Adapting LLMs for Structured Natural Language API Integration

Robin Chan[1,2] **Katsiaryna Mirylenka**[1] **Thomas Gschwind**[1]
Christoph Miksovic-Czasch[1] **Paolo Scotton**[1] **Enrico Toniato**[1] **Abdel Labbi**[1]

[1]IBM Research [2]ETH Zürich
robin.chan@inf.ethz.ch
{kmi, thg, cmi, psc, eto, abl}@zurich.ibm.com

Abstract

API integration is crucial for enterprise systems, as it enables seamless interaction between applications within workflows. However, the diversity and complexity of the API landscape present significant challenges in combining API calls based on user intent. Existing methods rely on named entity recognition (NER) and knowledge graphs, but struggle to generate more complex control flow structures, such as conditionals and loops. We propose a novel framework that leverages the success of large language models (LLMs) in code generation to integrate APIs based on natural language input. Our approach involves fine-tuning an LLM using automatically generated API flows derived from OpenAPI specifications. We further evaluate the effectiveness of enforcing the syntax and schema adherence through constrained decoding. To enable systematic comparison, we introduce targeted test suites to assess the generalization capabilities of these approaches and their ability to retain structured knowledge. Our findings show that LLMs fine-tuned on OpenAPI specifications can (a) learn structural API constraints implicitly during training, and (b) achieve significant improvements in both in-distribution and out-of-distribution performance over NER and retrieval-augmented generation (RAG)-based approaches.[1]

1 Introduction

The ability to integrate APIs of different software services is crucial for automating processes across applications. Industrial tools like IBM App Connect[2] or Zapier[3] provide visual interfaces for manual flow composition but they require users to possess API knowledge or tediously search through service catalogs. This motivates automatic flow generation from natural language descriptions.

GOFA (Brachman et al., 2022) demonstrates the feasibility of such solutions by implementing utterance-to-API generation with an NER-based approach. GOFA, however, struggles with variations in user utterances and limited support for complex flow control structures like conditionals and iterations, as they require more complex reasoning over the natural language query. Recent successes of large language models (LLMs) on related code generation tasks like text-to-SQL (Xie et al., 2022; Scholak et al., 2021; Giaquinto et al., 2023; Deng et al., 2022) encourage exploring their capabilities for this task. This requires the LLM to learn (a) mapping utterances to relevant APIs, (b) valid methods within those APIs, and (c) the syntactical constraints for composing API flows.

To this end, we propose a generic LLM-tuning approach where structured information is (a) implicitly learned through automatically generated samples and (b) optionally enforced at inference time with constrained decoding. Our contributions are summarized as follows:

1. We propose general synthetic data generation for learning API structure to implicitly adapt the LLM via fine-tuning and compare them to NER, prompt engineering, and RAG approaches.
2. We introduce problem-specific baselines to assess the in- and out-of-distribution generalization of the tuned LLM and compare them to baselines from previous work.
3. We demonstrate that in- and out-of-distribution generalization and structural reasoning can be improved by data augmentation and constrained decoding.
4. We implement a working system that translates natural language queries to API flows.[4]

[1]The code is public and available here: https://github.com/chanr0/api-integration

[2]https://ibm.com/cloud/app-connect

[3]https://zapier.com

[4]A video demonstration of the system can be found here: https://youtu.be/U0KNdnO92rk

Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track, pages 991–1000
November 12-16, 2024

2 Background

OpenAPI Specification. The OpenAPI Specification (OpenAPI Initiative, 2021), formerly known as Swagger, defines a language-agnostic way to describe interfaces for HTTP APIs. Many enterprise software systems provide an OpenAPI specification or document how users can interact with them through a REST-style API. OpenAPI defines how to specify API metadata, available endpoints, operations, parameters, and expected responses. API calls are uniquely identified by their *application*, *object*, and CRUD *operation*, where the former two define the endpoint to which the API call is made. We can therefore represent the available API calls for a set of applications as a forest of trees—one for each application. The application is stored at the root, and the available objects and operations are stored on the subsequent layers as shown in Figure 1. The forest stores all available application, object, and operation combinations along with relevant metadata.

API Flows. An API *flow* refers to a sequence of API calls across applications. These flows can be event-driven, where a *trigger* event initiates a set of *actions*, or they can involve chaining specific *actions* in response to a given *request*.

Decoding Algorithms. During text generation, standard decoding algorithms explore an exponentially large space of possible output strings. This computational complexity necessitates relying on heuristic decoding strategies without formal guarantees. Deterministic approaches like greedy search—selecting the most probable token at each step—prioritize efficiency, while beam search (Reddy, 1977; Sutskever et al., 2014) and its stochastic counterparts like top-k sampling (Wiher et al., 2022) aim for a balance between efficiency and generating diverse, natural outputs. This work focuses on beam search due to its simplicity and popularity. However, constrained decoding can be applied to any of the above decoding strategies.

Beam search employs a $k \in \mathbb{Z}_+$-pruned breadth-first search. At each decoding step, it only keeps the top k decoding paths based on the beams' cumulative probability. As such, it can be defined recursively (Meister et al., 2020). Namely, let $\boldsymbol{y}^{t-1}$ denote the previously generated sequence at some decoding timestep $t > 0$, y be the next candidate token in the language model vocabulary $\overline{\Sigma}$ which contains the end-of-string symbol EOSThen, beam search considers the candidate set

$$\mathcal{B}^t = \left\{ \boldsymbol{y}^{t-1} \circ y \mid y \in \overline{\Sigma} \wedge \boldsymbol{y}^{t-1} \in Y^{t-1} \right\}, \quad (1)$$

where for some LM p at each decoding step $t > 0$:

$$Y^t = \underset{Y' \subseteq \mathcal{B}^t, |Y'|=k}{\arg\max} \log p(Y'|x; \boldsymbol{\theta}), \quad (2)$$

and $Y^0 = \{\text{BOS}\}$, the set only containing the beginning-of-string symbol.

Constrained Decoding. During constrained decoding, the set of candidate tokens $\mathcal{B}^t$ is restricted to contain only continuation tokens adhering to a binary objective function $\mathcal{G} : \overline{\Sigma} \to \{0, 1\}$, i.e.,

$$\mathcal{B}^t = \Big\{ \boldsymbol{y}^{t-1} \circ y \mid y \in \overline{\Sigma} \wedge \boldsymbol{y}^{t-1} \in Y^{t-1} \wedge \mathcal{G}(\boldsymbol{y}^{t-1} \circ y) \Big\}. \quad (3)$$

This objective could represent a specific syntax or grammar that the generated sequence must adhere to. It is typically left-context dependent, meaning the constraint on the next token depends only on the previously generated sequence. In this work, we adopt incremental parsing for constrained generation, which has been shown to enhance performance in tasks such as text-to-SQL (Scholak et al., 2021; Poesia et al., 2022), and extend it to the text-to-API flow task.

3 Automatic Data Generation

Our goal is to automatically generate training data for LLM text-to-API flow fine-tuning, thereby aligning the model with API domain knowledge. To this end, we leverage the tree structure and node attributes of the API forest depicted in Figure 1.

Prompting with this large structured information is still difficult. Firstly, despite efforts to increase prompt length for modern LLMs (e.g., LongLLaMA; Tworkowski et al. 2024), maximum token limitations restrict the amount of structured information that can be passed during inference. Further, LLMs have been shown to perform worse for longer prompts due to the amount of irrelevant context (Shi et al., 2023). To this end, a common mitigation strategy is retrieving-augmented generation (RAG) (Khattab et al., 2022). However, as shown in section 5, RAG has limited impact on domain-specific tasks where semantic search over unseen concepts performs poorly. Therefore, we train models to learn structural knowledge implicitly through generated training samples. Generating samples manually is expensive and requires

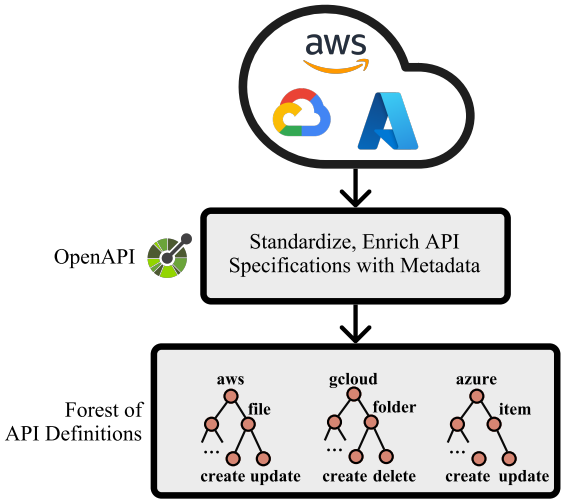

Figure 1: API definition ingestion and representation.

specific API flow expertise. We therefore propose a *synthetic* data generation approach to create a rich training set capturing diverse utterance-to-flow pairs. Namely, we first extract OpenAPI descriptions from the "description" and "responses" attributes of object methods (examples 1 & 2 in Figure 2). As these descriptions vary in quality, we additionally generate operation-specific templates filled with API details from the specification (examples 3 & 4 in Figure 2). This aims to generate a diverse set of API call descriptions covering potential user utterances.

Matching utterance intent to the appropriate API call requires considering the limited set of request methods. The same method can have different meanings depending on the endpoint it calls. For instance, the `CREATE` method on a Gmail message object sends an email, while on a Salesforce account, it creates the account. Matching intent might require more than simple semantic parsing and should incorporate API descriptions.

```
{
 info: ...
 base_path: /gmail/
 paths:{
  /mail:{
    get: {
      description: ...
      summary: ...
      operationId: create_Mail
          }
    ...
}
```

gmail/mail/create
↓
"Sends an email message from Gmail"
"Sends email"
"Create a gmail mail."
"Create a Google Mail mail."

Figure 2: Synthetic utterance generation from the OpenAPI specification.

We use the following categories for generating training samples, representing the building blocks of API flows:

- **ID1**: Single API call.
- **ID2**: Trigger followed by an action.
- **ID3**: Trigger followed by two actions.
- **ID4**: `if`-conditional (e.g., conditional branching based on a stated condition).
- **ID5**: `for`-loop (e.g., iteratively calling an action on retrieved items).
- **ID6**: Sync/Move/Copy operation (combinations of the above).

We provide examples for each category in Table 2. To mitigate spurious correlations and improve out-of-distribution generalization, we augment data with paraphrasing—a common technique used to increase data variability (McCoy et al., 2019; Feng et al., 2021; Chan et al., 2023). We paraphrase the synthetically generated samples using few-shot prompting. Finally, we filter out samples where application or object names are lost during paraphrasing using partial string matching.

4 Enforcing Valid API Flows

Event-driven API integrations can be represented by a constrained subset of Python code with API calls expressed as `app.object.operation` triplets. The precise grammar, which also directly encodes the set of available APIs, is specified in EBNF-like notation. For completeness, the grammar is shown in Figure 3. Note, that the schema is encoded in the set of terminals named `actions` and `triggers`, resulting in a rather large grammar. During incremental parsing, such terminals are split up to match the current generation with valid matching suffixes at each decoding step.

To enforce adherence to this structured knowledge and syntax, we employ constrained semantic decoding (Poesia et al., 2022) at inference time, as even tuned models can deviate from valid schemas, especially in ambiguous scenarios.

We adopt a faster implementation of constrained semantic decoding for beam search. Instead of building prefix trees to find all valid continuations at each step, we directly check whether the continuation token is valid on the most likely tokens until we find k valid continuations. Since beam search

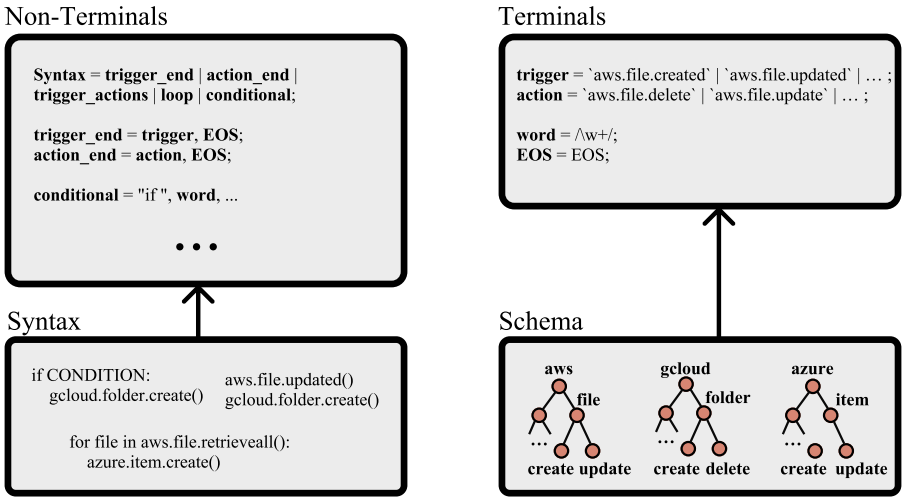

Figure 3: Syntax and schema ingestion into the grammar. The full grammar is listed in Appendix A.

samples generate at most k continuations per beam, we only need to keep the k most likely valid tokens per beam, i.e., its *beam width*. This significantly reduces the number of expensive `isValidPrefix` calls. The procedure is shown in Algorithm 1.

Algorithm 1 Fast Constrained Decoding

Require: Prefix p, grammar $\mathcal{G}$, tuned model $p_{\boldsymbol{\theta}}$, beam width k
Ensure: Masked next token scores

$l \leftarrow$ `sortProb(getNextTokenProb`$(p, p_{\boldsymbol{\theta}})$)
$ct, validTokens \leftarrow 0, []$
for tok in l **do**
  while $ct < k$ **do**
    if `isValidPrefix`$(p \cdot tok, \mathcal{G})$ **then**
      `append`$(validTokens, tok)$
      $ct \leftarrow ct + 1$
    end if
  end while
end for
return `maskInvalid`$(l, validTokens)$

5 Evaluation

We evaluate our approach by training several LLMs and comparing them to an NER baseline. All models are trained on the same APIs with 85 applications, 4'557 application-specific objects, and 21'712 unique API calls. As each application has an arbitrarily large number of objects/endpoints and each endpoint may support a different subset of the actions and triggers, the resulting trees are much differently sized. This is shown in Figure 4.

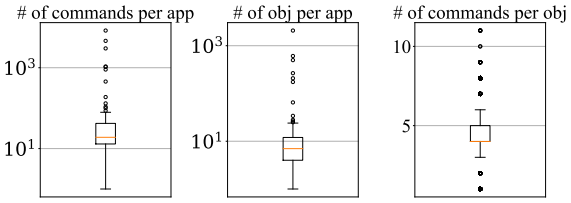

Figure 4: API tree node distribution statistics. The two plots on the left use a logarithmic scale.

The data is split into a 55k/7k/7k train/eval/test split, with the test set subsampled for equal distribution among in-distribution (ID) categories. We also define an out-of-distribution (OOD) test suite described in subsection 5.1.

LLM Baselines and Fine-tuning. We implement multiple approaches to assess their effectiveness in the text-to-API flow task: (a) an NER-based method serves as a baseline and is similar to GOFA (Brachman et al., 2022), (b) we prefix-tune (Li and Liang, 2021) T5-3B (Raffel et al., 2020) and BLOOM-3B (BigScience Workshop, 2023) at 0.35% of their parameters, (c) BLOOM-3B and LLaMA-13B (Touvron et al., 2023) are fully fine-tuned on a mixture of ShareGPT[5] data and our training samples, (d) conversational LLaMA-13B; a version of LLaMA-13B that is further fine-tuned on a dataset with instructions, enabling conversational prompting during testing. This may simulate a conversation scenario, where the initial utterance requires clarifications on the application to use as multiple solutions are possible.

[5] https://sharegpt.com/

5.1 Out-of-Distribution Test Suite

We define a set of out-of-distribution sample classes representing user input scenarios that may not be included in the training set, allowing us to evaluate the model's generalization ability:

- **OOD1**: Commonly known app name variations (e.g., "s3" instead of "Amazon Simple Storage Service").
- **OOD2**: Omitting application from utterance if clear from object and action (cf. Figure 5).
- **OOD3**: Omitting object from utterance if clear from the application (cf. Figure 5).
- **OOD4**: A flow containing a trigger followed by more than two actions.
- **OOD5**: User-collected full integration flows with potentially intricate reasoning.

OOD1 samples consist of a single API call and evaluate how well the model deals with references to domain-specific knowledge, for example, referencing *Amazon Simple Storage Service* as *s3*. Samples in OOD2 and OOD3, like OOD1, consist of a single API call and evaluate whether the model is able to use structural knowledge to make conclusions about implicit information in the utterance (Figure 5, top and bottom, respectively). Samples in OOD4 evaluate whether the model identifies and generalizes to the syntactic constraints of the grammar. Finally, the samples in OOD5 are a set of full human-generated integration flows with human annotation, which at times require significantly more intricate reasoning than what can be taken from the utterance, often significantly exceeding the training set coverage. We provide examples for each category in Table 3.

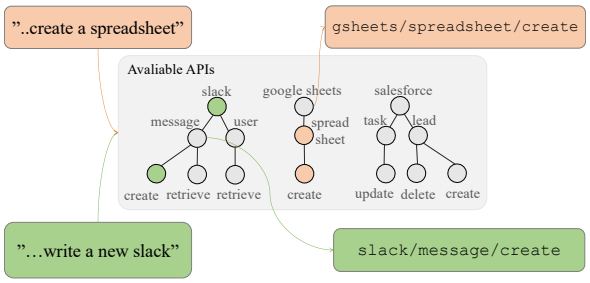

Figure 5: Using structural information to infer implicit knowledge in the utterance.

5.2 Metrics

We introduce the following metrics to evaluate the API generations:

- **Exact matching (EM)**: Checks if the generated string exactly matches the ground truth (except for irrelevant variable names).
- **Similarity Ratio (Sim)**: Token-based similarity between target and generated string (1 minus token-based edit distance).
- **Triplet Precision (TP)**: Fraction of generated API calls that exist in the API definitions.

5.3 Results and Discussion

NER-Based Baseline: Explicit Matching. The NER-based approach only considers a subset of the test set due to limitations in handling iterations and conditionals (Table 1). It performs well for individual API calls or simple trigger-action flows but struggles with more complex scenarios.

We see that the NER-based approach shows decent performance for extracting individual API calls from utterances, dealing well with common knowledge app aliases, as such are likely to be part of the NER train corpus. Triplet precision is high (and comparable to tuned LLMs), as candidates are mostly successfully matched to a knowledge graph of existing API calls. However, the NER-based model struggles with composite flows, where paraphrasing may yield a range of formulation variations, where entities cannot be extracted from the text.

Effects of Constrained Decoding. Table 1 shows that LLMs outperform the baseline in both in-distribution and out-of-distribution settings, with BLOOM generally outperforming T5 for smaller, prefix-tuned models. Constrained decoding (CD) significantly improves triplet precision for all models, especially for ambiguous samples (reflected in higher out-of-distribution accuracy). While CD enforces syntactic validity, the underlying model still influences semantic correctness. The increase in triplet precision with CD is not met with a proportional increase in Exact matching accuracy, suggesting errors beyond API call matching. Additionally, CD can lead to lower target similarity as the model might prefer generating existing (but slightly incorrect) API calls.

Model	In-Distribution Metrics			Out-of-Distribution Metrics		
	EM ↑	Similarity ↑	TP ↑	EM ↑	Similarity ↑	TP ↑
NER-based	n/a (8.7)	n/a (40.0)	n/a (79.4)	n/a (23.6)	n/a (56.4)	n/a (86.8)
Prefix-tuned T5-3B	58.9 (53.6)	92.3 (91.9)	79.4 (76.3)	22.9 (23.2)	76.6 (76.6)	66.6 (67.1)
Prefix-tuned BLOOM-3B	60.9 (59.0)	93.7 (91.9)	78.2 (77.1)	25.0 (25.4)	73.7 (73.5)	68.6 (68.7)
Prefix-tuned BLOOM-3B + CD	64.7 (66.2)	90.8 (90.0)	**100.0** (**100.0**)	32.6 (32.4)	72.0 (71.6)	**100.0** (**100.0**)
Fully tuned BLOOM-3B	83.6 (77.4)	97.6 (96.6)	88.8 (85.6)	34.4 (34.2)	77.1 (76.2)	81.2 (81.5)
Fully tuned BLOOM-3B + CD	85.2 (79.2)	97.3 (96.1)	**100.0** (**100.0**)	40.6 (40.5)	78.2 (77.9)	**100.0** (**100.0**)
Prompting fully tuned LLaMA-13B	44.3 (84.0)	68.3 (94.7)	78.4 (85.1)	26.7 (27.1)	66.5 (67.2)	75.3 (75.7)
LLaMA-13B + RAG	46.6 (46.7)	67.4 (71.5)	88.8 (86.2)	27.1 (27.5)	64.1 (64.6)	96.6 (96.5)
Fully tuned LLaMA-13B	92.1 (87.4)	98.2 (97.2)	89.5 (86.3)	44.8 (44.7)	80.8 (80.6)	78.9 (78.6)
Conversational LLaMA-13B	**92.5** (**88.6**)	**98.4** (**97.7**)	89.6 (86.6)	**57.6** (**57.7**)	**87.5** (**87.4**)	91.7 (91.6)

Table 1: Unweighted average performance metrics. Results in parenthesis refer to the NER-applicable subset.

Retaining Syntax and Schema. We observe that already the prefix-tuned bloom model shows some success in predicting an API component missing from the utterance if it is deducible from the list of available APIs (OOD2, OOD3), especially if decoding is constrained. However, the accuracy in case of a missing application is much lower. One can argue that this may be attributed to beam sampling during left-to-right decoding, as the uncertainty in the choice of application may result in the correct beam being removed in the early stages of decoding.

Retrieval-Augmented Prompting. For further comparison, we finally implement a RAG-based approach that leverages the API knowledge graph directly. To achieve this, we encode all potential API calls—comprising application, object, and operation triplets, along with their descriptions—using pre-trained LLM embeddings for semantic search. We employ ChromaDB for the semantic search, retrieving the top 5 most relevant API call paths based on the encoded utterance. These retrieve paths are then provided as potential solutions within a simple prompt for the LLM.

For the embeddings, we evaluate a fine-tuned LLaMA-13B model and the sentence transformer model all-MiniLM-L6-v2 (Reimers et al., 2023). The sentence transformer model retrieve the correct API call within the top 5 candidates for 54% of the utterances, significantly higher than the 20% achieved using LLaMA-based embeddings. The results for the retrieval-augmented LLaMA-13B with sentence transformer for semantic indexing are presented in Table 1. While the retrieval-augmented approach exhibit slightly better performance than simple prompting for in-distribution cases, it was still outperformed by the fully fine-tuned LLaMA model and models with constrained decoding.

Potential of Multi-Turn-Prompting LLaMA. The LLaMA model trained on a combined dataset of our task-specific data and instructional data achieve the overall best performance. While LLaMA exhibits similar tendencies to other unconstrained models, where it occasionally generates non-existent API calls, its conversational capabilities enable interactive corrections. We simulate a user correcting an ambiguous utterance by re-prompting the model with the intended component after an initial prediction. While not directly comparable to other approaches due to its interactive nature, multi-turn prompting with LLaMA yields significant performance improvements when dealing with ambiguous user requests.

Given the good performance of the fully fine-tuned LLaMA-13B model, both with and without the conversational simulation, we opted to forgo applying constrained decoding in this case, even though it may further improve schematic and syntactic adherance. Constrained decoding would introduce additional computational overhead, and the model already achieved satisfactory results without it. As future work, we plan to investigate the possibility of developing efficient constrained decoding techniques specifically suited for interactive API generation with LLaMA-like models.

6 Related Work

Database and knowledge graph querying are well-known NLP problems, often addressed through techniques such as NER, relation extraction, and query generation. These methods typically involve producing graph queries from natural language utterances and executing them against graph databases (Liang et al., 2021; Copestake and Jones, 1990; Krivosheev et al., 2021; Brachman et al., 2022; Krivosheev et al., 2023). Modern database

interfaces also employ LLMs for converting user queries into SQL (Toniato et al., 2023).

Web API search using deep learning models has been explored by Liu et al. (2020). This work proposes synthetic dataset generation and leverages deep learning for API integration. However, the approach relies on substring detection routines, which can be less flexible than the LLM-based adaptation presented here. Further, Gorilla (Patil et al., 2023) is an LLM trained to access APIs for interacting with ML models on platforms like Torch Hub, TensorFlow Hub, and HuggingFace. It excels at interacting with individual models but is not specifically tuned creating flows between APIs.

7 Limitations

A limitation of the proposed approach is that this work focuses solely on generating flows of API call triplets. As such, the trained models do not generate arguments for the API calls (cf. Appendix B). However, we note that including arguments in a written utterance is practically rather tedious and a semi-supervised approach may be better suited to address this need.

8 Conclusion

This work presents a novel approach to natural language-driven large-scale API integration using LLMs. We demonstrate that models trained with our approach exhibit strong generalization capabilities, both in-distribution and out-of-distribution. This is evident in their ability to: (a) **handle ambiguity** by leveraging structural knowledge to make informed decisions when user intent is unclear; (b) **learn domain knowledge** by adapting to domain-specific phrasing and terminology encountered during training; and (c) **generate unseen flow structures** by utilizing the capability of general-purpose LLMs, particularly LLaMA, to create novel API flow compositions that adhere to implicit syntactic constraints. These findings highlight the potential of LLMs to streamline API integration tasks.

Broader Impact

This paper presents research about generating API calls from natural language utterances. To the best of our knowledge, there are no ethical or negative societal implications to this work.

Acknowledgements

The authors thank the anonymous reviewers for their helpful and productive feedback.

References

BigScience Workshop. 2023. Bloom: A 176b-parameter open-access multilingual language model. *Preprint*, arXiv:2211.05100.

Michelle Brachman, Christopher Bygrave, Tathagata Chakraborti, Arunima Chaudhary, Zhining Ding, Casey Dugan, David Gros, Thomas Gschwind, James Johnson, Jim Laredo, Christoph Miksovic, Qian Pan, Priyanshu Rai, Ramkumar Ramalingam, Paolo Scotton, Nagarjuna Surabathina, and Kartik Talamadupula. 2022. A goal-driven natural language interface for creating application integration workflows. *AAAI*, 36(11):13155–13157.

Robin Chan, Afra Amini, and Mennatallah El-Assady. 2023. Which spurious correlations impact reasoning in nli models? a visual interactive diagnosis through data-constrained counterfactuals. In *The 61st Annual Meeting Of The Association For Computational Linguistics*.

Ann Copestake and Karen Sparck Jones. 1990. Natural language interfaces to databases. *The Knowledge Engineering Review*, 5(4):225–249.

Naihao Deng, Yulong Chen, and Yue Zhang. 2022. Recent advances in text-to-SQL: A survey of what we have and what we expect. In *Proceedings of the 29th International Conference on Computational Linguistics*, pages 2166–2187, Gyeongju, Republic of Korea. International Committee on Computational Linguistics.

Steven Y. Feng, Varun Gangal, Jason Wei, Sarath Chandar, Soroush Vosoughi, Teruko Mitamura, and Eduard Hovy. 2021. A survey of data augmentation approaches for NLP. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pages 968–988.

Robert Giaquinto, Dejiao Zhang, Benjamin Kleiner, Yang Li, Ming Tan, Parminder Bhatia, Ramesh Nallapati, and Xiaofei Ma. 2023. Multitask pretraining with structured knowledge for text-to-SQL generation. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 11067–11083, Toronto, Canada. Association for Computational Linguistics.

Omar Khattab, Keshav Santhanam, Xiang Lisa Li, David Hall, Percy Liang, Christopher Potts, and Matei Zaharia. 2022. Demonstrate-search-predict: Composing retrieval and language models for knowledge-intensive nlp. *arXiv preprint arXiv:2212.14024*.

Evgeny Krivosheev, Mattia Atzeni, Katsiaryna Mirylenka, Paolo Scotton, Christoph Miksovic, and Anton Zorin. 2021. Business entity matching with siamese graph convolutional networks. In *AAAI Conference on Artificial Intelligence*.

Evgeny Krivosheev, Katsiaryna Mirylenka, Mattia Atzeni, and Paolo Scotton. 2023. Graph neural networks for entity matching. In *2023 IEEE International Conference on Big Data (BigData)*, pages 6212–6214. IEEE Computer Society.

Xiang Lisa Li and Percy Liang. 2021. Prefix-tuning: Optimizing continuous prompts for generation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 4582–4597.

Shiqi Liang, Kurt Stockinger, Tarcisio Mendes de Farias, Maria Anisimova, and Manuel Gil. 2021. Querying knowledge graphs in natural language. *Journal of big data*, 8:1–23.

Lei Liu, Mehdi Bahrami, Junhee Park, and Wei-Peng Chen. 2020. Web api search: Discover web api and its endpoint with natural language queries. In *Web Services – ICWS 2020*, pages 96–113.

Tom McCoy, Ellie Pavlick, and Tal Linzen. 2019. Right for the wrong reasons: Diagnosing syntactic heuristics in natural language inference. In *ACL'19*.

Clara Meister, Ryan Cotterell, and Tim Vieira. 2020. If beam search is the answer, what was the question? In *EMNLP'20*, pages 2173–2185, Online. Association for Computational Linguistics.

OpenAPI Initiative. 2021. Openapi specification.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. 2023. Gorilla: Large language model connected with massive apis. *arXiv preprint arXiv:2305.15334*.

Gabriel Poesia, Alex Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Chris Meek, and Sumit Gulwani. 2022. Synchromesh: Reliable code generation from pre-trained language models. In *ICLR 2022*.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research*, 21(140):1–67.

Raj Reddy. 1977. Speech understanding systems: A summary of results of the five-year research effort at carnegie mellon university.

Nils Reimers, Omar Espejel, and Pedro Cuenca. 2023. Sentence transformer. https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2. Accessed: 2023-08-01.

Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 2021. PICARD: Parsing incrementally for constrained auto-regressive decoding from language models. In *EMNLP'21*, pages 9895–9901.

Freda Shi, Xinyun Chen, Kanishka Misra, Nathan Scales, David Dohan, Ed Chi, Nathanael Schärli, and Denny Zhou. 2023. Large language models can be easily distracted by irrelevant context. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org.

Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to sequence learning with neural networks. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*. Curran Associates, Inc.

Enrico Toniato, Abdel Labbi, Katsiaryna Mirylenka, Christoph Miksovic Czasch, Thomas Gschwind, Paolo Scotton, Francesco Fusco, and Diego Antognini. 2023. Flowpilot: An llm-powered system for enterprise data integration. In *Annual Conference on Neural Information Processing Systems*.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Szymon Tworkowski, Konrad Staniszewski, Mikołaj Pacek, Yuhuai Wu, Henryk Michalewski, and Piotr Miłoś. 2024. Focused transformer: Contrastive training for context scaling. *Advances in Neural Information Processing Systems*, 36.

Gian Wiher, Clara Meister, and Ryan Cotterell. 2022. On Decoding Strategies for Neural Text Generators. *Transactions of the Association for Computational Linguistics*, 10:997–1012.

Tianbao Xie, Chen Henry Wu, Peng Shi, Ruiqi Zhong, Torsten Scholak, Michihiro Yasunaga, Chien-Sheng Wu, Ming Zhong, Pengcheng Yin, Sida I. Wang, Victor Zhong, Bailin Wang, Chengzu Li, Connor Boyle, Ansong Ni, Ziyu Yao, Dragomir Radev, Caiming Xiong, Lingpeng Kong, Rui Zhang, Noah A. Smith, Luke Zettlemoyer, and Tao Yu. 2022. UnifiedSKG: Unifying and multi-tasking structured knowledge grounding with text-to-text language models. In *EMNLP'22*, pages 602–631.

A Grammar

For completeness, we show the complete EBNF grammar used for constrained semantic decoding.

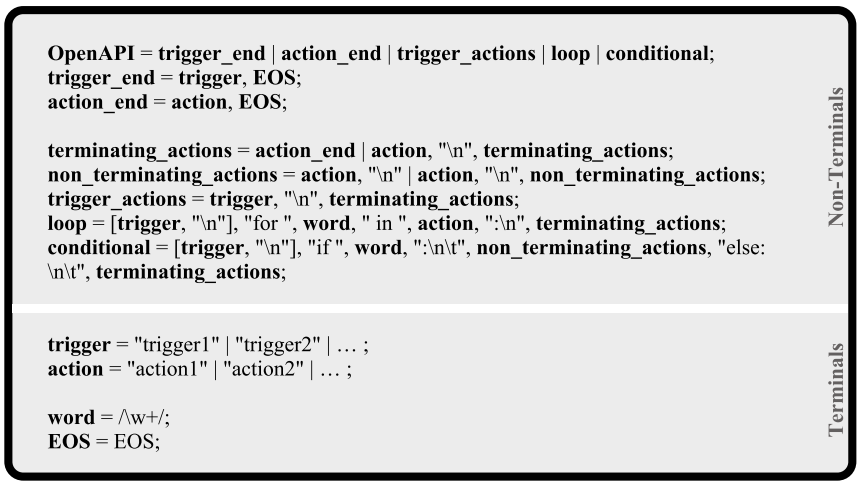

Figure 6: Full EBNF of the used API integration grammar.

B Data Generation

We provide a set of examples for each of the described integration flow types.

Description	Example Command	Example Utterance
Single API Calls	`slack.message.CREATE`, or `servicenow.lead.UPDATED`	*Send a message in slack*, or *Triggers a servicenow lead is deleted.*
Trigger + Action	`slack.message.CREATED` `box.folder.UPDATE`	*When a message is sent in slack, update a box folder.*
Trigger + Action + Action	`slack.message.CREATED` `box.folder.UPDATE` `salesforce.Note.UPDATE`	*When a message is sent in Slack, update both box folders and salesforce notes.*
`if`-Condition	`yammer.Message.CREATED if CONDITION: slack.message.CREATE else: salesforce.Note.UPDATE`	*For a new message in yammer, if CONDITION, forward it in slack, else update the corresponding salesforce notes.*
`for` Loop	`for var in slack.User.RETRIEVEALL: trello.Member.CREATE`	*Create a trello membership for every slack user*
Sync / Move / Copy	`for comment in trello.Comment.RETRIEVEALL: confluence.Comment.CREATE trello.Comment.DELETEALL`	*Move all trello comments to confluence.*

Table 2: Examples for the in-distribution sample categories.

Description	Example Command	Example Utterance
Referencing common knowledge app aliases	`ciscospark.groups.DELETEALL`	*Remove all Webex groups.*
Leave out app from utterance, if it should be clear from the object and action.	`gsheet.spreadsheet.CREATE`	*Create a spreadsheet.*
Leave out object from utterance, if it should be clear from the app.	`slack.message.CREATE`	*Write a slack.*

Trigger + Action + Action + Action	`slack.message.CREATED`
	`box.folder.UPDATE`
	`salesforce.Note.UPDATE`
	`maximo.message.UPDATEALL`
		When a message is sent in Slack, update both box folders and salesforce notes and update all Maximo messages.

Mix of user-Generated Flows	`slack.RawMessage.CREATED` `mailchimp.Members.CREATE`	*Add a Mailchimp subscriber from a Slack slash command.*

Table 3: Examples for the out-of-distribution sample categories.