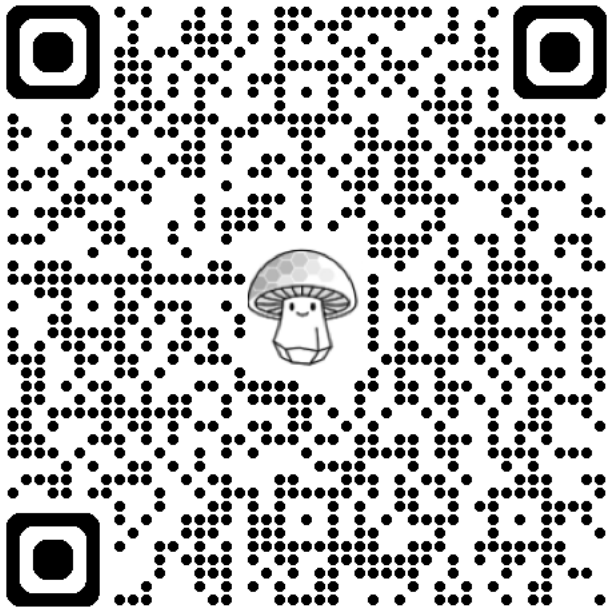
Generative Programming with Mellea

Before we get started...

Step 1: Go to CoLab



URL: gt.mellea.ai

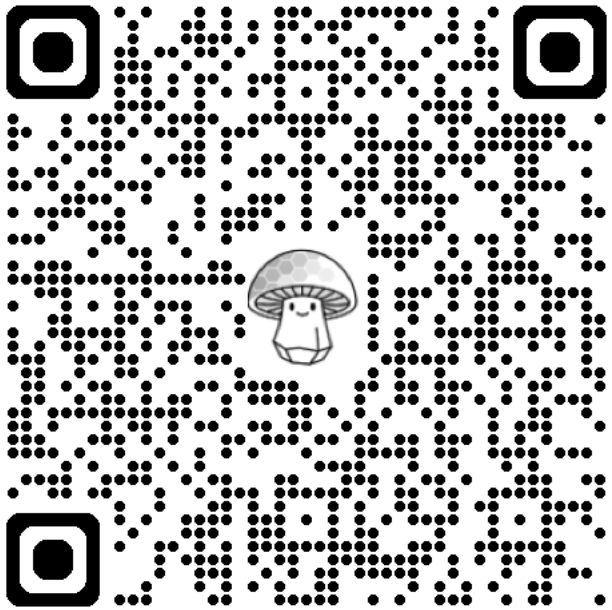
Step 2: Run only the first cell

```
# Install ollama.
!curl -fsSL https://ollama.com/install.sh | sh > /dev/null
!nohup ollama serve >/dev/null 2>&1 &

# Download the granite:3.38b weights.
!ollama pull granite3.3:8b

# install Mellea.
!uv pip install mellea[all] -q

# Some UI niceness.
from IPython.display import HTML, display
def set_css():
    display(HTML("\n<style>\n pre{\n white-space: pre-wrap;\n}\n</style>\n"))
get_ipython().events.register("pre_run_cell", set_css)
```

Before we get started...

Click!

Step 1: Go to CoLab



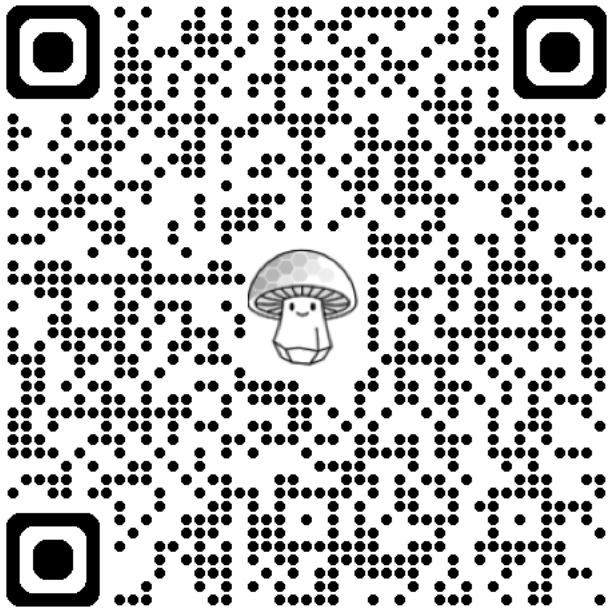
URL: gt.mellea.ai

Step 2: Run only the first cell

```
# Install ollama.
!curl -fsSL https://ollama.com/install.sh | sh > /dev/null
!nohup ollama serve >/dev/null 2>&1 &

# Download the granite:3.38b weights.
!ollama pull granite3.3:8b

# install Mellea.
!uv pip install mellea[all] -q

# Some UI niceness.
from IPython.display import HTML, display
def set_css():
    display(HTML("\n<style>\n pre{\n white-space: pre-wrap;\n}\n</style>\n"))
get_ipython().events.register("pre_run_cell", set_css)
```

Before we get started

Step 1: Go to CoLab



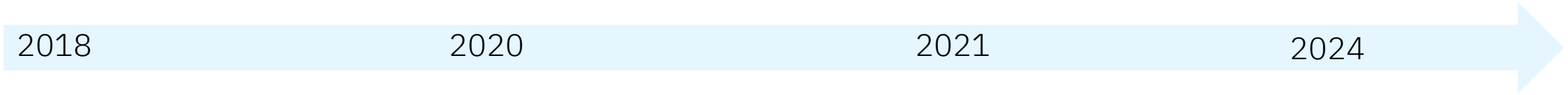
URL: gt.mellea.ai

```
# Install ollama.
!curl -fsSL https://ollama.com/install.sh | sh > /dev/null
!nohup ollama serve >/dev/null 2>&1 &

# Download the granite:3.3:8b weights.
!ollama pull granite3.3:8b
!ollama pull llama3.2:3b

# install Mellea.
!uv pip install mellea[all] -q

# Run docling once to download model weights.
from mellea.stdlib.docs.richdocument import RichDocument

RichDocument.from_document_file("https://mellea.ai")

# Some UI niceness.
from IPython.display import HTML, display


def set_css():
    display(HTML("\n<style>\n pre{\n white-space: pre-wrap;\n}\n</style>\n"))


get_ipython().events.register("pre_run_cell", set_css)
```

```
>>> Installing ollama to /usr/local
>>> Downloading Linux amd64 bundle
##############################                                38.6%
```

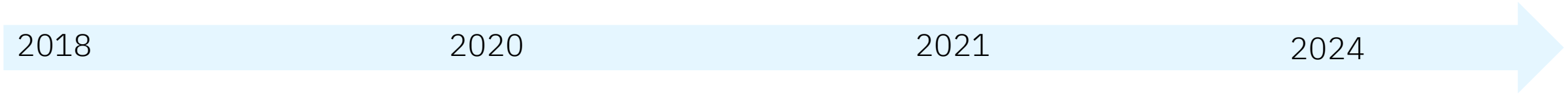
From auto-complete to automation

Early Transformers
Many of the algorithmic issues with early DL approaches are solved; a few labs start scaling up.



From auto-complete to automation

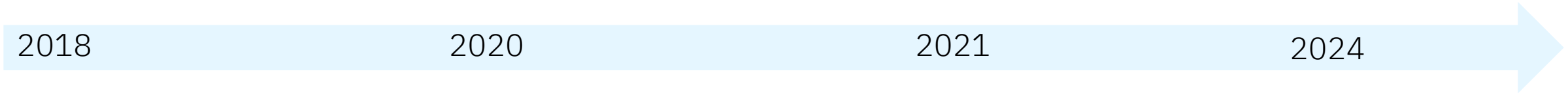
Early Transformers
Many of the algorithmic issues with early DL approaches are solved; a few labs start scaling up.



Code Generation, Primordial MUDs, FIM
Researchers and hobbyists begin experimenting with new interfaces; FIM training seems to work; CoPilot launches.

From auto-complete to automation

Early Transformers
Many of the algorithmic issues with early DL approaches are solved; a few labs start scaling up.

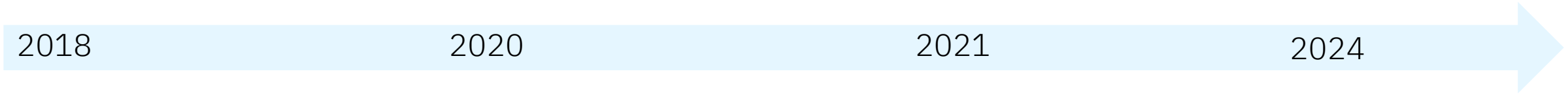


Code Generation, Primordial MUDs, FIM
Researchers and hobbyists begin experimenting with new interfaces; FIM training seems to work; CoPilot launches.

ChatGPT
Taking the chat modality seriously takes the world by storm.

From auto-complete to automation

Early Transformers
Many of the algorithmic issues with early DL approaches are solved; a few labs start scaling up.



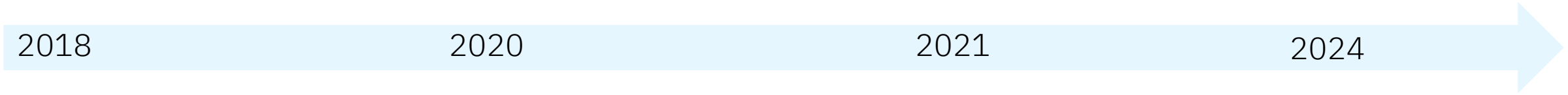
Code Generation, Primordial MUDs, FIM
Researchers and hobbyists begin experimenting with new interfaces; FIM training seems to work; CoPilot launches.

ChatGPT
Taking the chat modality seriously takes the world by storm.

Hobbyist hacking:
IRC bots,
MUD games,
...

From auto-complete to automation

Early Transformers
Many of the algorithmic issues with early DL approaches are solved; a few labs start scaling up.



Code Generation, Primordial MUDs, FIM
Researchers and hobbyists begin experimenting with new interfaces; FIM training seems to work; CoPilot launches.

ChatGPT
Taking the chat modality seriously takes the world by storm.

From auto-complete to automation

Early Transformers
Many of the algorithmic issues with early DL approaches are solved; a few labs start scaling up.

Tools and Prompt Eng
“Unhobbling” systems by providing access to tools.

2018 2020 2021 2024

Code Generation, Primordial MUDs, FIM
Researchers and hobbyists begin experimenting with new interfaces; FIM training seems to work; CoPilot launches.

ChatGPT
Taking the chat modality seriously takes the world by storm.

From auto-complete to automation

Early Transformers
Many of the algorithmic issues with early DL approaches are solved; a few labs start scaling up.

Tools and Prompt Eng
“Unhobbling” systems by providing access to tools.

2018 2020 2021 2024

Code Generation, Primordial MUDs, FIM
Researchers and hobbyists begin experimenting with new interfaces; FIM training seems to work; CoPilot launches.

ChatGPT
Taking the chat modality seriously takes the world by storm.

Agents
LLMs operating autonomously to solve tasks without human oversight.

From auto-complete to automation

Early Transformers
Many of the algorithmic issues with early DL approaches are solved; a few labs start scaling up.

Tools and Prompt Eng
"Unhobbling" systems by providing access to tools.

2018 2020 2021 2024

Code Generation, Primordial MUDs, FIM
Researchers and hobbyists begin experimenting with new interfaces; FIM training seems to work; CoPilot launches.

ChatGPT
Taking the chat modality seriously takes the world by storm.

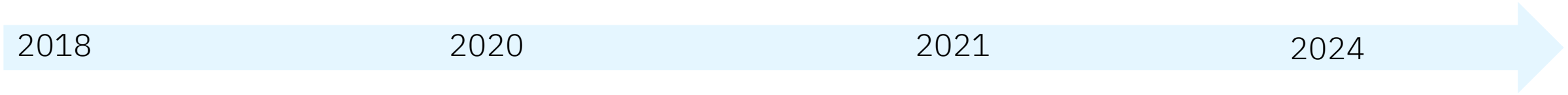
Agents
LLMs operating autonomously to solve tasks without human oversight.

"Think"
Models with internal monologues

From auto-complete to automation

Early Transformers
Many of the algorithmic issues with early DL approaches are solved; a few labs start scaling up.

Tools and Prompt Eng
"Unhobbling" systems by providing access to tools.



Code Generation, Primordial MUDs, FIM
Researchers and hobbyists begin experimenting with new interfaces; FIM training seems to work; CoPilot launches.

Hobbyist hacking:
IRC bots,
MUD games,
...

ChatGPT
Taking the chat modality seriously takes the world by storm.

Agents
LLMs operating autonomously to solve tasks without human oversight.

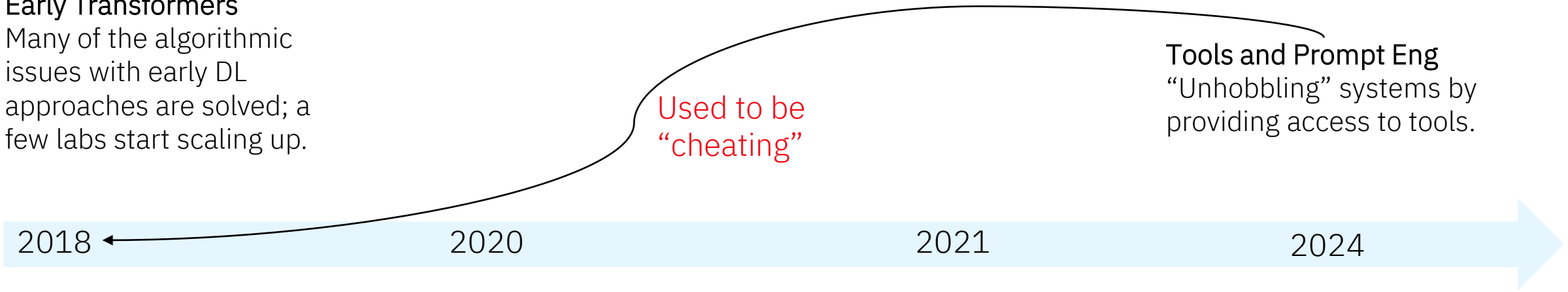
"Think"
Models with internal monologues

From auto-complete to automation

Early Transformers
Many of the algorithmic issues with early DL approaches are solved; a few labs start scaling up.

Used to be "cheating"

Tools and Prompt Eng
"Unhobbling" systems by providing access to tools.



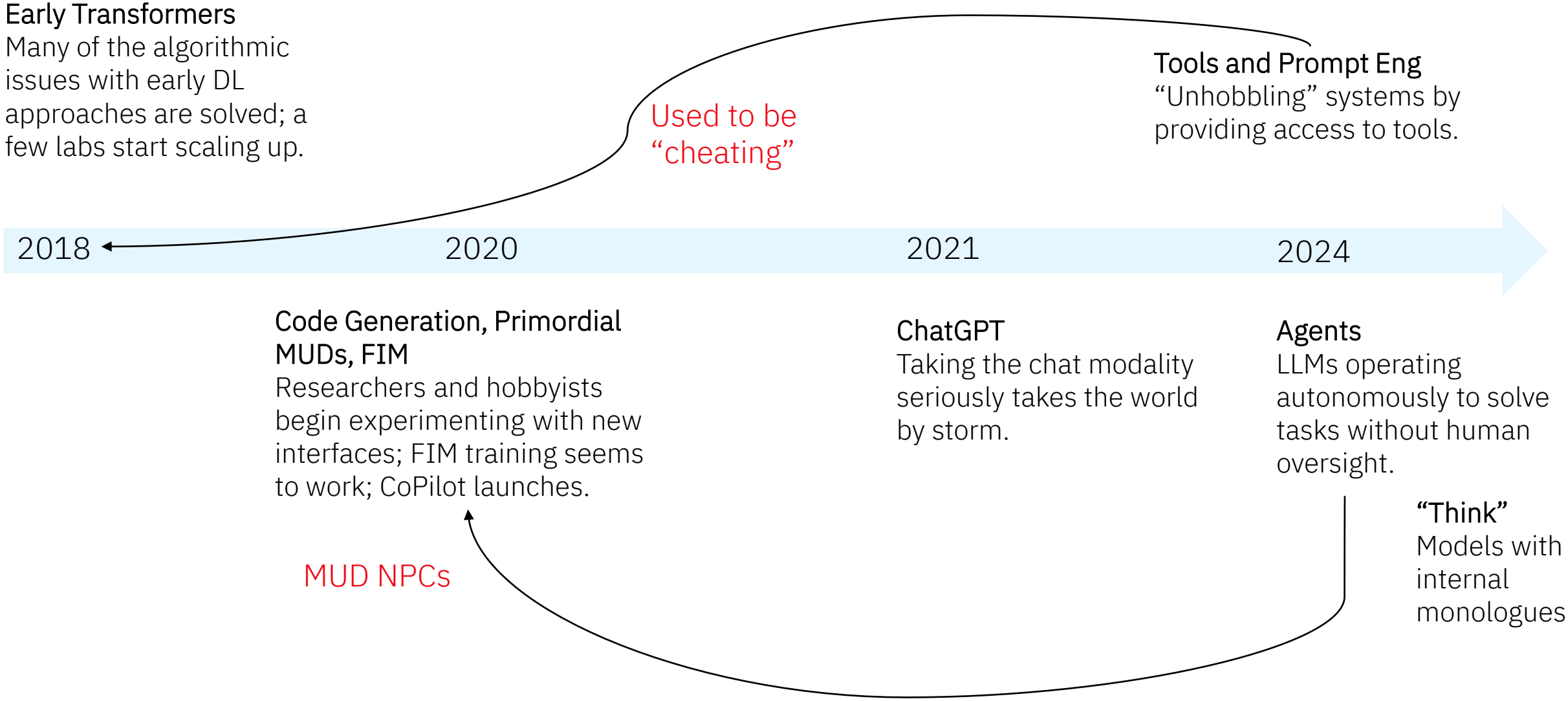
Code Generation, Primordial MUDs, FIM
Researchers and hobbyists begin experimenting with new interfaces; FIM training seems to work; CoPilot launches.

ChatGPT
Taking the chat modality seriously takes the world by storm.

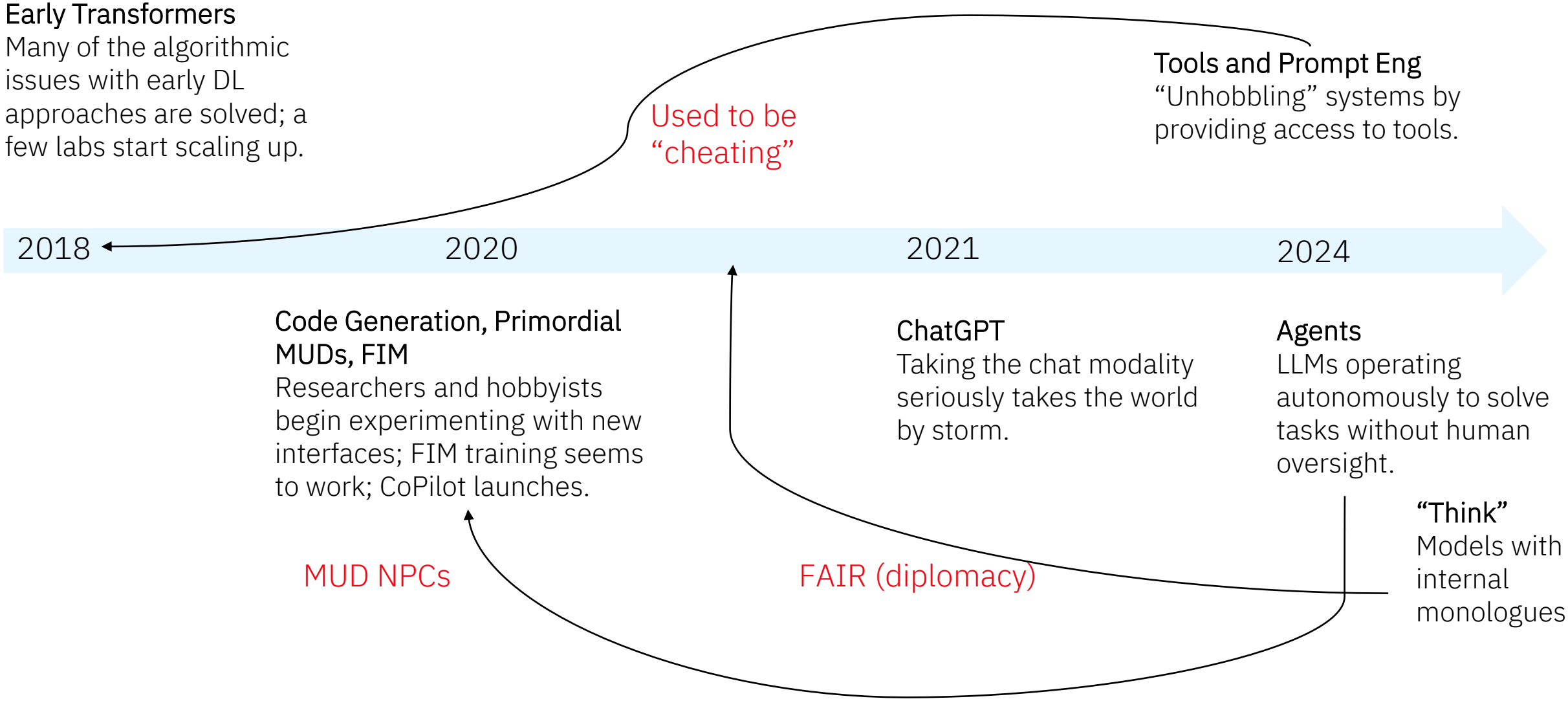
Agents
LLMs operating autonomously to solve tasks without human oversight.

"Think"
Models with internal monologues

From auto-complete to automation



From auto-complete to automation



Toys to Infrastructure

Primodial PoCs

Team Size: One person or a small team.

System Size: Greenfield.

Stakes: Low. Mostly for fun/R&D, not highly trusted, maybe not even deployed.

Autonomy: Human-in-the-loop

Toys to Infrastructure

Primodial PoCs

Team Size: One person or a small team.

System Size: Greenfield.

Stakes: Low. Mostly for fun/R&D, not highly trusted, maybe not even deployed.

Autonomy: Human-in-the-loop

Toward Productivity

Team Size: Massive.

- Data, Training, Inference Stack, Middleware, Application Itself, Integrators, …

System Size: Massive.

Stakes: Core business processes.

Autonomy: Even with HiL, fault models must be understood and designed for.

PROGRAMMING-IN-THE LARGE VERSUS PROGRAMMING-IN-THE-SMALL

Frank DeRemer
Hans Kron

University of California, Santa Cruz

Key words and phrases
Module interconnection language, visibility, accessibility, scope of definition, external name, linking, system hierarchy, protection, information hiding, virtual machine, project management tool.

Abstract
We distinguish the activity of writing large programs from that of writing small ones. By large programs we mean systems consisting of many small programs (modules), possibly written by different people.

We need languages for programming-in-the-small, i.e. languages not unlike the common programming languages of today, for writing modules. We also need a "module interconnection language" for knitting those modules together into an integrated whole and for providing an overview that formally records the intent of the programmer(s) and that can be checked for consistency by a compiler.

We explore the software reliability aspects of such an interconnection language. Emphasis is placed on facilities for information hiding and for defining layers of virtual machines.

1. Introduction

Programming a large system in any typical programming language available today is an exercise in obscuration. We work hard at discovering the inherent structure in a problem and then structuring our solution in a compatible way. Research into "structured programming" (Dijkstra 1972) tells us that this approach will lead to readable, understandable, provable, and modifiable solutions. However, current languages discourage the accurate recording of the overall solution structure; they force us to write programs in which we are so preoccupied with the trees that we lose sight of the forest, as do the readers of our programs!

Let us refer to typical languages as "languages for programming-in-the-small" (LPSs). Let us use the term "module" to refer to a segment of LPS code defining one or more named resources. Each "resource" is a variable, constant, procedure, data structure, mode, or whatever is definable in the LPS. Preferably a module is one to a few pages long and is easily comprehensible by a single person who understands the intended environment and function of the module.

We argue that structuring a large collection of modules to form a "system" is an essentially distinct and different intellectual activity from that of constructing the individual modules. That is, we distinguish programming-in-the-large from programming-in-the-small. Correspondingly, we believe that essentially distinct and different languages should be used for the two activities. We refer to a language for describing system structure as a "module interconnection language" (MIL); it is one necessity for supporting programming-in-the-large.

An MIL should provide a means for the programmer(s) of a large system to express their intent regarding the overall program structure in a concise, precise, and checkable form. Where an MIL is not available, module interconnectivity information is usually buried partly in the modules, partly in an often amorphous collection of linkage-editor instructions, and partly in the informal documentation of the project. Aside from the issue that each of these three areas is ill-suited to express interconnectivity, the smearing of the relevant information over disjoint media is highly unreliable. Even more unsatisfactory are the facilities for specifying and enforcing module disconnectivity via informating hiding, limiting access to resources, establishing protection layers, closing off subsystems, etc. The lack of such facilities invites undisciplined or even unsocial programming, as shown by one of Weinberg's case studies (Weinberg 1971, pp. 71-75), since there is no automated means of enforcing the surface consensus of the programming team.

That current languages fail to support the global task of composing large systems was well argued by Wulf and Shaw in their paper entitled "Global variables considered harmful" (Wulf 1973). A responding paper (George 1973) proposed a scheme of declarations to augment block structure as a solution to the problems associated with global variables. The scheme provided mechanisms to protect variables from violations of various sorts by contained blocks, and to allow limited access to certain variables by selected internal blocks. Similar approaches have been suggested by others (Clark 1971, White 1972, and Ichbiah 1974). We believe that some of the mechanisms proposed were appropriate, but that they were inappropriately placed in the LPS.

Work reported herein was supported in part by the National Science Foundation via grant number GJ 36339.

On the Criteria To Be Used in Decomposing Systems into Modules

D.L. Parnas
Carnegie-Mellon University

This paper discusses modularization as a mechanism for improving the flexibility and comprehensibility of a system while allowing the shortening of its development time. The effectiveness of a "modularization" is dependent upon the criteria used in dividing the system into modules. A system design problem is presented and both a conventional and unconventional decomposition are described. It is shown that the unconventional decompositions have distinct advantages for the goals outlined. The criteria used in arriving at the decompositions are discussed. The unconventional decomposition, if implemented with the conventional assumption that a module consists of one or more subroutines, will be less efficient in most cases. An alternative approach to implementation which does not have this effect is sketched.

Key Words and Phrases: software, modules, modularity, software engineering, KWIC index, software design
CR Categories: 4.0

Author's address: Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213.

Introduction

A lucid statement of the philosophy of modular programming can be found in a 1970 textbook on the design of system programs by Gouthier and Pont [1, ¶10.23], which we quote below:[1]

> A well-defined segmentation of the project effort ensures system modularity. Each task forms a separate, distinct program module. At implementation time each module and its inputs and outputs are well-defined, there is no confusion in the intended interface with other system modules. At checkout time the integrity of the module is tested independently; there are few scheduling problems in synchronizing the completion of several tasks before checkout can begin. Finally, the system is maintained in modular fashion; system errors and deficiencies can be traced to specific system modules, thus limiting the scope of detailed error searching.

Usually nothing is said about the criteria to be used in dividing the system into modules. This paper will discuss that issue and, by means of examples, suggest some criteria which can be used in decomposing a system into modules.

A Brief Status Report

The major advancement in the area of modular programming has been the development of coding techniques and assemblers which (1) allow one module to be written with little knowledge of the code in another module, and (2) allow modules to be reassembled and replaced without reassembly of the whole system. This facility is extremely valuable for the production of large pieces of code, but the systems most often used as examples of problem systems are highly-modularized programs and make use of the techniques mentioned above.

[1] Reprinted by permission of Prentice-Hall, Englewood Cliffs, N.J.

PROGRAMMING-IN-THE LARGE
VERSUS
PROGRAMMING-IN-THE-SMALL

Frank DeRemer
Hans Kron

University of California, Santa Cruz

Key words and phrases
Module interconnection language, visibility, accessibility, scope of definition, external name, linking, system hierarchy, protection, information hiding, virtual machine, project management tool.

Abstract
We distinguish the activity of writing large programs from that of writing small ones. By large programs we mean systems consisting of many small programs (modules), possibly written by different people.

We need languages for programming-in-the-small, i.e. languages not unlike the common programming languages of today, for writing modules. We also need a "module interconnection language" for knitting those modules together into an integrated whole and for providing an overview that formally records the intent of the programmer(s) and that can be checked for consistency by a compiler.

We explore the software reliability aspects of such an interconnection language. Emphasis is placed on facilities for information hiding and for defining layers of virtual machines.

1. Introduction

Programming a large system in any typical programming language available today is an exercise in obscuration. We work hard at discovering the inherent structure in a problem and then structuring our solution in a compatible way. Research into "structured programming" (Dijkstra 1972) tells us that this approach will lead to readable, understandable, provable, and modifiable solutions. However, current languages discourage the accurate recording of the overall solution structure; they force us to write programs in which we are so preoccupied with the trees that we lose sight of the forest, as do the readers of our programs!

Let us refer to typical languages as "languages for programming-in-the-small" (LPSs). Let us use the term "module" to refer to a segment of LPS code defining one or more named resources. Each "resource" is a variable, constant, procedure, data structure, mode, or whatever is definable in the LPS. Preferably a module is one to a few pages long and is easily comprehensible by a single person who understands the intended environment and function of the module.

Work reported herein was supported in part by the National Science Foundation via grant number GJ 36339.

We argue that structuring a large collection of modules to form a "system" is an essentially distinct and different intellectual activity from that of constructing the individual modules. That is, we distinguish programming-in-the-large from programming-in-the-small. Correspondingly, we believe that essentially distinct and different languages should be used for the two activities. We refer to a language for describing system structure as a "module interconnection language" (MIL); it is one necessity for supporting programming-in-the-large.

An MIL should provide a means for the programmer(s) of a large system to express their intent regarding the overall program structure in a concise, precise, and checkable form. Where an MIL is not available, module interconnectivity information is usually buried partly in the modules, partly in an often amorphous collection of linkage-editor instructions, and partly in the informal documentation of the project. Aside from the issue that each of these three areas is ill-suited to express interconnectivity, the smearing of the relevant information over disjoint media is highly unreliable. Even more unsatisfactory are the facilities for specifying and enforcing module disconnectivity via information hiding, limiting access to resources, establishing protection layers, closing off subsystems, etc. The lack of such facilities invites undisciplined or even unsocial programming, as shown by one of Weinberg's case studies (Weinberg 1971, pp. 71-75), since there is no automated means of enforcing the surface consensus of the programming team.

That current languages fail to support the global task of composing large systems was well argued by Wulf and Shaw in their paper entitled "Global variables considered harmful" (Wulf 1973). A responding paper (George 1973) proposed a scheme of declarations to augment block structure as a solution to the problems associated with global variables. The scheme provided mechanisms to protect variables from violations of various sorts by contained blocks, and to allow limited access to certain variables by selected internal blocks. Similar approaches have been suggested by others (Clark 1971, White 1972, and Ichbiah 1974). We believe that some of the mechanisms proposed were appropriate, but that they were inappropriately placed in the LPS.

1. Introduction

Programming a large system in any typical programming language available today is an exercise in obscuration. We work hard at discovering the inherent structure in a problem and then structuring our solution in a compatible way. Research into "structured programming" (Dijkstra 1972) tells us that this approach will lead to readable, understandable, provable, and modifiable solutions. However, current languages discourage the accurate recording of the overall solution structure; they force us to write programs in which we are so preoccupied with the trees that we lose sight of the forest, as do the readers of our programs!

6 Reasons why Langchain Sucks

Woyera Follow 4 min read · Sep 8, 2023

386 10

Why AI Agents Break in Production

Ryan Knight Follow 8 min read · Jul 16, 2025

why we no longer use LangChain for building our AI agents

When abstractions do more harm than good: Lessons learned using LangChain in production and what we should've done instead

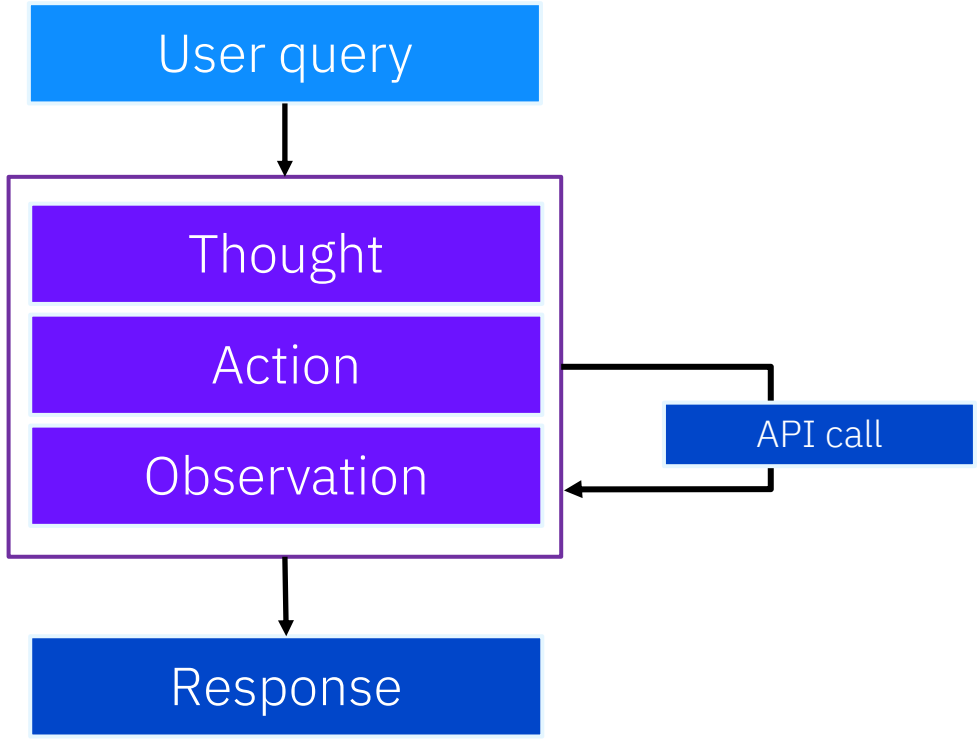
What is “generative computing”?

Generative computing is the idea that LLMs can function as computing elements that are a part of, not separate from, the rest of computer science.

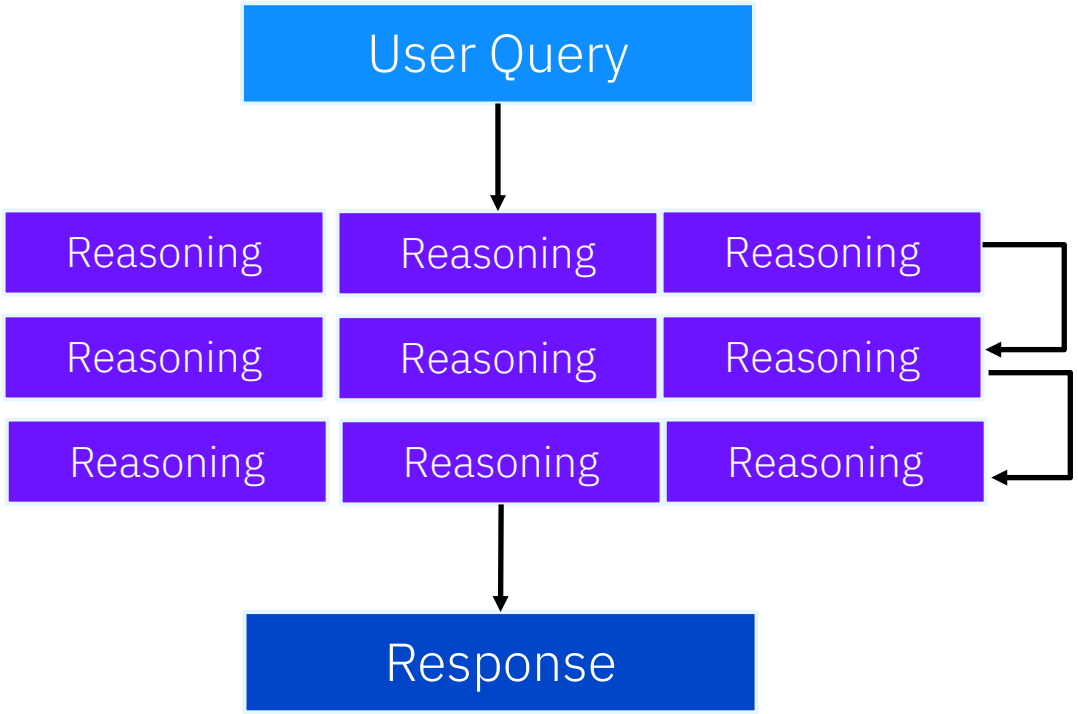
Like computing in general, this will have many facets, spanning engineering practice, system design, theory, hardware, etc.

Many aspects are already emerging in the field, but they could use a nudge, and we’re building tools to accelerate progress.

“Agents”



Inference Scaling



“Chain of agents”

Chain of Agents: Large Language Models Collaborating on Long-Context Tasks

Yusen Zhang[♣*], **Ruoxi Sun**[◇], **Yanfei Chen**[◇], **Tomas Pfister**[◇], **Rui Zhang**[♣†], **Sercan Ö. Arik**[◇†]
[♣] Penn State University, [◇] Google Cloud AI Research
{yfz5488, rmz5227}@psu.edu, {ruoxis, yanfeichen, tpfister, soarik}@google.com

Abstract

Addressing the challenge of effectively processing long contexts has become a critical issue for Large Language Models (LLMs). Two common strategies have emerged: 1) reducing the input length, such as retrieving relevant chunks by Retrieval-Augmented Generation (RAG), and 2) expanding the context window limit of LLMs. However, both strategies have drawbacks: input reduction has no guarantee of covering the part with needed information, while window extension struggles with focusing on the pertinent information for solving the task. To mitigate these limitations, we propose *Chain-of-Agents (CoA)*, a novel framework that harnesses multi-agent collaboration through natural language to enable information aggregation and context reasoning across various LLMs over long-context tasks. CoA consists of multiple worker agents who sequentially communicate to handle different segmented portions of the text, followed by a manager agent who synthesizes these contributions into a coherent final output. CoA processes the entire input by interleaving reading and reasoning, and it mitigates long context focus issues by assigning each agent a short context. We perform comprehensive evaluation of CoA on a wide range of long-context tasks in question answering, summarization, and code completion, demonstrating significant improvements by up to 10% over strong baselines of RAG, Full-Context, and multi-agent LLMs.

1 Introduction

Despite their impressive performance across a wide range of scenarios, LLMs struggle with tasks that involve long contexts [8, 57, 52]. Numerous application scenarios demand extremely long contexts, such as question answering [78, 20, 63], document and dialogue summarization [23, 84, 83, 82, 12], and code completion [18, 39], where the inputs contain entire books [29, 30] and long articles [14].

To tackle the challenge with long context tasks, two major directions have been explored as shown in Table 1: *input reduction* and *window extension*. *Input reduction* reduces the length of the input context before feeding to downstream LLMs. Truncation approaches [1, 61] directly truncate the input. Retrieval Augmented Generation (RAG) [74] extends this direction by retrieving the most relevant chunks through embedding similarity. However, because of low retrieval accuracy, LLMs could receive an incomplete context for solving the task, hurting performance. *Window extension* extends the context window of LLMs via finetuning to consume the whole input [13, 40, 43]. For example, Claude-3 [5] directly allows reading 200k tokens for each input. However, when the window becomes longer, LLMs struggle to focus on the needed information to solve the task, suffering from ineffective context utilization such as the *“lost in the middle”* issue [33, 3, 38].

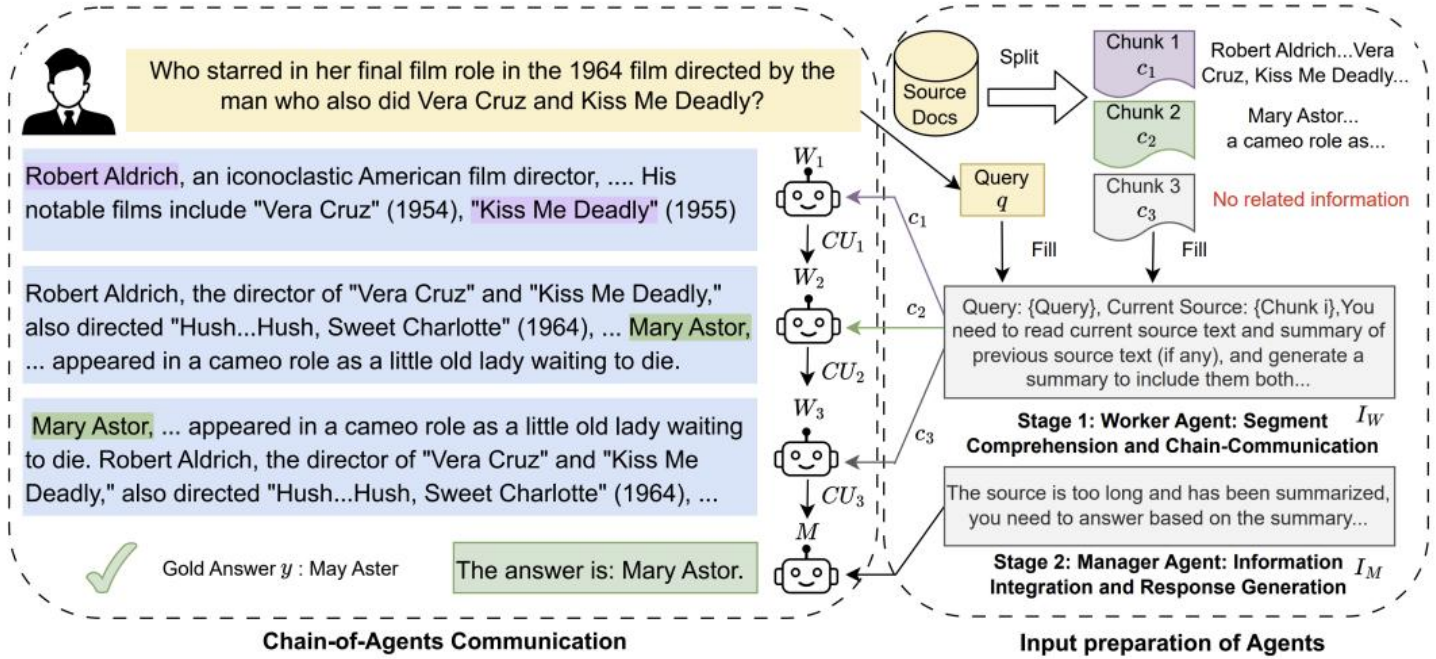


Figure 1: Overview of Chain-of-Agents, a training free, task agnostic, and highly-interpretable framework that harnesses multi-agent collaboration for long-context tasks. It consists of multiple worker agents who sequentially communicate to handle different segmented portions of the text, followed by a manager agent who synthesizes these contributions into a coherent final output.

Divide-and-conquer algorithm

35 languages

Article Talk Read Edit View history Tools

From Wikipedia, the free encyclopedia

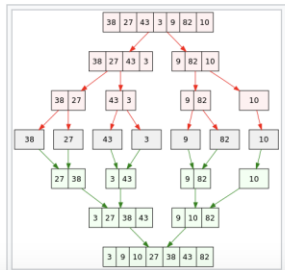
In computer science, **divide and conquer** is an algorithm design paradigm. A divide-and-conquer algorithm recursively breaks down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

The divide-and-conquer technique is the basis of efficient algorithms for many problems, such as sorting (e.g., quicksort, merge sort), multiplying large numbers (e.g., the Karatsuba algorithm), finding the closest pair of points, syntactic analysis (e.g., top-down parsers), and computing the discrete Fourier transform (FFT).[1]

Designing efficient divide-and-conquer algorithms can be difficult. As in mathematical induction, it is often necessary to generalize the problem to make it amenable to a recursive solution. The correctness of a divide-and-conquer algorithm is usually proved by mathematical induction, and its computational cost is often determined by solving recurrence relations.

Divide and conquer [edit]

The divide-and-conquer paradigm is often used to find an optimal solution of a problem. Its basic idea is to decompose a given problem into two or more similar, but simpler, subproblems, to solve them in turn, and to compose their solutions to solve the given problem. Problems of sufficient simplicity are solved directly. For example, to sort a given list of *n* natural numbers, split it into two lists of about *n*/2 numbers each, sort each of them in turn, and interleave both results appropriately to obtain the sorted version of the given list (see the picture). This approach is known as the merge sort algorithm.



The name "divide and conquer" is sometimes applied to algorithms that reduce each problem to

DEV Find related posts... Powered by Algolia

81 17 137 ...

Mikhail Levkovsky
Posted on Aug 17, 2019

65 16

The Holy Trinity of Functional Programming: Map, Filter and Reduce

#javascript #typescript #webdev #codenewbie

You've watched the Youtube videos, you've done the tutorials on Pluralsight and you consider yourself a Javascript expert.

However, now you want to hang with all of the cool kids and talk about functional programming. *For loops* are for suckers (well not really), higher order functions are all the rage and you want to understand what this means. Well the very first step is understanding the 3 most popular methods which are: Map, Filter and Reduce.

LISP I

PROGRAMMER'S MANUAL

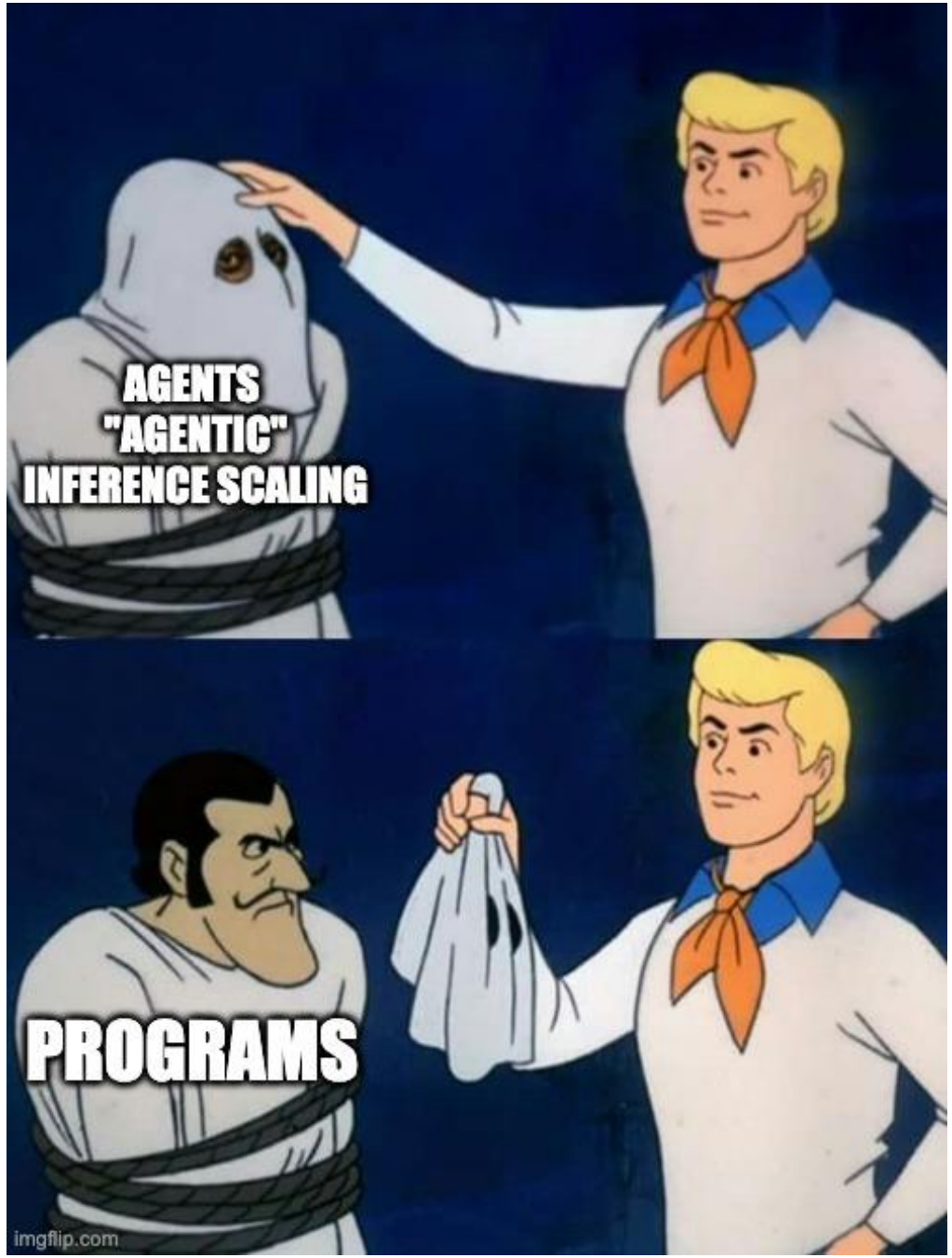
March 1, 1960

Artificial Intelligence Group

J. McCarthy
R. Brayton
D. Edwards
P. Fox
L. Hodes
D. Luckham
K. Maling
D. Park
S. Russell

COMPUTATION CENTER
and
RESEARCH LABORATORY OF ELECTRONICS
Massachusetts Institute of Technology
Cambridge, Massachusetts

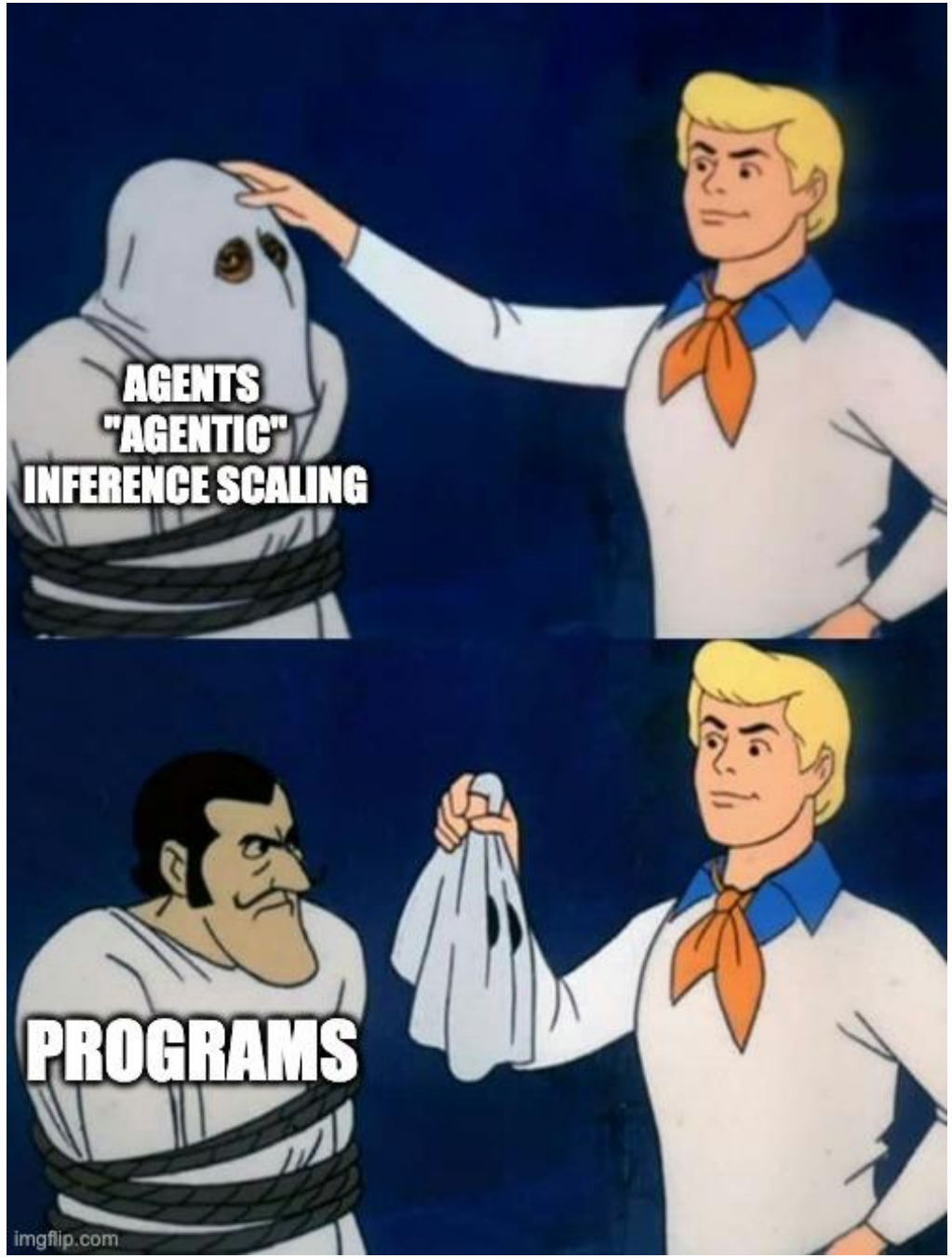
March 1, 1960


AGENTS
"AGENTIC"
INFERENCE SCALING
PROGRAMS
imgflip.com

[...Commercial Break...]

✓ 57s

```
# Install ollama.
!curl -fsSL https://ollama.com/install.sh | sh > /dev/null
!nohup ollama serve >/dev/null 2>&1 &

# Download the granite:3.38b weights.
!ollama pull granite3.3:8b

# install Mellea.
!uv pip install mellea[all] -q

# Run docling once to download model weights.
from mellea.stdlib.docs.richdocument import RichDocument
RichDocument.from_document_file("https://mellea.ai")

# Some UI niceness.
from IPython.display import HTML, display
def set_css():
    display(HTML("\n<style>\n pre{\n white-space: pre-wrap;\n}\n</style>\n"))
get_ipython().events.register("pre_run_cell", set_css)
```


AGENTS
"AGENTIC"
INFERENCE SCALING
PROGRAMS
imgflip.com

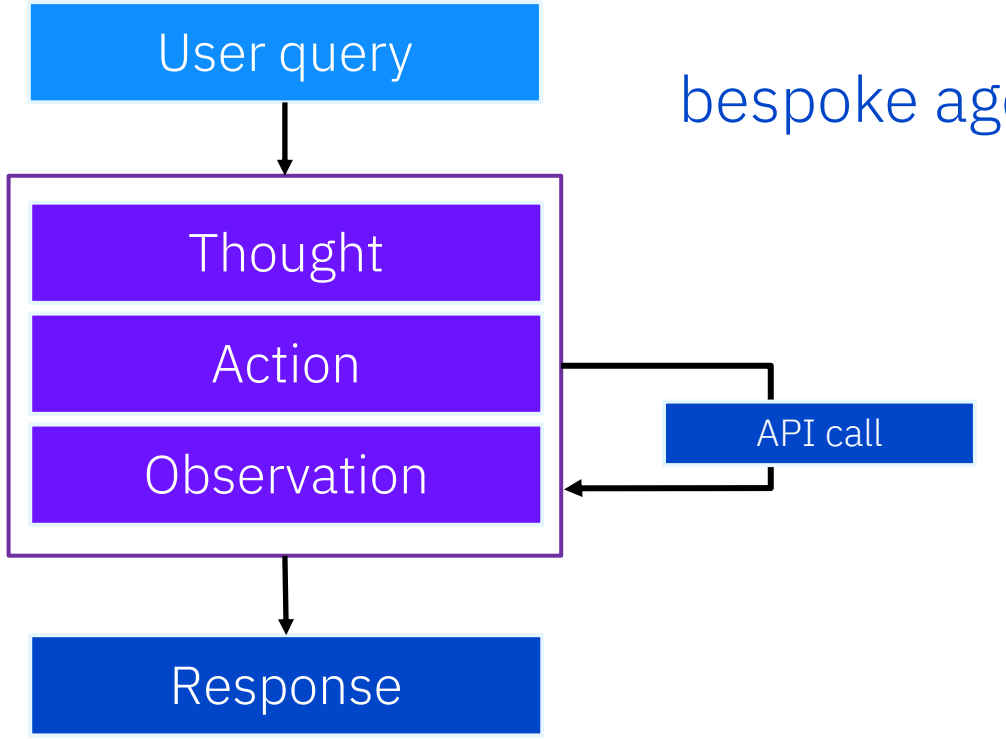
“Agents”

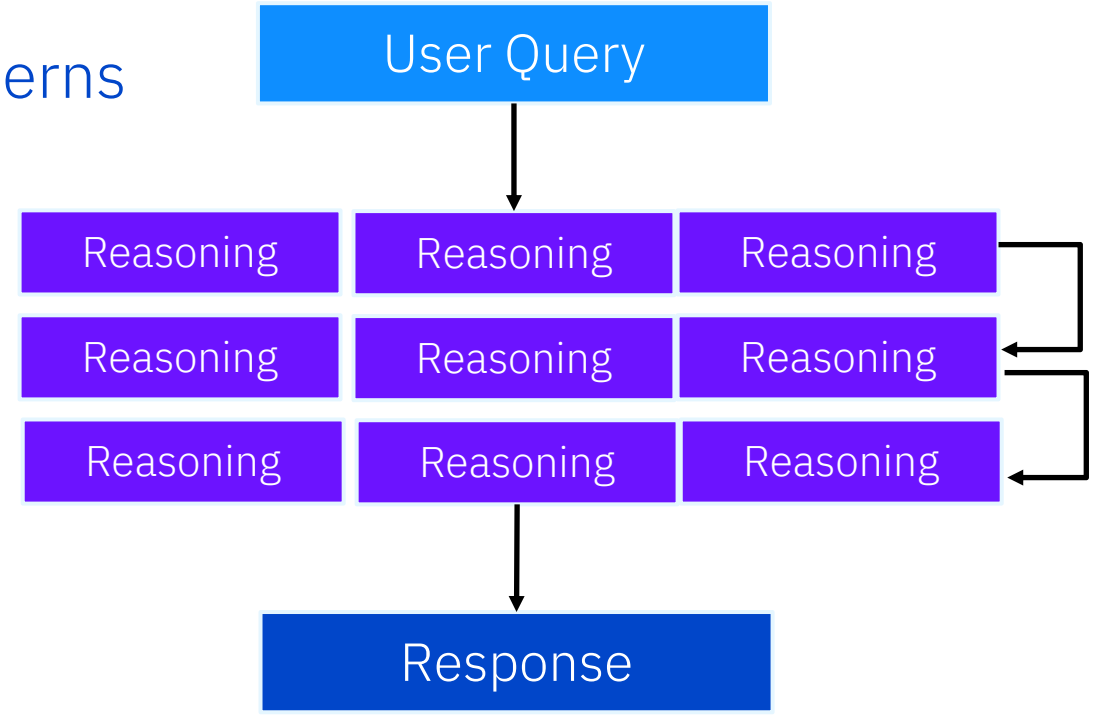
“Mixed” programs

“Inference Scaling”

Casually “agentic” programs

bespoke agent patterns





These are *design patterns*, and we are leaving a ton of opportunity on the table

What do we lose by not treating Gen AI like computation?

Abstraction boundaries

- Prompts are not portable across models, or even across point releases of the same model
- Hard to encapsulate functionality

Composability / Reusability

- “agent” level is basically microservice level composability
- Prompt libraries are cumbersome and fine-grained use is difficult if not impossible

Determinism / Predictability

- stochasticity of behavior
- LLMs don’t always follow instructions reliably

Control/Methodology/Maintainability

- what do you have to do to get the outcome you want?
- is a 10k word essay maintainable?

“The second most remarkable thing about LLMs is that they can understand and produce fluent natural language. The most remarkable thing about them is that they do the wrong thing 5-50% of the time”

— Winston Churchill

Consequence: we need structures that make it a first-class concern to check your work, and ideally, we should build new tools to allow interrogation of certainty

LLMs are particularly bad at control flow

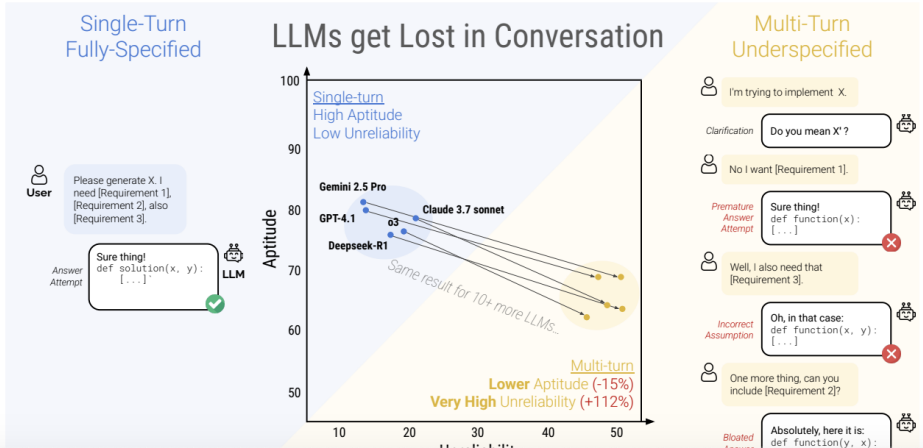
LLMs Get Lost In Multi-Turn Conversation

Philippe Laban[*,◇] **Hiroaki Hayashi**[*,♣] **Yingbo Zhou**[♣] **Jennifer Neville**[◇]
[◇]Microsoft Research [♣]Salesforce Research
{plaban,jenneville}@microsoft.com
{hiroakihayashi,yingbo.zhou}@salesforce.com

Abstract

Large Language Models (LLMs) are conversational interfaces. As such, LLMs have the potential to assist their users not only when they can fully specify the task at hand, but also to help them define, explore, and refine what they need through multi-turn conversational exchange. Although analysis of LLM conversation logs has confirmed that underspecification occurs frequently in user instructions, LLM evaluation has predominantly focused on the single-turn, fully-specified instruction setting. In this work, we perform large-scale simulation experiments to compare LLM performance in single- and multi-turn settings. Our experiments confirm that all the top open- and closed-weight LLMs we test exhibit significantly lower performance in multi-turn conversations than single-turn, with an average drop of 39% across six generation tasks. Analysis of 200,000+ simulated conversations decomposes the performance degradation into two components: a minor loss in aptitude and a significant increase in unreliability. We find that LLMs often make assumptions in early turns and prematurely attempt to generate final solutions, on which they overly rely. In simpler terms, we discover that **when LLMs take a wrong turn in a conversation, they get lost and do not recover**.

Microsoft/lost_in_conversation datasets/Microsoft/lost_in_conversation



Benchmarking Complex Instruction-Following with Multiple Constraints Composition

Bosi Wen[1,†,*] **Pei Ke**[3,*] **Xiaotao Gu**[2] **Lindong Wu**[2] **Hao Huang**[2] **Jinfeng Zhou**[1]

Category	And	Chain			Selection				Selection & Chain			All
Nesting Depth	1	1	2	Avg.	1	2	≥ 3	Avg.	2	≥ 3	Avg.	Avg.
Closed-Source Language Models												
GPT-4-1106	0.881	**0.787**	0.759	0.766	**0.815**	**0.772**	**0.694**	**0.765**	0.802	0.626	0.675	**0.800**
Claude-3-Opus	**0.886**	0.784	**0.779**	**0.780**	0.764	0.749	0.592	0.724	0.695	0.576	0.609	0.788
GLM-4	0.868	0.763	0.739	0.745	0.768	0.739	0.626	0.724	**0.809**	**0.647**	**0.692**	0.779
ERNIEBot-4	0.866	0.749	0.735	0.738	0.725	0.696	0.649	0.692	0.756	0.600	0.643	0.764
GPT-3.5-Turbo-1106	0.845	0.686	0.630	0.644	0.661	0.561	0.475	0.561	0.565	0.482	0.505	0.682
Open-Source Language Models												
Qwen1.5-72B-Chat	0.873	0.749	0.730	0.735	0.751	0.698	0.521	0.675	0.611	0.521	0.546	0.752
Llama-3-70B-Instruct	0.858	0.769	0.722	0.733	0.747	0.704	0.675	0.706	0.573	0.571	0.571	0.757
InternLM2-20B-Chat	0.796	0.666	0.648	0.652	0.648	0.599	0.543	0.597	0.611	0.488	0.522	0.678
Qwen1.5-14B-Chat	0.817	0.657	0.636	0.641	0.622	0.621	0.536	0.606	0.550	0.435	0.467	0.680
Baichuan2-13B-Chat	0.760	0.583	0.517	0.533	0.571	0.479	0.404	0.480	0.443	0.409	0.418	0.591
Llama-3-8B-Instruct	0.778	0.669	0.568	0.592	0.597	0.552	0.483	0.546	0.626	0.429	0.484	0.638
Mistral-7B-Instruct	0.737	0.574	0.556	0.560	0.554	0.493	0.411	0.488	0.534	0.374	0.418	0.592
Qwen1.5-7B-Chat	0.802	0.598	0.611	0.608	0.519	0.564	0.570	0.558	0.634	0.491	0.531	0.658
InternLM2-7B-Chat	0.755	0.633	0.598	0.607	0.532	0.568	0.525	0.555	0.550	0.432	0.465	0.634
ChatGLM3-6B-Chat	0.701	0.556	0.490	0.506	0.455	0.430	0.411	0.431	0.573	0.312	0.384	0.546

significant deficiencies in existing LLMs when dealing with complex instructions with multiple constraints composition[1].

Mellea Lab 1

Hello, Mellea

Hello, Mellea

```
import mellea


m = mellea.start_session()


answer = m.chat(”Tell me some fun historical trivia
about Computer Science at Georgia Tech.")

print(answer.content)
```

Hello, Mellea

Loads **granite3.3:8b** with a `SimpleContext`

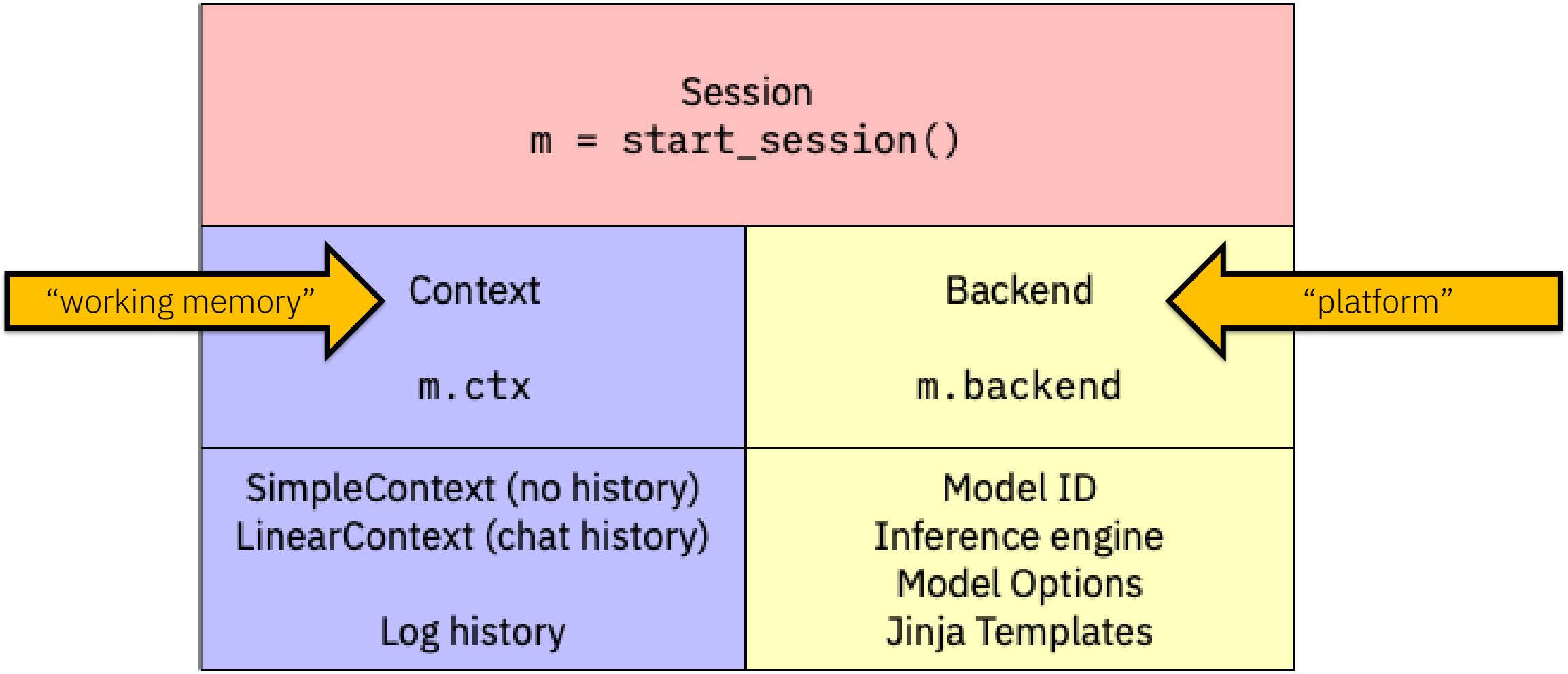
```python
import mellea


m = mellea.start_session()


answer = m.chat("Tell me some fun historical...")

print(answer.content)
```

Hello, Mellea

Loads **granite3.3:8b** with a `SimpleContext`

```python
import mellea


m: mellea.MelleaSession = mellea.start_session()


answer = m.chat("Tell me some fun historical...")

print(answer.content)
```

What's in a Session?



Hello, Mellea

User: Tell me...

```python
import mellea

m = mellea.start_session()

answer = m.chat("Tell me some fun historical...")

print(answer.content)
```

Hello, Mellea

...blocking call...

```python
import mellea

m = mellea.start_session()

answer = m.chat("Tell me some fun historical...")

print(answer.content)
```

Hello, Mellea

Prints out the model's response.

```python
import mellea

m = mellea.start_session()

answer = m.chat("Tell me some fun historical...")

print(answer.content)
```

Mellea Lab 2

Instruct→

Validate→

Repair

Instruct – Validate – Repair

```
email_candidate = m.instruct(
    "Write an email to Olivia using the notes following notes: ….",
)
```

Instruct – **Validate** – Repair

Default validation strategy is LLLM-as-a-Judge.

```
requirements = [
    req("The email should have a salutation"),
    req("Use only lower-case letters",
        validation_fn=simple_validate(lambda x: x.lower() == x)),
    check("Do not mention purple elephants."),
]

email_candidate = m.instruct(
    "Write an email to Olivia using the notes following notes: ….",
    requirements=requirements
)
```

Instruct – **Validate** – Repair

But provide your own logic when possible!

```
requirements = [
    req("The email should have a salutation"),
    req("Use only lower-case letters",
        validation_fn=simple_validate(lambda x: x.lower() == x)),
    check("Do not mention purple elephants."),
]

email_candidate = m.instruct(
    "Write an email to Olivia using the notes following notes: ….",
    requirements=requirements
)
```

Instruct – **Validate** – Repair

Enforce without prompting.

```
requirements = [
    req("The email should have a salutation"),
    req("Use only lower-case letters",
        validation_fn=simple_validate(lambda x: x.lower() == x)),
    check("Do not mention purple elephants."),
]

email_candidate = m.instruct(
    "Write an email to Olivia using the notes following notes: ….",
    requirements=requirements
)
```

Instruct – **Validate** – Repair

`instruct` with requirements

```python
requirements = [
    req("The email should have a salutation"),
    req("Use only lower-case letters",
        validation_fn=simple_validate(lambda x: x.lower() == x)),
    check("Do not mention purple elephants."),
]

email_candidate = m.instruct(
    "Write an email to Olivia using the notes following notes: ….",
    requirements=requirements
)
```

Instruct – Validate – **Repair**

If reqs fail, try again. (But not forever!)

```
requirements = [...]

email_candidate = m.instruct(
    "Write an email to Olivia using the notes following notes: ….",
    requirements=requirements,
    strategy=RejectionSamplingStrategy(loop_budget=5),
)
```

Instruct – Validate – **Repair**
...defines an abstraction boundary!

```python
def write_email(m, name: str, notes: str) -> str:
    email_candidate = m.instruct(
        "Write an email to {{name}} using the notes following: {{notes}}.",
        requirements=requirements,
        strategy=RejectionSamplingStrategy(loop_budget=5),
        user_variables={"name": name, "notes": notes},
        return_sampling_results=True,
    )
    if email_candidate.success:
        return str(email_candidate.result)
    else:
        return email_candidate.sample_generations[0].value
```

Instruct – Validate – **Repair**
...defines an abstraction boundary!

```
  0%|          | 0/5 [00:00<?, ?it/s]=== 13:32:21-INFO ======
FAILED. Valid: 2/3
INFO:fancy_logger:FAILED. Valid: 2/3
 20%|██        | 1/5 [00:09<00:37,  9.30s/it]=== 13:32:31-INFO ======
FAILED. Valid: 2/3
INFO:fancy_logger:FAILED. Valid: 2/3
 40%|████      | 2/5 [00:19<00:29,  9.86s/it]=== 13:32:42-INFO ======
FAILED. Valid: 2/3
INFO:fancy_logger:FAILED. Valid: 2/3
 60%|██████    | 3/5 [00:30<00:20, 10.40s/it]=== 13:32:51-INFO ======
FAILED. Valid: 2/3
INFO:fancy_logger:FAILED. Valid: 2/3
 80%|████████  | 4/5 [00:40<00:10, 10.11s/it]=== 13:33:02-INFO ======
SUCCESS
INFO:fancy_logger:SUCCESS
 80%|████████  | 4/5 [00:50<00:12, 12.61s/it]subject: heartfelt thanks for your recent support

dear olivia,

i hope this message finds you well. i'm writing to express our team's immense gratitude for the incredible effort you've put into supporting our lab over the past few weeks. your dedication
to organizing intern events has been nothing short of remarkable, creating a welcoming and engaging atmosphere that undoubtedly enriched their experience.

additionally, your initiative in advertising our speaker series has significantly boosted attendance and interest, demonstrating your keen ability to promote valuable learning opportunities
within our community. not to mention, the smooth handling of snack delivery issues ensured that even the smallest details were taken care of with your efficiency and grace.

thank you once again for your hard work and commitment. your contributions have made a substantial impact on our lab's recent successes, and we are truly appreciative of your support.

warm regards,

[your name]
```

Instruct – Validate – **Repair**
…defines an abstraction boundary!

```
  0%|          | 0/5 [00:00<?, ?it/s]=== 15:44:13-INFO ======
SUCCESS
INFO:fancy_logger SUCCESS
  0%|          | 0/5 [00:09<?, ?it/s]subject: appreciation for your recent support in the lab

dear olivia,

i hope this message finds you well. i am writing to express our heartfelt gratitude for the exceptional assistance you've provided to our lab over the past few weeks. your dedication and hard
work have not gone unnoticed.

your efforts in organizing intern events have been instrumental in fostering a collaborative and engaging environment. similarly, your proactive approach to advertising the speaker series has
significantly boosted participation and interest among our team members. additionally, your patience and diligence in resolving issues with snack delivery ensured that everyone stayed
nourished and focused throughout their busy schedules.

we are truly thankful for your contributions and selfless commitment to the lab's success. your positive impact is immeasurable, and we feel fortunate to have you as part of our team.

thank you once again for all that you've done. we look forward to continuing to work with you in the future.

warm regards,

[your name]
```

Instruct – Validate – **Repair**
...defines an abstraction boundary!

What if even after 5 attempts we still fail?

```python
def write_email(m, name: str, notes: str) -> str:
    email_candidate = m.instruct(
        "Write an email to {{name}} using the notes following: {{notes}}.",
        requirements=requirements,
        strategy=RejectionSamplingStrategy(loop_budget=5),
        user_variables={"name": name, "notes": notes},
        return_sampling_results=True,
    )
    if email_candidate.success:
        return str(email_candidate.result)
    else:
        return email_candidate.sample_generations[0].value
```

Instruct – Validate – **Repair**
…defines an abstraction boundary!

```python
def write_email(m, name: str, notes: str) -> str:
    email_candidate = m.instruct(
        "Write an email to {{name}} using the notes following: {{notes}}.",
        requirements=requirements,
        strategy=RejectionSamplingStrategy(loop_budget=5),
        user_variables={"name": name, "notes": notes},
        return_sampling_results=True,
    )
    if email_candidate.success:
        return str(email_candidate.result)
    else:
        raise AutomationFailedException(...)
```

Puts the app developer in charge.

Mellea Lab 3

Fun with `@generative` Functions

Building Libraries with Generative Stubs

```python
from mellea import generative

# The Summarizer Library
@generative
def summarize_meeting(transcript: str) -> str:
    """Summarize the meeting transcript into a concise paragraph of main points."""

@generative
def summarize_contract(contract_text: str) -> str:
    """Produce a natural language summary of contract obligations and risks."""”

@generative
def summarize_short_story(story: str) -> str:
    """Summarize a short story, with one paragraph on plot and one paragraph on
broad themes."""
```

Building Libraries with Generative Stubs

```python
# The Decision Aides Library
@generative
def propose_business_decision(summary: str) -> str:
    """Given a structured summary with clear recommendations, propose a business decision."""

@generative
def generate_risk_mitigation(summary: str) -> str:
    """If the summary contains risk elements, propose mitigation strategies."""

@generative
def generate_novel_recommendations(summary: str) -> str:
    """Provide a list of novel recommendations that are similar in plot or theme to the short story summary."""
```

Compositionality Checks

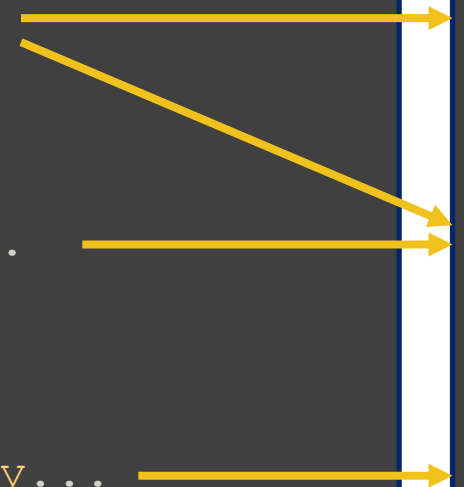
```
# The Summarizer Library
@generative
def summarize_meeting...


@generative
def summarize_contract...


@generative
def summarize_short_story...
```

```
# The Decision Aides Library
@generative
def generate_risk_mitigation...


@generative
def propose_business_decision...


@generative
def generate_novel_recommendations...
```

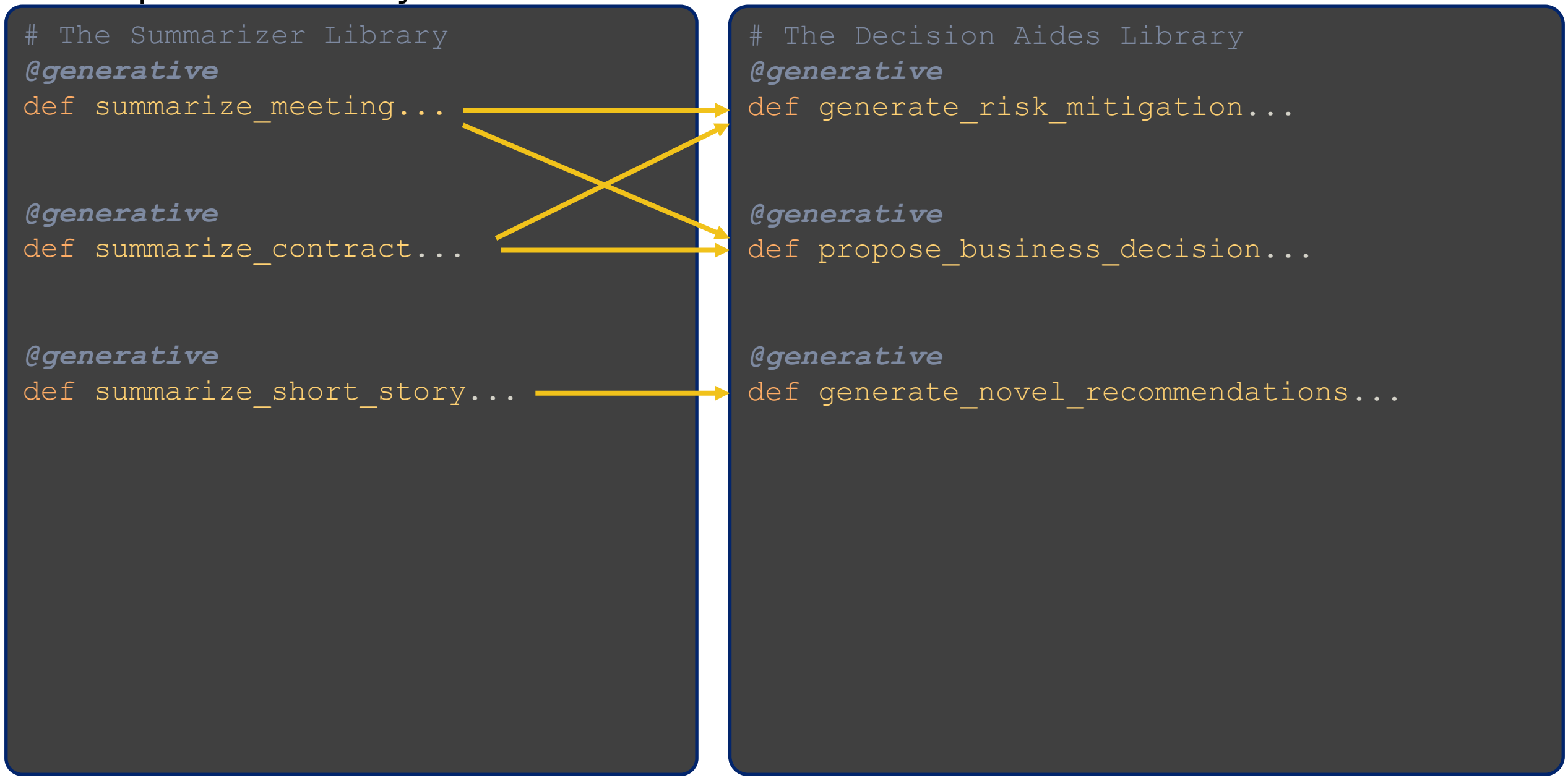
Compositionality Checks

```
# The Summarizer Library
@generative
def summarize_meeting...


@generative
def summarize_contract...


@generative
def summarize_short_story...
```

```
# The Decision Aides Library
@generative
def generate_risk_mitigation...


@generative
def propose_business_decision...


@generative
def generate_novel_recommendations...
```



Compositionality Checks



Compositionality Checks

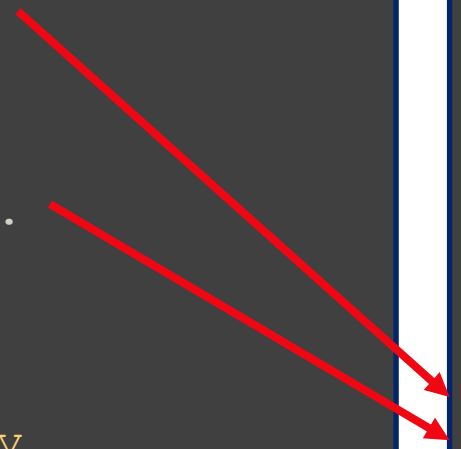
```
# The Summarizer Library
@generative
def summarize_meeting...


@generative
def summarize_contract...


@generative
def summarize_short_story...
```

```
# The Decision Aides Library
@generative
def generate_risk_mitigation...


@generative
def propose_business_decision...


@generative
def generate_novel_recommendations...
```

Compositionality Checks

```
# The Summarizer Library
@generative
def summarize_meeting...


@generative
def summarize_contract...


@generative
def summarize_short_story...
```

```
# The Decision Aides Library
@generative
def generate_risk_mitigation...


@generative
def propose_business_decision...


@generative
def generate_novel_recommendations...
```



Compositionality Checks

```
# The Summarizer Library
@generative
def summarize_meeting...


@generative
def summarize_contract...


@generative
def summarize_short_story...
```

```
# The Decision Aides Library
@generative
def generate_risk_mitigation...


@generative
def propose_business_decision...


@generative
def generate_novel_recommendations...
```

Compositionality Checks

composition check

```
generative
def summarize_meeting...


@generative
def summarize_contract...


@generative
def contains_actionable_risks(summary: str) -> Literal["yes", "no"]:
    """Check whether the summary contains references to business
risks or exposure."""

@generative
def generate_risk_mitigation...
```

Building Libraries with Generative Stubs

composition check (constrained decoding)

```
generative
def summarize_meeting...


@generative
def summarize_contract...


@generative
def contains_actionable_risks(summary: str) -> Literal["yes", "no"]:
    """Check whether the summary contains references to business
risks or exposure."""

@generative
def generate_risk_mitigation...
```

Building Libraries with Generative Stubs

```
@generative
def summarize_meeting...
@generative
def generate_risk_mitigation...
@generative
def contains_actionable_risks...


summary = summarize_meeting(m, ”meeting transcript here")

if contains_actionable_risks(m, summary=summary) == "yes":
    mitigation = generate_risk_mitigation(m, summary=summary)
    print(f"Mitigation: {mitigation}")
else:
    print("Summary does not contain actionable risks.")
```

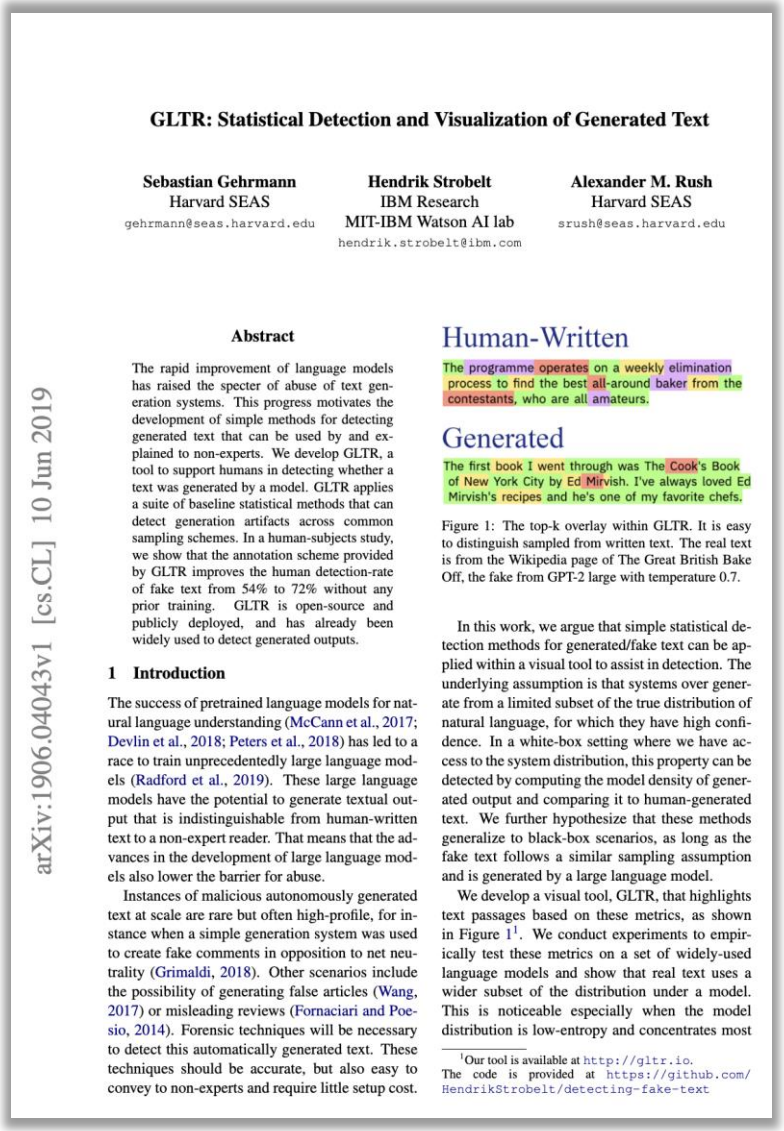
Mellea Lab 4

Mellea ❤️ Docling

Mellea ❤️ Docling

```
rd = RichDocument.from_document_file("...")

table1: Table = rd.get_tables()[0]

m = start_session()
for seed in [x * 12 for x in range(5)]:
    table2 = m.transform(
        table1,
        "Add a column 'Model' that extracts which model was used"
        " or 'None' if none.",
        model_options={ModelOption.SEED: seed},
    )
    if isinstance(table2, Table):
        print(table2.to_markdown())
        break
    else:
        print("==== TRYING AGAIN after non-useful output.====")
```

Mellea ❤️ Docling



GLTR: Statistical Detection and Visualization of Generated Text

Sebastian Gehrmann
Harvard SEAS
gehrmann@seas.harvard.edu

Hendrik Strobelt
IBM Research
MIT-IBM Watson AI lab
hendrik.strobelt@ibm.com

Alexander M. Rush
Harvard SEAS
srush@seas.harvard.edu

Abstract

The rapid improvement of language models has raised the specter of abuse of text generation systems. This progress motivates the development of simple methods for detecting generated text that can be used by and explained to non-experts. We develop GLTR, a tool to support humans in detecting whether a text was generated by a model. GLTR applies a suite of baseline statistical methods that can detect generation artifacts across common sampling schemes. In a human-subjects study, we show that the annotation scheme provided by GLTR improves the human detection-rate of fake text from 54% to 72% without any prior training. GLTR is open-source and publicly deployed, and has already been widely used to detect generated outputs.

1 Introduction

The success of pretrained language models for natural language understanding (McCann et al., 2017; Devlin et al., 2018; Peters et al., 2018) has led to a race to train unprecedentedly large language models (Radford et al., 2019). These large language models have the potential to generate textual output that is indistinguishable from human-written text to a non-expert reader. That means that the advances in the development of large language models also lower the barrier for abuse.

Instances of malicious autonomously generated text at scale are rare but often high-profile, for instance when a simple generation system was used to create fake comments in opposition to net neutrality (Grimaldi, 2018). Other scenarios include the possibility of generating false articles (Wang, 2017) or misleading reviews (Fornaciari and Poesio, 2014). Forensic techniques will be necessary to detect this automatically generated text. These techniques should be accurate, but also easy to convey to non-experts and require little setup cost.

Human-Written

The programme operates on a weekly elimination process to find the best all-around baker from the contestants, who are all amateurs.

Generated

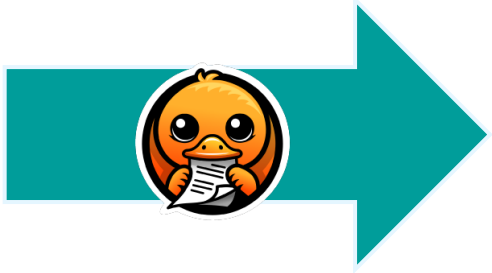
The first book I went through was The Cook's Book of New York City by Ed Mirvish. I've always loved Ed Mirvish's recipes and he's one of my favorite chefs.

Figure 1: The top-k overlay within GLTR. It is easy to distinguish sampled from written text. The real text is from the Wikipedia page of The Great British Bake Off, the fake from GPT-2 large with temperature 0.7.

In this work, we argue that simple statistical detection methods for generated/fake text can be applied within a visual tool to assist in detection. The underlying assumption is that systems over generate from a limited subset of the true distribution of natural language, for which they have high confidence. In a white-box setting where we have access to the system distribution, this property can be detected by computing the model density of generated output and comparing it to human-generated text. We further hypothesize that these methods generalize to black-box scenarios, as long as the fake text follows a similar sampling assumption and is generated by a large language model.

We develop a visual tool, GLTR, that highlights text passages based on these metrics, as shown in Figure 1[1]. We conduct experiments to empirically test these metrics on a set of widely-used language models and show that real text uses a wider subset of the distribution under a model. This is noticeable especially when the model distribution is low-entropy and concentrates most

[1]Our tool is available at http://gltr.io. The code is provided at https://github.com/HendrikStrobelt/detecting-fake-text



Feature	AUC
Bag of Words	0.63 ±0.11
(Test 1 - GPT-2) Average Probability	0.71 ±0.25
(Test 2 - GPT-2) Top-K Buckets	0.87 ±0.07
(Test 1 - BERT) Average Probability	0.70 ±0.27
(Test 2 - BERT) Top-K Buckets	0.85 ±0.09

Table 1: Cross-validated results of fake-text discriminators. Distributional information yield a higher informativeness than word-features in a logistic regression.

```
Add a column 'Model' that extracts which model was used or 'None' if none.
```

Feature	AUC	Model
Bag of Words	0.63 ± 0.11	None
(Test 1 - GPT-2) Average Probability	0.71 ± 0.25	(Test 1 - GPT-2)
(Test 2 - GPT-2) Top-K Buckets	0.87 ± 0.07	(Test 2 - GPT-2)
(Test 1 - BERT) Average Probability	0.70 ± 0.27	(Test 1 - BERT)
(Test 2 - BERT) Top-K Buckets	0.85 ± 0.09	

Mellea Lab 5 Generative Objects

Object-oriented generative programs

```
@mify(fields_include={"table"}, template="{{ table }}")
class MyCompanyDatabase:
    table: str = """| Store       | Sales   |
| ---------- | ------- |
| Northeast  | $250    |
| Southeast  | $80     |
| Midwest    | $420    |"""

    def update_sales(self, store: str, amount: str) -> MyCompanyDatabase: ...

db = MyCompanyDatabase()
db = m.transform(db, "Update the northeast sales to 1250.")
print(m.query(db, "What were sales for the Northeast branch this month?"))
```

Object-oriented generative programs

```
@mify(fields_include={"table"}, template="{{ table }}")
class MyCompanyDatabase:
    table: str = """| Store       | Sales   |
| ----------- | ------- |
| Northeast   | $250    |
| Southeast   | $80     |
| Midwest     | $420    |"""

    def update_sales(self, store: str, amount: str) -> MyCompanyDatabase:
        """ Updates the sales for a specific store. """

db = MyCompanyDatabase()
db = m.transform(db, "Update the northeast sales to 1250.")
print(m.query(db, "What were sales for the Northeast branch this month?"))
```

Fields are data

Object-oriented generative programs

```
@mify(fields_include={"table"}, template="{{ table }}")
class MyCompanyDatabase:
    table: str = """| Store       | Sales   |
| ---------- | ------- |
| Northeast  | $250    |
| Southeast  | $80     |
| Midwest    | $420    |"""

    def update_sales(self, store: str, amount: str) -> MyCompanyDatabase:
        """" Updates the sales for a specific store. """

db = MyCompanyDatabase()
db = m.transform(db, "Update the northeast sales to 1250.")
print(m.query(db, "What were sales for the Northeast branch this month?"))
```

Methods are tools

mellea.ai

An open-source library for writing generative programs.

Local-first Capability

Generative programs can do Big Model Things™ without Big Model hardware.

Robust and Composable

Requirement-driven inference pipelines result in rock-solid libraries and deployable apps.

Tutorial

mellea.ai

GitHub

generative-computing/mellea

Contact

Nathan Fulton

nathan@ibm.com

Star + Follow on GitHub

Ready to Scale

Swap out inference engines and models with ease. Massively scale inference in one line of code.

Open & Permissive

The open inference stack for building generative programs on open weight models.