DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines

Credits: Slides adapted from original authors

For reliability, we often build pipelines of FMs

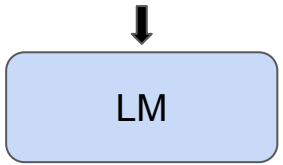
Break down problems and **delegate subtasks** to specialized FMs

How many storeys are there in the castle David Gregory inherited?

For reliability, we often build pipelines of FMs

Break down problems and **delegate subtasks** to specialized FMs

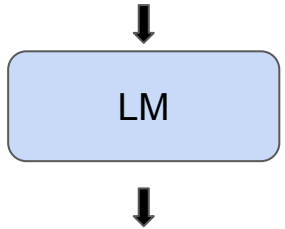
How many storeys are there in the castle David Gregory inherited?



For reliability, we often build pipelines of FMs

Break down problems and **delegate subtasks** to specialized FMs

How many storeys are there in the castle David Gregory inherited?

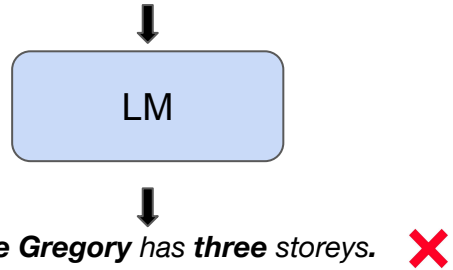


Castle Gregory** has **three** storeys**.

For reliability, we often build pipelines of FMs

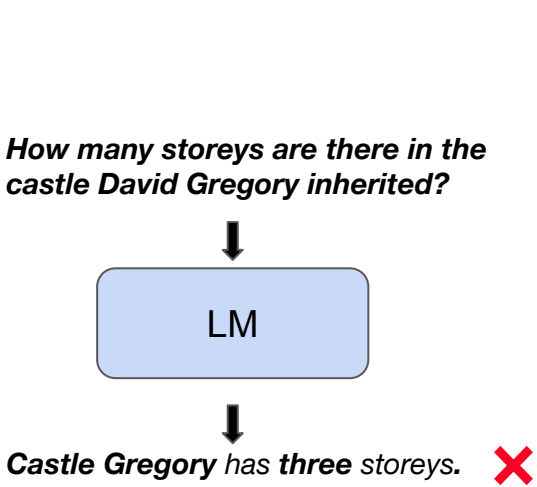
Break down problems and **delegate subtasks** to specialized FMs

How many storeys are there in the castle David Gregory inherited?

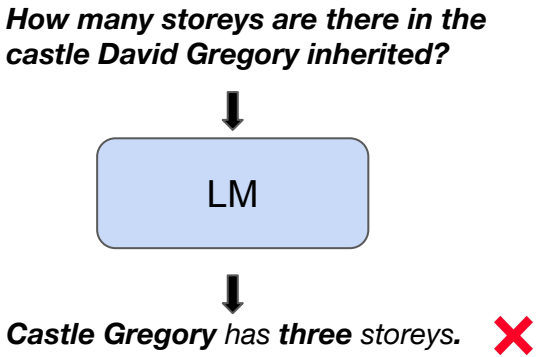


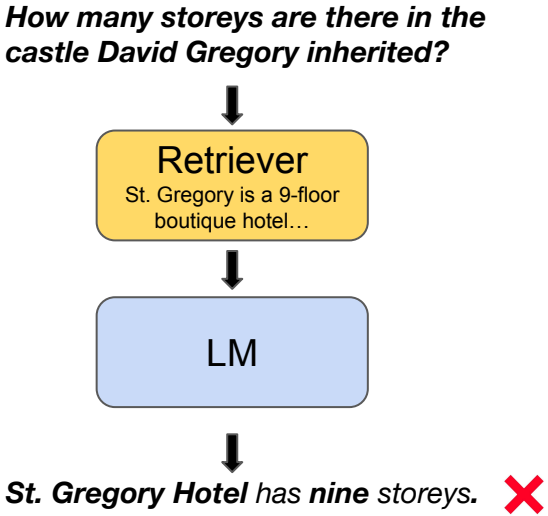
Castle Gregory** has **three** storeys**.

For reliability, we often build pipelines of FMs



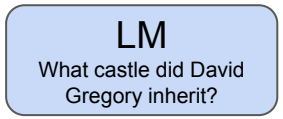
For reliability, we often build pipelines of FMs



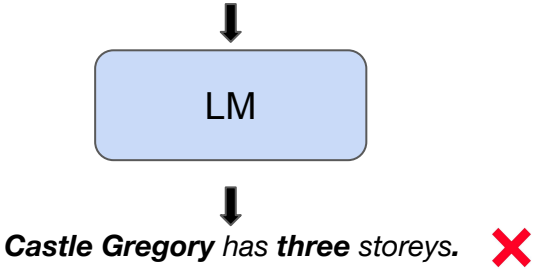


For reliability, we often build pipelines of FMs

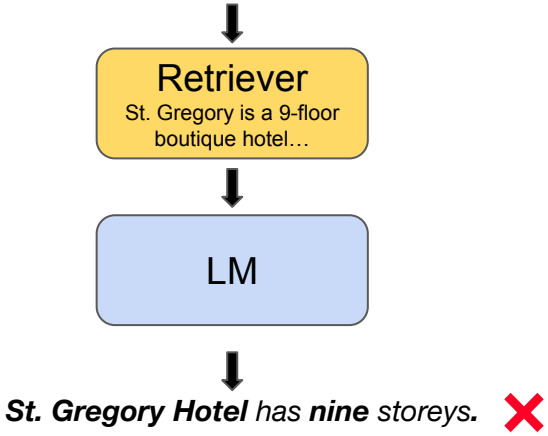
How many storeys are there in the castle David Gregory inherited?



How many storeys are there in the castle David Gregory inherited?

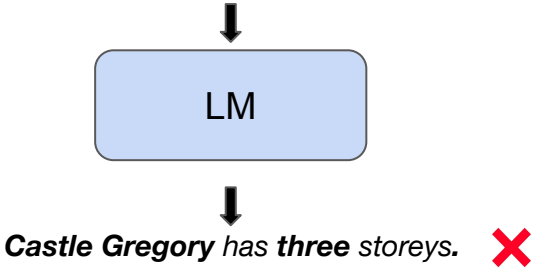


How many storeys are there in the castle David Gregory inherited?

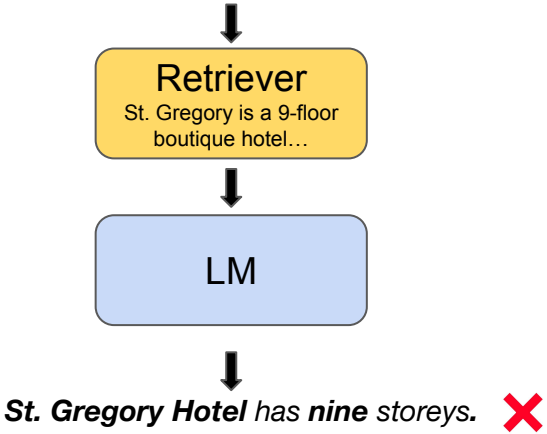


For reliability, we often build pipelines of FMs

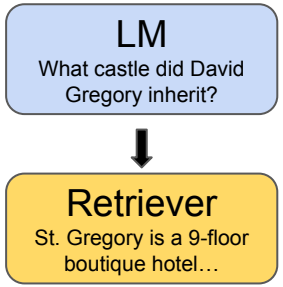
How many storeys are there in the castle David Gregory inherited?



How many storeys are there in the castle David Gregory inherited?

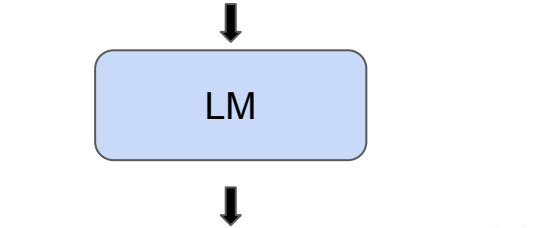


How many storeys are there in the castle David Gregory inherited?

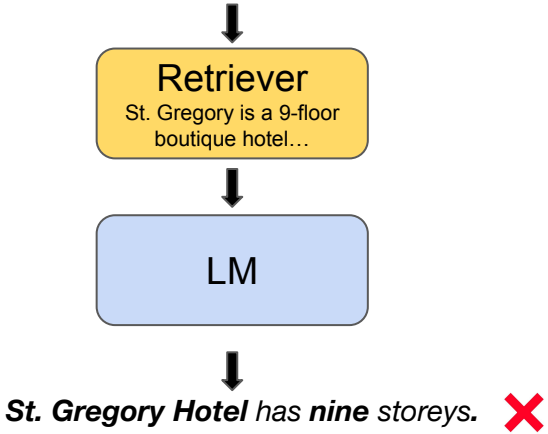


For reliability, we often build pipelines of FMs

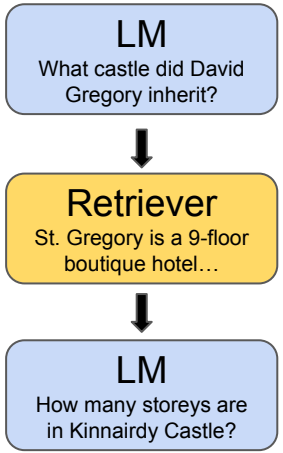
How many storeys are there in the castle David Gregory inherited?



How many storeys are there in the castle David Gregory inherited?

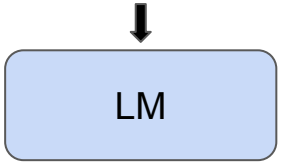


How many storeys are there in the castle David Gregory inherited?



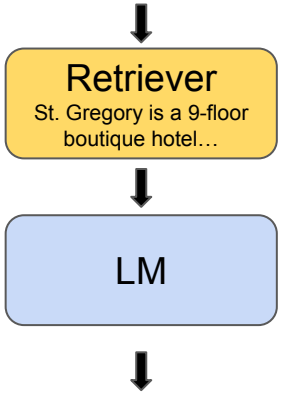
For reliability, we often build pipelines of FMs

How many storeys are there in the castle David Gregory inherited?



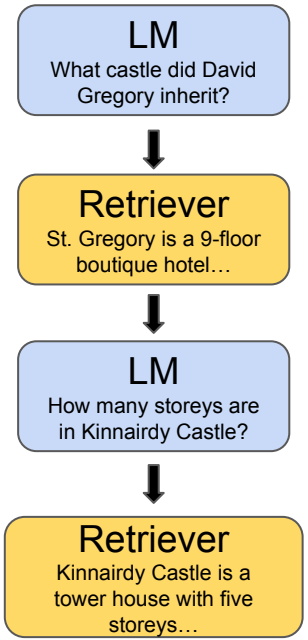
***Castle Gregory** has **three** storeys.* ✗

How many storeys are there in the castle David Gregory inherited?



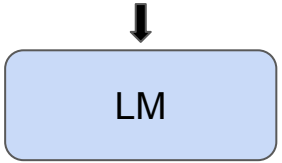
***St. Gregory Hotel** has **nine** storeys.* ✗

How many storeys are there in the castle David Gregory inherited?



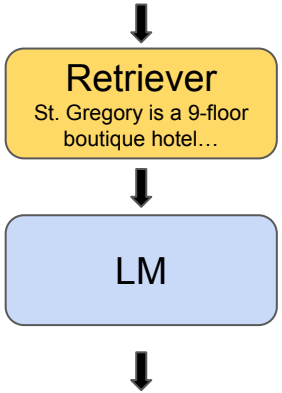
For reliability, we often build pipelines of FMs

How many storeys are there in the castle David Gregory inherited?



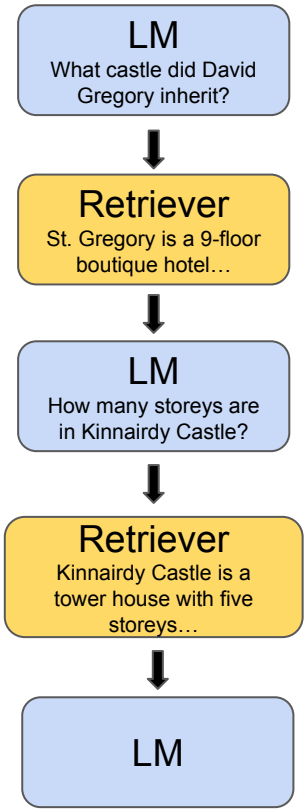
***Castle Gregory** has **three** storeys.* ✗

How many storeys are there in the castle David Gregory inherited?



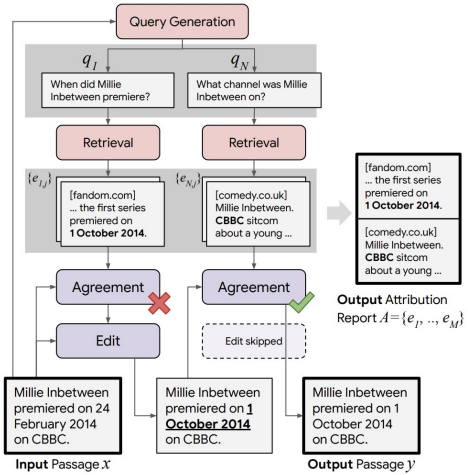
***St. Gregory Hotel** has **nine** storeys.* ✗

How many storeys are there in the castle David Gregory inherited?



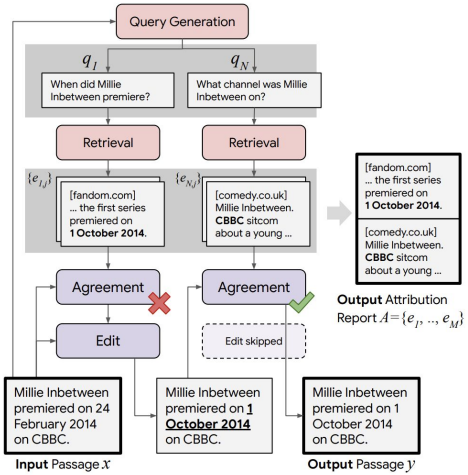
***Castle Gregory** has **five** storeys.* ✓

For reliability, we often build pipelines of FMs

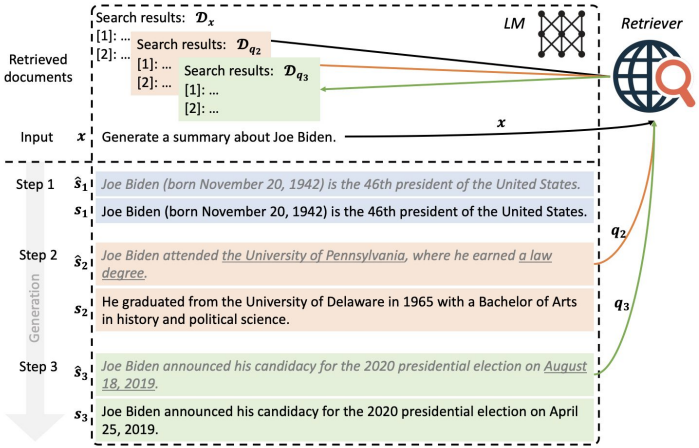


RARR - a pipeline of hand-prompted LLMs and a retriever [Gao et al. 2023]

For reliability, we often build pipelines of FMs

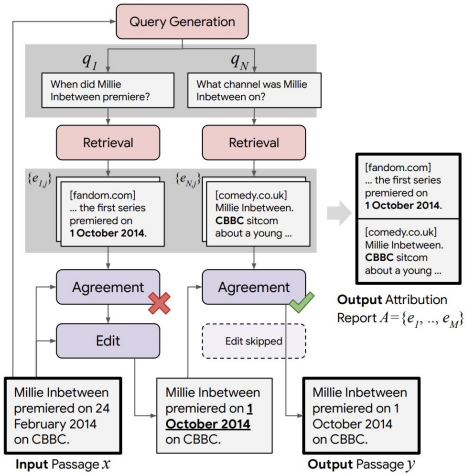


RARR - a pipeline of hand-prompted LLMs and a retriever [Gao et al. 2023]

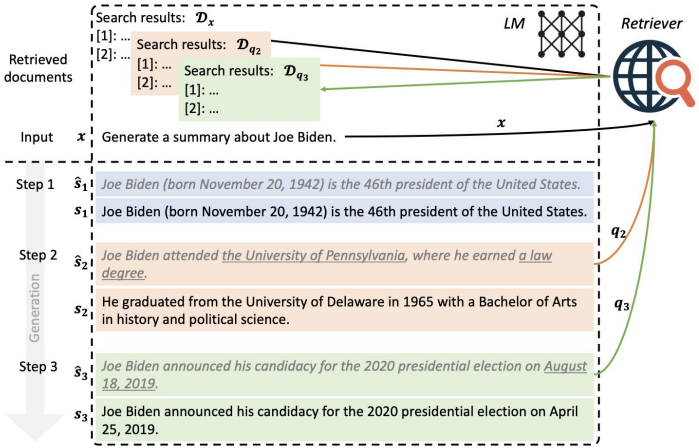


FLARE - a pipeline of hand-prompted LLMs and a retriever
[Jiang et al. 2023]

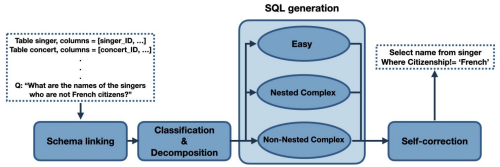
For reliability, we often build pipelines of FMs



RARR - a pipeline of hand-prompted LLMs and a retriever [Gao et al. 2023]

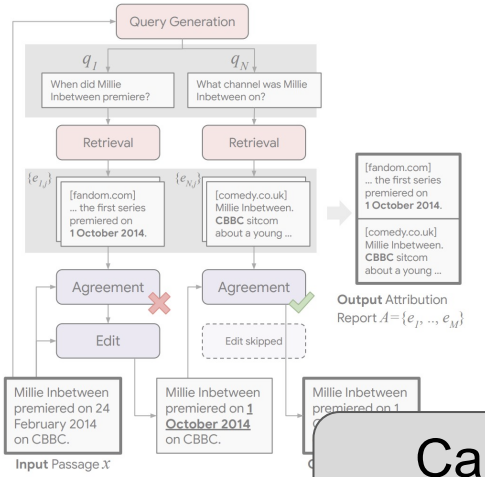


FLARE - a pipeline of hand-prompted LLMs and a retriever [Jiang et al. 2023]

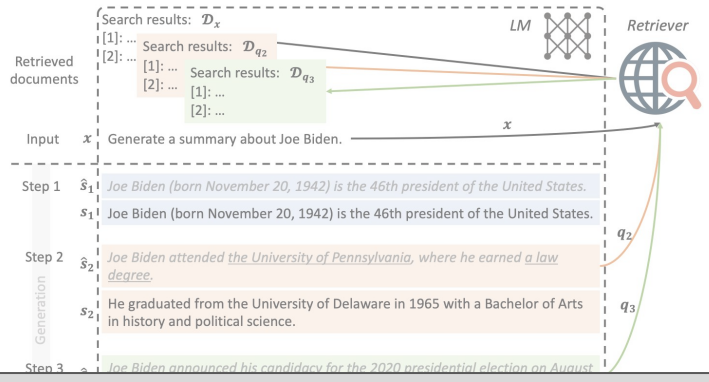


DIN-SQL - a pipeline of hand-prompted LLMs and a SQL DB [Pourreza et al. 2023]

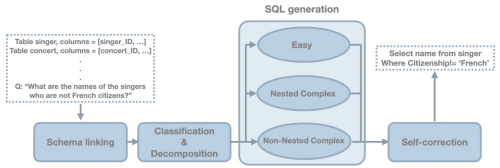
For reliability, we often build pipelines of FMs



RARR - a pipeline of hand-prompted LLMs and a retriever [Gao et al. 2023]



LLMs and a retriever [Jiang et al. 2023]



ine of LMs and a SQL DB 023]

Can we develop a **systematic** approach to design **LM pipelines**?

But building **effective pipelines** isn’t easy

It is not clear how to **connect and optimize all the pieces** of a pipeline together.

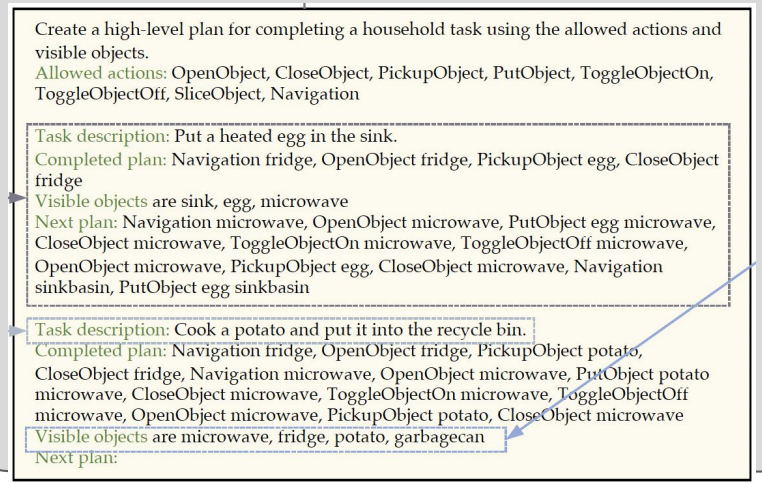
- Pipeline prompts are **extremely fragile** and non-transferable.
- Pipeline finetunes are **very complicated**!

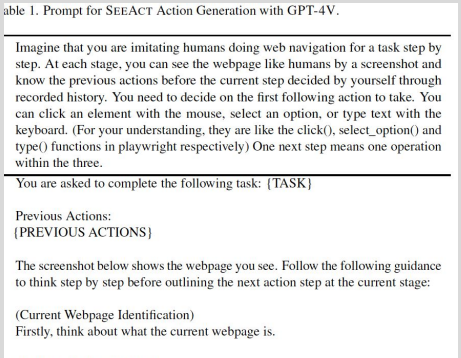
But building **effective pipelines** isn't easy

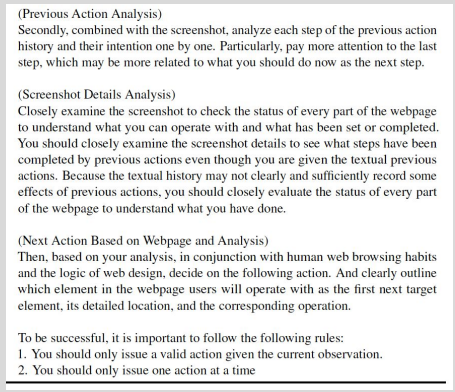
It is not clear how to **connect and optimize all the pieces** of a pipeline together.

- Pipeline prompts are **extremely fragile** and non-transferable.
- Pipeline finetunes are **very complicated**!

Right now, many NLP researchers are writing demonstrations - and often long free-form prompts by hand!







But building **effective pipelines** isn't easy

It is not clear how to **connect and optimize all the pieces** of a pipeline together.

- Pipeline prompts are **extremely fragile** and non-transferable.
- Pipeline finetunes are **very complicated**!

Right now, many NLP researchers are writing demonstrations - and often long free-form prompts by hand!

You are an intelligent shopping assistant that can help users find the right item. You are given an observation of the current web navigation session, in the following format:

Current Observation:
WebShop
Instruction:
{the user instruction}
[button] Search [button_] (generate a search query based on the user instruction and select this button to find relevant items)

Every button in the observation represents a possible action you can take. Based on the current observation, your task is to generate a rationale about the next action you should take. Note that if an history of past rationales and actions is provided, you should also consider the history when generating the rationale.

The problem

- Talking to ChatGPT requires lengthy instructions and lots of mistranslations
  - Sounds imperative?
  - Reminiscent of Siri

- Use declarative as cleaner solution
  - Like programming, needs a compiler to generate complex instructions from simpler commands

The **DSPy** paradigm: Let's *program* - not prompt - LMs

Shift focus from tweaking the LM or prompts to good overarching system design.

Declarative **S**elf-improving Language **P**rograms. How?

The **DSPy** paradigm: Let's *program* - not prompt - LMs

Shift focus from tweaking the LM or prompts to good overarching system design.

Declarative **S**elf-improving Language **P**rograms. How?

1. **Signatures** abstract prompts (and fine-tuned LM weights)

The **DSPy** paradigm: Let's *program* - not prompt - LMs

Shift focus from tweaking the LM or prompts to good overarching system design.

Declarative **S**elf-improving Language **P**rograms. How?

1. **Signatures** abstract prompts (and fine-tuned LM weights)

2. **Modules** generalize prompting techniques (and compose into multi-stage systems)

The **DSPy** paradigm: Let's *program* - not prompt - LMs

Shift focus from tweaking the LM or prompts to good overarching system design.

Declarative **S**elf-improving Language **P**rograms. How?

1. **Signatures** abstract prompts (and fine-tuned LM weights)

2. **Modules** generalize prompting techniques (and compose into multi-stage systems)

3. **Teleprompters** optimize systems (i.e. complex modules) toward an arbitrary metric

The **DSPy** paradigm: Let's *program* - not prompt - LMs

Shift focus from tweaking the LM or prompts to good overarching system design.

Declarative **S**elf-improving Language **P**rograms. How?

1. **Signatures** abstract prompts (and fine-tuned LM weights)

2. **Modules** generalize prompting techniques (and compose into multi-stage systems)

dspy.ChainOfThought ***dspy.ReAct*** ***dspy.MultiChainComparison*** ***dspy.ProgramOfThought***

3. **Teleprompters** optimize systems (i.e. complex modules) toward an arbitrary metric

The **DSPy** paradigm: Let's *program* - not prompt - LMs

Shift focus from tweaking the LM or prompts to good overarching system design.

Declarative **S**elf-improving Language **P**rograms. How?

1. **Signatures** abstract prompts (and fine-tuned LM weights)

question -> answer ***long_document -> summary*** ***context, question -> rationale, response*** ***passage -> main entities***

2. **Modules** generalize prompting techniques (and compose into multi-stage systems)

dspy.ChainOfThought ***dspy.ReAct*** ***dspy.MultiChainComparison*** ***dspy.ProgramOfThought***

3. **Teleprompters** optimize systems (i.e. complex modules) toward an arbitrary metric

The **DSPy** paradigm: Let's *program* - not prompt - LMs

Shift focus from tweaking the LM or prompts to good overarching system design.

Declarative **S**elf-improving Language **P**rograms. How?

1. **Signatures** abstract prompts (and fine-tuned LM weights)

question -> answer ***long_document -> summary*** ***context, question -> rationale, response*** ***passage -> main entities***

2. **Modules** generalize prompting techniques (and compose into multi-stage systems)

dspy.ChainOfThought ***dspy.ReAct*** ***dspy.MultiChainComparison*** ***dspy.ProgramOfThought***

3. **Teleprompters** optimize systems (i.e. complex modules) toward an arbitrary metric

BootstrapFewShot ***BootstrapFinetune*** ***BootstrapFewShotWithRandomSearch*** ***BootstrapFewShotWithOptuna***

The **DSPy** paradigm: Let's *program* - not prompt - LMs

Shift focus from tweaking the LM or prompts to good overarching system design.

Declarative **S**elf-improving Language **P**rograms. How?

1. **Signatures** abstract prompts (and fine-tuned LM weights)

question -> answer ***long_document -> summary*** ***context, question -> rationale, response*** ***passage -> main entities***

2. **Modules** generalize prompting techniques (and compose into multi-stage systems)

dspy.ChainOfThought ***dspy.ReAct*** ***dspy.MultiChainComparison*** ***dspy.ProgramOfThought***

3. **Teleprompters** optimize systems (i.e. complex modules) toward an arbitrary metric

BootstrapFewShot ***BootstrapFinetune*** ***BootstrapFewShotWithRandomSearch*** ***BootstrapFewShotWithOptuna***

When in doubt, assume it's PyTorch and LM is your device

The **DSPy** paradigm: Let's *program* - not prompt - LMs

Shift focus from tweaking the LM or prompts to good overarching system design.

Declarative **S**elf-improving Language **P**rograms. How?

1. **Signatures** abstract prompts (and fine-tuned LM weights)

2. **Modules** generalize prompting techniques (and compose into multi-stage systems)

   Modules are parameterized layers (or compositions).

3. **Teleprompters** optimize systems (i.e. complex modules) toward an arbitrary metric

When in doubt, assume it's PyTorch and LM is your device

The **DSPy** paradigm: Let's *program* - not prompt - LMs

Shift focus from tweaking the LM or prompts to good overarching system design.

Declarative **S**elf-improving Language **P**rograms. How?

1. **Signatures** abstract prompts (and fine-tuned LM weights)

   Signatures are the in/out dimensions (function signature) of a module.

2. **Modules** generalize prompting techniques (and compose into multi-stage systems)

   Modules are parameterized layers (or compositions).

3. **Teleprompters** optimize systems (i.e. complex modules) toward an arbitrary metric

When in doubt, assume it's PyTorch and LM is your device

The **DSPy** paradigm: Let's *program* - not prompt - LMs

Shift focus from tweaking the LM or prompts to good overarching system design.

Declarative **S**elf-improving Language **P**rograms. How?

1. **Signatures** abstract prompts (and fine-tuned LM weights)

   Signatures are the in/out dimensions (function signature) of a module.

2. **Modules** generalize prompting techniques (and compose into multi-stage systems)

   Modules are parameterized layers (or compositions).

3. **Teleprompters** optimize systems (i.e. complex modules) toward an arbitrary metric

   Teleprompters are optimizers.

When in doubt, assume it's PyTorch and LM is your device

Signatures

- A signature consists of three simple elements
  - A minimal **description of the sub-task** that the LM is supposed to solve
  - A description of one of more **input fields** (e.g., input question) that is given as input to the LM
  - A description of one of more **output fields** (e.g., answer) that we expect from the LM.
  - DSPy will parse this notation and expand the field names into meaningful instructions for the LM

Signatures

- A signature consists of three simple elements
  - A minimal **description of the sub-task** that the LM is supposed to solve
  - A description of one of more **input fields** (e.g., input question) that is given as input to the LM
  - A description of one of more **output fields** (e.g., answer) that we expect from the LM.
  - DSPy will parse this notation and expand the field names into meaningful instructions for the LM

```
qa = dspy.Predict("question -> answer")
qa(question="Where is Guaraní spoken?")
# Out: Prediction(answer='Guaraní is spoken mainly in South America.')
```

What about more complex tasks?

For some advanced tasks, you need more verbose signatures. This is typically to:

1. Clarify something about the nature of the task (expressed below as a docstring).
2. Supply hints on the nature of an input field, expressed as a ‘desc’ keyword argument for ‘dspy.InputField’.
3. Supply constraints on an output field, expressed as a ‘desc’ keyword argument for ‘dspy.OutputField’.

What about more complex tasks?

Example D: A metric that evaluates faithfulness to citations

```
class CheckCitationFaithfulness(dspy.Signature):
    """Verify that the text is based on the provided context."""

    context = dspy.InputField(desc="facts here are assumed to be true")
    text = dspy.InputField()
    faithfulness = dspy.OutputField(desc="True/False indicating if text is faithful to context")

context = "The 21-year-old made seven appearances for the Hammers and netted his only goal for them in

text = "Lee scored 3 goals for Colchester United."

faithfulness = dspy.ChainOfThought(CheckCitationFaithfulness)
faithfulness(context=context, text=text)
```

docstring

Input field description

Output field description

Teleprompters

- *Tele*-prompters: Abstracting and automating the task of prompting
- Teleprompters are powerful optimizers that can take any program and learn to bootstrap and select effective prompts for its modules.
- Hence the name "*prompting at a distance*".

```
tp = BootstrapFewShotWithRandomSearch(metric=gsm8k_accuracy)
bootstrap = tp.compile(program, trainset=trainset, valset=devset)
```

Teleprompter Examples

- **LabeledFewshot**
  - Sample random GT question–answer pairs from train set.

Teleprompter Examples

- **BootstrapFewShot**
  - Bootstrapping few-shot examples and their reasoning traces for ICL, optionally using a teacher program.
  - Keep those demonstrations which satisfy a given metric.
  - Advantages: Store reasoning traces from a teacher, can be used in CoT.

Teleprompter Examples

- **BootstrapFewShotWithRandomSearch**
  - Random search in the dataset for few-shot examples in ICL.
  - The training dataset is shuffled each time.
  - Use the validation set to select the best performing candidate program according to specified metric.

Teleprompter Examples

- **BootstrapFewShotWithOptuna**
  - Hyperparam optimization with Optuna
  - Optimize hyperparameters such as demonstrations, instructions, etc.

Teleprompter Examples

- **BootstrapFinetune**
  - Fine-tune a target LM on demonstrations generated by a teacher program.

Teleprompter Examples

- **Optimizers for instructions**
  - SignatureOptimizer
  - BayesianSignatureOptimizer

Let’s get concrete: Math Word Problems with GSM8K

Question: Joseph and his friends watched two movies in his house. The first movie is 1 hour and 30 minutes long while the second movie is 30 minutes longer than the first. Before the movies, they spent 10 minutes making popcorn and twice as long making fries. How long, in hours, did it take Joseph and his friends to cook and watch the movies?

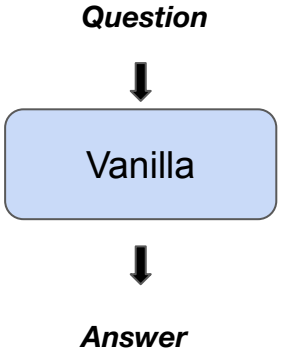
Let’s get concrete: Math Word Problems with GSM8K

Question: Joseph and his friends watched two movies in his house. The first movie is 1 hour and 30 minutes long while the second movie is 30 minutes longer than the first. Before the movies, they spent 10 minutes making popcorn and twice as long making fries. How long, in hours, did it take Joseph and his friends to cook and watch the movies?

Answer: 4

Case Study: GSM8K

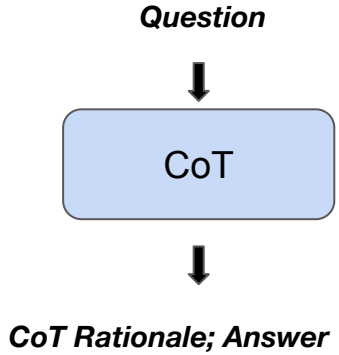
```
vanilla = dspy.Predict("question -> answer")
```



Case Study: GSM8K

```
vanilla = dspy.Predict("question -> answer")
```

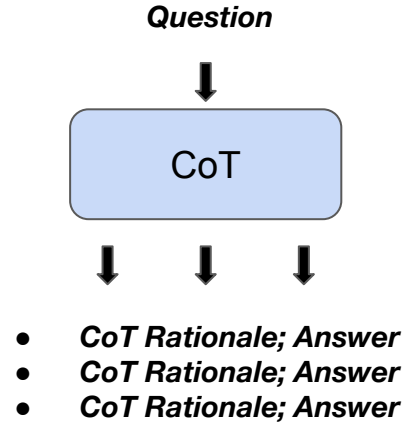
```
CoT = dspy.ChainOfThought("question -> answer")
```



Case Study: GSM8K

```
vanilla = dspy.Predict("question -> answer")
```

```
CoT = dspy.ChainOfThought("question -> answer")
```



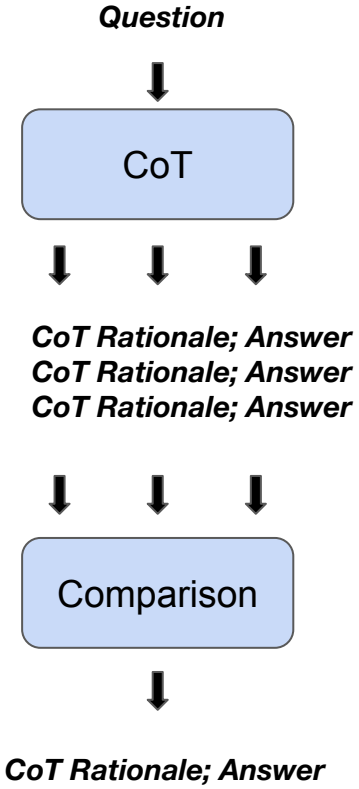
Case Study: GSM8K

vanilla = dspy.Predict(“question -> answer”)

CoT = dspy.ChainOfThought(“question -> answer”)

```python
vanilla = dspy.Predict("question -> answer")  # GSM8K Program ‘vanilla‘

CoT = dspy.ChainOfThought("question -> answer")  # GSM8K Program ‘CoT‘
```

```python
class ThoughtReflection(dspy.Module):
    def __init__(self, num_attempts):
        self.predict = dspy.ChainOfThought("question -> answer", n=num_attempts)
        self.compare = dspy.MultiChainComparison('question -> answer', M=num_attempts)

    def forward(self, question):
        completions = self.predict(question=question).completions
        return self.compare(question=question, completions=completions)

reflection = ThoughtReflection(num_attempts=5) # GSM8K Program ‘reflection‘
```



Now that we have programs, we can **compile** them.

```
improved_dspy_program  = teleprompter.compile(dspy_program, few_examples)
```

Let’s compile our program

```python
trainset = [ds.py.Example(question="Joseph and his friends watched two movies...", answer="4"), ...]
CoT = ds.py.ChainOfThought("question -> answer")

teleprompter = ds.py.BootstrapFewShotWithRandomSearch(metric=gsm8k_accuracy)
bootstrapped_program = teleprompter.compile(CoT, trainset=trainset)
```

```
Given the fields ‘question‘, produce the fields ‘answer‘.

---

Follow the following format.

Question: ${question}
Reasoning: Let’s think step by step in order to $produce the answer. We ...
Answer: ${answer}

---

Question: Mark is baking bread. He has to let it rise for 120 minutes twice. He also needs to spend 10 minutes kneading it and 30 minutes baking it. How many minutes does it take Mark to finish making the bread?
Reasoning: Let’s think step by step in order to find out how many minutes it takes Mark to finish making the bread. We know that he needs to let it rise for 120 minutes twice, so that’s 240 minutes. Then, he needs to spend 10 minutes kneading it and 30 minutes baking it. So, in total, it will take Mark 240 + 10 + 30 = 280 minutes to finish making the bread.
Answer: 280 ---

---

Question: Ben has $2000 for his business operations costs. He orders goods from his supplier and writes them a cheque for $600. His debtor pays him $800 from the purchases they had made on credit. Mr. Ben then decides to do equipment maintenance and spends $1200 on the whole operation. How much money is Mr. Ben remaining with?
Reasoning: Let’s think step by step in order to find out how much money Mr. Ben is remaining with. We know that he had $2000 to begin with, and he spent $600 on goods from his supplier, so he has $2000 - $600 = $1400 left. Then, his debtor paid him $800, so he has $1400 + $800 = $2200. Finally, he spent $1200 on equipment maintenance, so he has $2200 - $1200 = $1000 left.
Answer: $1000 ---

---

... several other demonstrations here ...

---

Question:
```

Figure 10: Shortened copy of the prompt automatically generated by DSPy for GSM8K Llama2-13b-chat CoT program compiled with `bootstrap`.

Let’s compile our program

```
fewshot = dspy.LabeledFewShot(k=8).compile(program, trainset=trainset)
```

Let’s compile our program

```
fewshot = dspy.LabeledFewShot(k=8).compile(program, trainset=trainset)
```

```
tp = BootstrapFewShotWithRandomSearch(metric=gsm8k_accuracy)
bootstrap = tp.compile(program, trainset=trainset, valset=devset)
```

Let’s compile our program

```
fewshot = dspy.LabeledFewShot(k=8).compile(program, trainset=trainset)
```

```
tp = BootstrapFewShotWithRandomSearch(metric=gsm8k_accuracy)
bootstrap = tp.compile(program, trainset=trainset, valset=devset)
```

```
bootstrap2 = tp.compile(program, teacher=bootstrap, trainset=trainset, valset=devset)
```

Let’s compile our program

```
fewshot = dspy.LabeledFewShot(k=8).compile(program, trainset=trainset)
```

```
tp = BootstrapFewShotWithRandomSearch(metric=gsm8k_accuracy)
bootstrap = tp.compile(program, trainset=trainset, valset=devset)
```

```
bootstrap2 = tp.compile(program, teacher=bootstrap, trainset=trainset, valset=devset)
```

```
# A program that ensembles the top-7 candidate programs from a bootstrapping compiler run
    (in particular ‘bootstrap‘ or, when applicable, ‘bootstrap2‘) with majority voting.
ensemble = Ensemble(reduce_fn=dspy.majority).compile(bootstrap.programs[:7])
```

GSM8K Results

Program	Compilation	Training	GPT-3.5 Dev	GPT-3.5 Test	Llama2-13b-chat Dev	Llama2-13b-chat Test
vanilla	none	n/a	24.0	25.2	7.0	9.4
	fewshot	trainset	33.1	–	4.3	–
	bootstrap	trainset	44.0	–	28.0	–
	bootstrap×2	trainset	64.7	61.7	37.3	36.5
	+ensemble	trainset	62.7	61.9	39.0	34.6
CoT	none	n/a	50.0	–	26.7	–
	fewshot	trainset	63.0	–	27.3	–
	fewshot	+human_CoT	78.6	72.4	34.3	33.7
	bootstrap	trainset	80.3	72.9	43.3	–
	+ensemble	trainset	**88.3**	81.6	43.7	–
reflection	none	n/a	65.0	–	36.7	–
	fewshot	trainset	71.7	–	36.3	–
	bootstrap	trainset	83.0	76.0	44.3	40.2
	+ensemble	trainset	86.7	–	**49.0**	**46.9**

Model size	**Acc (8-shot) (%)**
13b	28.7
34b	42.2
70b	56.8

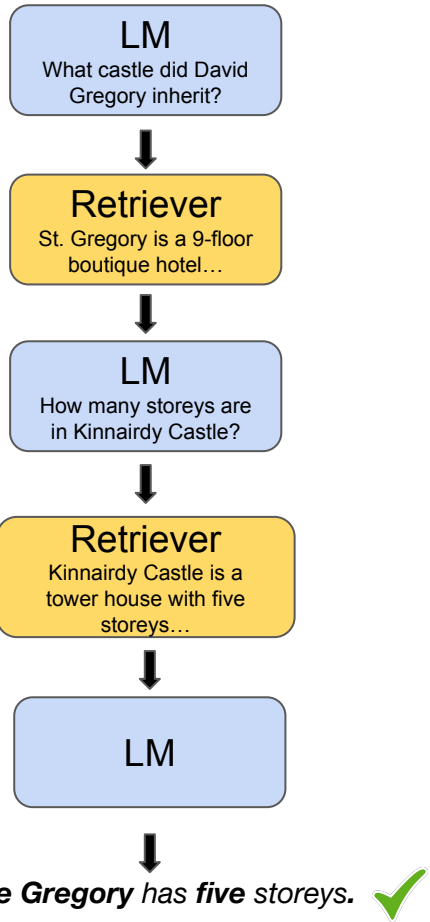
LLaMA2 results (GSM8K)

- DSPy program with LLaMA 13b is competitive with their 34b-based results even though they don't use human reasoning chains in our program.
- Auto-CoT reports 48% for GPT 3 (text-davinci-002), which aligns closely with llama2-13b-chat results.

Case Study: Complex QA

```python
class BasicMultiHop(dspy.Module):
    def __init__(self, passages_per_hop):
        self.retrieve = dspy.Retrieve(k=passages_per_hop)
        self.generate_query = dspy.ChainOfThought("context, question -> search_query")
        self.generate_answer = dspy.ChainOfThought("context, question -> answer")

    def forward(self, question):
        context = []

        for hop in range(2):
            query = self.generate_query(context=context, question=question).search_query
            context += self.retrieve(query).passages

        return self.generate_answer(context=context, question=question)

multihop = BasicMultiHop(passages_per_hop=3)
```

How many storeys are there in the castle David Gregory inherited?



Why DSPy

- Reliability, predictability, scalability
- Other solutions such as LangChain and LLamaIndex still have manual prompt engineering prevalent in their codebase
- DSPy provides a structured framework that automatically bootstraps prompts.

Key Takeaways

- Don’t mess with the weights (prompts) by hand. Define new modules or teleprompters.
- Don’t expect “one layer” to do the work. Add inductive biases and “depth”.
- Use local models for reproducible research. They work really well when compiled.