CODE4STRUCT: Code Generation for Few-Shot Event Structure Prediction

Xingyao Wang and **Sha Li** and **Heng Ji**
University of Illinois Urbana-Champaign, IL, USA
{xingyao6, shal2, hengji}@illinois.edu

Abstract

Large Language Model (LLM) trained on a mixture of text and code has demonstrated impressive capability in translating natural language (NL) into structured code. We observe that semantic structures can be conveniently translated into code and propose CODE4STRUCT to leverage such text-to-structure translation capability to tackle structured prediction tasks. As a case study, we formulate Event Argument Extraction (EAE) as converting text into event-argument structures that can be represented as a class object using code. This alignment between structures and code enables us to take advantage of Programming Language (PL) features such as inheritance[1] and type annotation[2] to introduce external knowledge or add constraints. We show that, with sufficient in-context examples, formulating EAE as a code generation problem is advantageous over using variants of text-based prompts. Despite only using 20 training event instances for each event type, CODE4STRUCT is comparable to supervised models trained on 4,202 instances and outperforms current state-of-the-art (SOTA) trained on 20-shot data by 29.5% absolute F1. By leveraging the inheritance feature of PL, CODE4STRUCT can use 10-shot training data from a sibling event type to predict arguments for zero-resource event types and outperforms the zero-shot baseline by 12% absolute F1. [3]

1 Introduction

Large Language Model (LLM) trained on massive corpora of code mixed with natural language (NL) comments and docstrings[4] (*e.g.*, Chen et al. 2021, Nijkamp et al. 2022) has demonstrated the ability to translate natural language instructions into structured code. We ask if this conversion between language and code can serve as a bridge to build a connection between language and semantic structure, which is the goal of many structured prediction tasks (*e.g.*, semantic parsing, information extraction) in Natural Language Processing (NLP). In particular, the target structure (*e.g.*, event-argument graph in Figure 1) can be mapped to code more straightforwardly compared to natural language, which often requires careful prompt engineering (Hsu et al. 2022, Li et al. 2021, Table 2). In addition, code written in programming languages has an inherent advantage in representing complex and

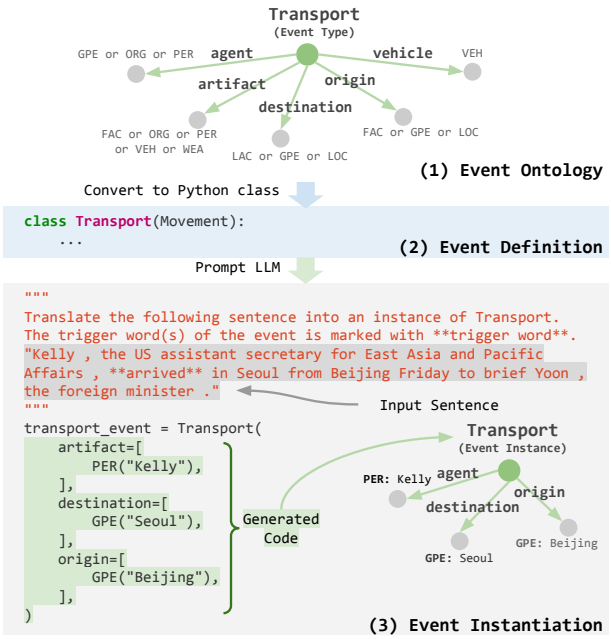


Figure 1: Event Argument Extraction using code generation. We convert the existing event type ontology to PYTHON class definitions. Conditioned on these definitions, we put the input sentence for event argument extraction into a docstring as the prompt for code generation. The generated code (colored in green) can be mapped to an instance graph of `Transport` event.

[1] Inheritance is a way to create a hierarchy of classes in PL. A child class can base upon another class, retaining similar implementation.

[2] Developers use type annotations to indicate the data types of variables and input/outputs of functions.

[3] All code and resources are publicly available at https://github.com/xingyaoww/code4struct.

[4] Text used to document a specific segment of code.

Event Argument Extraction	Programming Language (Python)
Event / Entity Type `Transport, VEH`	**Class definition** `class Transport, class VEH`
Hierarchical Event Ontology `Movement:Transport`	**Inheritance** Inheritance is a way to create a hierarchy of classes in PL. A child class can base upon another class, retaining similar implementation. `class Transport(Movement)`
Event Arguments `vehicle`	**Function arguments** `def function(vehicle=...)`

Argument Constraint
`Each argument can has a list of multiple entities; Argument vehicle should be entities of type VEH.`
	Type Annotation & Argument Default Value
	Type annotations are used by developers to indicate the data types of variables and input/outputs of functions. If a function is called without the argument, the argument gets its default value (a list in this case).
	`def function(`
	`  vehicle: List[VEH] = [], ...`
	`)`

Weakly-supervised Information
`Transport Event describes someone transporting something in a vehicle from one place to another place.`
	Docstring or Comments
	`class Transport(Movement):`
	`  """`
	`  self.agent transported self.artifact in self.vehicle vehicle from self.origin place to self.destination place.`
	`  """`

Table 1: Mapping between Event Argument Extraction requirements and features of Python programming language.

interdependent structures (Miller, 1981; Sebrechts and Gross, 1985) with features such as inheritance and type annotation.

As a case study, we showcase our proposed CODE4STRUCT on the Event Argument Extraction (EAE) task, which aims to extract event structures from unstructured text. EAE is the ideal testbed for our method due to the close alignment between EAE and PL as shown in Table 1. In CODE4STRUCT (Figure 1), we first translate the entity and event type ontology into Python class definitions. Conditioned on the relevant class definitions and the input sentence, we prompt an LLM to generate an instantiation of the event class, from which we can extract the predicted arguments.

By leveraging the alignment between PL and NLP problems, CODE4STRUCT enjoys various advantages as shown in Table 1. Using PL features like type annotation and argument default value, we can naturally enforce argument constraints for output structures. This allows CODE4STRUCT to handle multiple or zero argument fillers for the same argument role by annotating the expected type (*i.e.*, expect a list of entities) and setting the default value for each argument (*i.e.*, an empty list without any entity by default). Furthermore, we can naturally utilize the event hierarchy by leveraging inheritance. Inheritance allows a child event class (*e.g.*, `Transport`) to reuse most components of its parent class (*e.g.*, `Movement`) while preserving its unique properties. We demonstrate that hierarchical event types allow zero-resource event types to use annotated training examples from their high-resource sibling types (§4.6).

We outline our contributions as follows:

- We propose CODE4STRUCT to tackle structured prediction problems in NLP using code generation. As a case study, we use CODE4STRUCT for Event Argument Extraction (EAE).
- We perform extensive experiments contrasting the performance of code-based prompt and two variants of text prompt on different LLMs and show that code prompt is generally advantageous over text prompt when sufficient in-context examples are provided (§4.2).
- We demonstrate that 20-shot CODE4STRUCT rivals fully-supervised methods trained on 4,202 instances. CODE4STRUCT outperforms a SOTA approach by 29.5% absolute F1 gain when 20-shot data are given to both. 0-shot CODE4STRUCT can even outperform the SOTA on both 20 and 50 shots (§4.5).
- We show that integrating the event ontology hierarchy by class inheritance can improve prediction. Compared to the zero-shot baseline, we see 12% F1 gains for zero-resource event types when using 10-shot examples from their sibling event types (§4.6).

2 Code Generation Prompt Construction

In Event Argument Extraction (EAE) task, a model is provided with an event ontology and the tar-

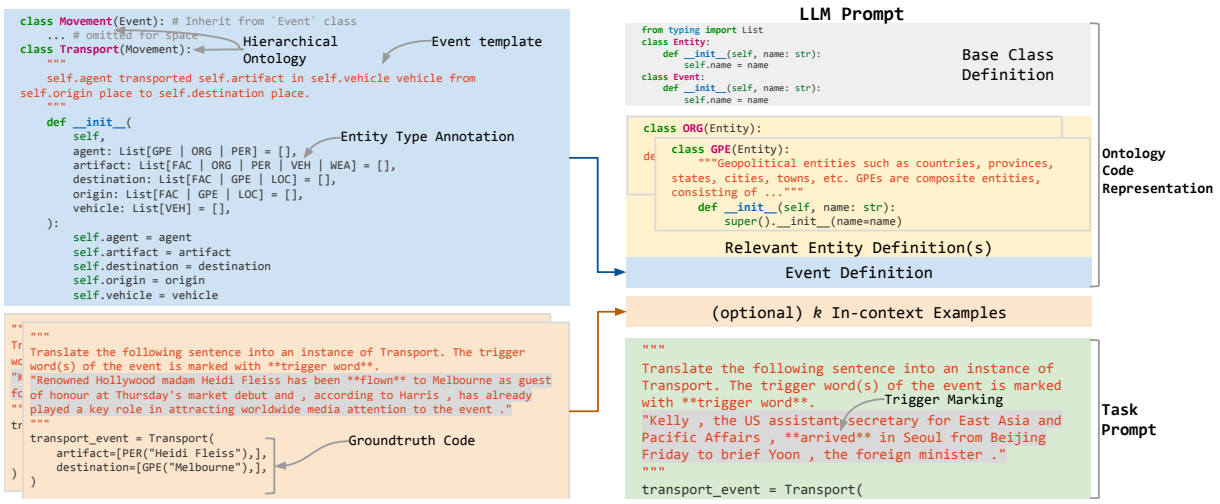


Figure 2: Prompt components. **(1)** Ontology code representation contains definitions of entity and event classes, colored in yellow and blue (§2.1). **(2)** k-shot examples for in-context learning, colored in orange (§2.3). **(3)** The task prompt, appended at the end with partial class instantiation for LLM completion, colored in green (§2.2).

get text to extract from. Similarly, we prompt an LLM with the ontology that consists of definitions of event types and argument roles, and input sentences to generate code that instantiates the given event type. We breakdown the input prompt into three components: (1) ontology code representation which consists of Python class definitions for entity types and an event type (§2.1); (2) optional k-shot in-context learning examples for the event type defined in (1) (§2.3); (3) task prompt for completion (§2.2). We show a breakdown of the full prompt in Figure 2.

2.1 Ontology Code Representation

To represent the event ontology as code, we concatenate the base class definition, entity class definitions, and event class definitions.

Base Class Definition We define base type `Entity` and `Event` to be inherited by other classes.

Entity Class Definition We use entity type definitions from the Automatic Content Extraction (ACE) program[5]. We construct Python classes that inherit from `Entity` and use the entity type as the class name (*e.g.*, `class GPE(Entity)`). We add a natural language description as a docstring of the defined class for each entity type.

2.1.1 Event Class Definition

We define the event class using the name of the event type (*e.g.*, `class Transport`). As ACE defines its event types in a hierarchical ontology, mimicking class definitions in Object-Oriented PL, we inherit the event class definition from its parent (*e.g.*, `class Transport(Movement)`) or root event type if the event class does not has a parent (*e.g.*, `class Movement(Event)`). An example of hierarchical event definition can be found in Figure A.9.

We define the argument roles (*e.g.*, destination of `Transport`) as input arguments of the constructor `__init__`[6]. We specify the type of each argument role using Python type annotation, a commonly used PL feature: For example, `agent: List[GPE | ORG | PER]` means that the `agent` argument accepts a list of entities which could be either of type GPE (Geo-Political Entity), ORG (Organization), or PER (Person). We assign each input argument (*e.g.*, `agent`) to a class member variable of the same name.

We include event description templates into the docstring of the class definition. The event description templates are modified from Li et al. (2021) by replacing each role with their corresponding member variable (*e.g.*, `self.agent`).

2.2 Task Prompt

The task prompt consists of a docstring describing the task and incomplete event instantiation code for

[5] https://www.ldc.upenn.edu/collaborations/past-projects/ace

[6] A constructor is a special function that initializes an instance of a class.

Prior Work	Language Template
DEGREE (Hsu et al., 2022)	**somebody** was moved to **somewhere** from **some place** by **some way**. **somebody** or **some organization** was responsible for the movement. **something** was sent to **somewhere** from **some place**. **somebody** or **some organization** was responsible for the **transport**.
BART-Gen (Li et al., 2021)	**<arg1> transported <arg2>** in **<arg3>** vehicle from **<arg4>** place to **<arg5>** place
Text2Event (Lu et al., 2021)	((**Transport** returned (Agent **<arg>**) (Artifact **<arg>**) (Destination **<arg>**) (Origin **<arg>**) (Vehicle **<arg>**))

Table 2: Example of language templates for Event Argument Extraction used by Hsu et al. (2022); Li et al. (2021); Lu et al. (2021).

completion. An example of a task prompt can be found in Figure 2. The text-based docstring contains a task instruction and an input sentence. We mark the ground truth trigger words for the input text by surrounding them with `**`. We choose to use `**` as it is used to set text to bold in `Markdown` (a markup language for creating formatted text), which is commonly found in code bases and web data on which our LLM is trained. The incomplete code prompt assigns a partial instantiation of an event class to a variable to trigger the model for completion, for example, `transport_event = Transport(`.

We observed that LLM tends to generate additional sentences paired with extracted arguments if no stopping constraint is applied. To focus on the given EAE task, we stop the code generation whenever any of the following patterns is generated by the model: `"""`, `class`, `print`, or `#`.

2.3 In-context Learning

Optionally, we can include in-context learning examples, which are task prompts (§2.2) paired with completed event instantiations using ground-truth arguments (see Figure 2 for a specific example). For k-shot learning, we concatenate k such examples together. Given a task prompt, we deterministically gather k learning examples by collecting training instances with the same event type, following the order of occurrences in the training set.

3 Why Represent Event Structure in PL?

A wide range of NLP tasks have benefited from LLM (Brown et al., 2020; Hoffmann et al., 2022; Chowdhery et al., 2022) trained on web-scale language corpora. To effectively use LLM trained on language for EAE, one of the biggest challenges is to specify the desired output, namely event structures in our case, using natural language.

There is a tradeoff between the effort put into defining the output or designing the prompt (*e.g.*, Text2Event in Table 2) and the benefit from pre-training in natural language (*e.g.*, DEGREE and BART-Gen in Table 2). Text2Event (Lu et al., 2021) resides at one end of the spectrum with a concise but unnatural output format. As a result, this formulation under-utilizes the pretraining power of the model and does not work in low-resource settings as shown in Table 4. Towards the other end, Hsu et al. (2022); Li et al. (2021) design manual templates for the model to fill in. We also design two variants of language prompt as shown in Figure A.5 and A.6 miciking our code prompt and BART-Gen style prompt for comparison. Note that these natural language prompts are much more verbose and, as shown in §4.2, usually result in sub-optimal performance with sufficient in-context examples.

Essentially, this tradeoff is a result of the mismatch between the pretraining corpora and task output formats. Instead of using LLM trained on only unstructured text, we turn to LLM trained with a mixture of text and code, where the text is often aligned in semantics with the accompanying code. Such Code-LLMs have the ability to convert text into corresponding code as demonstrated by (Chen et al., 2021; Nijkamp et al., 2022). Then we can map the desired output event structure into code in a straightforward manner and leverage the full pretraining power of these models. PLs like Python offer features (*e.g.*, class, docstrings, type annotations, inheritance) that have a significant presence in the pre-training corpus of Code-LLM due to frequent usage. CODE4STRUCT leverages these features to succinctly describe event structures, which makes it better aligned with Code-LLM. By leveraging LLM's learned knowledge from diverse pre-training domains, CODE4STRUCT can work well in open-domain, achieving non-trivial zero-shot performance given unseen event types (§4.5). CODE4STRUCT is also data-efficient as exemplified by reaching comparable performance to fully-supervised methods with much fewer annotated examples (20 per event type) (§4.5).

4 Experiments

4.1 Experiment Setup

LLM We use CODEX `code-davinci-002` (Chen et al., 2021), a GPT-3 (Brown et al., 2020) model finetuned on code, which supports up to 8k input tokens. We compare its performance with InstructGPT (Ouyang et al., 2022) `text-davinci-002` and its improved version `text-davinci-003`, both support up to 4k input tokens. We access these LLMs through OpenAI API[7].

Hyperparameters We prompt LLM to generate code that instantiates an event using sampling temperature $t = 0$ (*i.e.*, greedy decoding). We set the max number of new tokens for each generation to 128, which fits all code outputs for the test set.

Evaluation Tasks We use ground truth event type and gold-standard trigger words to perform Event Argument Extraction.

Dataset We evaluate our performance of EAE on the English subset of Automatic Content Extraction 2005 dataset (ACE05-E)[8] (Doddington et al., 2004). We follow Wadden et al. (2019); Lin et al. (2020) for dataset processing. ACE05-E has hierarchical event types with 8 parent types and 33 child types. Among all child types, roughly half of the event types (14 out of 33) in ACE05-E have less than 50 event instances in the training set. We show statistics for each event type in Table A.4.

Evaluation metrics We use **Argument F1-score** following prior work (Ji and Grishman, 2008; Li et al., 2021; Hsu et al., 2022): We consider an argument to be correctly identified when the head word span of predicted text[9] matches that of the human-annotated text (denoted as **Arg-I**); We consider an argument to be correctly classified if the role (*e.g.*, `agent`) of a *correctly identified* argument matches that of the human annotation (denoted as **Arg-C**).

4.2 Comparison with Text Prompt

To compare our code-based prompt with text-based prompts, we design two variants of text prompt: $T^{(1)}$ mimicking our code prompt (*i.e.*, code imitation, Figure A.5) and $T^{(2)}$ following BART-Gen style prompt (Li et al., 2021) (Figure A.6) which resembles natural language more compared to $T^{(1)}$. Both text prompts have similar components as our code-based prompt in Figure 2. Text prompts rely on natural language to define the requirement and format of the desired output, while the code prompt utilizes PL syntax. We compare the F1 score difference between the code prompt (§2) and two variants of text prompts (*i.e.*, $\Delta^{(i)}_{C-T} = \text{F1}_{\text{code}} - \text{F1}^{(i)}_{\text{text}}, i \in \{1, 2\}$) on different LLMs in Table 3. We include exact performance numbers of text prompts in Table A.3. We summarize our findings as follows:

- Code prompt outperforms both text prompts on Arg-C F1 (*i.e.*, $\Delta^{(i)}_{C-T} > 0$) for two text prompt variants and all LLMs except `text-davinci-003` when sufficient in-context examples are given (*i.e.*, $k \geq 5$).
- For `*-davinci-002` LLMs, there are more significant performance gains from using a code prompt (*i.e.*, increasing $\Delta^{(i)}_{C-T}$ for all i) when the number of in-context examples k increases (for $k \geq 5$).
- There is no clear trend on Arg-I F1 to differentiate code and text prompts, except for `text-davinci-003`, which exhibits similar behavior that code prompt performs better with larger k.
- Text prompt $T^{(2)}$ (BART-Gen style), which resembles natural language more, performs poorly under low-shot ($k \leq 1$), primarily due to the LLM being unable to produce the desired structure output described using language in $T^{(2)}$, causing the low-shot code-text performance gap $\Delta^{(2)}_{C-T}$ to be larger compared to $T^{(1)}$. These low-shot performance differences between $T^{(1)}$ and $T^{(2)}$ further signify the need to prompt engineering for language-based prompts to work well in a low-shot setting.

4.3 Comparison with different LLM

We measure the performance of the same CODE4STRUCT code prompt across different foundational LLMs in §4.1. LLM performance comparison can be found in Figure 3. `text-davinci-002` is an InstructGPT

[7] https://openai.com/api/

[8] https://www.ldc.upenn.edu/collaborations/past-projects/ace

[9] We find the span of predicted text in the given sentence, then use `spacy` library to find its head word.

Table 3: Performance of the code prompt on the Arg-I and Arg-C metrics and its F1 score difference $\Delta^{(i)}_{C-T}$ with two text prompt variants described in §4.2 (*i.e.*, $\text{F1}_{\text{code}} - \text{F1}^{(i)}_{\text{text}}$). On Arg-C, there is a trend that the code prompt performs better (*i.e.*, $\Delta^{(i)}_{C-T} > 0$) when more in-context examples are provided, except on `text-davinci-003`.

Model	code-davinci-002						text-davinci-002						text-davinci-003					
k-shot	Arg-I	$\Delta^{(1)}_{C-T}$	$\Delta^{(2)}_{C-T}$	Arg-C	$\Delta^{(1)}_{C-T}$	$\Delta^{(2)}_{C-T}$	Arg-I	$\Delta^{(1)}_{C-T}$	$\Delta^{(2)}_{C-T}$	Arg-C	$\Delta^{(1)}_{C-T}$	$\Delta^{(2)}_{C-T}$	Arg-I	$\Delta^{(1)}_{C-T}$	$\Delta^{(2)}_{C-T}$	Arg-C	$\Delta^{(1)}_{C-T}$	$\Delta^{(2)}_{C-T}$
0	50.6	0.7	50.6	36.0	-2.2	36.0	48.9	-2.6	20.2	35.0	-2.4	13.1	49.9	-2.1	15.3	37.8	-1.4	12.6
1	57.3	0.1	4.7	47.8	-1.0	4.7	55.8	1.8	5.3	45.2	3.0	4.9	56.0	-1.5	1.1	44.7	-3.2	1.1
5	58.0	1.1	1.9	52.5	2.9	1.1	56.0	-2.0	1.0	48.8	3.0	1.4	59.2	-0.9	-0.7	51.7	1.4	-2.1
10	57.2	-1.4	-0.2	52.8	0.8	0.1	60.6	2.7	2.9	53.9	6.4	5.0	62.8	3.1	0.6	56.3	5.0	-1.2
20	62.1	1.7	0.2	58.5	3.6	2.4	59.9	0.9	3.7	56.5	8.0	5.8	65.0	3.5	0.7	60.4	7.8	-0.4

(Ouyang et al., 2022) model finetuned with human demonstrations based on `code-davinci-002`, yet these two LLMs perform similarly in Arg-C F1. Although having a similar code prompt Arg-C performance, `text-davinci-002` generally has a larger $\Delta^{(i)}_{C-T}$ compared to `code-davinci-002` of the same k in Table 3 (*e.g.*, +3.6 vs. +8.0, +2.4 vs. +5.8 on 20-shot for both text prompt variants), suggesting the degradation of text prompt performance after finetuning with human demonstrations.

`text-davinci-003`, which uses reinforcement learning (RL) with reward models to align with human preference[10] (Ouyang et al., 2022), outperforms other LLMs for $k > 5$. In Table 3, `text-davinci-003` obtains superior Arg-C F1 performance (60.4% vs. 56.5% on 20-shot) compared to `text-davinci-002`. This suggests RL with reward models effectively improves EAE performance (*i.e.*, Arg-C) on code prompt.

Interestingly, `text-davinci-003` has a very different $\Delta^{(i)}_{C-T}$ pattern for text prompt $T^{(2)}$ compared to $T^{(1)}$. Like `text-davinci-002`, in Table 3, Arg-C $\Delta^{(1)}_{C-T}$ for text prompt $T^{(1)}$ has an increasing trend with a similar magnitude (*e.g.*, +7.8 vs. +8.0 on 20-shot). That is, in both LLMs, the code prompt is always better than text prompt $T^{(1)}$ with $k \geq 5$. However, for text prompt $T^{(2)}$ which is more similar to natural sentences, the gap $\Delta^{(2)}_{C-T}$ exhibits a vastly different pattern compared to other models: code prompt performs on par or even slightly worse than $T^{(2)}$ for $k \geq 5$. We also notice that for zero-shot prediction, $T^{(2)}$ on `text-davinci-003` performs better compared to other LLMs. This indicates that aligning LLM with RL and reward models helps improve LLM's ability to follow zero-shot language instructions.

Even though code prompt still performs superior to both text prompt variants on `002` LLMs, results from `text-davinci-003` suggest a better-aligned language model can perform equally well on a natural text prompt $T^{(2)}$ when sufficient in-context examples are provided.

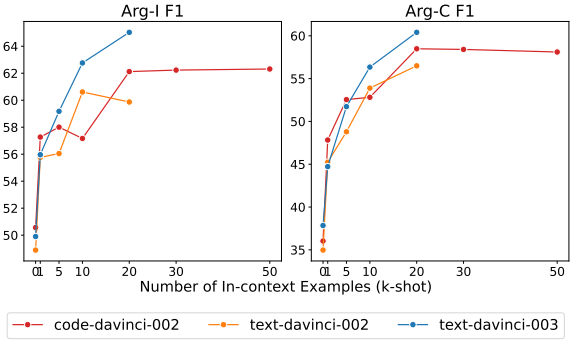


Figure 3: CODE4STRUCT performance (in F1%) with different k. We observe improvements with diminishing returns when we increase the number of in-context examples. Exact performance numbers can be found in Table A.3 (code prompt). We stop at $k = 20$ for `text-davinci` and $k = 50$ for `code-davinci-002` as including more examples would exceed the input length limitation imposed by corresponding LLM.

4.4 Comparison with different k

We examine the performance of code prompts with varying numbers of examples in Figure 3. We observe that F1 scores for all metrics generally increase with diminishing returns when providing more in-context learning examples. The initial in-context example ($k = 1$) brings the largest absolute performance gain (+11.8, +10.2, +6.9 Arg-C F1 for three LLMs). For $k \geq 20$ on `code-davinci-002`, the Arg-I and Arg-C performance plateaus or even slightly degrade, as not all event types have enough in-context examples to benefit from increasing k (*i.e.*, only 19 out of 33 event types have more than 50 examples for in-context learning). To further investigate why the performance plateaus, we analyze how the sentence variability (or diversity) of in-context examples influences Arg-C performance in §A.4; We find that

[10] https://beta.openai.com/docs/model-index-for-researchers

Arg-C performance is positively correlated with the variability of in-context examples which plateaus as k increases, hinting that in-context learning performance may eventually plateau with increasing k due to little variability gains from the additional data.

4.5 Comparison with Supervised Models

Baselines Unlike prior methods trained on the entire training set, CODE4STRUCT learns from up to 50 examples (*i.e.*, 39 examples per event type on average, roughly 1% among all training instances) to predict arguments for each test event type. To ensure a fair comparison, for each event type *t* in the test set, we train a Text2Event model (Lu et al., 2021) and a DEGREE model (SOTA, Hsu et al. (2022)) on 20-shot and 50-shot in-context examples CODE4STRUCT used while providing gold-standard trigger words. We evaluate both models trained on event type *t* on a partition of the test set that only contains instances of event type *t*. We then aggregate F1 scores (micro F1) across all 31 event types on the test set and report them in Table 4. Following Hsu et al. (2022), we also compare with classification-based (DyGIE++ Wadden et al. (2019), BERT_QA Du and Cardie (2020), OneIE Lin et al. (2020)) or generation-based (TANL (Paolini et al., 2021), BART-Gen Li et al. (2021), DEGREE Hsu et al. (2022)) models trained on the full training set.

Results We report the performance of CODE4STRUCT using LLMs (§4.1) in comparison with prior work in Table 4. We report the performance of supervised models using the full dataset from Hsu et al. (2022). Note that 50-shot results for `text-davinci` are not available as the 50-shot input prompt will exceed LLM's input token length limitation, hence we use `code-davinci-002` for 50-shot comparison.

In the few-shot setting, 20-shot CODE4STRUCT using `text-davinci-003` can surpass DEGREE (Hsu et al., 2022), the current state-of-the-art, by a large margin (+29.5% Arg-C F1). Our zero-shot CODE4STRUCT using the best-performing `text-davinci-003` model can already achieve higher Arg-I and Arg-C performance than the 20-shot and 50-shot DEGREE. Despite only learning from 20 examples, 20-shot CODE4STRUCT achieves comparable performance with other fully-supervised models trained on 100% of the training data (4,202 instances).

Model	Data	Arg-I F1	Arg-C F1
DyGIE++	Full	66.2	60.7
BERT-QA	Full	68.2	65.4
OneIE	Full	73.2	69.3
TANL	Full	65.9	61.0
BART-Gen	Full	69.9	66.7
DEGREE	Full	**76.0**	**73.5**
CODE4STRUCT$_{\texttt{text-davinci-003}}$	0-shot	49.9	37.8
Text2Event	20-shot*	23.1	19.1
DEGREE	20-shot*	33.0	30.9
CODE4STRUCT$_{\texttt{text-davinci-003}}$	20-shot*	**65.0**	**60.4**
Text2Event	50-shot*	30.6	26.0
DEGREE	50-shot*	40.8	37.3
CODE4STRUCT$_{\texttt{code-davinci-002}}$	50-shot*	**62.3**	**58.1**

Table 4: Performance (in F1%) comparison between best-performing CODE4STRUCT LLM and existing supervised approaches. Performance numbers for all LLMs can be found in Table A.3. *Some event types do not have 20 or 50 examples for in-context learning; on average, we have 39 examples per type for a 50-shot prompt and 18 examples per type for 20-shot.

4.6 Event Type Hierarchy Improves Zero-resource EAE

In this section, we show that CODE4STRUCT, when provided with hierarchical event definitions and few-shot training instances D_{e_s} from a sibling event type $e_s \in \text{Siblings}(e)$ under the same parent event type, can improve performance for child event type e as good as if training instances D_e from the same event type e were used. This allows zero-resource event types without annotated data to exploit the event type hierarchy and benefit from their high-resource siblings. We include an example task prompt with sibling examples in Figure A.11 and report our results in Table 5.

Setup We split the child types for each parent type into training and testing types by selecting the high-resource child type with the largest amount of training instances to be the training type and have the rest be testing types. The train-test split for ACE types can be found in Table A.5. Under the same parent event type, we use data instances from the training type (*i.e.*, a sibling of testing types) as in-context examples to predict arguments for each testing type. We include event class definition (Figure 2) for parent event type (*e.g.*, `Transaction`), child training (sibling) event type (*e.g.*, `Transfer_Money`), and child testing event type (*e.g.*, `Transfer_Ownership`). We show an example of event definition with sibling type in Figure A.10. The few-shot performance when using data from a sibling type D_{e_s} is denoted with *(sibling type)* in Table 5. To demonstrate the

effectiveness of using data from sibling event types, we compare it with using training instances from the testing event type itself D_e (denoted as *(same type)*) and from a random non-sibling event type (denoted as *(non-sibling type)*).

	Arg-I	Arg-C
0-shot	52.8	42.9
1-shot (same type)	54.3	50.2
1-shot (sibling type)	**57.2**	**51.9**
1-shot (non-sibling type)	56.3	50.3
10-shot (same type)	58.7	**55.2**
10-shot (sibling type)	**60.8**	54.9
10-shot (non-sibling type)	58.5	51.0

Table 5: `code-davinci-002` performance (in F1%) when using examples from the same, sibling or non-sibling event types for in-context learning. To ensure a fair comparison, F1 scores are aggregated from 23 test event types in Table A.5 that contains more than 10 training instances.

Results We observe that CODE4STRUCT, when prompted with training examples from sibling type, performs on par with the prompt that uses training examples from the testing type itself on 1-shot and 10-shot. The substantial performance gain (+9% Arg-C F1 on 1-shot, +12% Arg-C F1 on 10-shot, compared with 0-shot) contributed by sibling-type training examples demonstrate the potential of applying CODE4STRUCT to zero-resource event types with no training data by exploiting their hierarchical relationship with other high-resource event types. Surprisingly, similar to the observation made by Min et al. (2022), using in-context examples from a random non-sibling type also benefits CODE4STRUCT performance, albeit not as helpful as sibling examples under 10-shot.

5 Related Work

Code-LLM for Structured Task Sun et al. (2019); Singh et al. (2022) focus on procedural tasks that aim to control situated agents in an embodied environment by representing the procedure plan in code. Madaan et al. (2022) uses Code-LLM to generate a structured commonsense reasoning graph represented in code, which is similar in spirit to our work but in a different task. Gao et al. (2022) tackles math and symbolic reasoning tasks by decomposing the natural language problem into runnable steps using Code-LLM and delegating solution calculation to a PL interpreter. We leverage PL features (*e.g.*, inheritance, type annotation) to introduce extra information and constraints for structured prediction, which is largely overlooked by prior work.

Event Extraction Li et al. (2013); Nguyen et al. (2016); Yang and Mitchell (2016); Wadden et al. (2019); Lin et al. (2020) use classification models and mitigate error propagation from pipeline models by leveraging global features to predict event triggers and arguments jointly. Recent work such as Liu et al. (2020) formulates event extraction as a reading comprehension problem and Li et al. (2021); Huang et al. (2021); Paolini et al. (2021); Hsu et al. (2022) converts event extraction to a text generation task to better exploit label semantics from pretrained language models. The most similar work to ours is Text2Event (Lu et al., 2021), which uses controlled generation to generate structures in a manually specified linearized format directly, hindering the model in leveraging pre-trained NL knowledge. On the other hand, our approach CODE4STRUCT directly generates structure in PL instead of using a manually designed format to fully exploit LLM's knowledge of PL.

6 Conclusions and Future Work

We propose CODE4STRUCT for structured prediction tasks in NLP by leveraging LLMs trained on language and code. As a case study, we use CODE4STRUCT to extract event arguments from natural language sentences through code generation. We show that, with sufficient in-context examples, formulating EAE as a code generation problem is advantageous over using text-based prompts. Our proposed CODE4STRUCT rivals fully-supervised models trained on 4,202 data instances only using 20-shot. It also outperforms a SOTA model by 29.5% absolute F1 when both are given the same 20-shot data. Furthermore, benefitting from hierarchical event definitions, CODE4STRUCT can predict arguments for zero-resource event types only using 10-shot training instances from its sibling event type and outperforms 0-shot baseline by 12% absolute F1 score. Going forward, we plan to expand CODE4STRUCT to a broader range of more complex structured prediction tasks (*e.g.*, relation prediction, schema matching). We would further explore the executable nature of PL to improve LLM's ability for structured prediction.

Limitations

In this work, our approach assumes event triggers and argument templates (*i.e.*, ontology) are given. This limits our approach's applicability, as it requires an event detection system to produce event triggers and event types before LLMs can be prompted to generate event arguments.

We only explore hierarchical events with only 2 levels from the ACE05-E ontology and data, which has limited coverage of real-world complex event hierarchy. Similar to prior event argument extraction work, our approach relies on a human-curated hierarchical ontology. We leave automatically discover hierarchical ontology for future work.

Despite LLMs performing well on EAE with few-shot data, compared to existing supervised approaches, their inference is relatively slow and costly[11] since the LLMs we used are generally more than 100x larger in the number of parameters. Prior work (Zhao et al., 2021; Lu et al., 2022) has demonstrated a strong relationship between performance and in-context demonstrations; however, for ease of comparison to supervised baselines, we use the same set of examples from the training set for in-context learning. We expect better selecting (Liu et al., 2021) and ordering (Lu et al., 2022) in-context examples can benefit CODE4STRUCT performance, which we leave for future work.

Ethical Considerations

Since event argument extraction only requires predicting arguments from the given text, the risk of generating toxic languages is relatively low as long as the given test is not toxic. This is because the prediction can be grounded in the input sentence, eliminating potential toxic tokens that did not appear in the original sentence. However, discrimination and bias are possible, as observed in the foundational LLMs we used (Brown et al., 2020; Chen et al., 2021; Ouyang et al., 2022), which we refer to Brown et al. (2020) for detailed discussion.

Acknowledgement

We thank the anonymous reviewers for their helpful suggestions and comments. This research is based upon work supported by U.S. DARPA KAIROS Program No. FA8750-19-2-1004, U.S. DARPA AIDA Program No. FA8750-18-2-0014 and U.S. DARPA ITM FA8650-23-C-7316. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of DARPA, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation therein.

[11] We perform most of our experiments on `code-davinci-002` which is in free public beta at the time of the experiment. For `text-davinci` models, around 700 USD was used to access its API to perform relevant experiments in this paper.

References

Laura Banarescu, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer, and Nathan Schneider. 2013. Abstract Meaning Representation for sembanking. In *Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse*, pages 178–186, Sofia, Bulgaria. Association for Computational Linguistics.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2022. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*.

George Doddington, Alexis Mitchell, Mark Przybocki, Lance Ramshaw, Stephanie Strassel, and Ralph Weischedel. 2004. The automatic content extraction (ACE) program – tasks, data, and evaluation. In *Proceedings of the Fourth International Conference on Language Resources and Evaluation (LREC'04)*, Lisbon, Portugal. European Language Resources Association (ELRA).

Xinya Du and Claire Cardie. 2020. Event extraction by answering (almost) natural questions. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 671–683, Online. Association for Computational Linguistics.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2022. Pal: Program-aided language models. *ArXiv*, abs/2211.10435.

Maria Halkidi, Yannis Batistakis, and Michalis Vazirgiannis. 2001. On clustering validation techniques. *Journal of Intelligent Information Systems*, 17:107–145.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. 2022. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*.

I-Hung Hsu, Kuan-Hao Huang, Elizabeth Boschee, Scott Miller, Prem Natarajan, Kai-Wei Chang, and Nanyun Peng. 2022. DEGREE: A data-efficient generation-based event extraction model. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1890–1908, Seattle, United States. Association for Computational Linguistics.

Kung-Hsiang Huang, Sam Tang, and Nanyun Peng. 2021. Document-level entity-based extraction as template generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 5257–5269, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Heng Ji and Ralph Grishman. 2008. Refining event extraction through unsupervised cross-document inference. In *In Proceedings of the Annual Meeting of the Association of Computational Linguistics (ACL 2008). Ohio, USA*.

Qi Li, Heng Ji, and Liang Huang. 2013. Joint event extraction via structured prediction with global features. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 73–82, Sofia, Bulgaria. Association for Computational Linguistics.

Sha Li, Heng Ji, and Jiawei Han. 2021. Document-level event argument extraction by conditional generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 894–908, Online. Association for Computational Linguistics.

Ying Lin, Heng Ji, Fei Huang, and Lingfei Wu. 2020. A joint neural model for information extraction with global features. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7999–8009, Online. Association for Computational Linguistics.

Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weizhu Chen. 2021. What makes good in-context examples for gpt-3? In *Workshop on Knowledge Extraction and Integration for Deep Learning Architectures; Deep Learning Inside Out*.

Jian Liu, Yubo Chen, Kang Liu, Wei Bi, and Xiaojiang Liu. 2020. Event extraction as machine reading comprehension. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1641–1651, Online. Association for Computational Linguistics.

Yao Lu, Max Bartolo, Alastair Moore, Sebastian Riedel, and Pontus Stenetorp. 2022. Fantastically ordered prompts and where to find them: Overcoming few-shot prompt order sensitivity. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8086–8098, Dublin, Ireland. Association for Computational Linguistics.

Yaojie Lu, Hongyu Lin, Jin Xu, Xianpei Han, Jialong Tang, Annan Li, Le Sun, Meng Liao, and Shaoyi Chen. 2021. Text2Event: Controllable sequence-to-structure generation for end-to-end event extraction. 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 2795–2806, Online. Association for Computational Linguistics.

Aman Madaan, Shuyan Zhou, Uri Alon, Yiming Yang, and Graham Neubig. 2022. Language models of code are few-shot commonsense learners. *arXiv preprint arXiv:2210.07128*.

L. A. Miller. 1981. Natural language programming: Styles, strategies, and contrasts. *IBM Systems Journal*, 20(2):184–215.

Sewon Min, Xinxi Lyu, Ari Holtzman, Mikel Artetxe, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. 2022. Rethinking the role of demonstrations: What makes in-context learning work? *ArXiv*, abs/2202.12837.

Thien Huu Nguyen, Kyunghyun Cho, and Ralph Grishman. 2016. Joint event extraction via recurrent neural networks. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 300–309, San Diego, California. Association for Computational Linguistics.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. A conversational paradigm for program synthesis. *arXiv preprint*.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke E. Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Francis Christiano, Jan Leike, and Ryan J. Lowe. 2022. Training language models to follow instructions with human feedback. *ArXiv*, abs/2203.02155.

Giovanni Paolini, Ben Athiwaratkun, Jason Krone, Jie Ma, Alessandro Achille, Rishita Anubhai, Cicero Nogueira dos Santos, Bing Xiang, and Stefano Soatto. 2021. Structured prediction as translation between augmented natural languages. *arXiv preprint arXiv:2101.05779*.

Marc M. Sebrechts and Paul Gross. 1985. Programming in natural language: A descriptive analysis. *Behavior Research Methods, Instruments, & Computers*, 17:268–274.

Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. 2022. ProgPrompt: Generating situated robot task plans using large language models.

Shao-Hua Sun, Te-Lin Wu, and Joseph J Lim. 2019. Program guided agent. In *International Conference on Learning Representations*.

David Wadden, Ulme Wennberg, Yi Luan, and Hannaneh Hajishirzi. 2019. Entity, relation, and event extraction with contextualized span representations. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 5784–5789, Hong Kong, China. Association for Computational Linguistics.

Bishan Yang and Tom M. Mitchell. 2016. Joint extraction of events and entities within a document context. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 289–299, San Diego, California. Association for Computational Linguistics.

Zixuan Zhang and Heng Ji. 2021. Abstract Meaning Representation guided graph encoding and decoding for joint information extraction. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 39–49, Online. Association for Computational Linguistics.

Tony Zhao, Eric Wallace, Shi Feng, Dan Klein, and Sameer Singh. 2021. Calibrate before use: Improving few-shot performance of language models. *ArXiv*, abs/2102.09690.

A Appendix

A.1 Qualitative Analysis

We show examples of 0-shot and 50-shot CODE4STRUCT argument extraction result in Figure A.1. CODE4STRUCT can leverage implicit commonsense knowledge in LLM to infer arguments not presented in the text. In the first 0-shot example, the model inferred the `place` of Welch's retirement is in the *United States*. This is a reasonable guess since Welch, in this example, is the former CEO of General Electric (GE), whose headquarter is in the United States. In the second 0-shot example, our model inferred that the `Justice:Fine` event should take `place` in a *court*, which matches our commonsense knowledge. Interestingly, we observe that increasing the number of in-context examples from 0-shot to 50-shot inhibits LLM from generating arguments (*i.e.*, making LLMs more conservative), including these inferred arguments and a correctly predicted argument (*i.e.*, `SEC`) in 0-shot predictions.

A.2 Prompt Component Analysis

In this section, we present an empirical analysis of other prompt component candidates. We compare different prompt components in Table A.1 using `code-davinci-002` and following the same hyper-parameters described in §4.1.

- **Event Keywords** We augment event-related keywords into the docstring of event definition for CODE4STRUCT (illustrated in Figure A.8). We follow the same keywords used by Li et al. (2021).

- **AMR** Zhang and Ji (2021) have demonstrated the effectiveness of utilizing Abstract Meaning Representation (AMR) (Banarescu et al., 2013) for information extraction. We experiment with AMR-augmented prompts. We use `armlib` [12] to predict AMR, and append the AMR structure after the NL sentence in the task prompt §2.2 (see Figure A.7 for an example).

Prompts that include event keywords and AMR all perform slightly better than CODE4STRUCT under the zero-shot setting on all metrics (Table A.1).

[12] https://github.com/bjascob/amrlib, parse_xfm_bart_large v0.1.0

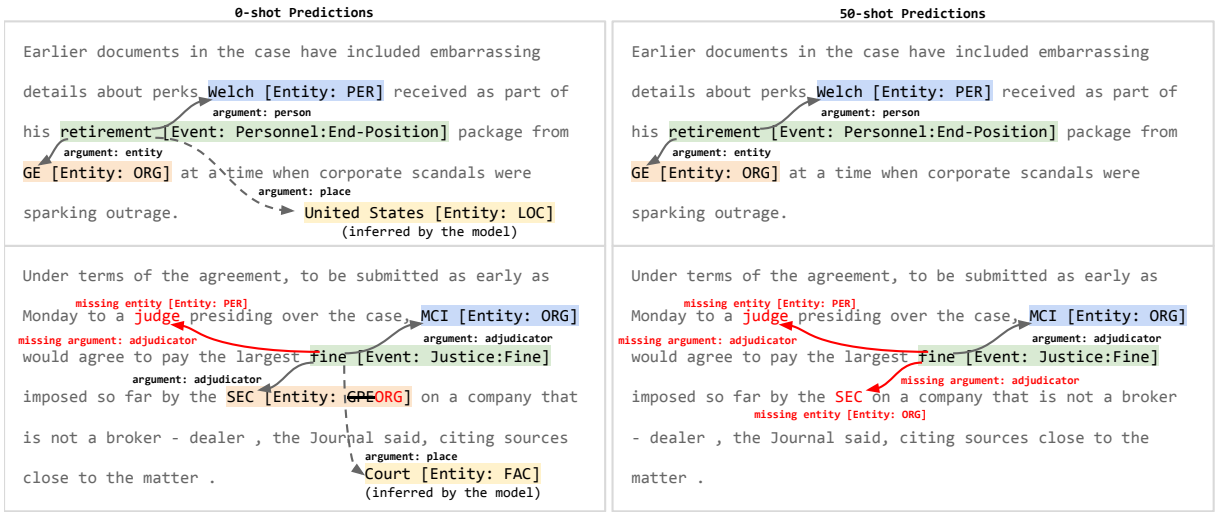


Figure A.1: Examples of 0-shot and 50-shot CODE4STRUCT event argument prediction using `code-davinci-002` on ACE05-E. In both 0-shot examples, LLM can infer an entity that does not present in the text as an argument (marked with a yellow span). CODE4STRUCT predicts fewer arguments when the examples are increased to 50-shot. We mark incorrect predictions with strikethrough text. Entities that LLM failed to predict are marked in red font.

	Arg-I F1					**Arg-C F1**				
k-shot	0	1	10	20	50	0	1	10	20	50
CODE4STRUCT	50.6	**57.3**	57.2	**62.1**	**62.3**	36.0	**47.8**	52.8	**58.5**	**58.1**
+ amr	51.1	54.7	55.6	-	-	**37.2**	44.2	51.3	-	-
+ keywords	**52.3**	**57.3**	**58.0**	61.7	61.7	36.4	47.3	**53.5**	57.7	57.9

Table A.1: Prompt components analysis on `code-davinci-002`. The best scores (in %) are bolded. - means the result is unavailable due to the input prompt exceeding the corresponding LLM's supported input token length.

	Arg-I F1					**Arg-C F1**				
k-shot	0	1	10	20	50	0	1	10	20	50
CODE4STRUCT	50.6	**57.3**	57.2	**62.1**	**62.3**	36.0	47.8	52.8	**58.5**	**58.1**
- trigger	48.8	54.4	53.0	57.6	56.6	33.8	44.1	48.9	53.8	51.5
- description	**51.4**	56.7	56.2	61.1	61.6	**36.1**	47.2	51.6	57.1	57.8
- type annotation	49.4	57.2	**58.0**	61.5	61.4	35.7	**48.0**	**54.5**	57.6	57.5
- hierarchy	49.4	56.6	55.5	59.9	60.4	34.3	46.8	50.0	55.4	55.9

Table A.2: Ablation study on `code-davinci-002`. The best scores (in %) are bolded. - means the result is unavailable due to the input prompt exceeding the corresponding LLM's supported input token length.

A.3 Ablation Study

In Table A.2, we ablate different prompt components described in §2, including event trigger marking, event description in natural language, type annotation, and hierarchical ontology. We perform this ablation study using `code-davinci-002`.

Event Trigger Marking We find that removing event trigger marking consistently degrades performance on all metrics over varying numbers of in-context examples.

Event Description Event descriptions generally provide a small F1 gain under the few-shot setting. However, removing event descriptions improves CODE4STRUCT's zero-shot performance on argument identification. 0-shot Arg-I precision is relatively unchanged after removing event descriptions

(37.4 vs. 37.2). We argue that removing event descriptions loosens entity-related constraints and allows LLM to identify more relevant entities. This is supported by the improvement of 0-shot Arg-I recall (78.7 to 81.8) after description removal, which mainly accounts for the increase in 0-shot Arg-I F1. Despite being helpful in argument identification by boosting 0-shot Arg-I recall, we do not see the benefit of removing descriptions in few-shot Arg-C, where it performs consistently worse compared to CODE4STRUCT.

Type Annotation Type annotation is more helpful when more in-context examples are provided ($k \geq 20$). Under a low-shot setting, the F1 difference resulting from type annotation removal is small and inconsistent across different shots. Prompts with type annotation consistently outperforms prompts without it when sufficient in-context examples are provided ($k \geq 20$). We hypothesize that type annotations help disambiguate entity types accepted for each argument, and such disambiguation ability is only needed when the number of entity instances that appeared in in-context examples passes a certain threshold (*e.g.*, $k \geq 20$).

Hierarchical Event Definition Providing hierarchical event definition (i.e., the parent class definition of a given child event class) benefits CODE4STRUCT performance in high-shot setting ($k \geq 20$). Prompts without parent class definition perform on par with CODE4STRUCT under $k < 20$.

A.4 In-context Example Variability Analysis

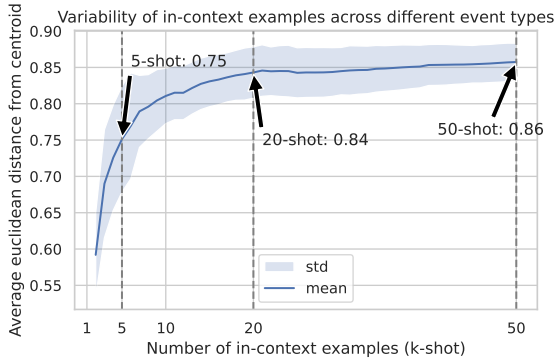


Figure A.2: The variability of k-shot in-context examples (*i.e.*, average euclidean distance from centroid example) across different event types increases with diminishing returns when k increases.

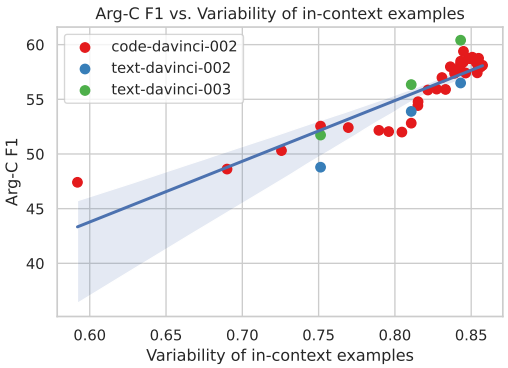


Figure A.3: The variability of in-context examples is positively correlated with code prompt Arg-C performance.

To investigate why the Arg-C performance plateaus with an increasing number of in-context examples k as shown in Figure 3, we analyze the sentence variability of a fixed set of in-context examples (§2.3). We consider the set of k-shot in-context examples for each event type e as a cluster D_e where $|D_e| \leq k$ and use `sentence-transformer`[13] to embed all the input sentences from D_e into a cluster of vectors V_e.

We use the average euclidean distance from the centroid example similar to (Halkidi et al., 2001) to measure the variability of in-context examples for each event type e:

$$\text{Variability}(e) = \frac{1}{|V_e|} \sum_{v \in V_e} d(v, \bar{v})$$

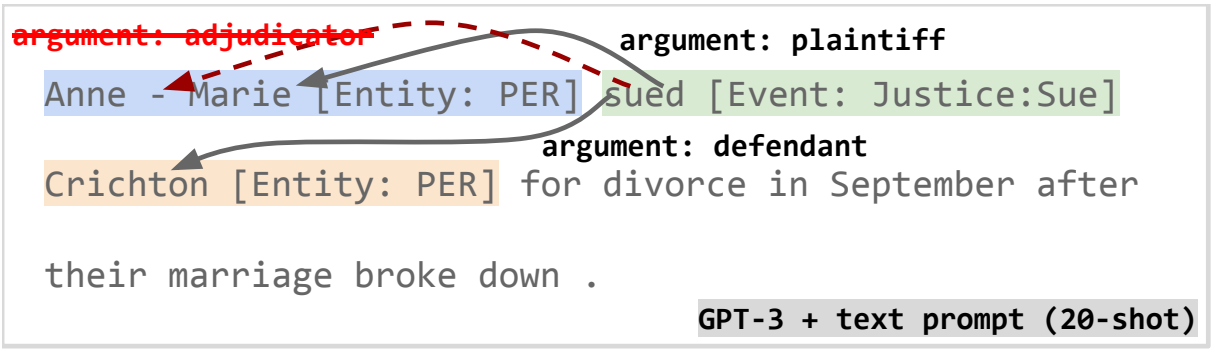
where $d(\cdot, \cdot)$ is the euclidean distance between two vectors and $\bar{v} = \frac{1}{|V_e|} \sum_{v \in V_e} v$ is the centroid example of the cluster V_e.

We calculate the mean Variability(e) across all e for $k \in \{1, \cdots, 50\}$. In Figure A.2, similar to Arg-C performance in Figure 3, we find the mean Variability(e) across all e increases with diminishing returns with increasing k. Furthermore, we find that, in Figure A.3, Arg-C F1 performance is positively correlated with the mean Variability(e) across all e. This suggests the lack of in-context example variability improvement could be one of the reasons Arg-C F1 plateaus, even when more examples are given.

[13] `all-mpnet-base-v2` model

Prompt	Model	code-davinci-002		text-davinci-002		text-davinci-003	
	Metric	Arg-I	Arg-C	Arg-I	Arg-C	Arg-I	Arg-C
	k-shot						
code	0	50.6	36.0	48.9	35.0	49.9	37.8
	1	57.3	47.8	55.8	45.2	56.0	44.7
	5	58.0	52.5	56.0	48.8	59.2	51.7
	10	57.2	52.8	60.6	53.9	62.8	56.3
	20	62.1	58.5	59.9	56.5	65.0	60.4
	30	62.2	58.4	-	-	-	-
	50	62.3	58.1	-	-	-	-
text (code imitation)	0	49.9	38.2	51.5	37.4	52.0	39.2
	1	57.2	48.8	54.0	42.2	57.5	47.9
	5	56.9	49.6	58.0	45.8	60.1	50.3
	10	58.6	52.0	57.9	47.5	59.7	51.3
	20	60.4	54.9	59.0	48.5	61.5	52.6
text (BART-Gen style Li et al. (2021))	0	0.0	0.0	28.7	21.9	34.6	25.2
	1	52.6	43.1	50.5	40.3	54.9	43.6
	5	56.1	51.4	55.0	47.4	59.9	53.8
	10	57.4	52.7	57.7	48.9	62.2	57.5
	20	61.9	56.1	56.2	50.7	64.3	60.8

Table A.3: Performance of the code and two variants of the text prompts on the Arg-I and Arg-C metrics. 50-shot results for `text-davinci` and text prompts are unavailable since the 50-shot prompt length exceeds such LLM's input token limitation. Examples of text prompt variants can be found in Figure A.5 (code imitation) and Figure A.6 (BART-Gen style).



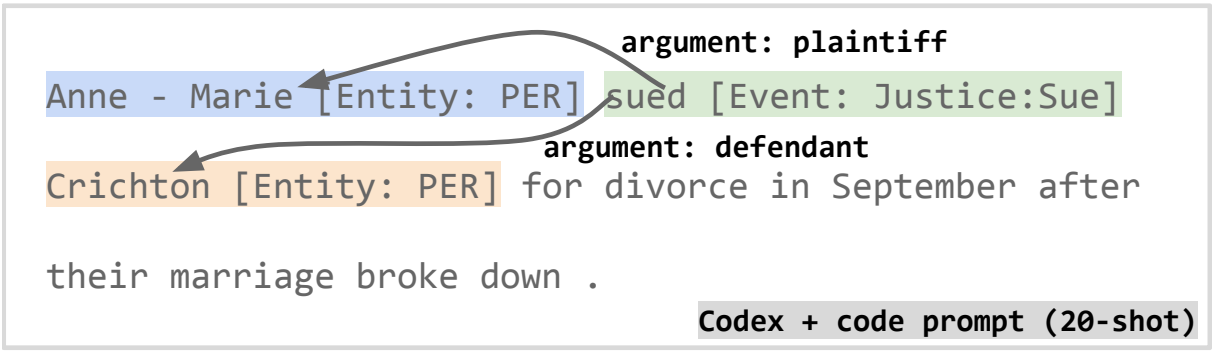


Figure A.4: Example prediction of 20-shot text prompt $T^{(1)}$ using `text-davinci-002` and code prompt using `code-davinci-002`. In this example, 20-shot text prompt using `text-davinci-002` incorrectly predicts the same entity *Anne-Marie* as both *adjudicator* and *plaintiff* of the `Justice:Sue` event.

```
Description of base entity types:
GPE: Geopolitical entities such as countries, provinces, states, cities, towns,
etc. GPEs are composite entities, consisting of a physical location, a government,
and a population. All three of these elements must be present for an entity to be
tagged as a GPE. A GPE entity may be a single geopolitical entity or a group.
... (other types omitted for space)
```

(1) Entity Definition(s)

```
Role definition of event type Movement (Parent type: Event):
1. agent (need to be one of GPE or ORG or PER)
2. artifact (need to be one of FAC or ORG or PER or VEH or WEA)
3. destination (need to be one of FAC or GPE or LOC)
4. origin (need to be one of FAC or GPE or LOC)
5. vehicle (need to be one of VEH)
Role definition of event type Transport (Parent type: Movement):
1. agent (need to be one of GPE or ORG or PER)
2. artifact (need to be one of FAC or ORG or PER or VEH or WEA)
3. destination (need to be one of FAC or GPE or LOC)
4. origin (need to be one of FAC or GPE or LOC)
5. vehicle (need to be one of VEH)
Multiple entities can be extracted for the same role, each entity is a
double-quote enclosed string.
Each extracted entity should look like: (Base Entity Type) "content of extracted
string"
If entity is not present in the text, write: () ""
Different entities are delimited by a comma.
In this event: [agent] transported [artifact] in [vehicle] vehicle from [origin]
place to [destination] place.
```

(2) Event Definition

```
Translate the following sentence into an instance of Transport event. The
trigg
"Kel
Sunda
1. ag
2. ar
3. de
4. or
5. ve
```

```
Translate the following sentence into an instance of Transport event. The
trigger word(s) of the event is marked with **trigger word**.
"Renowned Hollywood madam Heidi Fleiss has been **flown** to Melbourne as
guest of honour at Thursday's market debut and , according to Harris , has
already played a key role in attracting worldwide media attention to the
event ."
1. agent: () ""
2. artifact: (PER) "Heidi Fleiss"
3. destination: (GPE) "Melbourne"
4. origin: () ""
5. vehicle: () ""
```

(3) *k* In-context Examples

```
Translate the following sentence into an instance of Transport event. The trigger
word(s) of the event is marked with **trigger word**.
"Kelly , the US assistant secretary for East Asia and Pacific Affairs ,
**arrived** in Seoul from Beijing Friday to brief Yoon , the foreign minister ."
1. agent: () ""
2. artifact: (PER) "Kelly"
3. destination: (GPE) "Seoul"
4. origin: (GPE) "Beijing"
5. vehicle: () ""
```

(4) Event Instantiation

Figure A.5: Natural language prompt for EAE task following our code prompt design described in section 2. We ask a LLM to generate event instantiation marked in green.

Description of base entity types:
GPE: Geopolitical entities such as countries, provinces, states, cities, towns, etc. GPEs are composite entities, consisting of a physical location, a government, and a population. All three of these elements must be present for an entity to be tagged as a GPE. A GPE entity may be a single geopolitical entity or a group.
... (other types omitted for space)

(1) Entity Definition(s)

Role definition of event type **Movement** (Parent type: Event):
1. agent (need to be one of GPE or ORG or PER)
2. artifact (need to be one of FAC or ORG or PER or VEH or WEA)
3. destination (need to be one of FAC or GPE or LOC)
4. origin (need to be one of FAC or GPE or LOC)
5. vehicle (need to be one of VEH)

Role definition of event type **Transport** (Parent type: Movement):
1. agent (need to be one of GPE or ORG or PER)
2. artifact (need to be one of FAC or ORG or PER or VEH or WEA)
3. destination (need to be one of FAC or GPE or LOC)
4. origin (need to be one of FAC or GPE or LOC)
5. vehicle (need to be one of VEH)

Multiple entities can be extracted for the same role, each entity is a double-quote enclosed string.
Different entities are delimited by a comma.
Each pair of brackets below contains a role name (e.g., [role_1])
Fill in the corresponding role [brackets] with the extracted entities (e.g., ["entity_1_for_role_1", "entity_2_for_role_1"]).
If an entity is not present in the text, write: []
In this event: [agent] transported [artifact] in [vehicle] vehicle from [origin] place to [destination] place.

(2) Event Definition

Translate the following sentence into an instance of Transport event. The trigger word(s) of the event is marked with **trigger word**.
"Kell
Sunda
In th
["Tok

Translate the following sentence into an instance of Transport event. The trigger word(s) of the event is marked with **trigger word**.
"Renowned Hollywood madam Heidi Fleiss has been **flown** to Melbourne as guest of honour at Thursday 's market debut and , according to Harris , has already played a key role in attracting worldwide media attention to the event ."
In this event: [] transported ["Heidi Fleiss"] in [] vehicle from [] place to ["Melbourne"] place.

(3) *k* In-context Examples

Translate the following sentence into an instance of Transport event. The trigger word(s) of the event is marked with **trigger word**.
"Kelly , the US assistant secretary for East Asia and Pacific Affairs , **arrived** in Seoul from Beijing Friday to brief Yoon , the foreign minister ."
In this event: [] transported ["Kelly"] in [] vehicle from ["Beijing"] place to ["Seoul"] place.

(4) Event Instantiation

Figure A.6: BART-Gen style (Li et al., 2021) natural language prompt for EAE task. We ask a LLM to generate event instantiation marked in green. Brackets and double-enclosed strings are designed for ease of parsing free form natural language.

```
"""
Translate the following sentence into an instance of Transport. The trigger
word(s) of the event is marked with **trigger word**.
"Kelly , the US assistant secretary for East Asia and Pacific Affairs ,
**arrived** in Seoul from Beijing Friday to brief Yoon , the foreign minister ."

Abstract Meaning Representation of the given sentence:
(a / arrive-01
      :ARG1 (p / person
            :name (n / name
                  :op1 "Kelly")
            :ARG0-of (h / have-org-role-91
                  :ARG1 (g / government-organization
                        :name (n2 / name
                              :op1 "East"
                              :op2 "Asia"
                              :op3 "and"
                              :op4 "Pacific"
                              :op5 "Affairs")
                        :poss (c / country
                              :name (n3 / name
                                    :op1 "US")))
                  :ARG2 (s / secretary
                        :mod (a2 / assistant))))
      :ARG3 (c2 / city
            :name (n4 / name
                  :op1 "Beijing"))
      :ARG4 (c3 / city
            :name (n5 / name
                  :op1 "Seoul"))
      :time (d / date-entity
            :weekday (f / friday))
      :purpose (b / brief-01
            :ARG0 p
            :ARG1 (p2 / person
                  :name (n6 / name
                        :op1 "Yoon")
                  :ARG0-of (h2 / have-org-role-91
                        :ARG2 (m / minister
                              :topic (f2 / foreign))))))
"""
transport_event = Transport(
```

Figure A.7: Example of an AMR-augmented task prompt. We append the AMR prediction after the input sentence. Different prompt components compared to CODE4STRUCT are highlighted in yellow.

```
class Transport(Movement):
    """
    self.agent transported self.artifact in self.vehicle vehicle from self.origin place to self.destination place.
    Event keywords: transport, move, travel, head.
    """
    def __init__(
        self,
        agent: List[GPE | ORG | PER] = [],
        artifact: List[FAC | ORG | PER | VEH | WEA] = [],
        destination: List[FAC | GPE | LOC] = [],
        origin: List[FAC | GPE | LOC] = [],
        vehicle: List[VEH] = [],
    ):
        self.agent = agent
        self.artifact = artifact
        self.destination = destination
        self.origin = origin
        self.vehicle = vehicle
```

Event Keywords

Figure A.8: Example of an event-keywords-augmented event definition. Different prompt components compared to CODE4STRUCT are highlighted in yellow. We use event keywords from Li et al. (2021).

```
class Event:
    def __init__(self, name: str):
        self.name = name

class Movement(Event):
    def __init__(
        self,
        agent: List[GPE | ORG | PER] = [],
        artifact: List[FAC | ORG | PER | VEH | WEA] = [],
        destination: List[FAC | GPE | LOC] = [],
        origin: List[FAC | GPE | LOC] = [],
        vehicle: List[VEH] = [],
    ):
        self.agent = agent
        self.artifact = artifact
        self.destination = destination
        self.origin = origin
        self.vehicle = vehicle

class Transport(Movement):
    """self.agent transported self.artifact in self.vehicle
vehicle from self.origin place to self.destination place."""
    def __init__(
        self,
        agent: List[GPE | ORG | PER] = [],
        artifact: List[FAC | ORG | PER | VEH | WEA] = [],
        destination: List[FAC | GPE | LOC] = [],
        origin: List[FAC | GPE | LOC] = [],
        vehicle: List[VEH] = [],
    ):
        super().__init__(
            agent=agent,
            artifact=artifact,
            destination=destination,
            origin=origin,
            vehicle=vehicle,
        )
```

Parent Event Type

Child event Transport inherit from parent Movement

Transport calls the __init__ method of its parent Movement

Figure A.9: Example of a hierarchical event definition. Different prompt components compared to CODE4STRUCT are highlighted in yellow.

```
class Transaction(Event):
    def __init__(
        self,
        artifact: List[FAC | ORG | PER | VEH | WEA] = [],
        beneficiary: List[GPE | ORG | PER] = [],
        buyer: List[GPE | ORG | PER] = [],
        giver: List[GPE | ORG | PER] = [],
        place: List[FAC | GPE | LOC] = [],
        recipient: List[GPE | ORG | PER] = [],
        seller: List[GPE | ORG | PER] = [],
    ):
        self.artifact = artifact
        self.beneficiary = beneficiary
        self.buyer = buyer
        self.giver = giver
        self.place = place
        self.recipient = recipient
        self.seller = seller

class Transfer_Money(Transaction):
    """self.giver gave money to self.recipient for the benefit of
self.beneficiary in self.place place."""
    def __init__(
        self,
        beneficiary: List[GPE | ORG | PER] = [],
        giver: List[GPE | ORG | PER] = [],
        place: List[FAC | GPE | LOC] = [],
        recipient: List[GPE | ORG | PER] = [],
    ):
        super().__init__(
            beneficiary=beneficiary,
            giver=giver,
            place=place,
            recipient=recipient,
        )

class Transfer_Ownership(Transaction):
    """self.seller gave self.artifact to self.buyer for the benefit of
self.beneficiary at self.place place."""
    def __init__(
        self,
        artifact: List[FAC | ORG | PER | VEH | WEA] = [],
        beneficiary: List[GPE | ORG | PER] = [],
        buyer: List[GPE | ORG | PER] = [],
        place: List[FAC | GPE | LOC] = [],
        seller: List[GPE | ORG | PER] = [],
    ):
        super().__init__(
            artifact=artifact,
            beneficiary=beneficiary,
            buyer=buyer,
            place=place,
            seller=seller,
        )
```

Parent Event Type

Child event Transfer_Money inherit from parent Transaction

Transfer_Ownership is a sibling event type of Transfer_Money

Child event Transfer_Ownership inherit from parent Transaction

Figure A.10: Example of a hierarchical event definition with a sibling event type. Different prompt components compared to Figure A.9 are highlighted in yellow.

```
"""
Translate the following sentence into an instance of Transfer_Money. The
trigger word(s) of the event is marked with **trigger word**.
"If the budget goes through as is , why do n't Mr. Begala and Mr. Carville
just **donate** the extra tax money they do n't want ?"
"""
transfer_money_event = Transfer_Money(
    giver=[
        PER("Begala"),
        PER("Carville"),
    ],
)

"""
Translate the following sentence into an instance of Transfer_Ownership.
The trigger word(s) of the event is marked with **trigger word**.
"" The **acquisition** of Banco Zaragozano builds on our existing business
creating the sixth largest private sector banking group in Spain " by
assets , added Jacobo Gonzalez - Robatto , chief executive of Barclays
Spain ."
"""
transfer_ownership_event = Transfer_Ownership(
    artifact=[
        ORG("Banco Zaragozano"),
    ],
)
```

In-context example from sibling type **Transaction:Transfer_Money**

Make prediction for **Transaction:Transfer_Ownership**

Figure A.11: Example of a task prompt with a 1-shot example from sibling event type. Event definitions for the task prompt is shown in Figure A.10. Groundtruth prediction is colored green.

Parent Event Type	Child Event Type	# of Test Instances	# of Train Example
Business	Declare-Bankruptcy	2	39
	End-Org	5	24
	Merge-Org	0	13
	Start-Org	17	21
Conflict	Attack	90	1211
	Demonstrate	7	62
Contact	Meet	49	194
	Phone-Write	8	104
Justice	Acquit	1	4
	Appeal	6	30
	Arrest-Jail	6	72
	Charge-Indict	8	95
	Convict	6	61
	Execute	2	12
	Extradite	1	6
	Fine	6	22
	Pardon	0	2
	Release-Parole	1	44
	Sentence	11	83
	Sue	4	60
	Trial-Hearing	5	103
Life	Be-Born	3	44
	Die	17	516
	Divorce	9	20
	Injure	1	125
	Marry	10	71
Movement	Transport	47	561
Personnel	Elect	13	156
	End-Position	17	143
	Nominate	1	11
	Start-Position	11	87
Transaction	Transfer-Money	12	121
	Transfer-Ownership	27	85

Table A.4: The number of Train/Test event instances for 33 event types in ACE05-E.

Parent Event Type	Child Event Type (Train)	Child Event Type (Test)
Business	Declare-Bankruptcy	End-Org Merge-Org* Start-Org
Conflict	Attack	Demonstrate
Contact	Meet	Phone-Write
Justice	Trial-Hearing	Acquit Appeal Arrest-Jail Charge-Indict Convict Execute Extradite Fine Pardon* Release-Parole Sentence Sue
Life	Die	Be-Born Divorce Injure Marry
Personnel	Elect	End-Position Nominate Start-Position
Transaction	Transfer-Money	Transfer-Ownership

Table A.5: Train/Test split for each parent event type. * denotes child event types that do not have examples in the ACE05-E test set.