ConCodeEval: Evaluating Large Language Models for Code Constraints in Domain-Specific Languages

Anonymous ACL submission

Abstract

Recent work shows Large Language Models (LLMs) struggle to understand natural language constraints for various text generation tasks in zero- and few-shot settings. While, in the code domain, there is wide usage of constraints in code format to maintain the integrity of code written in Domain-Specific Languages[1] (DSLs) like JSON and YAML which are widely used for system-level programming tasks in enterprises. Given that LLMs are increasingly used for system-level code tasks, evaluating if they can comprehend these code constraints is crucial. However, no work has been done to evaluate their controllability over code constraints. Hence, we introduce `ConCodeEval` first-of-its-kind benchmark having two novel tasks for code constraints across five representations. Our findings suggest that language models struggle with code constraints. Code languages that perform excellently for normal code tasks do not perform well when the same languages represent fine-grained constraints.

1 Introduction

Large language models (LLMs) have shown promising results (Brown et al., 2020) in generating coherent text and code in zero and few-shot settings, especially for resource-rich languages. However, their practical utility depends on their ability to follow instructions or constraints at various granularity, encompassing user and system requirements. Recent studies (Sun et al., 2023) indicate that LLMs struggle with fine-grained constraints in natural language prompts, especially in paraphrase generation and numerical planning tasks.

While previous work studied the controllability of language models (LMs) using fine-grained instructions in natural language (NL) format, the necessity to represent instructions in code format arises from a critical enterprise use case involving system-level programming. In enterprises,

Listing 1: The JSON sample generated (highlighted in yellow) by the Granite 20B model does not adhere to the *minContains* and subsequent numerical constraints.

```
Write a JSON sample with field values as per the
     JSON format schema given below.
{
    "type": "array",
    "contains": {
        "type": "number",
        "multipleOf": 2.66,
        "exclusiveMinimum": 0.082318859954352 84,
        "exclusiveMaximum": 5.1100233535478 },
    "minContains": 7
}
JSON sample:
```
[2.66, 5.22, 8.88, 12.54, 16.2, 19.86, 23.52, 27.18]
"‘
```

system-level code integrity is maintained through use-case-specific constraints, typically encoded using schemas. Schemas are instructions in structured code languages like JSON, YAML, XML, or Python to enforce constraints like data types and required fields, ensuring code integrity. For example, the schema in Listing 1 requires the sample to be an array of numbers. Each number must be a multiple of 2.66 and fall within the range defined by the *exclusiveMinimum* and *exclusiveMaximum* fields. Additionally, the array must contain at least seven elements. Following such schema constraints, developers write system-level code in Domain-Specific Languages (DSLs) in a format similar to JSON, YAML, or XML. DSLs are custom languages with specialized schemas and syntax suitable for a particular domain or application. These DSLs are common for tasks like data exchange and system configuration, such as in Kubernetes[2]. Writing DSL code requires deep domain expertise and a significant learning process for developers. This has led to a growing adoption of LLMs for system-level programming in several products such as Ansible Lightspeed[3].

[1] https://w.wiki/6jCH

[2] https://w.wiki/3kbZ

[3] https://developers.redhat.com/products/ansible/lightspeed
```

Given the cruciality of factoring in schemas with LLMs, there is increasing interest in using constrained decoding for DSLs (Pimparkhede et al., 2024; Wang et al., 2024a). However, given its limitations (Appendix A.4), it is necessary to evaluate if LMs are cognizant of code constraints when directly presented as a part of the prompt. Therefore, we aim to study the controllability of LMs through two novel seed tasks: (i) Data as Code generation: valid sample generation factoring in constraints (ii) DSL validation: validate code against constraints. We evaluate two model families, Llama and Granite, ranging from 8B to 70B parameters, aligning with enterprise needs for system-level tasks, where open-source models provide an economical and transparent alternative to black-box models like GPT-4. Both tasks are highly motivated from research and enterprise use case point of view as detailed in Appendix A.5.

Our contributions are:

1. We introduce two novel NLP tasks for enterprise system-level code: code generation from fine-grained schema instructions and code validation against schemas. To the best of our knowledge, we are the first to evaluate LMs on these tasks.

2. A benchmark test set consisting of 602 schema samples, each containing multiple instructions. Each schema sample in our test set is represented in 5 different language formats (JSON, YAML, XML, Python, and NL).

3. Comparative and qualitative analysis of state-of-the-art code models for two novel tasks with different schema languages. Our findings show that language models best comprehend JSON schema (Tables 1, 2) and are agnostic of language proportions in pre-training data.

2 Data as Code Generation in DSL

Task description: Given the schema, the generation task (see Listing 1) aims to produce a compliant data sample in DSL code format. We draw inspiration from several use cases (see Appendix A.5), including synthesizing schema-compliant data from LLMs' parametric memory to train and evaluate smaller-sized models (Song et al., 2020) and generating diverse sets of samples to be used in product test pipelines. For reliable DSL code generation, LLMs need to be schema-aware.

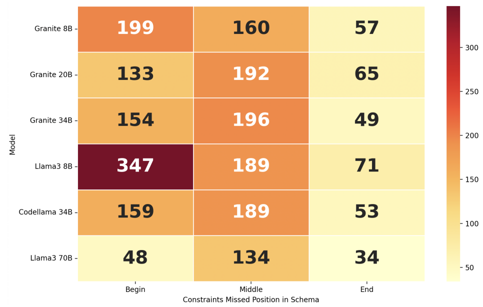

Figure 1: Uniform trend of steep decline in performance across models for constraints positioned in the middle and beginning of the JSON schema context and output for task 1. We divide the schema in to 3 equal portions Begin, Middle, and End, and put the violated constraints based on their locality into either of these three buckets.

Dataset: We synthetically prepare 602 schemas for each of the 5 representations having combinations of various constraints (Appendix A.6). First, we prepare JSON schemas using our combinatorial tool to generate a good mix of constraints. We then convert each JSON schema to XML and YAML schemas using automated tools. Further, we include resource-rich general-purpose language - Python using the Pydantic library generated using the Gemini-1.0-pro (Team et al., 2024) model as a code translation task. We extend our evaluation to NL representation generated using rule-based templates. We[4] ensure equivalence of the generated schemas across languages. More details are in Appendix A.8.

Evaluation metric: Each schema-compliant code output LLM generates is awarded one point where schema compliance is checked using a schema validator tool. We then utilize the accuracy metric (Gen Acc) over all samples to benchmark performance across the models. Additionally, we also report the percentage of samples generated with the invalid root data type (RTV%) and invalid samples (IS%) in Table 5. The root data type is the data type of the whole DSL sample. For example, the root data type of sample represented in Listing 1 is *array*. For IS and RTV metrics, the lesser the number, the better the performance.

Experimental setup: We report greedy decoding results since it performed slightly better than beam search with a beam width of 3. More details including hyper-parameters in Appendix A.7.

[4]The schemas are manually validated by the paper's authors.

Prompts: We experiment with zero-shot and 3-shot prompting for each model. For 3-shot prompting, we identify errors from the zero-shot setting, then select shots similar to the most frequent errors. Examples of prompts are in Appendix 1. While we represent the zero-shot results in the main paper, few-shot results are in the Appendix (Table 2).

Results: Among the 5 schema representations studied, NL is best understood by models across all outputs. JSON and YAML schemas perform well for constraints in code despite their limited presence in pre-training data. Surprisingly, models struggle with constraints in Python, though it is the major portion of the pre-training data, and models also find XML schemas challenging. Notably, models did not exhibit a performance boost when schema and output representation languages were the same. As shown in Figure 1, models are sensitive to the locality of the constraints in the schema and struggle to factor in constraints present in the beginning and middle portions of the schema. Irrespective of their overall performance, models show a similar distribution of mistakes across all constraint types (see Table 4) and show improved performance in few-shot results (see Table 2) when shots relevant to the mistakes used. Among the two family of models, Llama3 70B performed the best followed by Granite 34B.

3 DSL Validation

Listing 2: In the JSON sample, values for fields *stingo* and *anisic* do not adhere to schema constraints. But the Granite 34B model gives the incorrect answer (highlighted in yellow) as *yes*.

~~~~
Question:
Does the JSON sample { "tamil": false, "baser": null
     , "anisic": 1906.34, "stingo": "officiis tellus
     . illum modi odit quas mattis nunc", "
     pigheadedness": 52.0 } adhere to all the
     constraints defined in JSON format schema
{
  "type": "object",
  "properties": {
    "tamil": { "type": "boolean" },
    "baser": { "type": "null" },
    "anisic": { "type": "number", "multipleOf": 17.0
        2 },
    "stingo": { "type": "string", "maxLength": 20 },
    "pigheadedness": {"type": "number", "
        exclusiveMinimum": 27.65410407394338, "
        maximum": 93.85523810367313 } },
  "additionalProperties": false
}
Respond to yes or no.
Answer:
```
yes
“‘
~~~~

Task description: There is a growing body of work (Hada et al., 2024) on showing promising usage of LLMs as evaluators in many tasks. On similar lines, given the DSL sample and schema to validate, this task (see Listing 2) aims to determine the validity of the provided sample against the constraints through boolean question answering (QA). Also, the task is highly motivated from various use cases (see Appendix A.5) and throws light on LM's understanding of the relation between requirements and output in various representations.

Dataset: For each of the 602 schemas across 5 representations as described in Section 2, we generate 3 data samples across JSON, XML, and YAML languages. First, these data samples are synthetically generated by parsing through the JSON schema by randomly pruning and selecting constraints, resulting in data samples of different lengths and constraints. We then convert the generated JSON data samples to equivalent YAML and XML formats. Dataset consists of 3076 instances with 45% of *no* and 55% *yes* instances.

Evaluation metric: Since it is a boolean QA task, we use Macro average F1 (see Table 6) and Accuracy (Val Acc) as evaluation metrics (see Table 1).

Experimental setup: The decoding strategy used here is similar to the generation task as mentioned in Section 2. More details in Appendix A.7.

Prompts: The goal of this task is to answer with either *yes* or *no*. We experiment with zero- and few-shot prompting. With few shot prompting we provide one example each of *yes* and *no* answer. Results for few-shot prompting and examples of prompts are given in Appendix (Table 2).

Results: Although NL representation excels in generation tasks, it degrades the validation performance of larger models like 70B. JSON, YAML and Python representations show effectiveness in one of the output formats however poorly perform for others. Like in task 1, models perform sub-optimally when both schema and output representations are same. In lines with task 1, XML stands as a challenging language for models. The Llama3 70B model perform best in validation like in task 1, with other models hovering around 50% Val Acc, likely reflecting random choice given the binary nature of the task. Smaller models, particularly the Llama3-8B with natural language representation,

		Output Representation					
		JSON		YAML		XML	
Model	Schema	Gen Acc	Val Acc	Gen Acc	Val Acc	Gen Acc	Val Acc
Llama3 8B		28.2	56.0	29.2	45.0	7.9	47.0
Granite 8B		47.5	56.0	24.7	55.0	5.1	45.0
Granite 20B	JSON	50.4	52.0	37.7	44.0	10.1	53.0
Granite 34B		53.3	64.0	32.2	57.0	11.2	**65.0**
Codellama 34B		58.4	64.0	23.0	54.0	9.4	53.0
🏆 **Llama3 70B**		**62.8**	**67.0**	**40.1**	**58.4**	**18.9**	55.7
Llama3 8B		10.2	37.0	22.5	42.0	10.2	46.0
Granite 8B		18.9	47.0	12.1	44.0	8.4	52.0
Granite 20B	XML	24.0	37.0	12.4	47.0	8.6	57.0
Granite 34B		18.7	68.0	18.1	58.0	8.6	**58.0**
Codellama 34B		8.8	46.0	14.2	46.0	8.6	50.0
🏆 **Llama3 70B**		**28.4**	**70.3**	**24.8**	**60.1**	**16.6**	54.2
Llama3 8B		25.9	46.0	8.1	44.0	6.4	45.0
Granite 8B		47.0	47.0	15.7	50.0	8.6	44.0
Granite 20B	YAML	34.7	31.0	25.9	38.0	8.4	47.0
Granite 34B		52.1	68.0	26.4	61.0	8.6	**58.0**
Codellama 34B		48.0	59.0	27.9	53.0	9.1	**58.0**
🏆 **Llama3 70B**		**56.0**	**71.0**	**32.4**	**63.2**	**14.6**	56.9
Llama3 8B		13.7	43.0	10.2	42.0	**11.6**	43.0
Granite 8B		10.2	54.0	11.9	58.0	11.1	**55.0**
Granite 20B	Python	14.6	45.0	11.7	67.0	7.3	44.0
Granite 34B		17.7	54.0	13.9	67.0	10.6	46.0
Codellama 34B		13.7	49.0	11.6	53.0	8.4	44.0
🏆 **Llama3 70B**		**24.7**	**57.2**	**18.9**	**70.4**	**14.9**	52.1
Llama3 8B		30.2	63.0	24.5	56.0	9.6	57.0
Granite 8B		52.3	59.0	42.1	61.0	11.1	58.0
Granite 20B	NL	65.4	54.0	46.0	48.0	10.9	**60.0**
Granite 34B		69.7	55.0	55.1	46.0	10.9	56.0
Codellama 34B		60.4	57.0	40.6	57.0	8.69	50.0
🏆 **Llama3 70B**		**75.2**	**67.7**	**57.2**	**64.2**	**13.4**	58.1

Table 1: Zero shot results for tasks 1 and 2. Models scoring the highest accuracy the majority of times across all output representations for a particular schema are labeled with 🏆. Gen Acc represents the accuracy of valid samples for DSL generation tasks. Val Acc represents the accuracy of the binary classification validation task.

show notable improvement, as its pre-training is a combination of NL and code.

4 Related Work

Generation: There is extensive work (Muennighoff et al., 2024; Cassano et al., 2022) on evaluating capabilities of LLMs for various code tasks such as code completion, translation, etc for resource-rich languages like Python. Despite there being work (Cassano et al., 2022) on multi-lingual code, there is scant attention to low-resource languages such as DSLs, though having crucial importance. In parallel, using LLMs as evaluators for low-resource languages is gaining interest, however limited, mainly focusing on languages like XML and INI (Lian et al., 2024).

Controllability of LLMs: While LLMs can handle coarse-grained constraints like sentiment, they struggle with fine-grained constraints, such as ending a text with a specific word (Sun et al., 2023). Code schemas often require such fine-grained control, and to our knowledge, we are the first to explore LLM controllability for constraints in code.

5 Conclusion

We introduce two novel tasks - Data as Code generation and DSL validation to test the controllability of LLMs when constraints are in code format, which is a crucial use case for system-level programming tasks in enterprises. We evaluate LLMs over 5 schema representations, including YAML, JSON, Python, XML, and NL, and 3 output representations, including YAML, JSON, and XML. We conclude that model performance does not directly correlate with language's portion in pre-training data. Models for task 1 best understand NL. However, this is not the case with the validation task. Interestingly, models underperformed when the schema and output representations were the same, and the locality of the constraints in the schema impacted their performance. Task 2 results show that most of the models, irrespective of their size, found it very challenging since they performed just above or under the 50% accuracy. We hope our work can help innovation in improving the capabilities of LLMs for such challenging use cases and serve as a valuable reference.

6 Limitations

While we explore the DSL validation task by generating *yes* or *no*, exploring the model's reasoning can give a more comprehensive analysis of LLM's understanding. Further, one can include more complex constraints in the future for general-purpose programming languages, like coding style constraints to write code along with natural language prompts and schema.

Ethics Statement

Custom-created datasets have been created synthetically using open-source tools. The language models, tools, and frameworks used for evaluation are open source and can be used without copyright issues.

References

Lukas Berglund, Meg Tong, Max Kaufmann, Mikita Balesni, Asa Cooper Stickland, Tomasz Korbak, and Owain Evans. 2024. The reversal curse: Llms trained on "a is b" fail to learn "b is a". *Preprint*, arXiv:2309.12288.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. 2022. Multipl-e: A scalable and extensible approach to benchmarking neural code generation. *Preprint*, arXiv:2208.08227.

Xinyun Chen, Ryan A. Chi, Xuezhi Wang, and Denny Zhou. 2024. Premise order matters in reasoning with large language models. *Preprint*, arXiv:2402.08939.

Saibo Geng, Martin Josifoski, Maxime Peyrard, and Robert West. 2023. Grammar-constrained decoding for structured NLP tasks without finetuning. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 10932–10952, Singapore. Association for Computational Linguistics.

Rishav Hada, Varun Gumma, Adrian Wynter, Harshita Diddee, Mohamed Ahmed, Monojit Choudhury, Kalika Bali, and Sunayana Sitaram. 2024. Are large language model-based evaluators the solution to scaling up multilingual evaluation? In *Findings of the Association for Computational Linguistics: EACL 2024*, pages 1051–1070, St. Julian's, Malta. Association for Computational Linguistics.

Xinyu Lian, Yinfang Chen, Runxiang Cheng, Jie Huang, Parth Thakkar, Minjia Zhang, and Tianyin Xu. 2024. Configuration validation with large language models. *Preprint*, arXiv:2310.09690.

Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. 2024. Octopack: Instruction tuning code large language models. *Preprint*, arXiv:2308.07124.

Sameer Pimparkhede, Mehant Kammakomati, Srikanth Tamilselvam, Prince Kumar, Ashok Pon Kumar, and Pushpak Bhattacharyya. 2024. Doccgen: Document-based controlled code generation. *Preprint*, arXiv:2406.11925.

Saurabh Pujar, Luca Buratti, Xiaojie Guo, Nicolas Dupuis, Burn Lewis, Sahil Suneja, Atin Sood, Ganesh Nalawade, Matthew Jones, Alessandro Morari, and Ruchir Puri. 2023. Automated code generation for information technology tasks in yaml through large language models. *Preprint*, arXiv:2305.02783.

Kaitao Song, Hao Sun, Xu Tan, Tao Qin, Jianfeng Lu, Hongzhi Liu, and Tie-Yan Liu. 2020. Lightpaff: A two-stage distillation framework for pre-training and fine-tuning. *Preprint*, arXiv:2004.12817.

Jiao Sun, Yufei Tian, Wangchunshu Zhou, Nan Xu, Qian Hu, Rahul Gupta, John Wieting, Nanyun Peng, and Xuezhe Ma. 2023. Evaluating large language models on controlled generation tasks. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 3155–3168, Singapore. Association for Computational Linguistics.

Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M. Dai, Anja Hauth, Katie Millican, David Silver, Melvin Johnson, Ioannis Antonoglou, Julian Schrittwieser, Amelia Glaese, Jilin Chen, Emily Pitler, Timothy Lillicrap, Angeliki Lazaridou, Orhan Firat, James Molloy, Michael Isard, Paul R. Barham, Tom Hennigan, Benjamin Lee, Fabio Viola, Malcolm Reynolds, Yuanzhong Xu, Ryan Doherty, Eli Collins, Clemens Meyer, Eliza Rutherford, Erica Moreira, Kareem Ayoub, Megha Goel, Jack Krawczyk, Cosmo Du, Ed Chi, Heng-Tze Cheng, Eric Ni, Purvi Shah, Patrick Kane, Betty Chan, Manaal Faruqui, Aliaksei Severyn, Hanzhao Lin, YaGuang Li, Yong Cheng, Abe Ittycheriah, Mahdis Mahdieh, Mia Chen, Pei Sun, Dustin Tran, Sumit Bagri, Balaji Lakshminarayanan, Jeremiah

Liu, Andras Orban, Fabian Güra, Hao Zhou, Xinying Song, Aurelien Boffy, Harish Ganapathy, Steven Zheng, HyunJeong Choe, Ágoston Weisz, Tao Zhu, Yifeng Lu, Siddharth Gopal, Jarrod Kahn, Maciej Kula, Jeff Pitman, Rushin Shah, Emanuel Taropa, Majd Al Merey, Martin Baeuml, Zhifeng Chen, Laurent El Shafey, Yujing Zhang, Olcan Sercinoglu, George Tucker, Enrique Piqueras, Maxim Krikun, Iain Barr, Nikolay Savinov, Ivo Danihelka, Becca Roelofs, Anaïs White, Anders Andreassen, Tamara von Glehn, Lakshman Yagati, Mehran Kazemi, Lucas Gonzalez, Misha Khalman, Jakub Sygnowski, Alexandre Frechette, Charlotte Smith, Laura Culp, Lev Proleev, Yi Luan, Xi Chen, James Lottes, Nathan Schucher, Federico Lebron, Alban Rrustemi, Natalie Clay, Phil Crone, Tomas Kocisky, Jeffrey Zhao, Bartek Perz, Dian Yu, Heidi Howard, Adam Bloniarz, Jack W. Rae, Han Lu, Laurent Sifre, Marcello Maggioni, Fred Alcober, Dan Garrette, Megan Barnes, Shantanu Thakoor, Jacob Austin, Gabriel Barth-Maron, William Wong, Rishabh Joshi, Rahma Chaabouni, Deeni Fatiha, Arun Ahuja, Gaurav Singh Tomar, Evan Senter, Martin Chadwick, Ilya Kornakov, Nithya Attaluri, Iñaki Iturrate, Ruibo Liu, Yunxuan Li, Sarah Cogan, Jeremy Chen, Chao Jia, Chenjie Gu, Qiao Zhang, Jordan Grimstad, Ale Jakse Hartman, Xavier Garcia, Thanumalayan Sankaranarayana Pillai, Jacob Devlin, Michael Laskin, Diego de Las Casas, Dasha Valter, Connie Tao, Lorenzo Blanco, Adrià Puigdomènech Badia, David Reitter, Mianna Chen, Jenny Brennan, Clara Rivera, Sergey Brin, Shariq Iqbal, Gabriela Surita, Jane Labanowski, Abhi Rao, Stephanie Winkler, Emilio Parisotto, Yiming Gu, Kate Olszewska, Ravi Addanki, Antoine Miech, Annie Louis, Denis Teplyashin, Geoff Brown, Elliot Catt, Jan Balaguer, Jackie Xiang, Pidong Wang, Zoe Ashwood, Anton Briukhov, Albert Webson, Sanjay Ganapathy, Smit Sanghavi, Ajay Kannan, Ming-Wei Chang, Axel Stjerngren, Josip Djolonga, Yuting Sun, Ankur Bapna, Matthew Aitchison, Pedram Pejman, Henryk Michalewski, Tianhe Yu, Cindy Wang, Juliette Love, Junwhan Ahn, Dawn Bloxwich, Kehang Han, Peter Humphreys, Thibault Sellam, James Bradbury, Varun Godbole, Sina Samangooei, Bogdan Damoc, Alex Kaskasoli, Sébastien M. R. Arnold, Vijay Vasudevan, Shubham Agrawal, Jason Riesa, Dmitry Lepikhin, Richard Tanburn, Srivatsan Srinivasan, Hyeontaek Lim, Sarah Hodkinson, Pranav Shyam, Johan Ferret, Steven Hand, Ankush Garg, Tom Le Paine, Jian Li, Yujia Li, Minh Giang, Alexander Neitz, Zaheer Abbas, Sarah York, Machel Reid, Elizabeth Cole, Aakanksha Chowdhery, Dipanjan Das, Dominika Rogozińska, Vitaliy Nikolaev, Pablo Sprechmann, Zachary Nado, Lukas Zilka, Flavien Prost, Luheng He, Marianne Monteiro, Gaurav Mishra, Chris Welty, Josh Newlan, Dawei Jia, Miltiadis Allamanis, Clara Huiyi Hu, Raoul de Liedekerke, Justin Gilmer, Carl Saroufim, Shruti Rijhwani, Shaobo Hou, Disha Shrivastava, Anirudh Baddepudi, Alex Goldin, Adnan Ozturel, Albin Cassirer, Yunhan Xu, Daniel Sohn, Devendra Sachan, Reinald Kim Amplayo, Craig Swanson, Dessie Petrova, Shashi Narayan, Arthur Guez, Siddhartha Brahma, Jessica Landon, Miteyan Patel, Ruizhe Zhao, Kevin Villela, Luyu Wang, Wenhao Jia, Matthew Rahtz, Mai Giménez, Legg Yeung, James Keeling, Petko Georgiev, Diana Mincu, Boxi Wu, Salem Haykal, Rachel Saputro, Kiran Vodrahalli, James Qin, Zeynep Cankara, Abhanshu Sharma, Nick Fernando, Will Hawkins, Behnam Neyshabur, Solomon Kim, Adrian Hutter, Priyanka Agrawal, Alex Castro-Ros, George van den Driessche, Tao Wang, Fan Yang, Shuo yiin Chang, Paul Komarek, Ross McIlroy, Mario Lučić, Guodong Zhang, Wael Farhan, Michael Sharman, Paul Natsev, Paul Michel, Yamini Bansal, Siyuan Qiao, Kris Cao, Siamak Shakeri, Christina Butterfield, Justin Chung, Paul Kishan Rubenstein, Shivani Agrawal, Arthur Mensch, Kedar Soparkar, Karel Lenc, Timothy Chung, Aedan Pope, Loren Maggiore, Jackie Kay, Priya Jhakra, Shibo Wang, Joshua Maynez, Mary Phuong, Taylor Tobin, Andrea Tacchetti, Maja Trebacz, Kevin Robinson, Yash Katariya, Sebastian Riedel, Paige Bailey, Kefan Xiao, Nimesh Ghelani, Lora Aroyo, Ambrose Slone, Neil Houlsby, Xuehan Xiong, Zhen Yang, Elena Gribovskaya, Jonas Adler, Mateo Wirth, Lisa Lee, Music Li, Thais Kagohara, Jay Pavagadhi, Sophie Bridgers, Anna Bortsova, Sanjay Ghemawat, Zafarali Ahmed, Tianqi Liu, Richard Powell, Vijay Bolina, Mariko Iinuma, Polina Zablotskaia, James Besley, Da-Woon Chung, Timothy Dozat, Ramona Comanescu, Xiance Si, Jeremy Greer, Guolong Su, Martin Polacek, Raphaël Lopez Kaufman, Simon Tokumine, Hexiang Hu, Elena Buchatskaya, Yingjie Miao, Mohamed Elhawaty, Aditya Siddhant, Nenad Tomasev, Jinwei Xing, Christina Greer, Helen Miller, Shereen Ashraf, Aurko Roy, Zizhao Zhang, Ada Ma, Angelos Filos, Milos Besta, Rory Blevins, Ted Klimenko, Chih-Kuan Yeh, Soravit Changpinyo, Jiaqi Mu, Oscar Chang, Mantas Pajarskas, Carrie Muir, Vered Cohen, Charline Le Lan, Krishna Haridasan, Amit Marathe, Steven Hansen, Sholto Douglas, Rajkumar Samuel, Mingqiu Wang, Sophia Austin, Chang Lan, Jiepu Jiang, Justin Chiu, Jaime Alonso Lorenzo, Lars Lowe Sjösund, Sébastien Cevey, Zach Gleicher, Thi Avrahami, Anudhyan Boral, Hansa Srinivasan, Vittorio Selo, Rhys May, Konstantinos Aisopos, Léonard Hussenot, Livio Baldini Soares, Kate Baumli, Michael B. Chang, Adrià Recasens, Ben Caine, Alexander Pritzel, Filip Pavetic, Fabio Pardo, Anita Gergely, Justin Frye, Vinay Ramasesh, Dan Horgan, Kartikeya Badola, Nora Kassner, Subhrajit Roy, Ethan Dyer, Víctor Campos Campos, Alex Tomala, Yunhao Tang, Dalia El Badawy, Elspeth White, Basil Mustafa, Oran Lang, Abhishek Jindal, Sharad Vikram, Zhitao Gong, Sergi Caelles, Ross Hemsley, Gregory Thornton, Fangxiaoyu Feng, Wojciech Stokowiec, Ce Zheng, Phoebe Thacker, Çağlar Ünlü, Zhishuai Zhang, Mohammad Saleh, James Svensson, Max Bileschi, Piyush Patil, Ankesh Anand, Roman Ring, Katerina Tsihlas, Arpi Vezer, Marco Selvi, Toby Shevlane, Mikel Rodriguez, Tom Kwiatkowski, Samira Daruki, Keran Rong, Allan Dafoe, Nicholas FitzGerald, Keren Gu-Lemberg, Mina Khan, Lisa Anne Hendricks, Marie Pellat, Vladimir Feinberg, James Cobon-Kerr, Tara Sainath, Maribeth Rauh, Sayed Hadi Hashemi, Richard Ives, Yana Hasson, Eric Noland, Yuan Cao, Nathan Byrd, Le Hou, Qingze Wang, Thibault Sottiaux, Michela Paganini, Jean-Baptiste Lespiau, Alexandre Moufarek, Samer Hassan, Kaushik Shivakumar, Joost van Amersfoort, Amol Mandhane, Pratik Joshi, Anirudh Goyal, Matthew Tung, Andrew Brock, Hannah Sheahan, Vedant Misra, Cheng Li, Nemanja Rakićević, Mostafa Dehghani, Fangyu Liu, Sid Mittal, Junhyuk Oh, Seb Noury, Eren Sezener, Fantine Huot, Matthew Lamm, Nicola De Cao, Charlie Chen, Sidharth Mudgal, Romina Stella, Kevin Brooks, Gautam Vasudevan, Chenxi Liu, Mainak Chain, Nivedita Melinkeri, Aaron Cohen, Venus Wang, Kristie Seymore, Sergey Zubkov, Rahul Goel, Summer Yue, Sai Krishnakumaran, Brian Albert, Nate Hurley, Motoki Sano, Anhad Mohananey, Jonah Joughin, Egor Filonov, Tomasz Kępa, Yomna Eldawy, Jiawern Lim, Rahul Rishi, Shirin Badiezadegan, Taylor Bos, Jerry Chang, Sanil Jain, Sri Gayatri Sundara Padmanabhan, Subha Puttagunta, Kalpesh Krishna, Leslie Baker, Norbert Kalb, Vamsi Bedapudi, Adam Kurzrok, Shuntong Lei, Anthony Yu, Oren Litvin, Xiang Zhou, Zhichun Wu, Sam Sobell, Andrea Siciliano, Alan Papir, Robby Neale, Jonas Bragagnolo, Tej Toor, Tina Chen, Valentin Anklin, Feiran Wang, Richie Feng, Milad Gholami, Kevin Ling, Lijuan Liu, Jules Walter, Hamid Moghaddam, Arun Kishore, Jakub Adamek, Tyler Mercado, Jonathan Mallinson, Siddhinita Wandekar, Stephen Cagle, Eran Ofek, Guillermo Garrido, Clemens Lombriser, Maksim Mukha, Botu Sun, Hafeezul Rahman Mohammad, Josip Matak, Yadi Qian, Vikas Peswani, Pawel Janus, Quan Yuan, Leif Schelin, Oana David, Ankur Garg, Yifan He, Oleksii Duzhyi, Anton Älgmyr, Timothée Lottaz, Qi Li, Vikas Yadav, Luyao Xu, Alex Chinien, Rakesh Shivanna, Aleksandr Chuklin, Josie Li, Carrie Spadine, Travis Wolfe, Kareem Mohamed, Subhabrata Das, Zihang Dai, Kyle He, Daniel von Dincklage, Shyam Upadhyay, Akanksha Maurya, Luyan Chi, Sebastian Krause, Khalid Salama, Pam G Rabinovitch, Pavan Kumar Reddy M, Aarush Selvan, Mikhail Dektiarev, Golnaz Ghiasi, Erdem Guven, Himanshu Gupta, Boyi Liu, Deepak Sharma, Idan Heimlich Shtacher, Shachi Paul, Oscar Akerlund, François-Xavier Aubet, Terry Huang, Chen Zhu, Eric Zhu, Elico Teixeira, Matthew Fritze, Francesco Bertolini, Liana-Eleonora Marinescu, Martin Bölle, Dominik Paulus, Khyatti Gupta, Tejasi Latkar, Max Chang, Jason Sanders, Roopa Wilson, Xuewei Wu, Yi-Xuan Tan, Lam Nguyen Thiet, Tulsee Doshi, Sid Lall, Swaroop Mishra, Wanming Chen, Thang Luong, Seth Benjamin, Jasmine Lee, Ewa Andrejczuk, Dominik Rabiej, Vipul Ranjan, Krzysztof Styrc, Pengcheng Yin, Jon Simon, Malcolm Rose Harriott, Mudit Bansal, Alexei Robsky, Geoff Bacon, David Greene, Daniil Mirylenka, Chen Zhou, Obaid Sarvana, Abhimanyu Goyal, Samuel Andermatt, Patrick Siegler, Ben Horn, Assaf Israel, Francesco Pongetti, Chih-Wei "Louis" Chen, Marco Selvatici, Pedro Silva, Kathie Wang, Jackson Tolins, Kelvin Guu, Roey Yogev, Xiaochen Cai, Alessandro Agostini, Maulik Shah, Hung Nguyen, Noah Ó Donnaile, Sébastien Pereira, Linda Friso, Adam Stambler, Adam Kurzrok, Chenkai Kuang, Yan Romanikhin, Mark Geller, ZJ Yan, Kane Jang, Cheng-Chun Lee, Wojciech Fica, Eric Malmi, Qijun Tan, Dan Banica, Daniel Balle, Ryan Pham, Yanping Huang, Diana Avram, Hongzhi Shi, Jasjot Singh, Chris Hidey, Niharika Ahuja, Pranab Saxena, Dan Dooley, Srividya Pranavi Potharaju, Eileen O'Neill, Anand Gokulchandran, Ryan Foley, Kai Zhao, Mike Dusenberry, Yuan Liu, Pulkit Mehta, Ragha Kotikalapudi, Chalence Safranek-Shrader, Andrew Goodman, Joshua Kessinger, Eran Globen, Prateek Kolhar, Chris Gorgolewski, Ali Ibrahim, Yang Song, Ali Eichenbaum, Thomas Brovelli, Sahitya Potluri, Preethi Lahoti, Cip Baetu, Ali Ghorbani, Charles Chen, Andy Crawford, Shalini Pal, Mukund Sridhar, Petru Gurita, Asier Mujika, Igor Petrovski, Pierre-Louis Cedoz, Chenmei Li, Shiyuan Chen, Niccolò Dal Santo, Siddharth Goyal, Jitesh Punjabi, Karthik Kappaganthu, Chester Kwak, Pallavi LV, Sarmishta Velury, Himadri Choudhury, Jamie Hall, Premal Shah, Ricardo Figueira, Matt Thomas, Minjie Lu, Ting Zhou, Chintu Kumar, Thomas Jurdi, Sharat Chikkerur, Yenai Ma, Adams Yu, Soo Kwak, Victor Ähdel, Sujeevan Rajayogam, Travis Choma, Fei Liu, Aditya Barua, Colin Ji, Ji Ho Park, Vincent Hellendoorn, Alex Bailey, Taylan Bilal, Huanjie Zhou, Mehrdad Khatir, Charles Sutton, Wojciech Rzadkowski, Fiona Macintosh, Konstantin Shagin, Paul Medina, Chen Liang, Jinjing Zhou, Pararth Shah, Yingying Bi, Attila Dankovics, Shipra Banga, Sabine Lehmann, Marissa Bredesen, Zifan Lin, John Eric Hoffmann, Jonathan Lai, Raynald Chung, Kai Yang, Nihal Balani, Arthur Bražinskas, Andrei Sozanschi, Matthew Hayes, Héctor Fernández Alcalde, Peter Makarov, Will Chen, Antonio Stella, Liselotte Snijders, Michael Mandl, Ante Kärrman, Paweł Nowak, Xinyi Wu, Alex Dyck, Krishnan Vaidyanathan, Raghavender R, Jessica Mallet, Mitch Rudominer, Eric Johnston, Sushil Mittal, Akhil Udathu, Janara Christensen, Vishal Verma, Zach Irving, Andreas Santucci, Gamaleldin Elsayed, Elnaz Davoodi, Marin Georgiev, Ian Tenney, Nan Hua, Geoffrey Cideron, Edouard Leurent, Mahmoud Alnahlawi, Ionut Georgescu, Nan Wei, Ivy Zheng, Dylan Scandinaro, Heinrich Jiang, Jasper Snoek, Mukund Sundararajan, Xuezhi Wang, Zack Ontiveros, Itay Karo, Jeremy Cole, Vinu Rajashekhar, Lara Tumeh, Eyal Ben-David, Rishub Jain, Jonathan Uesato, Romina Datta, Oskar Bunyan, Shimu Wu, John Zhang, Piotr Stanczyk, Ye Zhang, David Steiner, Subhajit Naskar, Michael Azzam, Matthew Johnson, Adam Paszke, Chung-Cheng Chiu, Jaume Sanchez Elias, Afroz Mohiuddin, Faizan Muhammad, Jin Miao, Andrew Lee, Nino Vieillard, Jane Park, Jiageng Zhang, Jeff Stanway, Drew Garmon, Abhijit Karmarkar, Zhe Dong, Jong Lee, Aviral Kumar, Luowei Zhou, Jonathan Evens, William Isaac, Geoffrey Irving, Edward Loper, Michael Fink, Isha Arkatkar, Nanxin Chen, Izhak Shafran, Ivan Petrychenko, Zhe Chen, Johnson Jia, Anselm Levskaya, Zhenkai Zhu, Peter Grabowski, Yu Mao, Alberto Magni, Kaisheng Yao, Javier Snaider, Norman Casagrande, Evan Palmer, Paul Suganthan, Alfonso Castaño, Irene Giannoumis, Wooyeol Kim, Mikołaj Rybiński, Ashwin Sreevatsa, Jennifer Prendki, David Soergel, Adrian Goedeckemeyer, Willi Gierke, Mohsen Jafari, Meenu Gaba, Jeremy Wiesner, Diana Gage Wright, Yawen Wei, Harsha Vashisht, Yana Kulizhskaya, Jay Hoover, Maigo Le, Lu Li, Chimezie Iwuanyanwu, Lu Liu, Kevin Ramirez, Andrey Khorlin, Albert Cui, Tian LIN, Marcus Wu, Ricardo Aguilar, Keith Pallo, Abhishek Chakladar, Ginger Perng, Elena Allica Abellan, Mingyang Zhang, Ishita Dasgupta, Nate Kushman, Ivo Penchev, Alena Repina, Xihui Wu, Tom van der Weide, Priya Ponnapalli, Caroline Kaplan, Jiri Simsa, Shuangfeng Li, Olivier Dousse, Fan Yang, Jeff Piper, Nathan Ie, Rama Pasumarthi, Nathan Lintz, Anitha Vijayakumar, Daniel Andor, Pedro Valenzuela, Minnie Lui, Cosmin Paduraru, Daiyi Peng, Katherine Lee, Shuyuan Zhang, Somer Greene, Duc Dung Nguyen, Paula Kurylowicz, Cassidy Hardin, Lucas Dixon, Lili Janzer, Kiam Choo, Ziqiang Feng, Biao Zhang, Achintya Singhal, Dayou Du, Dan McKinnon, Natasha Antropova, Tolga Bolukbasi, Orgad Keller, David Reid, Daniel Finchelstein, Maria Abi Raad, Remi Crocker, Peter Hawkins, Robert Dadashi, Colin Gaffney, Ken Franko, Anna Bulanova, Rémi Leblond, Shirley Chung, Harry Askham, Luis C. Cobo, Kelvin Xu, Felix Fischer, Jun Xu, Christina Sorokin, Chris Alberti, Chu-Cheng Lin, Colin Evans, Alek Dimitriev, Hannah Forbes, Dylan Banarse, Zora Tung, Mark Omernick, Colton Bishop, Rachel Sterneck, Rohan Jain, Jiawei Xia, Ehsan Amid, Francesco Piccinno, Xingyu Wang, Praseem Banzal, Daniel J. Mankowitz, Alex Polozov, Victoria Krakovna, Sasha Brown, MohammadHossein Bateni, Dennis Duan, Vlad Firoiu, Meghana Thotakuri, Tom Natan, Matthieu Geist, Ser tan Girgin, Hui Li, Jiayu Ye, Ofir Roval, Reiko Tojo, Michael Kwong, James Lee-Thorp, Christopher Yew, Danila Sinopalnikov, Sabela Ramos, John Mellor, Abhishek Sharma, Kathy Wu, David Miller, Nicolas Sonnerat, Denis Vnukov, Rory Greig, Jennifer Beattie, Emily Caveness, Libin Bai, Julian Eisenschlos, Alex Korchemniy, Tomy Tsai, Mimi Jasarevic, Weize Kong, Phuong Dao, Zeyu Zheng, Frederick Liu, Fan Yang, Rui Zhu, Tian Huey Teh, Jason Sanmiya, Evgeny Gladchenko, Nejc Trdin, Daniel Toyama, Evan Rosen, Sasan Tavakkol, Linting Xue, Chen Elkind, Oliver Woodman, John Carpenter, George Papamakarios, Rupert Kemp, Sushant Kafle, Tanya Grunina, Rishika Sinha, Alice Talbert, Diane Wu, Denese Owusu-Afriyie, Cosmo Du, Chloe Thornton, Jordi Pont-Tuset, Pradyumna Narayana, Jing Li, Saaber Fatehi, John Wieting, Omar Ajmeri, Benigno Uria, Yeongil Ko, Laura Knight, Amélie Héliou, Ning Niu, Shane Gu, Chenxi Pang, Yeqing Li, Nir Levine, Ariel Stolovich, Rebeca Santamaria-Fernandez, Sonam Goenka, Wenny Yustalim, Robin Strudel, Ali Elqursh, Charlie Deck, Hyo Lee, Zonglin Li, Kyle Levin, Raphael Hoffmann, Dan Holtmann-Rice, Olivier Bachem, Sho Arora, Christy Koh, Soheil Hassas Yeganeh, Siim Põder, Mukarram Tariq, Yanhua Sun, Lucian Ionita, Mojtaba Seyedhosseini, Pouya Tafti, Zhiyu Liu, Anmol Gulati, Jasmine Liu, Xinyu Ye, Bart Chrzaszcz, Lily Wang, Nikhil Sethi, Tianrun Li, Ben Brown, Shreya Singh, Wei Fan, Aaron Parisi, Joe Stanton, Vinod Koverkathu, Christopher A. Choquette-Choo, Yunjie Li, TJ Lu, Abe Ittycheriah, Prakash Shroff, Mani Varadarajan, Sanaz Bahargam, Rob Willoughby, David Gaddy, Guillaume Desjardins, Marco Cornero, Brona Robenek, Bhavishya Mittal, Ben Albrecht, Ashish Shenoy, Fedor Moiseev, Henrik Jacobsson, Alireza Ghaffarkhah, Morgane Rivière, Alanna Walton, Clément Crepy, Alicia Parrish, Zongwei Zhou, Clement Farabet, Carey Radebaugh, Praveen Srinivasan, Claudia van der Salm, Andreas Fidjeland, Salvatore Scellato, Eri Latorre-Chimoto, Hanna Klimczak-Plucińska, David Bridson, Dario de Cesare, Tom Hudson, Piermaria Mendolicchio, Lexi Walker, Alex Morris, Matthew Mauger, Alexey Guseynov, Alison Reid, Seth Odoom, Lucia Loher, Victor Cotruta, Madhavi Yenugula, Dominik Grewe, Anastasia Petrushkina, Tom Duerig, Antonio Sanchez, Steve Yadlowsky, Amy Shen, Amir Globerson, Lynette Webb, Sahil Dua, Dong Li, Surya Bhupatiraju, Dan Hurt, Haroon Qureshi, Ananth Agarwal, Tomer Shani, Matan Eyal, Anuj Khare, Shreyas Rammohan Belle, Lei Wang, Chetan Tekur, Mihir Sanjay Kale, Jinliang Wei, Ruoxin Sang, Brennan Saeta, Tyler Liechty, Yi Sun, Yao Zhao, Stephan Lee, Pandu Nayak, Doug Fritz, Manish Reddy Vuyyuru, John Aslanides, Nidhi Vyas, Martin Wicke, Xiao Ma, Evgenii Eltyshev, Nina Martin, Hardie Cate, James Manyika, Keyvan Amiri, Yelin Kim, Xi Xiong, Kai Kang, Florian Luisier, Nilesh Tripuraneni, David Madras, Mandy Guo, Austin Waters, Oliver Wang, Joshua Ainslie, Jason Baldridge, Han Zhang, Garima Pruthi, Jakob Bauer, Feng Yang, Riham Mansour, Jason Gelman, Yang Xu, George Polovets, Ji Liu, Honglong Cai, Warren Chen, XiangHai Sheng, Emily Xue, Sherjil Ozair, Christof Angermueller, Xiaowei Li, Anoop Sinha, Weiren Wang, Julia Wiesinger, Emmanouil Koukoumidis, Yuan Tian, Anand Iyer, Madhu Gurumurthy, Mark Goldenson, Parashar Shah, MK Blake, Hongkun Yu, Anthony Urbanowicz, Jennimaria Palomaki, Chrisantha Fernando, Ken Durden, Harsh Mehta, Nikola Momchev, Elahe Rahimtoroghi, Maria Georgaki, Amit Raul, Sebastian Ruder, Morgan Redshaw, Jinhyuk Lee, Denny Zhou, Komal Jalan, Dinghua Li, Blake Hechtman, Parker Schuh, Milad Nasr, Kieran Milan, Vladimir Mikulik, Juliana Franco, Tim Green, Nam Nguyen, Joe Kelley, Aroma Mahendru, Andrea Hu, Joshua Howland, Ben Vargas, Jeffrey Hui, Kshitij Bansal, Vikram Rao, Rakesh Ghiya, Emma Wang, Ke Ye, Jean Michel Sarr, Melanie Moranski Preston, Madeleine Elish, Steve Li, Aakash Kaku, Jigar Gupta, Ice Pasupat, Da-Cheng Juan, Milan Someswar, Tejvi M., Xinyun Chen, Aida Amini, Alex Fabrikant, Eric Chu, Xuanyi Dong, Amruta Muthal, Senaka Buthpitiya, Sarthak Jauhari, Nan Hua, Urvashi Khandelwal, Ayal Hitron, Jie Ren, Larissa Rinaldi, Shahar Drath, Avigail Dabush, Nan-Jiang Jiang, Harshal Godhia, Uli Sachs, Anthony Chen, Yicheng Fan, Hagai Taitelbaum, Hila Noga, Zhuyun Dai, James Wang, Chen Liang, Jenny Hamer, Chun-Sung Ferng, Chenel Elkind, Aviel Atias, Paulina Lee, Vít Listík, Mathias Carlen, Jan van de Kerkhof, Marcin Pikus, Krunoslav Zaher, Paul Müller, Sasha Zykova, Richard Stefanec, Vitaly Gatsko, Christoph Hirn-

schall, Ashwin Sethi, Xingyu Federico Xu, Chetan Ahuja, Beth Tsai, Anca Stefanoiu, Bo Feng, Keshav Dhandhania, Manish Katyal, Akshay Gupta, Atharva Parulekar, Divya Pitta, Jing Zhao, Vivaan Bhatia, Yashodha Bhavnani, Omar Alhadlaq, Xiaolin Li, Peter Danenberg, Dennis Tu, Alex Pine, Vera Filippova, Abhipso Ghosh, Ben Limonchik, Bhargava Urala, Chaitanya Krishna Lanka, Derik Clive, Yi Sun, Edward Li, Hao Wu, Kevin Hongtongsak, Ianna Li, Kalind Thakkar, Kuanysh Omarov, Kushal Majmundar, Michael Alverson, Michael Kucharski, Mohak Patel, Mudit Jain, Maksim Zabelin, Paolo Pelagatti, Rohan Kohli, Saurabh Kumar, Joseph Kim, Swetha Sankar, Vineet Shah, Lakshmi Ramachandruni, Xiangkai Zeng, Ben Bariach, Laura Weidinger, Tu Vu, Amar Subramanya, Sissie Hsiao, Demis Hassabis, Koray Kavukcuoglu, Adam Sadovsky, Quoc Le, Trevor Strohman, Yonghui Wu, Slav Petrov, Jeffrey Dean, and Oriol Vinyals. 2024. Gemini: A family of highly capable multimodal models. *Preprint*, arXiv:2312.11805.

Bailin Wang, Zi Wang, Xuezhi Wang, Yuan Cao, Rif A Saurous, and Yoon Kim. 2024a. Grammar prompting for domain-specific language generation with large language models. *Advances in Neural Information Processing Systems*, 36.

Bailin Wang, Zi Wang, Xuezhi Wang, Yuan Cao, Rif A. Saurous, and Yoon Kim. 2024b. Grammar prompting for domain-specific language generation with large language models. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA. Curran Associates Inc.

A Appendix

A.1 Prompts

This section defines the prompts which are used for models. We report different prompts for every model tried here and report the best-performing prompt results. Generally, the model consists of a System Prompt followed by a prompt template specific to the model.

A.1.1 Common prompt

For zero shot inference, we use a common prompt as it is for all the models irrespective of the model's prompt format and we observe best results for Task-1 with this prompt. The prompt is as follows.

Listing 3: common prompt

```
Write an {input_representation} sample with field
     values as per the {output_representation}
     format schema given below.

{schema}

{output_representation} sample:
```
```

A.1.2 Granite model family

The granite model generally follows the question-answering format. Task-1 prompts for granite family models are as follows.

System prompt:
System:
You are an intelligent AI programming assistant, utilizing a Granite code language model developed by IBM. Your primary function is to assist users in code explanation, code generation and other software engineering tasks. You MUST follow these guidelines: - Your responses must be factual. Do not assume the answer is *yes* when you do not know, and DO NOT SHARE FALSE INFORMATION. - You should give concise answers. You should follow the instruction and provide the answer in the specified format and DO NOT SHARE FALSE INFORMATION.
Prompt 2:

Listing 4: QA-prompt-1

```
{System prompt}

Question:
Write an {input_representation} sample with field
     values as per the {input_representation} format
      schema given below.

{schema}

Answer:
```
```

**Prompt 3:**

Listing 5: QA-prompt-2

```
{System prompt}

Question:
Write an {input_representation} sample with field
     values as per the {output_representation}
     format schema given below. Please wrap your
     code
answer using ```

{schema}

Answer:
```
```

{output_representation} and {input_representation} are the variables where {input_representation} take the values JSON, YAML, XML, Python, and natural language. {output_representation} takes the values JSON, YAML, and XML.

A.1.3 Llama family

For codellama $34B$ model we wrap the common prompt in [INST] and [/INST] tags. For the llama3-
```

8B model, we use the System prompt along with user tags [5].

**System prompt:** You are a helpful, respectful, and honest assistant. Always answer as helpfully as possible, while being safe. Your answers should not include any harmful, unethical, racist, sexist, toxic, dangerous, or illegal content. Please ensure that your responses are socially unbiased and positive. If a question does not make any sense or is not factually coherent, explain why instead of answering something not correct. If you don't know the answer to a question, please don't share false information.

Other than this, similar to the granite family we try Question answering format and instruction to wrap the output in quotes (“‘).

### A.2 Data statistics

This section represents schema length comparison for various languages.

### A.3 Few shot prompting results

Below are the results for Few-shot prompting. We experiment with 3 shot prompting. We observe that the majority of errors made by all the models are regarding short schema and the schema having root type of array as shown in sample 1. An example of a 3-shot prompt for a DSL generation task is shown below.

**Few shot prompt**

Listing 6: Few shot prompt

```
{System prompt}

Your task is to write a JSON sample with field
    values as per JSON format schema.
You are given a few examples demonstrating the same.

JSON format schema:
{
    "type": "array",
    "contains": {
        "type": "boolean"
    },
    "minContains": 0
}
JSON sample:
```
[true, true, false]
```
JSON format schema:
{
    "type": "string",
    "format": "idn-email"
}
JSON sample:
```
"hchavezexample.org"“‘JSON format schema:"type":
    "array","items": "type": "number","multipleOf":
    5.82,"exclusiveMinimum": 3.069158195370172JSON sample:“‘
```

A.4 Limitations of Constrained Decoding

This section outlines some common problems with constrained decoding and emphasises on why it cannot be a complete and a viable solution for factoring in schemas to generate compliant text using language models.

A.4.1 Inference Performance Bottleneck

Constrained decoding often negatively effects inference throughput widely mentioned as one of the major drawbacks in many works (Wang et al., 2024b; Pimparkhede et al., 2024; Geng et al., 2023) due to involvement of token-level operations keeping track of the schema constraints and tokens generated so far. This latency can be a factor of the complexity of the schema, tokens generated so far, and the nature of the constrained decoding implementation. Further, advances such as batched inference [6] are not yet there for constrained decoding limiting their scalability and practical use.

A.4.2 Complex Engineering Effort

Implementing a constrained decoding system can involve instrumenting at the decoding phase of the language model while keeping track of the tokens generated so far and structured schema adherence which can involve implementation specific to a schema representation and may not be possible to generalize to any schema representation. For instance, most of the openly available constrained decoding systems [7] have limited support and not generalized to various schemas such as XML and output formats such as YAML and others. It is worthwhile to note that some approaches tend to convert scehmas to context free grammars, however, this approach is possible with common schema representations such as Python pydantic. Additionally, implementing such a system requires deep domain expertise.

A.4.3 Model Performance Bottleneck

LLMs have multiple failure modes that can likely be triggered through constrained decoding. Many works show that LLMs are sensitive to the text being fed into them and often deteriorate the model's performance. Some examples being the reverse

[5] https://huggingface.co/meta-llama/Meta-Llama-3-8B-Instruct

[6] https://github.com/microsoft/batch-inference

[7] https://github.com/outlines-dev/outlines
```

| | | Output Representation | | | | | |
|---|---|---|---|---|---|---|---|
| | | JSON | | YAML | | XML | |
| Model | Schema | Gen Acc | Val Acc | Gen Acc | Val Acc | Gen Acc | Val Acc |
| Llama3 8B | JSON | 48.3 | 71.2 | 46.6 | 68.1 | 39.2 | 64.1 |
| Granite 8B | | 51.2 | 69.2 | 52.3 | 66.1 | 47.8 | 65.8 |
| Granite 20B | | 58.3 | 73.5 | 56.4 | 72.3 | 50.2 | 68.2 |
| Granite 34B | | 66.3 | 76.2 | 64.5 | 75.4 | 51.3 | 73.2 |
| Codellama 34B | | 65.1 | 75.1 | 63.4 | 73.2 | 50.6 | 71.2 |
| 🏆 **Llama3 70B** | | **70.1** | **79.3** | **69.4** | **77.9** | **58.6** | **74.2** |
| Llama3 8B | XML | 46.6 | 65.8 | 42.3 | 63.4 | 36.6 | 60.1 |
| Granite 8B | | 46.2 | 64.8 | 44.5 | 63.2 | 34.5 | 57.3 |
| Granite 20B | | 50.4 | 66.7 | 48.2 | 64.1 | 36.4 | 56.1 |
| Granite 34B | | 52.3 | 68.5 | 51.1 | 63.4 | 39.2 | 53.2 |
| Codellama 34B | | 49.2 | 66.2 | 49.2 | 63.2 | 35.1 | 52.1 |
| 🏆 **Llama3 70B** | | **56.4** | **70.3** | **55.6** | **68.2** | **43.6** | **66.3** |
| Llama3 8B | YAML | 46.7 | 67.2 | 45.3 | 64.2 | 43.5 | 63.2 |
| Granite 8B | | 48.1 | 65.2 | 46.2 | 61.2 | 44.2 | 61.2 |
| Granite 20B | | 52.3 | 68.9 | 49.7 | 66.7 | 47.8 | 65.1 |
| Granite 34B | | 54.2 | 67.7 | 51.3 | 65.3 | 45.3 | 56.4 |
| Codellama 34B | | 56.8 | 66.4 | 50.2 | 64.3 | 47.8 | 56.2 |
| 🏆 **Llama3 70B** | | **60.4** | **76.3** | **57.3** | **69.1** | **49.6** | **68.3** |
| Llama3 8B | Python | 43.2 | 60.1 | 41.1 | 58.9 | 39.2 | 57.6 |
| Granite 8B | | 45.1 | 60.5 | 46.7 | 59.4 | 37.4 | 56.0 |
| Granite 20B | | 48.2 | 57.2 | 45.9 | 57.8 | 38.4 | 58.2 |
| Granite 34B | | 50.6 | 59.2 | 47.1 | 55.6 | 41.3 | 57.3 |
| Codellama 34B | | 47.2 | 56.4 | 45.3 | 57.2 | 39.2 | 55.1 |
| 🏆 **Llama3 70B** | | **56.2** | **65.1** | **50.7** | **64.2** | **43.4** | **60.6** |

Table 2: Few shot results for task 1 (3 shots) and 2 (2 shots). Models scoring highest accuracy majority number of times across all output representations for a particular schema are labeled with 🏆. Gen Acc represents the accuracy of valid samples for DSL generation tasks. Val Acc represents the accuracy of the binary classification validation task.

| **Language** | **Max schema length** | **Average schema length** |
|---|---|---|
| XML | 3316 | 364.82 |
| JSON | 1954 | 208.23 |
| YAML | 1295 | 135.09 |

Table 3: Schema length comparison

| **Constraint** | **Llama3 8B** | **Llama3 70B** |
|---|---|---|
| type | 302 | 49 |
| exclusiveMinimum | 18 | 44 |
| multipleOf | 170 | 42 |
| minLength | 47 | 21 |
| contains | 22 | 12 |
| exclusiveMaximum | 22 | 12 |
| maximum | 11 | 2 |
| maxLength | 7 | 19 |
| additionalProperties | 4 | 0 |
| minimum | 4 | 15 |

Table 4: Both the models, least and best performing irrespective of their performance show a similar distribution of mistakes for each constraint.

curse from (Berglund et al., 2024), where LLM understanding "A is B" may not guarantee to learn "B is A". Another work (Chen et al., 2024) shows that the order of the premises can have a substantial impact on the performance often affecting negatively. Such failures can be triggered when the natural flow of text generation is interrupted through constrained decoding over autoregressive generation. The problem can worsen when it involves mixed generation of structured output and unstructured NL text.

#### A.4.4 Limited Scope

Since constrained decoding needs access to the decoding phase of the language model, its often not possible to apply such decoding to hosted or gated LLM deployments.

Applying constrained decoding to some common use cases is not obvious. Given $n$ structured schemas from $s_1$ to $s_n$, unstructured NL text output as $k$ and structured output as $u$. Common use cases in natural language processing (NLP) such as summarization involve the following input-output relationship. For some arbitrary schema $i$, $s_i \rightarrow u$. Further typical use cases involve factoring in $n$ multiple schemas and generate $m$ multiple structured outputs $(s_1 ... s_n) \rightarrow (k_1 ... k_m)$.

Employing constrained decoding in such use cases is not viable since in the first use case, tasks that output $u$ cannot leverage constrained decoding and schema has to go into LLMs as input. When multiple schemas and structured outputs are involved, its not obvious to choose the right schema for decoding a particular structured output. Such common use cases substantially limit the scope of using constrained decoding.

### A.5 Task Motivation

#### A.5.1 Data as Code Generation Task

This section describes use cases from enterprise and research point of view motivating data as code generation seed task in our study.

**Enterprise Use Cases:** (i) Test case structured data generation to test application interfaces such as REST API endpoints. Often enterprises have large number of services exposing API endpoints that have to be tested and LLMs can be a drop in solution to generate test case data at scale. (ii) Structured configuration data generation for a particuilar use case and domain. Enterprise applications such as Kubernetes use DSLs for configuration and usage, preparing them require deep domain expertise and there is increasing motivation (Pujar et al., 2023) to employ LLMs in enterprises to generate DSL code. (iii) Some more downstream tasks involving structured data such as forms and tables often represented in programmable format such as JSON can leverage LLMs to generate structured data to fill forms or tables leveraging the schema.

**Research Use Cases:** (i) Since DSLs are typically low resource languages, LLMs are often employed (Song et al., 2020) to synthesize data from LLMs to train and evaluate smaller-sized models. (ii) This task acts a as a seed for similar NLP use cases such as code translation.

#### A.5.2 DSL Validation Task

This section describes use cases from enterprise and research point of view motivating DSL validation seed task in our study.

**Enterprise Use Cases:** (i) Given the schema, employing LLMs to generate domain aware suggestions over the provided structured data which is not viable with traditional schema validators which only pinpoint syntactic errors and cannot provide semantic suggestions. Such as providing optimizations over the existing resource YAML in Kubernetes while complying with resource schema. (ii) In a assistive chat system, often the constraints are in NL representation from the user which are not machine readable and LLMs should be able to understand such constraints. (iii) Quick interoperability across different schema and data representation versions. Often in enterprises, schemas can be in a particular version not compatible with structured data's version. For instance, the schema could be in an older JSON schema version such as Draft 0 and data in Draft 7, in such cases LLMs can come handy to perform validation at scale.

**Research Use Case:** Understanding LLMs' capability in validating the given structured data against the schema across representations can provide seed evidence for more complex tasks such as automatic fixing of data in compliance with given schema.

### A.6 Schema Examples

This section provides schemas across 5 representations from Listings 7 to 11. All the schemas are equivalent in terms of constraints.

Listing 7: Sample schema using JSON Schema

```
{
    "type": "object",
    "properties": {
        "footbaths": {
            "type": "boolean"
        },
        "deluded": {
            "type": "null"
        },
        "bravadoing": {
            "type": "number",
```

		Output Representation					
		JSON		YAML		XML	
Model	Schema	IS (%)	RTV (%)	IS (%)	RTV (%)	IS (%)	RTV (%)
Llama3 8B		**1.9**	50.1	1.8	49.8	**1.6**	73.9
Granite 8B		2.9	31.0	2.8	57.3	17.1	**70.26**
Granite 20B	JSON	13.9	**15.6**	2.3	**38.0**	7.9	71.92
Granite 34B		2.6	23.5	2.6	48.6	4.1	73.08
Codellama 34B		3.6	17.9	**1.8**	51.4	3.7	71.12
Llama3 8B		12.9	64.1	6.1	52.8	**4.8**	73.5
Granite 8B		3.6	60.7	2.8	70.9	10.7	72.0
Granite 20B	XML	2.1	**53.3**	1.9	73.9	12.2	**70.5**
Granite 34B		**1.9**	56.9	1.6	63.1	10.6	71.9
Codellama 34B		2.3	71.2	1.6	56.9	10.2	71.7
Llama3 8B		1.3	53.3	3.1	62.4	0.4	74.5
Granite 8B		11.2	**13.7**	1.8	63.9	12.2	70.5
Granite 20B	YAML	**1.6**	39.8	1.4	56.6	10.7	72.0
Granite 34B		3.1	14.9	**1.1**	**40.6**	10.6	71.9
Codellama 34B		7.1	24.9	1.4	50.3	12.6	71.0
Llama3 8B		5.4	64.9	3.1	72.9	**3.1**	72.9
Granite 8B		2.4	73.0	2.3	70.9	10.7	72.71
Granite 20B	Python	**1.6**	64.7	2.4	68.7	16.6	71.42
Granite 34B		2.6	**61.2**	**2.4**	**66.9**	8.9	69.35
Codellama 34B		5.6	65.1	2.9	64.1	14.1	**69.1**
Llama3 8B		5.8	50.4	3.4	54.1	5.6	73.9
Granite 8B		**2.1**	28.9	2.6	29.2	8.3	69.24
Granite 20B	NL	2.9	0.6	2.8	30.2	7.97	69.24
Granite 34B		2.3	**1.9**	**2.4**	**8.9**	9.86	**63.42**
Codellama 34B		2.8	60.4	2.9	34.5	**7.88**	65.51

Table 5: Task 1 zero shot results having IS and RTV metric values. IS denotes the percentage of invalid samples and RTV denotes the percentage of sample root data type errors. For IS and RTV, the lesser the value better the performance.

		Output Representation		
		JSON	YAML	XML
Model	Schema	Macro-F1	Macro-F1	Macro-F1
Llama3 8B		0.55	0.37	0.40
Granite 8B		0.55	0.55	0.42
Granite 20B	JSON	0.48	0.37	0.47
Granite 34B		0.60	**0.56**	**0.63**
Codellama 34B		**0.64**	0.53	0.50
Llama3 8B		0.44	0.35	0.41
Granite 8B		0.45	0.44	0.50
Granite 20B	XML	0.24	0.45	**0.56**
Granite 34B		**0.52**	**0.47**	0.39
Codellama 34B		0.41	0.41	0.48
Llama3 8B		0.38	0.40	0.40
Granite 8B		0.45	0.50	0.44
Granite 20B	YAML	0.24	0.31	0.45
Granite 34B		0.52	**0.55**	0.47
Codellama 34B		**0.59**	0.52	**0.58**
Llama3 8B		0.37	0.36	0.38
Granite 8B		0.54	0.44	**0.54**
Granite 20B	Python	0.34	0.45	0.36
Granite 34B		**0.53**	**0.47**	0.40
Codellama 34B		0.48	0.45	0.46
Llama3 8B		**0.63**	**0.55**	0.57
Granite 8B		0.45	0.51	0.39
Granite 20B	NL	0.53	0.45	**0.57**
Granite 34B		0.45	0.46	0.38
Codellama 34B		0.52	0.54	0.42

Table 6: Task 2 zero shot Macro-F1 scores. Task 2 is a binary classification task.


```
        "exclusiveMaximum": 5.131849487240756
    },
    "queintise": {},
    "manucodia": {
        "type": "number"
    },
    "antagonized": {},
```

```
    "outbacker": {
        "type": "number"
    },
    "sphenotripsy": {
        "type": "boolean"
    },
    "hw": {
```

```
            "type": "null"
        }
    },
    "additionalProperties": true,
    "required": []
}
```

Listing 8: Sample schema using YAML

```
additionalProperties: true
properties:
  antagonized: {}
  bravadoing:
    exclusiveMaximum: 5.131849487240756
    type: number
  deluded:
    type: 'null'
  footbaths:
    type: boolean
  hw:
    type: 'null'
  manucodia:
    type: number
  outbacker:
    type: number
  queintise: {}
  sphenotripsy:
    type: boolean
required: []
type: object
```

Listing 9: Sample schema using Python

```
from pydantic import BaseModel, Field

class Schema(BaseModel):
    footbaths: bool
    deluded: None = Field(None, alias="null")
    bravadoing: float = Field(..., exclusive_maximum
        =5.131849487240756)
    queintise: None = {}
    manucodia: float
    antagonized: None = {}
    outbacker: float
    sphenotripsy: bool
    hw: None = Field(None, alias="null")
```

Listing 10: Sample schema using XML

```
<?xml version="1.0" ?>
<all>
        <type type="str">object</type>
        <properties type="dict">
                <footbaths type="dict">
                        <type type="str">boolean</
                            type>
                </footbaths>
                <deluded type="dict">
                        <type type="str">null</type>
                </deluded>
                <bravadoing type="dict">
                        <type type="str">number</
                            type>
                        <exclusiveMaximum type="
                            float">5.13184948724075
                            6</exclusiveMaximum>
                </bravadoing>
                <queintise type="dict"/>
                <manucodia type="dict">
                        <type type="str">number</
                            type>
                </manucodia>
                <antagonized type="dict"/>
                <outbacker type="dict">
                        <type type="str">number</
                            type>
                </outbacker>
                <sphenotripsy type="dict">
                        <type type="str">boolean</
                            type>
                </sphenotripsy>
                <hw type="dict">
                        <type type="str">null</type>
                </hw>
        </properties>
        <additionalProperties type="bool">true</
            additionalProperties>
        <required type="list"/>
</all>
```

Listing 11: Sample schema in NL

```
This is a JSON schema that defines the structure of
     an object. Here's a breakdown of the schema:

# **Top-level properties**

# * `type`: The type of the JSON data, which is an
     object (`"object"`).
# * `properties`: An object that defines the
     properties of the object.
# * `additionalProperties`: A boolean value that
     indicates whether additional properties not
     specified in the schema are allowed. In this
     case, it is set to True
* required: An empty array that specifies no
     properties are required in the object.

**Properties object**

The `properties` object defines the structure of
     each property in the object. Here's a brief
     description of each property:

footbaths: A boolean
deluded: A null
bravadoing: A number that must be strictly lesser
     than 5.131849487240756,
queintise: An object with no specific type or
     constraints.
manucodia: A number
antagonized: An object with no specific type or
     constraints.
outbacker: A number
sphenotripsy: A boolean
hw: A null
```

A.7 Hyperparameter details

We perform inference for all the models in $float16$ precision and a max new token limit of 1024 tokens. For beam search decoding, we use the beam width of 3.

A.8 Data Generation Scripts

A combinatorial data generation tool is created, which factors in constraints of interest, constraint-specific information, and combinatorial preferences, to generate the schemas. We also use openly available automatic lossless language to language translation tools for translation code from JSON to YAML etc. We plan to open-source all the scripts used for data preparation.