# Multiple Schema-Conformant Declarative Code Generation

Mehant Kammakomati
*IBM Research*
Bengaluru, India
mehant.kammakomati2@ibm.com

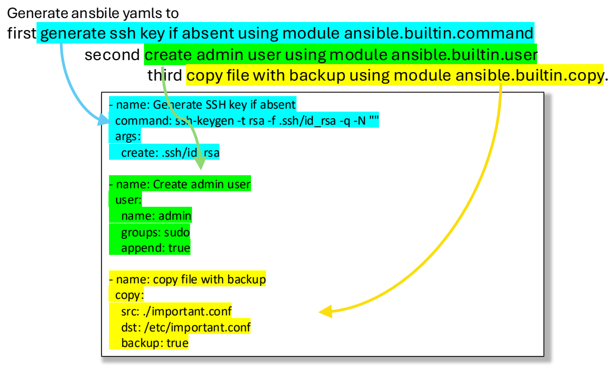
Srikanth G. Tamilselvam
*IBM Research*
Bengaluru, India
srikanth.tamilselvam@in.ibm.com

***Abstract*—Many enterprise systems including large-scale deployment platforms like Ansible provide a declarative user interface through programming languages like JavaScript Object Notation (JSON). These systems maintain integrity through validation rules, typically enforced via JSON schemas. However, enterprise tasks in these systems are often complex, involving multiple schemas, which makes it challenging for the developers to select the appropriate ones and write schema-compliant code snippets for each task. Recently, Large Language Models (LLMs) have shown promising performance for many declarative code generation tasks when adopted with constrained generation using a pre-known schema. However, to cater to real-world enterprise tasks, each task often requiring multiple code snippets to generate while ensuring compliance with their respective schemas, we introduce a novel framework that allows LLMs to generate multiple code snippets while choosing an appropriate schema for each of the snippets for constrained generation. To the best of our knowledge, we are the first to study this crucial enterprise problem for declarative systems and preliminary results on two real-world use cases demonstrate substantial improvements in both syntactic and semantic task performance. These findings highlight the potential of the approach to enhance the reliability and scalability of LLMs in declarative enterprise systems, indicating a promising direction for future research and development.**

***Index Terms*—code generation, constrained generation, LLMs**

## I. Introduction

JavaScript Object Notation (JSON) and YAML Ain't Markup Language (YAML) are widely used programming languages to enable declarative user interfaces that are adopted by many modern enterprise systems for seamless large-scale operations. Some examples of such systems are Ansible [1] and Kubernetes [3]. Each of the code snippets written is validated against a set of rules as supported by the enterprise system. These rules are often articulated as schemas in languages such as JSON Schema [2]. For instance, deploying an Ansible task requires the developer to write a schema-compliant YAML code snippet. Enterprise systems often support a large number of schemas, for example, *ansible.builtin* collection has over 100+ variety of modules, each having a respective schema. Furthermore, an enterprise can have its own set of custom modules and schemas for internal use cases. Writing multiple code snippets for a task involving such a variety of schemas to choose from and comply with is tedious work for developers.


Generate ansbile yamls to
first generate ssh key if absent using module ansible.builtin.command
second create admin user using module ansible.builtin.user
third copy file with backup using module ansible.builtin.copy.

```
- name: Generate SSH key if absent
  command: ssh-keygen -t rsa -f .ssh/id_rsa -q -N ""
  args:
    create: .ssh/id_rsa

- name: Create admin user
  user:
    name: admin
    groups: sudo
    append: true

- name: copy file with backup
  copy:
    src: ./important.conf
    dst: /etc/important.conf
    backup: true
```

Fig. 1. Illustration of an enterprise scenario where answering the prompt requires generating three schema compliant code snippets.

To mitigate this problem, Large Language Models (LLMs) have been adopted [5]–[8] in multiple code generation tasks. There is a growing body of work in enabling LLMs with constrained generation techniques [13], [17] to generate schema-compliant declarative code in languages such as JSON and YAML. However, current works in both industry [12] and research primarily focus on generating a single code snippet compliant with a single pre-known schema. This is insufficient for the usage of LLMs in enterprises, where individual tasks often involve choosing from multiple schemas and generating multiple code snippets. For instance, as shown in Fig. 1, a user may provide a prompt to accomplish 3 tasks through Ansible, such as *Generate ansible yamls to first generate ssh key if absent using module ansible.builtin.command second create admin user using module ansible.builtin.user third copy file with backup using module ansible.builtin.copy*, In response, the LLM is expected to generate 3 code snippets, one for each task involving command, user, and copy modules, while requiring all the code snippets to be compliant with their respective schemas.

To the best of our knowledge, this is the first work to study and systematically address this important problem. We introduce a novel framework (see Fig. II) comprising three key modules—*Speculator*, *Backtracker*, and *Constrained Decoder (CD)*—that together enable large language models to generate multiple schema-compliant code snippets (see section II).

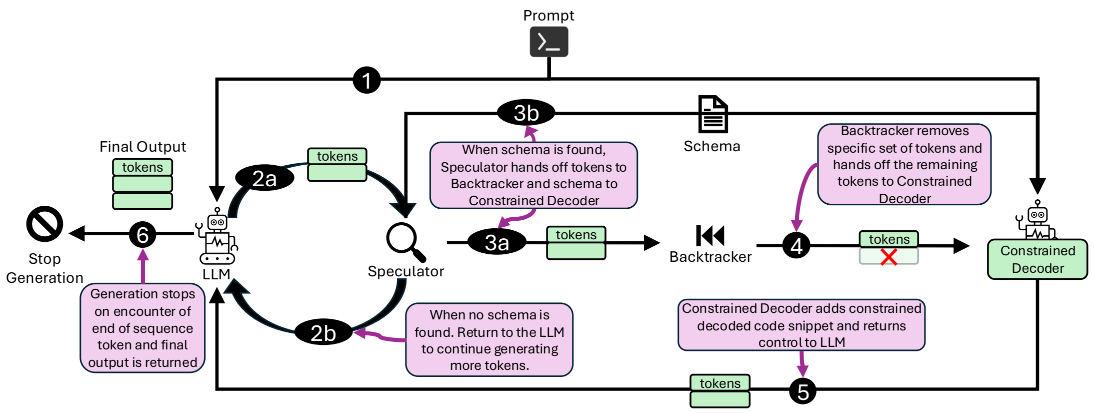

Prompt
1
3b
Schema
Final Output
tokens
2a
tokens
When schema is found, Speculator hands off tokens to Backtracker and schema to Constrained Decoder
Backtracker removes specific set of tokens and hands off the remaining tokens to Constrained Decoder
Stop Generation
6
LLM
Speculator
3a
tokens
Backtracker
4
tokens
Constrained Decoder
Generation stops on encounter of end of sequence token and final output is returned
2b
When no schema is found. Return to the LLM to continue generating more tokens.
Constrained Decoder adds constrained decoded code snippet and returns control to LLM
tokens
5

Fig. 2. Overview of our framework.

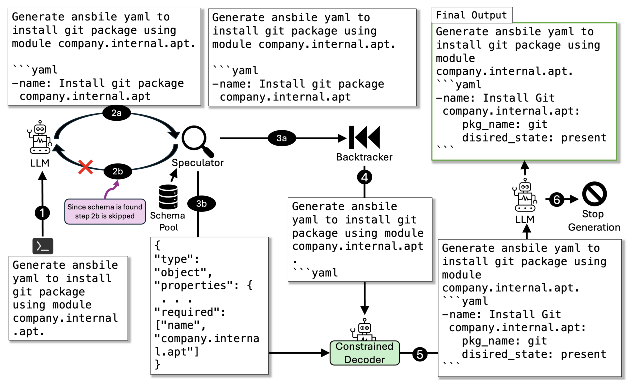
## II. Framework

### A. Speculator

*1) Functionality:* Speculator aims to find the most relevant schema that can be used for constrained generation of the current code snippet that the LLM is about to generate. It accomplishes this by leveraging partially generated response tokens from the LLM to search for a relevant schema from the pool. If no schema is found, the control is returned to the LLM to generate an additional set of tokens. This process continues to loop (steps $2a$ and $2b$ in Fig. II) between the Speculator and the LLM until there are enough response tokens to aid the Speculator in finding one potential schema candidate. The optimal number of tokens to generate before invoking the Speculator module to reduce the number of Speculator-LLM iterations depends on the use case. We provide these details in II-A2 and II-A3 for the 2 use cases (details in III-A) studied.

*2) Speculator for Ansible:*

*a) Schema:* As shown in Fig. 3, Ansible YAMLs are categorized by the module name. For instance, YAML for the prompt *Generate ansible yaml to install git package using module company.internal.apt* uses the module *apt*. Each module is accompanied by a schema.


~~~
Generate ansbile yaml to
install git package using
module company.internal.apt.

```yaml
-name: Install git package
 company.internal.apt
~~~

~~~
Generate ansbile yaml to
install git package using
module company.internal.apt.

```yaml
-name: Install git package
 company.internal.apt
~~~

~~~
Final Output
Generate ansbile yaml to
install git package using
module
company.internal.apt.
```yaml
-name: Install Git
 company.internal.apt:
   pkg_name: git
   disired_state: present
~~~
```

```
Generate ansbile
yaml to install
git package
using module
company.internal
.apt.
```

```
{
"type":
"object",
"properties": {
 . . .
"required":
["name",
"company.interna
l.apt"]
}
```

```
Generate ansbile
yaml to install git
package using module
company.internal.apt
.
```yaml
```

~~~
Generate ansbile yaml to
install git package using
module
company.internal.apt.
```yaml
-name: Install Git
 company.internal.apt:
   pkg_name: git
   disired_state: present
```
~~~

Fig. 3. An example flow of our framework involving generation of Ansible code snippet for apt module.

*b) Search heuristic:* Since the module determines the schema, we use the module present in the response tokens to retrieve the corresponding schema from the pool.

*c) Optimal tokens:* The LLM is allowed to generate as many tokens as needed until the start of the code block (identified as ```). From that point, only $x$ number of tokens are generated before invoking the Speculator. To identify the optimal value for $x$, all the modules having a schema in the pool are tokenized using the LLM, and $x$ is set to the average number of tokens. Intuitively, the module (search term) is often generated after the code block tokens, and $x$ represents the average tokens needed to represent a module name.

*3) Speculator for application configuration:*

*a) Schema:* For each application module, keys in the configuration (config) YAML are accompanied by constraints articulated in a schema. For instance, the value for *per_device_train_batch_size* in the Trainer module config should be a positive integer.

*b) Search heuristic:* Key names used in the YAML config determine which module it belongs to. The tokens generated after the code block token are used to do a fuzzy search over all the keys present in the schema pool. The intuition behind fuzzy search is that the generated tokens can be incomplete and may not exactly match a key name from the schemas.

*c) Optimal tokens:* Similar to the Ansible use case, the optimal value for $x$ is computed as the average number of tokens of all the keys in the pool.

### B. Backtracker

*1) Functionality:* The Backtracker supports the framework by removing the partially generated code snippet (step $3a$ in Fig. II), which previously served the purpose of finding the right schema through the Speculator. This enables the CD module to regenerate the entire code snippet from scratch, rather than extending a potentially flawed partial snippet.

*2) Model specific Backtracker:* Unlike the Speculator, which is use case-bound, Backtracker is model-bound. The reason is that the Backtracker requires the boundary of the

TABLE I
Dataset Statistics

| # of Code Snippets | # of Samples in Dataset | |
|---|---|---|
| | **Ansible** | **Application Configuration** |
| 1 | 50 | 10 |
| 2 | 60 | 10 |
| 3 | 90 | 10 |
| 4 | 120 | 10 |
| 5 | 150 | 10 |
| **Total** | **470** | **50** |

TABLE II
Some Task Examples with Code Snippet Count

| Task | # Expected Code Snippets in Output |
|---|---|
| Generate train configuration with learning rate 10 and model granite. | 1 |
| Generate ansible yamls to first create admin user using module company.internal.user second create a file from inline content using module company.internal.copy. | 2 |
| Generate ansible yamls to first generate ssh key if absent using module company.internal.command second create admin user using module company.internal.user third copy file with backup using module company.internal.copy | 3 |
| Generate ansible yamls to first deploy startup script using module company.internal.template second downgrade openssl using module company.internal.apt third check disk usage using module company.internal.command fourth dump database to file using module company.internal.shell | 4 |
| Generate ansible yamls to first pull latest changes using module company.internal.git second copy file with backup using module company.internal.copy third downgrade openssl using module company.internal.apt fourth remove nginx package using module company.internal.yum fifth ensure correct permissions on script using module company.internal.file | 5 |

code snippet so that the token removal factors in the code-related tokens. The model used in our study uses markdown code block (```) as boundaries for the code snippet.

## C. Constrained Decoder

*1) Functionality:* The CD module aims to generate schema-compliant code snippets. To accomplish this, the module takes the schema from the Speculator (step $3b$ in Fig. II) and backtracked response tokens from the Backtracker (step 4 in Fig. II) and performs constrained generation following the constraints articulated in the schema. Constrained generation is a well-known technique and has been applied extensively [13], [17] in the past.

## D. LLM

*1) Functionality:* Apart from generating initial set of tokens for the given prompt (step 1 in Fig. II) and iterative interaction with the Speculator, LLM receives the compliant code snippet from the CD (step 5 in Fig. II) and continues to generate next set of tokens that follow. The entire process repeats until an end-of-sequence token is encountered (step 6 in Fig. II).

## III. Preliminary Evaluation

## A. Evaluation Setup

*1) Dataset:* To demonstrate the effectiveness and practicality of our framework, we apply it to 2 settings: In-Domain (ID) and Out-of-Domain (OOD). For ID setting, we pick a popular known domain that is Ansible, and we manually curate Ansible task data consisting of 470 samples distributed across multiple output code snippet counts as shown in Table I. We curate a schema pool of size 100 schemas. For the OOD setting, we hypothesize an application configuration scenario consisting of 50 samples in total as shown in Table I and a schema pool of size 5. The application has 5 modules, namely Trainer, Evaluator, DataLoader, Inferencer, and Metrics, where each module can accept a specific YAML config. Some examples of the dataset are shown in Table II.

*2) Language Model:* We employ the open-source Deepseek Coder 33B instruction tuned model[1], which shows state-of-the-art capabilities [4], [10] in the open category for code generation when used as is, making it a challenging baseline for our framework. Further, the model includes crucial languages of our study: YAML and JSON in the pretraining data.

*3) Prompting:* We wrap each user query into the model's standard instruction prompt format [4] for aligned generation.

*4) Constrained Decoder Tool:* We use outlines [15].

## B. Baselines

To demonstrate the value of our framework, we propose the following challenging baselines.

*1) LLM:* Pass the user's query, wrapped in the instruction prompt, directly to the LLM, allowing the LLM to generate the code snippets using greedy decoding. In this baseline, LLM is unaware of the pool of schemas supported by the system.

*2) In-context schema pool (ICS) + LLM:* All schemas in the pool supported by the enterprise system are prepended to the user query and wrapped in the instruction prompt, which is then passed to the LLM. The LLM is allowed to generate the answer using greedy decoding, having awareness of the pool of supported schemas.

## C. Evaluation Metrics

*1) Code Snippet Count Ratio (CSCR):* The ratio of the number of code snippets generated to the expected number. The average across all the samples is reported. CSCR evaluates the capability to generate the required number of code snippets. The higher the values better the performance. CSCR # $k$ used in Table IV denotes the ratio at $k$ expected number of code snippets. Where $k$ is ranging from 1 to 5.

*2) Schema Compliance Ratio (SCR):* The ratio of the number of code snippets that are schema-compliant out of all the generated code snippets. The average across all the samples is reported. SCR evaluates the capability to generate schema-correct code snippets. The higher the values better the performance. SCR # $k$ is used in Table III to denotes the ratio at $k$ generated code snippets.

*3) Bilingual Evaluation Understudy (BLEU) Score [18]:* Measures the n-gram similarity between the output and ground truth code snippet. Evaluates semantic correctness.

## IV. Results and Discussion

## A. RQ1: What Is the Performance in Generating Schema-Compliant Code Snippets?

Table III presents the performance across different output code snippet counts ($1-5$) in generating schema-compliant code.

***LLM performance and issues.*** The LLM baseline fails to generate at least a single schema-correct code snippet in

[1]https://huggingface.co/deepseek-ai/deepseek-coder-33b-instruct

TABLE III
SCHEMA COMPLIANCE RATIO EVALUATION

| Method | SCR (%) | | | | |
|---|---|---|---|---|---|
| | **Ansible Use Case (In-Domain)** | | | | |
| | **# 1** | **# 2** | **# 3** | **# 4** | **# 5** |
| **LLM** | 0 | 0 | 0.02 | 0 | 0.03 |
| **ICS + LLM** | 0.41 (↑0.41) | 0.43 (↑0.43) | 0.34 (↑0.32) | 0.33 (↑0.33) | 0.43 (↑0.41) |
| **Ours** | **1 (↑1)** | **1 (↑1)** | **1 (↑0.98)** | **1 (↑1)** | **1 (↑0.97)** |
| | **Application Configuration Use Case (Out-of-Domain)** | | | | |
| **LLM** | 0.01 | 0 | 0.02 | 0.01 | 0 |
| **ICS + LLM** | 0.51 (↑0.5) | 0.48 (↑0.48) | 0.43 (↑0.41) | 0.42 (↑0.41) | 0.45 (↑0.45) |
| **Ours** | **1 (↑0.99)** | **1 (↑1)** | **1 (↑0.98)** | **1 (↑0.99)** | **1 (↑1)** |

TABLE IV
CODE SNIPPET COUNT RATIO EVALUATION

| Method | CSCR (%) | | | | |
|---|---|---|---|---|---|
| | **Ansible Use Case (In-Domain)** | | | | |
| | **# 1** | **# 2** | **# 3** | **# 4** | **# 5** |
| **LLM** | 0.9 | **1** | 0.96 | **1** | **1** |
| **ICS + LLM** | **0.98 (↑0.08)** | **1 (0)** | **1 (↑0.03)** | **1 (0)** | **1 (0)** |
| **Ours** | 0.94 (↑0.04) | **1 (0)** | 0.91 (↓0.06) | 0.93 (↓0.07) | 0.83 (↓0.17) |
| | **Application Configuration Use Case (Out-of-Domain)** | | | | |
| **LLM** | 0.9 | 0.92 | 0.9 | **1** | 0.96 |
| **ICS + LLM** | **1 (↑0.1)** | **1 (↑0.08)** | **1 (↑0.1)** | **1 (0)** | **1 (↑0.04)** |
| **Ours** | **1 (↑0.1)** | **1 (↑0.08)** | **1 (↑0.1)** | 0.94 (↓0.06) | 0.95 (↓0.01) |

the majority of the cases. This demonstrates that the LLM is devoid of prior knowledge of the constraints to follow while generating the code.

***ICS+LLM performance and issues.*** In the second baseline, we provide the full pool of schemas as context to the language model, equipping it with schema information and improving the Schema Compliance Ratio (SCR) to a stable average of $\sim 0.38$ across different output counts. However, the performance remains limited due to the model's inability to consistently adhere to constraints and its tendency to hallucinate keys. These findings are consistent with recent studies [11], which highlight the challenges LLMs face with long input contexts—often missing critical information and leading to degraded output quality. Although ICS+LLM outperforms the vanilla baseline, it becomes impractical when the schema pool exceeds the context window, and incurs higher costs for long-context models due to per-token-based pricing[2]. These limitations motivate our proposed framework, which avoids passing all schemas in-context and instead dynamically selects relevant schemas during generation.

***Our framework performance.*** Our framework outperforms all the baselines for both ID and OOD settings across all the output counts, around $\sim 2.5X$ the performance of the ICS + LLM baseline. The performance is attributed to the Speculator module for choosing the right schema and the CD module for generating code snippets that fully comply with the schema.

### *B. RQ2: What Is the Performance in Generating the Required Number of Code Snippets?*

Table IV presents the performance in generating the required number of code snippets with a breakdown across different output counts through Code Snippet Count Ratio (CSCR).

***Performance of our framework and issues.*** Though our framework outperforms the LLM and is at par with ICS + LLM baselines in lower orders of output count, it has slightly underperformed for higher orders of output counts. We study this gap, and the issue is narrowed down to the CD module. Due to the absence of the token healing feature[3], the generation of longer responses gets distorted, thereby not producing enough code blocks as required by the prompt.

[2]https://research.aimultiple.com/llm-pricing

TABLE V
BLEU SCORE WITH GROUND TRUTH

| Method | BLEU Score | |
|---|---|---|
| | **Ansible Use Case (In-Domain)** | **Application Configuration Use Case (Out-of-Domain)** |
| **LLM** | 0.38 | 0.42 |
| **ICS + LLM** | 0.43 (↑0.05) | 0.54 (↑0.12) |
| **Ours** | **0.68 (↑0.3)** | **0.73 (↑0.31)** |

### *C. RQ3: What Is the Task Performance?*

Table III reflects the syntactic requirement of the task performance, while Tables IV and V provide insights on the semantic requirement. We compute the BLEU [18] score between the ground truth and the response code snippet.

***Our framework performance and issues.*** Our framework performs around $\sim 1.6X$ the baselines in both the settings. Though our framework indirectly covers some aspects of semantics by including required keys in the YAML, thereby covering the keys in the ground truth, it does not fully control the semantic quality of the output and is mostly driven by the underlying LLM's understanding of the prompt. This can be a potential extension to our work in the future.

## V. RELATED WORK

DocCGen [13] performs declarative code generation for languages such as YAML using schemas; however, the solution does not study generating multiple code snippets, each having a respective schema for adherence. Recently, there is a large body of work [9], [14], [16], [19] in generating multiple code snippets or code for large codebases using agentic workflows; however, none of the works study declarative code languages and associated schemas for adherence, though having vital importance in enterprise use cases. Our framework complement such agentic workflows by enabling them with multi-schema compliant code snippet generation capabilities.

## VI. CONCLUSION

In this paper, we explore a promising new direction toward enhancing LLMs for practical usage in generating multiple schema-conformant code snippets for enterprise systems having declarative user interfaces and introduce a potent framework. Preliminary results show that our framework demonstrates promising syntactic and semantic task performance outperforming 2 challenging baselines for ID and OOD settings. We believe our work motivates further research towards supporting code generation for declarative enterprise systems. In the future, we plan to study full semantic control over the output code snippets and setup full-fledged environment for end-to-end execution-based performance study.

[3]https://github.com/dottxt-ai/outlines/issues/161

## REFERENCES

[1] "ansible/ansible," GitHub, Feb. 29, 2024. https://github.com/ansible/ansible

[2] "JSON Schema," Json-schema.org, 2025. http://json-schema.org/ (accessed Jul. 07, 2025).

[3] Production-Grade Container Orchestration, "Production-Grade Container Orchestration," Kubernetes, Jun. 27, 2025. http://kubernetes.io/ (accessed Jul. 07, 2025).

[4] D. Guo et al., "DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence," arXiv.org, Jan. 26, 2024. https://arxiv.org/abs/2401.14196

[5] N. S. Mathews and M. Nagappan, "Test-driven development and LLM-based code generation," Sacramento, CA, USA: Association for Computing Machinery, 2024, pp. 1583–1594. doi: https://doi.org/10.1145/3691620.3695527.

[6] P. Sahoo, S. Pujar, G. Nalawade, R. Genhardt, L. Mandel, and L. Buratti, "Ansible lightspeed: A code generation service for IT automation," Sacramento, CA, USA: Association for Computing Machinery, 2024, pp. 2148–2158. doi: https://doi.org/10.1145/3691620.3695277.

[7] X. Jiang et al., "Self-planning code generation with large language models," ACM Trans. Softw. Eng. Methodol., vol. 33, Art. no. 7, Sep. 2024, doi: https://doi.org/10.1145/3672456.

[8] Y. Dong, X. Jiang, Z. Jin, and G. Li, "Self-collaboration code generation via ChatGPT," ACM Trans. Softw. Eng. Methodol., vol. 33, Art. no. 7, Sep. 2024, doi: https://doi.org/10.1145/3672459.

[9] F. Lei et al., "Spider 2.0: Evaluating language models on real-world enterprise text-to-SQL workflows," 2025. Available: https://openreview.net/forum?id=XmProj9cPs

[10] J. Li et al., "DevEval: A manually-annotated code generation benchmark aligned with real-world code repositories," L.-W. Ku, A. Martins, and V. Srikumar, Eds., Association for Computational Linguistics, Aug. 2024, pp. 3603–3614. doi: https://doi.org/10.18653/v1/2024.findings-acl.214.

[11] J. Li, M. Wang, Z. Zheng, and M. Zhang, "LooGLE: Can long-context language models understand long contexts?," L.-W. Ku, A. Martins, and V. Srikumar, Eds., Association for Computational Linguistics, Aug. 2024, pp. 16304–16333. doi: https://doi.org/10.18653/v1/2024.acl-long.859.

[12] "OpenAI Platform," Openai.com, 2025. https://platform.openai.com/docs/guides/structured-outputs

[13] S. Pimparkhede, M. Kammakomati, Tamilselvam, Srikanth G, P. Kumar, A. P. Kumar, and P. Bhattacharyya, "DocCGen: Document-based controlled code generation," Y. Al-Onaizan, M. Bansal, and Y.-N. Chen, Eds., Association for Computational Linguistics, Nov. 2024, pp. 18681–18697. doi: https://doi.org/10.18653/v1/2024.emnlp-main.1040.

[14] Z. Rasheed et al., "CodePori: Large-scale system for autonomous software development using multi-agent technology," 2024. https://arxiv.org/abs/2402.01411

[15] B. T. Willard and Rémi Louf, "Efficient guided generation for large language models," 2023. https://arxiv.org/abs/2307.09702

[16] S. Yao et al., "ReAct: Synergizing reasoning and acting in language models," 2023. Available: https://openreview.net/forum?id=WE_vluYUL-X

[17] N. Mündler, J. He, H. Wang, K. Sen, D. Song, and M. Vechev, "Type-constrained code generation with language models," Proc. ACM Program. Lang., vol. 9, Art. no. PLDI, Jun. 2025, doi: https://doi.org/10.1145/3729274.

[18] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," P. Isabelle, E. Charniak, and D. Lin, Eds., Association for Computational Linguistics, Jul. 2002, pp. 311–318. doi: https://doi.org/10.3115/1073083.1073135.

[19] [19]Y. Ishibashi and Y. Nishimura, "Self-organized agents: A LLM multi-agent framework toward ultra large-scale code generation and optimization," 2024. https://arxiv.org/abs/2404.02183