can weave together software and AI.

REFERENCES

[1] D. Crockford, "The application/json Media Type for JavaScript Object Notation (JSON)," 2006.

[2] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, "Extensible Markup Language (XML) 1.0," 2008.

[3] G. van Rossum, B. Warsaw, and N. Coghlan, "PEP 484 – Type Hints," 2021.

[4] G. Bierman, M. Abadi, and M. Torgersen, "Understanding TypeScript," Microsoft Research, 2014.

[5] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Ph.D. dissertation, University of California, Irvine, 2000.

[6] R. T. Fielding and R. N. Taylor, "Principled Design of the Modern Web Architecture," *ACM Transactions on Internet Technology (TOIT)*, vol. 2, no. 2, pp. 115–150, 2002.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, 1994.

[8] T. B. Brown et al., "Language Models are Few-Shot Learners," *arXiv preprint arXiv:2005.14165*, 2020.

[9] C. J. Huang, J. Qiu, and J. T. Huang, "Conversational Agents in Customer Service Applications," *arXiv preprint arXiv:2005.05635*, 2020.

[10] P. Rajpurkar, R. Jia, and P. Liang, "Know What You Don't Know: Unanswerable Questions for SQuAD," 2018.

[11] B. C. Pierce, *Types and Programming Languages*, MIT press, 2002.

[12] S. Lambert, "Pydantic: Data Validation and Settings Management Using Python Type Annotations," 2020.

[13] Y. LeCun, Y. Bengio, and G. Hinton, "Deep Learning," *Nature*, vol. 521, no. 7553, pp. 436–444, Nature Publishing Group, 2015.