Chapter 13: In-Context Learning, Prompting, and Reasoning

Learning Objectives

After reading this chapter, the reader should be able to:

1. Define in-context learning (ICL) and explain how a pre-trained autoregressive model can perform new tasks at inference time without any gradient updates, using only demonstrations provided in the prompt.
2. Apply systematic prompt engineering principles – including role specification, few-shot exemplar selection, output formatting, and system prompts – to measurably improve model performance on classification, extraction, and generation tasks.
3. Implement chain-of-thought (CoT) prompting and its variants (zero-shot CoT, self-consistency, tree-of-thought), and explain why eliciting intermediate reasoning steps improves performance on multi-step tasks.
4. Describe how tool use and function calling extend language model capabilities beyond text generation, and identify the practical boundaries where prompting fails and fine-tuning becomes necessary.

In Chapter 12, we aligned language models to be helpful, harmless, and honest through supervised fine-tuning, reward modeling, and preference optimization. An aligned model follows instructions, refuses harmful requests, and produces responses that humans prefer. But we left a question dangling: when a user types "Summarize this article in three bullet points" into a chat interface, what mechanism actually enables the model to comply? The model was never explicitly trained on that specific article. It received no gradient signal for the particular three-bullet format the user wants. It was never fine-tuned on the act of summarizing that particular text in that particular style. And yet it performs the task, often remarkably well. Something is happening at inference time that looks, from the outside, like learning – but no parameters move. Not one weight changes. The model receives a prompt, processes it through the same frozen computation graph it uses for every other input, and produces an output that is tailored to a task it has never been explicitly trained to perform. This phenomenon – in-context learning – is the subject of this chapter, and it is one of the most consequential surprises in the history of artificial intelligence.

In-context learning (ICL) was first demonstrated at scale by Brown et al. [Brown et al., 2020] with GPT-3, a 175-billion-parameter autoregressive language model. The authors showed that by simply placing a few input-output examples into the prompt – a technique they called "few-shot" learning – the model could perform translation, question answering, arithmetic, and dozens of other tasks without any parameter updates. The accuracy was not perfect, but it was startling: a model trained only to predict the next token could, at inference time, behave as though it had been fine-tuned for a specific task, simply because the task was described in its input. This chapter explores how ICL works, how to exploit it through systematic prompt engineering, how to push it further through chain-of-thought reasoning and tool use, and where it ultimately breaks down. The unifying insight, consistent with the thesis of this book established in Chapter 1, is that ICL is not a new mechanism. The model is still predicting the next token. The task specification has simply moved from the parameters to the context. We begin with the phenomenon itself and end with its limits.

13.1 In-Context Learning

Can a model learn a new task from three examples without changing a single parameter?

The phrase "in-context learning" describes a phenomenon that, upon first encounter, feels like it should not work. We take a pre-trained autoregressive language model – one whose parameters are entirely frozen – and we place a small number of input-output demonstrations into the prompt, followed by a new input. The model generates an output for the new input that is consistent with the demonstrated pattern. No gradient computation occurs. No optimizer runs. No loss function is evaluated against the demonstrations. The model's weights before and after generating the output are bit-for-bit identical. And yet the model behaves differently depending on which demonstrations we provide, as though it has learned a task-specific mapping from the examples in the prompt. This section formalizes the phenomenon, surveys the theoretical explanations proposed for it, and examines its surprising fragility with respect to seemingly trivial formatting choices.

13.1.1 Zero-Shot, One-Shot, Few-Shot

The landscape of in-context learning is organized along a single axis: the number of demonstrations placed in the prompt before the test input. At one extreme is zero-shot inference, where the model receives only a natural-language instruction describing the task and a test input, with no demonstrations at all. At the other extreme is few-shot inference, where the prompt contains anywhere from two to several dozen input-output pairs illustrating the desired behavior. Between them sits the one-shot case, where a single demonstration anchors the model's understanding of the expected format and task. This taxonomy, introduced by Brown et al. [Brown et al., 2020] in their GPT-3 paper, has become the standard vocabulary for describing how language models are deployed in practice.

To make the taxonomy concrete, consider a sentiment classification task. In the zero-shot setting, we might prompt the model with: "Classify the following movie review as Positive or Negative. Review: 'A breathtaking achievement in storytelling.' Sentiment:" – and nothing else. The model must infer, purely from the instruction and its pre-training knowledge, that it should output a single word, either "Positive" or "Negative," and it must determine which label applies to the given review. In the one-shot setting, we prepend a single labeled example: "Review: 'Terrible waste of time.' Sentiment: Negative" before presenting the test review. This demonstration serves two functions: it clarifies the expected output format (a single word on the same line) and it provides one instance of the input-output mapping the model should follow. In the few-shot setting, we provide three to five such labeled examples, covering both classes, before the test input. Each additional demonstration further constrains the model's interpretation of the task, reducing ambiguity about format, label space, and decision criteria. The formal mechanism underlying all three settings is the same autoregressive factorization we studied in Chapter 9. Given a prompt x consisting of optional instructions and demonstrations, and a desired output y, the model generates y by computing:

$$P(\mathbf{y} \mid \mathbf{x}; \theta) = \prod_{t=1}^{T} P(y_t \mid y_{<t}, \mathbf{x}; \theta), \quad \theta \text{ frozen} \tag{13.1}$$

The critical detail is that θ – the model's parameters – does not change between the zero-shot, one-shot, and few-shot settings. The only thing that changes is $\mathbf{x}$, the conditioning context. The "learning" in in-context learning is entirely a property of what the model conditions on, not of any parameter update. This is a profound reframing: the demonstrations are not training data in the

traditional sense. They are part of the input to a fixed function. The model that processes three sentiment examples followed by a test review is the same model, with the same weights, that would process a recipe request or a code-completion task if the prompt were different. What changes is the statistical context in which the next token is predicted, and that change in context is sufficient to elicit dramatically different behavior.

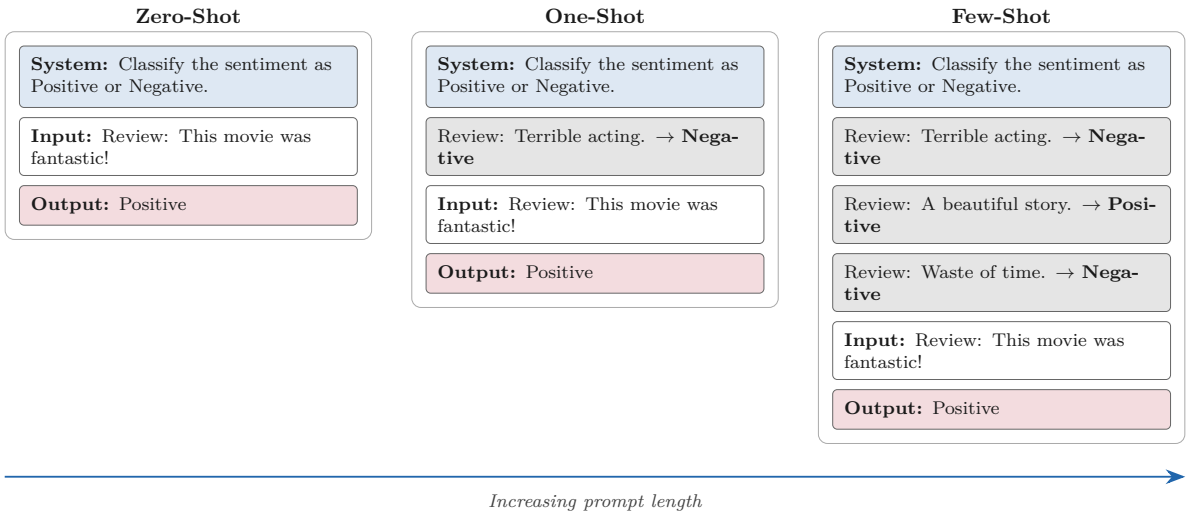

Figure 1: Figure 13.1 – Zero-Shot, One-Shot, and Few-Shot Prompts for Sentiment Classification

The empirical pattern that Brown et al. documented across dozens of tasks is straightforward: performance generally increases as we move from zero-shot to one-shot to few-shot, with the sharpest improvement often occurring between zero and one shot. Adding the first demonstration provides the model with critical information about the output format and the nature of the mapping. Subsequent demonstrations provide additional calibration, but with diminishing returns. For GPT-3 (175B parameters), few-shot performance on many benchmarks approached or matched the performance of models that had been explicitly fine-tuned for those tasks – a result that was, at the time, difficult to reconcile with the conventional understanding of learning as weight optimization. Smaller models showed weaker ICL capabilities, establishing a relationship between model scale and in-context learning that has been a recurring theme in the scaling literature. We will not claim that ICL is entirely understood – it is not, as the next subsection will make clear – but we can state with confidence that it is real, reproducible, and practically transformative.

```
def build_few_shot_prompt(exemplars, test_input, task_desc=""):
    """Construct a few-shot prompt from exemplars and test input."""
    prompt = task_desc + "\n\n" if task_desc else ""
    for inp, out in exemplars:
        prompt += f"Input: {inp}\nOutput: {out}\n\n"
    prompt += f"Input: {test_input}\nOutput:"
    return prompt

exemplars = [
    ("I love this movie!", "Positive"),
    ("Terrible waste of time.", "Negative"),
```

```
    ("A masterpiece of storytelling.", "Positive"),
]
# Zero-shot: no exemplars
print("=== Zero-shot ===")
print(build_few_shot_prompt([], "The acting was phenomenal.",
      "Classify the sentiment as Positive or Negative."))
# Few-shot: with exemplars
print("\n=== Few-shot ===")
print(build_few_shot_prompt(exemplars, "The acting was phenomenal."))
# Effect of ordering: reversed exemplars
print("\n=== Reversed order ===")
print(build_few_shot_prompt(exemplars[::-1], "The acting was phenomenal."))
```

13.1.2 How Does ICL Work? Theoretical Perspectives

We have established that in-context learning works empirically. The harder question is *why* it works. The model changed zero parameters. Not one weight moved. Yet it performed the task. Three competing hypotheses have emerged, each with supporting evidence, and the honest answer is that the field has not settled on a single explanation. We present all three, along with their strengths and limitations, because the current state of understanding is genuinely uncertain.

The first hypothesis frames ICL as *implicit Bayesian inference*. Under this view, developed by Xie et al. [Xie et al., 2022], the model has internalized a prior distribution over tasks during pre-training. When demonstrations appear in the prompt, they function as observed data that updates this prior, and the model's output represents the posterior-predictive distribution over the next token given the inferred task. Concretely, if the model has learned that certain prompt patterns correspond to sentiment classification and others to translation, the demonstrations serve as evidence for which task is being requested, and the model generates accordingly. The Bayesian framing is appealing because it explains why ICL is sensitive to the distribution of demonstrations: if the demonstrations are inconsistent (mixing positive and negative labels randomly), the posterior over tasks becomes diffuse, and performance degrades. The limitation of this hypothesis is that it operates at a high level of abstraction – it explains the functional behavior of ICL but says little about the computational mechanism through which Transformer layers implement Bayesian inference.

The second hypothesis, proposed by Akyurek et al. [Akyurek et al., 2023] and developed independently by von Oswald et al. [von Oswald et al., 2023], makes a more specific and mechanistic claim: Transformers implement gradient descent internally, and in-context demonstrations function as training data for this implicit optimization. The attention mechanism, in this view, computes something analogous to a gradient step: it reads the input-output pairs, adjusts an internal representation, and produces an output that is equivalent to what a linear model would produce after being trained on those pairs by gradient descent. Akyurek et al. demonstrated this equivalence rigorously for linear regression tasks, showing that Transformers trained to perform ICL on synthetic data produce outputs that are numerically equivalent to the outputs of gradient descent applied to the same data. This is a striking result: the Transformer's forward pass, with no explicit optimization loop, reproduces the effect of gradient-based learning. The limitation, which the authors acknowledge, is that the equivalence has been established primarily for linear problems. Whether the same mechanism operates on the complex, nonlinear NLP tasks where ICL is most practically useful remains an open question. The gradient-descent metaphor may be literally true for simple tasks and only approximately or metaphorically true for harder ones.

The third hypothesis is the most deflationary: ICL is *pattern matching* against the pre-training corpus. Under this view, when a model sees a few sentiment-labeled examples followed by a test input, it is not performing any kind of learning, even implicitly. Instead, it is recognizing that the prompt resembles the format of sentiment-labeled data in its training corpus and generating text that is statistically consistent with that format. The model has seen millions of examples of "Review: ... Sentiment: Positive" patterns during pre-training, and the demonstrations simply activate this pre-existing association. This hypothesis explains why ICL works well on tasks and formats that are common in web text (sentiment, translation, question answering) and poorly on tasks with unusual formats that the model is unlikely to have encountered. It also explains why changing the label names from "Positive/Negative" to "Foo/Bar" drastically reduces accuracy: the model has no pre-training association between "Foo" and positive sentiment. The limitation is that ICL sometimes succeeds on tasks that are genuinely novel – tasks that could not plausibly have appeared in the training data in the demonstrated format – which suggests that something beyond pure pattern matching is occurring.

The most intellectually honest position, and the one we adopt in this book, is that ICL likely involves a mixture of all three mechanisms. For tasks and formats that are well-represented in the pre-training data, pattern matching dominates. For tasks that are novel in format but familiar in structure, something like implicit Bayesian inference or gradient descent may operate. The relative contribution of each mechanism almost certainly varies with task complexity, model scale, and prompt design. This uncertainty is not a weakness of the field – it is a reflection of how recently ICL was discovered and how surprising its effectiveness has been.

13.1.3 Sensitivity to Prompt Format and Exemplar Choice

The practical power of in-context learning comes with a practical warning: ICL is brittle in ways that can catch practitioners off guard. Small changes to the prompt that a human reader would consider inconsequential – reordering the demonstrations, changing the label names, altering the separator character – can shift accuracy by twenty or thirty percentage points. This sensitivity is not a minor implementation detail. It means that ICL performance is not a reliable measure of a model's underlying capability for a task. It is, instead, a joint measure of capability and prompt design, and the prompt-design component can dominate.

The sensitivity manifests along several dimensions that practitioners must understand. First, exemplar ordering matters more than most people expect. Autoregressive models process tokens sequentially, and empirical evidence shows that most models exhibit a *recency bias*: the demonstrations closest to the test input exert the strongest influence on the output. Placing the most relevant or most informative exemplar last, immediately before the test input, typically improves accuracy by several percentage points. Conversely, burying the most relevant exemplar at the beginning of a long prompt can cause the model to partially ignore it, particularly in models with weaker long-range attention patterns. Second, label names carry semantic weight that interacts with the model's pre-training knowledge. Changing sentiment labels from "Positive" and "Negative" to "A" and "B" can halve accuracy, because the model has strong pre-training associations between the word "Positive" and favorable sentiment but no such association for the letter "A." This observation has a practical implication: label names should be chosen to be semantically aligned with the task, not arbitrary. Third, the number of exemplars affects performance with diminishing returns. Moving from zero to one demonstration typically produces the largest accuracy gain. Moving from one to four or five produces further improvement. Beyond eight demonstrations, additional exemplars rarely help and may actually hurt, because they consume context-window space that could be used

for the test input itself, and they can introduce contradictory patterns if the exemplars are not perfectly consistent.

Fourth, exemplar selection – which specific examples are chosen from a pool of candidates – has a larger effect on accuracy than most practitioners realize. Random selection from a labeled dataset produces high variance in ICL performance: one random draw might yield 85% accuracy while another yields 65%, on the same model and the same test set. Similarity-based selection, where exemplars are chosen to be semantically close to the test input in embedding space, consistently outperforms random selection by five to fifteen percentage points, as demonstrated by Liu et al. [Liu et al., 2022]. The intuition is that similar exemplars provide more relevant pattern information: if the test input is a movie review about acting quality, exemplars that are also about acting quality provide tighter constraints on the expected output than exemplars about plot or soundtrack. Fifth, the template format – the specific separator characters, instruction phrasing, and whitespace patterns – affects performance in ways that can seem almost random from the human perspective but reflect the model's sensitivity to distributional patterns in its training data. A prompt that uses "Input:" and "Output:" as separators may perform differently from one that uses "Question:" and "Answer:", because the model has different pre-training associations with each format. The practical lesson is clear: prompt design is not a write-once activity. It requires systematic experimentation, evaluation on held-out data, and an awareness that the "best" prompt for a given model may not transfer to a different model, a different model version, or even a different sample of test inputs. We return to this theme in Section 13.2, where we formalize prompt engineering as a systematic discipline rather than an ad hoc art.

13.2 Prompt Engineering

If the prompt is the program and the model is the computer, what are the principles of good programming?

Section 13.1 established that in-context learning is real, powerful, and fragile. The fragility creates a need for systematic methods to design prompts that reliably elicit the desired behavior from language models. This need has given rise to the discipline of *prompt engineering* – the systematic design, evaluation, and optimization of prompts to maximize model performance on a target task. The term carries connotations of both craft and science, and the tension between those connotations is real. Early prompt engineering was largely trial and error: practitioners tested dozens of phrasings, shared tips on forums, and developed intuitions that were difficult to formalize or transfer. The field has since matured, and we now have principles, frameworks, and even automated optimization tools that bring rigor to the process. This section presents the anatomy of an effective prompt, the strategies for selecting and ordering few-shot exemplars, the role of system prompts in deployed applications, and the emerging frontier of automated prompt optimization.

13.2.1 The Anatomy of an Effective Prompt

An effective prompt is not a single undifferentiated block of text. It is a structured composition of distinct components, each serving a specific function in guiding the model's behavior. Through extensive empirical work across many models and tasks, the community has converged on five core components that, when present and well-designed, consistently produce better results than unstructured prompts. These components are: the system instruction, the task description, the

few-shot exemplars, the user input, and the output format specification. Understanding each component and its role is the foundation of prompt engineering as a discipline.

The *system instruction* establishes the model's persona, expertise level, and behavioral constraints. It answers the question: "Who is the model pretending to be, and what rules govern its behavior?" A system instruction like "You are an expert financial analyst with twenty years of experience. You provide precise, data-driven answers. You never recommend specific stocks or investment products" simultaneously sets the expertise level (expert, not beginner), the communication style (precise, data-driven), and a behavioral constraint (no specific recommendations). The system instruction works because it biases the model's next-token predictions toward text that is consistent with the described persona. A model conditioned on an expert persona generates more technical vocabulary, more hedged claims, and more structured responses than the same model conditioned on a casual persona – not because it "understands" the instruction as a directive, but because the persona description shifts the statistical context toward the distribution of expert-written text in the model's training data. The *task description* specifies what the model should do with the input. It should be explicit, unambiguous, and complete. Vague task descriptions like "analyze this text" produce vague outputs because the model cannot determine whether "analyze" means summarize, critique, extract entities, classify sentiment, or identify the author's argument. A precise task description like "Extract all organization names mentioned in the following text and return them as a JSON array" leaves far less room for misinterpretation. The *few-shot exemplars*, when included, provide concrete demonstrations of the expected input-output mapping. Their role was discussed in Section 13.1; here we note that their placement within the prompt – after the system instruction and task description, before the user input – follows a logic of progressive specification: first the model learns who it is and what the task is, then it sees examples, then it receives the specific input to process.

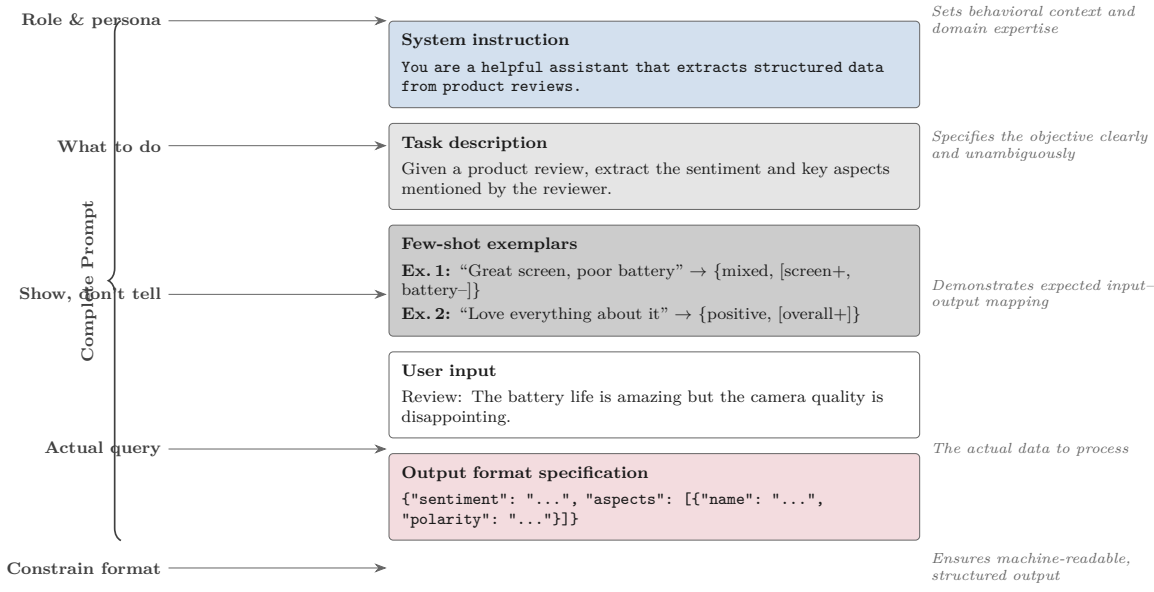

Figure 2: Figure 13.2 – The Anatomy of an Effective Prompt

The *user input* is the specific instance the model should process – the review to classify, the document to summarize, the question to answer. Its position after the exemplars and before the output format specification ensures that the model has maximum context about the task before encountering the

specific instance. The *output format specification* constrains the structure of the model's response. For programmatic consumption – where the model's output will be parsed by downstream code rather than read by a human – format specifications are essential. Specifying "Respond with a JSON object containing the fields 'sentiment' (string: 'positive' or 'negative') and 'confidence' (float between 0 and 1)" dramatically increases the probability that the model produces valid, parseable output rather than a free-text explanation that requires additional processing. Some modern APIs provide explicit "JSON mode" that enforces syntactically valid JSON output, but even without such enforcement, clear format specifications in the prompt substantially reduce formatting errors. The theatrical analogy captures the architecture well: the system instruction sets the character, the task description sets the scene, the exemplars are the dress rehearsal, the user input is opening night, and the output format is the stage direction that ensures the performance hits its marks.

```
def build_structured_prompt(system, task, exemplars, user_input, fmt):
    """Build a prompt with system instruction, task, examples, and format."""
    parts = [f"System: {system}", f"Task: {task}"]
    if exemplars:
        parts.append("Examples:")
        for i, (inp, out) in enumerate(exemplars, 1):
            parts.append(f"  {i}. Input: {inp}\n     Output: {out}")
    parts.append(f"Format: {fmt}")
    parts.append(f"Input: {user_input}")
    parts.append("Output:")
    return "\n\n".join(parts)

prompt = build_structured_prompt(
    system="You are a medical triage assistant.",
    task="Classify the urgency of the patient's symptoms.",
    exemplars=[
        ("chest pain, shortness of breath",
         '{"urgency":"high","action":"call 911"}'),
        ("mild headache, no fever",
         '{"urgency":"low","action":"rest and hydrate"}')],
    user_input="fever 103F, stiff neck, sensitivity to light",
    fmt='Respond with JSON: {"urgency":"low|medium|high","action":"..."}'
)
print(prompt)
```

13.2.2 Few-Shot Exemplar Selection and Ordering

We argued in Section 13.1.3 that exemplar choice is a major source of variance in ICL performance. Here we formalize the strategies that practitioners use to select and order exemplars, transforming what might otherwise be a random process into a systematic one. The goal is to choose a small set of demonstrations that maximally constrains the model's interpretation of the task, given the limited budget of context-window tokens available for exemplars.

The simplest strategy is *random selection*: draw exemplars uniformly at random from a labeled dataset. This approach serves as a baseline but produces high variance – different random draws can shift accuracy by ten to twenty percentage points on the same test set. The variance arises because random selection may produce exemplars that are unrepresentative of the task distribution,

that cluster in one region of the input space while leaving other regions unillustrated, or that happen to share a superficial feature (such as all being short sentences) that is irrelevant to the task. A more principled approach is *similarity-based selection*, where exemplars are chosen to be semantically close to the specific test input. The procedure is straightforward: encode all candidate exemplars and the test input into the same embedding space (using a sentence encoder such as a pre-trained Transformer), compute cosine similarity between the test input embedding and each candidate embedding, and select the top-k most similar candidates as exemplars. This approach, demonstrated by Liu et al. [Liu et al., 2022] under the name KATE (kNN-Augmented in-conText Example selection), consistently outperforms random selection by five to fifteen percentage points across a range of tasks. The intuition is that similar exemplars provide tighter constraints on the expected output: if the test input is a customer complaint about billing, exemplars about billing complaints are more informative than exemplars about product defects.

A third strategy is *diversity-based selection*, which prioritizes coverage of the output space over similarity to the test input. For a five-class classification task, diversity-based selection ensures that each class is represented among the exemplars, preventing the model from developing a bias toward overrepresented classes. Stratified selection – choosing an equal number of exemplars from each class – is the simplest implementation. For generation tasks, diversity-based selection might choose exemplars that demonstrate different output styles, lengths, or structures, giving the model a broader template for what constitutes an acceptable response. In practice, the best strategy often combines similarity and diversity: select a diverse set of exemplars that collectively covers the output space, then within each class, prioritize exemplars that are similar to the test input. Ordering the selected exemplars also matters. The recency bias of autoregressive models means the last exemplar in the prompt exerts the strongest influence. Best practice places the most relevant exemplar last, immediately before the test input, and interleaves classes rather than grouping them to prevent the model from developing a bias toward the final class in the sequence. The number of exemplars should typically be between four and eight; beyond this point, additional exemplars consume context-window tokens without producing meaningful accuracy gains and may introduce inconsistencies that degrade performance.

13.2.3 System Prompts and Output Formatting

In deployed applications – chatbots, customer service systems, coding assistants, data extraction pipelines – the system prompt is the primary mechanism through which developers control model behavior without retraining or fine-tuning. The system prompt is architecturally distinct from the user prompt in most modern APIs: it occupies a privileged position in the context window (typically at the very beginning) and is invisible to the end user. This architectural separation allows developers to embed instructions that shape every interaction without the user seeing or being able to modify those instructions. The system prompt defines the persona ("You are a helpful coding assistant specializing in Python"), the behavioral constraints ("Never execute code on the user's behalf; only suggest code"), the output format ("Always respond with markdown-formatted code blocks"), and edge-case handling ("If the user's request is ambiguous, ask a clarifying question before proceeding"). A well-designed system prompt can transform a general-purpose language model into a domain-specific assistant that behaves consistently across thousands of user interactions.

Output formatting deserves special emphasis because it is the bridge between natural-language generation and programmatic consumption. When a model's output will be parsed by downstream code – fed into a database, rendered in a user interface, or processed by another model – the format must be consistent and machine-readable. Several techniques enforce formatting. The most

direct is explicit specification in the prompt: "Respond with a JSON object containing exactly two fields: 'category' (string) and 'confidence' (float between 0.0 and 1.0)." This works well in practice but is not foolproof; the model may occasionally produce malformed output, especially for complex schemas. Some APIs provide a dedicated "JSON mode" or "structured output" mode that constrains the model's generation to produce syntactically valid JSON, eliminating parsing errors. XML tags offer another structured format, particularly useful for extracting multiple fields from unstructured text: the model can be instructed to wrap each extracted entity in tags like `<name>...</name>` and `<date>...</date>`, which are easy to parse programmatically. Markdown formatting with headers and bullet points is appropriate for human-readable outputs that still benefit from consistent structure. The choice of format depends on the downstream consumer: JSON and XML for code, markdown for human readers, plain text for simple tasks. A common failure mode in production systems is neglecting to specify the output format, which allows the model to produce free-text responses that vary in structure from call to call and require complex, fragile parsing logic. Specifying the format explicitly, with a concrete example of the expected output structure, eliminates most of this variance. We should note, however, that system prompts are guidelines rather than guarantees. They are not cryptographic boundaries: adversarial users can sometimes override system-prompt constraints through carefully crafted inputs, a phenomenon known as *jailbreaking* that we discussed in Chapter 12. System prompts should be designed with defense in depth, combining prompt-level instructions with application-level validation of the model's outputs.

Sidebar: The Prompt Engineering Profession – Art or Science?

When GPT-3 was released in 2020, researchers noticed that its performance varied wildly with prompt phrasing. The same model could score 90% or 60% on a benchmark depending on how the instructions were worded. This sensitivity spawned a new discipline: prompt engineering. By 2022, "prompt engineer" had become a job title at major technology companies, with salaries reaching six figures in Silicon Valley. The role combined linguistics (understanding how instructions are parsed), cognitive science (reasoning about model behavior), and empirical experimentation (systematically testing prompt variations). Critics dismissed prompt engineering as a temporary skill that would be automated away, comparing it to search engine optimization – useful but shallow. Proponents argued it was the natural-language analog of programming: just as programmers write code to instruct deterministic computers, prompt engineers write natural language to instruct probabilistic language models. The formalization came with tools like DSPy [Khattab et al., 2023], which reframed prompt engineering as an optimization problem with automatic tuning. By 2025, a consensus had begun to form: prompt engineering is real and measurable (optimized prompts consistently outperform naive prompts by ten to thirty percent), but it is being increasingly automated. The manual art is giving way to systematic science, just as manual hyperparameter tuning gave way to automated search. The profession has not disappeared, but its center of gravity has shifted from "writing clever prompts" to "designing prompt optimization pipelines."

13.2.4 Automated Prompt Optimization

The sensitivity of ICL performance to prompt design creates an obvious engineering challenge: manually searching the space of possible prompts is slow, labor-intensive, and explores only a tiny fraction of the possibilities. If we view the prompt as a hyperparameter of the ICL system –

analogous to learning rate or batch size in traditional machine learning – then the natural next step is to optimize it automatically. This insight has driven a wave of research into automated prompt optimization, which we survey in this subsection.

The simplest form of automated prompt optimization is *instruction generation and selection*. Zhou et al. [Zhou et al., 2023] introduced Automatic Prompt Engineer (APE), which uses a language model to generate candidate instructions for a given task, evaluates each candidate on a validation set, and selects the best-performing one. The procedure is elegant in its simplicity: given a few input-output examples of the target task, the system prompts a language model with "Write an instruction that would cause a language model to produce the following outputs given the following inputs," generates dozens or hundreds of candidate instructions, scores each on held-out validation data, and retains the highest-scoring instruction. The resulting prompts frequently outperform human-written prompts, sometimes substantially, because the automated search explores far more of the instruction space than any human could. A more ambitious framework is DSPy [Khattab et al., 2023], which treats prompting pipelines as programs that can be compiled and optimized. In DSPy, a practitioner defines a pipeline of prompting steps – for example, retrieve relevant exemplars, format them into a prompt, generate a candidate output, and parse the result – and the framework automatically optimizes the prompt templates and few-shot examples at each step to maximize end-to-end performance on a labeled dataset. DSPy represents a shift from *prompt engineering* to *prompt programming*: the human defines the pipeline architecture, and the system optimizes the specific prompts within that architecture. The analogy to traditional machine learning is deliberate: DSPy's creators explicitly model the relationship between a prompt pipeline and its optimization as analogous to the relationship between a neural network architecture and gradient-based training. The practical implication for practitioners is that prompt optimization should be treated as a standard step in any production ICL system, just as hyperparameter tuning is a standard step in any production machine learning system. A hand-written prompt is a reasonable starting point, but it is almost certainly suboptimal, and systematic optimization – whether through APE-style instruction search or DSPy-style pipeline optimization – will typically yield meaningful improvements. This optimization requires a labeled validation set, which may seem to undercut the appeal of ICL as a "zero-data" approach. The resolution is that the validation set for prompt optimization is typically very small (fifty to a few hundred examples) and is used to select a prompt, not to train model parameters. The cost of labeling a small validation set is trivial compared to the cost of fine-tuning, and the resulting optimized prompt generalizes to the test distribution because it identifies a clearer way to specify the task, not a pattern in the validation data.

13.3 Chain-of-Thought Reasoning

What happens when you ask a model to show its work?

A student who is asked "What is 23 times 47?" and must answer immediately, in a single step, will frequently make errors. The same student, asked to show her work – "First compute 23 times 40, then 23 times 7, then add the results" – will arrive at the correct answer far more often, not because she has become smarter, but because the problem has been decomposed into a sequence of simpler subproblems, each within her reliable capability. This observation, applied to language models, is the core idea behind chain-of-thought (CoT) prompting, introduced by Wei et al. [Wei et al., 2022]. CoT prompting adds intermediate reasoning steps to the prompt's demonstrations, and the model, following the demonstrated pattern, generates its own intermediate steps before

producing a final answer. The result is a dramatic improvement in accuracy on tasks that require multi-step reasoning – arithmetic, commonsense reasoning, symbolic manipulation – with no change to the model's parameters and no additional training. This section presents CoT prompting, its zero-shot variant, the self-consistency enhancement, tree-of-thought reasoning, and the deep question of whether the model's stated reasoning traces are faithful to its actual computation.

13.3.1 Chain-of-Thought Prompting

The discovery that launched the chain-of-thought paradigm was reported by Wei et al. [Wei et al., 2022] in a paper that demonstrated a strikingly simple intervention with outsized effects. The authors took standard few-shot prompts for arithmetic and reasoning tasks and modified them in one way: instead of showing only input-output pairs (question followed by answer), they included intermediate reasoning steps between the question and the answer. For a question like "Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?", a standard few-shot demonstration would show only the answer: "11." A chain-of-thought demonstration would show: "Roger started with 5 balls. He bought 2 cans of 3 balls each, so 2 times 3 equals 6 new balls. 5 plus 6 equals 11. The answer is 11." When the model encountered a new question after seeing several such demonstrations, it generated its own chain of reasoning steps before arriving at a final answer – and that answer was correct far more often than the answer produced by a standard prompt.

The improvement was not marginal. On the GSM8K benchmark (grade school math word problems), chain-of-thought prompting improved PaLM 540B from 17.9% accuracy to 57.1% – a threefold improvement from a change that required no additional training data, no fine-tuning, and no architectural modification. On StrategyQA (a commonsense reasoning benchmark), accuracy rose from 58.1% to 74.4%. These gains were concentrated in the largest models; smaller models showed little or no benefit from CoT prompting, suggesting that the ability to generate coherent multi-step reasoning is a capability that emerges with scale. The theoretical explanation for why CoT works is elegant in its simplicity: it converts a hard single-step prediction problem into a sequence of easier prediction problems. Predicting the final numerical answer to a multi-step word problem in a single forward pass requires the model to perform all the reasoning implicitly, within the hidden representations of a single generation step. This is asking a lot of the model's latent computation. By contrast, generating each reasoning step as explicit text allows the model to "offload" intermediate results into the token sequence, where they become part of the context for subsequent generation steps. Each step – "2 times 3 equals 6," "5 plus 6 equals 11" – is a relatively simple prediction that the model can perform reliably. The chain connects these simple predictions into a complex computation that the model could not reliably perform in a single step.

We should note what CoT does *not* do. It does not add reasoning capability to a model that lacks it. A model that cannot perform single-digit addition will not benefit from chain-of-thought prompting on multi-digit arithmetic, because the individual reasoning steps are themselves beyond the model's capability. CoT is an *elicitation* technique: it surfaces capability that the model already possesses but that standard prompting fails to activate. This distinction matters for setting realistic expectations about what CoT can achieve and where it will fail.

13.3.2 Zero-Shot CoT and Self-Consistency

A natural question about chain-of-thought prompting is whether it requires carefully crafted demonstrations with handwritten reasoning steps. Kojima et al. [Kojima et al., 2022] provided

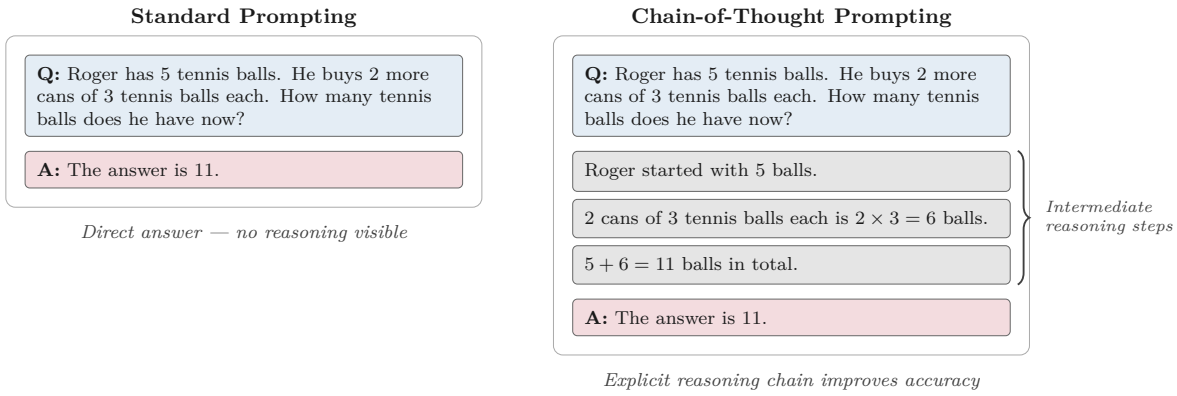

Figure 3: Figure 13.3 – Standard Prompting vs

a surprising answer: no. They showed that appending a single phrase – "Let's think step by step" – to the end of a zero-shot prompt, with no demonstrations at all, is sufficient to trigger chain-of-thought reasoning in large models. This technique, called zero-shot CoT, works because the phrase "Let's think step by step" is a pattern the model has encountered many times in its pre-training data, in contexts where step-by-step reasoning follows. The phrase activates the model's latent chain-of-thought generation capability without requiring any task-specific demonstrations. Zero-shot CoT is less effective than few-shot CoT with carefully designed demonstrations, but it is dramatically easier to apply – a single appended sentence, applicable to any task – and it consistently outperforms zero-shot prompting without the phrase.

The more substantial advancement came from Wang et al. [Wang et al., 2023], who introduced *self-consistency*, a technique that addresses a fundamental limitation of single-chain CoT: the reasoning chain can go wrong at any step, and a single error propagates to the final answer. The idea behind self-consistency is to sample multiple independent reasoning chains from the model (at temperature greater than zero, so each chain may take a different reasoning path), extract the final answer from each chain, and select the answer that appears most frequently across all chains. Formally, given a question and K independently sampled reasoning chains $r_1, r_2, \ldots, r_K$, the self-consistency answer is:

$$\hat{y} = \arg\max_{y} \sum_{i=1}^{K} \mathbb{1}[\text{answer}(r_i) = y] \tag{13.2}$$

where $\text{answer}(r_i)$ extracts the final answer from reasoning chain r_i. The insight that makes self-consistency work is an asymmetry between correct and incorrect reasoning: correct reasoning paths tend to converge on the same answer (because there is typically one correct answer), while incorrect paths produce diverse wrong answers (because there are many ways to err). Majority voting exploits this asymmetry – the correct answer accumulates votes while wrong answers scatter theirs. Self-consistency improved CoT on GSM8K from 57.1% to 74.4% for PaLM 540B, and similar gains were observed across arithmetic, commonsense, and symbolic reasoning benchmarks. The technique requires K forward passes instead of one, making it K times more expensive, but K values of ten to forty are typically sufficient, and the inference cost is often acceptable given the

accuracy improvement. The connection to the Condorcet jury theorem from political science is worth noting: if each individual "voter" (reasoning chain) has a probability of being correct that exceeds 50%, the majority vote converges to the correct answer as the number of voters increases. Self-consistency fails when individual chain accuracy is below 50% (the majority is then more likely wrong) or when errors are correlated (all chains make the same mistake).

```
import random
from collections import Counter

def simulate_cot_response(question, correct_answer, error_rate=0.3):
    """Simulate a CoT response with some error rate."""
    if random.random() > error_rate:
        return {"reasoning": "step-by-step...", "answer": correct_answer}
    wrong = correct_answer + random.choice([-1, 1, -2, 2])
    return {"reasoning": "step-by-step (with error)...", "answer": wrong}

def self_consistency(question, correct, K=20, error_rate=0.3):
    """Sample K chains and majority-vote on the final answer."""
    answers = []
    for _ in range(K):
        resp = simulate_cot_response(question, correct, error_rate)
        answers.append(resp["answer"])
    vote = Counter(answers).most_common(1)[0]
    return vote[0], vote[1] / K  # answer, confidence

# Compare single chain vs self-consistency
question = "What is 23 x 47?"
correct = 1081
single = simulate_cot_response(question, correct, error_rate=0.3)
sc_answer, sc_conf = self_consistency(question, correct, K=20)
print(f"Single chain: {single['answer']} (correct={single['answer']==correct})")
print(f"Self-consistency (K=20): {sc_answer} (confidence={sc_conf:.0%})")
```

13.3.3 Tree-of-Thought and Structured Reasoning

Chain-of-thought prompting generates a single linear chain from question to answer. But some problems resist linear reasoning: they require exploring multiple possibilities, evaluating partial solutions, and backtracking from dead ends. Consider the "Game of 24" – given four numbers, use arithmetic operations to make 24. A human solver might try one combination, realize it does not work, back up, and try another. A linear chain of thought cannot backtrack; once a step is generated, it is committed to the context. Tree-of-Thought (ToT), introduced by Yao et al. [Yao et al., 2024], addresses this limitation by generalizing CoT from a linear chain to a tree structure with explicit branching and evaluation.

The ToT procedure works as follows. At each reasoning step, instead of generating a single continuation, the model generates multiple candidate next-steps (typically three to five). Each candidate represents a different reasoning direction. The model itself is then used as a heuristic evaluator: it is prompted to assess whether each candidate is on a promising path toward the solution, and candidates judged unpromising are pruned. The surviving candidates are expanded further in

the same manner, creating a tree of reasoning paths. The tree is explored using standard search algorithms – breadth-first search (BFS) for exhaustive exploration of shallow trees, or depth-first search (DFS) for deeper exploration with backtracking. ToT dramatically outperforms linear CoT on problems with combinatorial structure: on the Game of 24, ToT achieved 74% accuracy compared to 4% for standard CoT prompting. On creative writing tasks where the model must plan a coherent narrative structure, ToT produced outputs rated significantly higher by human evaluators. However, ToT comes with a substantial computational cost: evaluating every node in the tree requires a separate model call, and a tree of depth four with branching factor three requires dozens to hundreds of calls per problem. For straightforward reasoning tasks where linear CoT suffices – most arithmetic, most commonsense reasoning – ToT is overkill, slower, and more expensive without meaningful accuracy improvement. ToT represents the frontier of structured reasoning through prompting alone: it adds search, evaluation, and backtracking to the model's reasoning process, all without parameter updates. The techniques previewed here – branching, evaluation, backtracking – are the same techniques that will reappear in Chapter 14 when we discuss agentic systems, where the "reasoning steps" become interactions with external tools and environments rather than purely internal text generation.

13.3.4 Faithfulness and Limitations of Stated Reasoning

Chain-of-thought prompting improves accuracy. The empirical evidence for this claim is overwhelming and we do not dispute it. But a harder and more unsettling question lurks behind the performance numbers: when a model generates a chain of reasoning steps and then produces a final answer, are the reasoning steps *actually driving* the answer? Or does the model arrive at the answer through some opaque internal computation and then *confabulate* a plausible-looking chain of reasoning that bears no causal relationship to how the answer was actually produced? If the latter, then CoT gives us the illusion of transparency without actual transparency – a deeply concerning prospect for any application where the reasoning matters, not just the answer.

> **Sidebar: Does the Model Really Reason? The Faithfulness Debate**
>
> Chain-of-thought prompting produces impressive results: models that show their work, explain their reasoning, and arrive at correct answers through what appears to be step-by-step logic. But a nagging question haunts the field – are these reasoning traces *faithful* to the model's actual computation? The debate intensified in 2023 when researchers demonstrated troubling inconsistencies. Turpin et al. [Turpin et al., 2023] showed that adding biased features to CoT prompts (such as "The answer is always A" in the system prompt) changed the model's final answer even when the reasoning trace appeared to ignore the bias entirely. The model claimed to reason from first principles while actually being influenced by the planted bias – a form of confabulation. Lanham et al. [Lanham et al., 2023] found that truncating reasoning traces at various points sometimes had no effect on the final answer, suggesting the model had already "decided" before generating the justification. On the other side, Wei et al.'s original CoT paper showed that corrupting intermediate reasoning steps *did* change the final answer, suggesting genuine causal dependence. The likely resolution is nuanced: reasoning traces are *partially* faithful. They influence the answer to some degree, but the model also relies on information channels invisible in the trace – attention patterns, residual-stream computation, compressed representations. For practitioners, the takeaway is clear: use CoT for its accuracy benefits but do not treat the reasoning traces as reliable explanations of model behavior. If interpretability is the goal, dedicated mechanistic interpretability

techniques are needed, not just prompting.

The evidence on both sides deserves careful examination. In favor of faithfulness, Wei et al.'s original experiments showed that corrupting the intermediate steps in CoT demonstrations – replacing correct arithmetic with incorrect arithmetic – caused the model to produce wrong final answers, even when the wrong answers were inconsistent with the corrupted steps. This suggests that the model is, at least partially, computing through the generated text rather than ignoring it. Complementary evidence comes from studies showing that the quality of reasoning steps correlates with the quality of final answers: better-articulated intermediate steps produce more accurate conclusions, which is what we would expect if the steps are causally involved in the computation. Against faithfulness, Turpin et al. [Turpin et al., 2023] demonstrated that models can be influenced by features in the prompt that are never mentioned in the reasoning trace, suggesting that the trace does not capture all the information the model uses. If a bias planted in the system prompt changes the answer without appearing in the reasoning, then the reasoning is incomplete at best and confabulatory at worst. Lanham et al.'s truncation experiments further complicate the picture: if removing the last several reasoning steps does not change the answer, then those steps were not contributing to the computation, and the model was generating them *after* having already determined its answer.

The resolution we advocate in this book is a pragmatic one. CoT should be understood as a *performance technique*, not a *transparency technique*. It improves accuracy by structuring the generation process in a way that decomposes hard predictions into sequences of easier ones. The reasoning traces it produces are partially faithful – they influence the answer more than random text would – but they are not a reliable window into the model's full computational process. For applications where the reasoning matters (medical diagnosis, legal analysis, educational tutoring), CoT traces should be treated as suggestive rather than authoritative, and critical decisions should not rest solely on the model's stated reasoning. The deeper question of what the model is "really doing" when it generates a chain of thought belongs to the field of mechanistic interpretability, which examines the model's internal representations and computations directly, rather than relying on the model's self-report. That field is beyond the scope of this chapter, but we flag it as one of the most important open problems in the study of language models.

13.4 Tool Use and Function Calling

What happens when prediction alone is not enough?

Every capability we have discussed so far – in-context learning, prompt engineering, chain-of-thought reasoning – operates within a single constraint: the model generates text and nothing else. It cannot execute code, query a database, search the web, perform precise arithmetic, or interact with any system outside the boundary of its own generation process. This constraint is fundamental: the model is, at bottom, a next-token predictor, and tokens are text. But many real-world tasks require capabilities that text prediction alone cannot provide. Computing $7{,}392 \times 4{,}583$ requires arithmetic, not statistical pattern matching over digit sequences. Answering "What is today's weather in Berlin?" requires access to real-time data that postdates the model's training cutoff. Executing a database query requires a connection to a database engine. The solution, which has become one of the most practically important developments in applied NLP, is *tool use*: connecting the language model to external systems that complement its language capabilities with computational, informational, and interactive capabilities the model lacks.

13.4.1 Why LLMs Need External Tools

The failures that motivate tool use are instructive because they reveal the boundary between what language models learn from text and what they cannot learn from text, no matter how much text they consume. Multi-digit arithmetic is the canonical example. Ask a large language model to compute 7,392 $\times$ 4,583, and it will frequently produce an answer that is close to correct but not exactly right. The model has seen many arithmetic examples during pre-training and has learned statistical patterns – it knows the answer should be roughly in the range of thirty million, and it can often get several of the leading digits right – but text prediction is not arithmetic. The model does not perform multiplication; it predicts which digit sequences are statistically likely to follow a multiplication prompt. For single-digit and some two-digit operations, this statistical approximation is often exact, because the model has memorized these common cases. For larger numbers, the statistical approach produces errors that a calculator would never make. Real-time information presents a different but equally fundamental limitation. A language model's training data has a cutoff date – it has seen no information after the date its training data was collected. Questions about current events, stock prices, weather, or any rapidly changing information are inherently unanswerable from the model's parameters alone. The model may fabricate a plausible-sounding answer (a phenomenon we discussed under "hallucination" in Chapter 12), but it cannot know what it has not seen.

Database queries, code execution, API calls, and file system operations represent a third category: capabilities that require interaction with external systems. No amount of text prediction can retrieve the contents of a specific database row or execute a Python script. The model can *generate* the SQL query or the Python code – this is a text-prediction task at which language models excel – but *executing* that query or code requires a computational environment outside the model's generation loop. The insight that resolves all three limitations is the separation of generation from execution. The model is the *planner*: it decides what action to take and expresses that action as structured text (a function call, an API request, a code snippet). An external *orchestrator* is the executor: it receives the model's action, executes it in the appropriate environment, and returns the result to the model's context. The model then incorporates the result into its ongoing generation, producing a final response that combines its language capabilities with the tool's computational capabilities. This separation is natural and mirrors how humans work: we do not perform long division in our heads when a calculator is available, and we do not memorize weather forecasts when a phone can check the current conditions. Language models using tools are doing the same thing, delegating specific subtasks to specialized systems that perform them more reliably.

13.4.2 Function Calling: Architecture and Interface

Function calling is the standardized interface through which language models invoke external tools. The architecture follows a structured protocol that has been adopted, with minor variations, by all major language model providers. Understanding this protocol is essential for building production systems that combine language model reasoning with external capabilities.

The protocol proceeds in five steps. First, the developer defines a set of available tools as structured schemas. Each schema specifies the tool's name, a natural-language description of what it does, and the parameters it accepts with their types and descriptions. For example, a calculator tool might be defined as: name "calculator," description "Evaluates a mathematical expression and returns the numerical result," parameters: expression (string, required). A weather tool might be: name "get_weather," description "Returns current weather conditions for a given city," parameters: city

(string, required), units (string, optional, default “celsius”). These schemas are included in the model’s prompt, giving it “awareness” of what tools are available and how to call them. Second, the user’s query is sent to the model along with the tool schemas. Third, the model decides whether to respond directly or to invoke a tool. This decision is itself a prediction: the model generates either a natural-language response or a structured tool-call object (typically JSON) specifying which tool to call and with what arguments. If the model determines that it can answer the user’s question from its own knowledge, it responds directly. If the question requires capabilities it lacks – precise computation, real-time data, code execution – it generates a tool call. Fourth, the orchestrator receives the tool call, validates it against the schema (checking that required parameters are present and correctly typed), executes the function in a controlled environment, and appends the result to the model’s context. Fifth, the model generates its final response, incorporating the tool’s result into a natural-language answer. This five-step cycle can repeat multiple times within a single interaction if the model determines that multiple tool calls are needed.

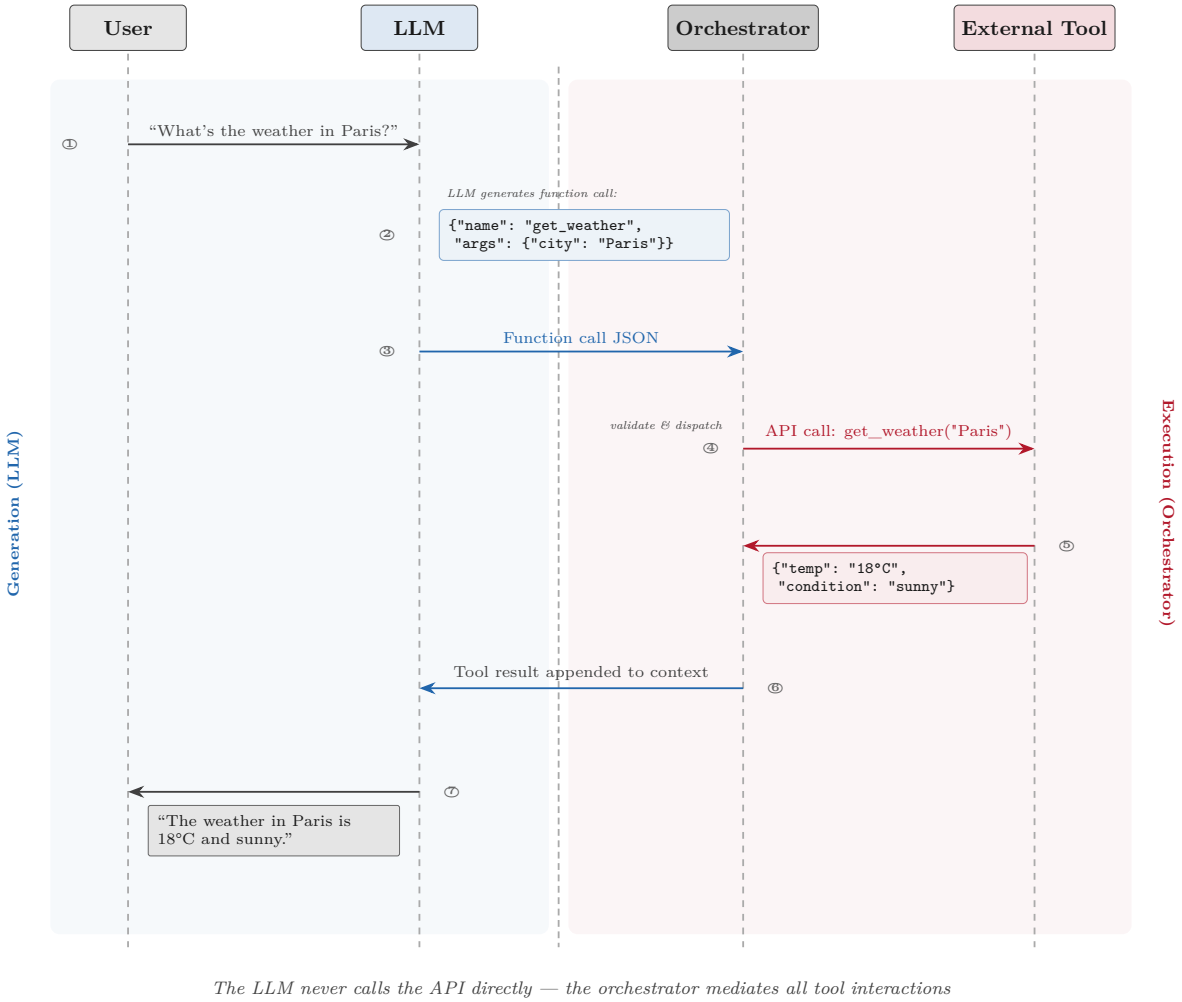

Figure 4: Figure 13.4 – The Function-Calling Interaction Loop

A critical architectural point deserves emphasis: the model never executes anything. It only generates text – text that happens to be a structured function call. The execution is performed by external

code that the developer controls. This separation is essential for safety: the model can propose actions, but a trusted orchestrator decides which actions to actually execute, can impose rate limits and access controls, and can reject dangerous requests. The Thought-Action-Observation pattern that emerges from function calling – the model thinks about what it needs, proposes an action, observes the result, and continues reasoning – is the precursor to the agentic systems we will study in Chapter 14. Function calling is, in this sense, the simplest form of agency: a single round of tool use within an otherwise standard generation process.

```
import json, math

TOOLS = {
    "calculator": lambda expr: str(eval(expr)),  # toy example only
    "sqrt": lambda x: str(math.sqrt(float(x))),
}

def simulate_tool_use(question):
    """Simulate the Thought -> Action -> Observation loop."""
    # Step 1: Model decides to use a tool (simulated)
    if "calculate" in question.lower() or any(c.isdigit() for c in question):
        tool_call = {"tool": "calculator", "input": "7392 * 4583"}
        print(f"Thought: I need to compute 7392 * 4583")
        print(f"Action: {json.dumps(tool_call)}")
        # Step 2: Orchestrator executes the tool
        result = TOOLS[tool_call["tool"]](tool_call["input"])
        print(f"Observation: {result}")
        # Step 3: Model incorporates result
        print(f"Answer: 7392 x 4583 = {result}")
        return result
    return "I can answer directly: ..."

simulate_tool_use("Calculate 7392 times 4583")
```

13.4.3 Toolformer and Self-Taught Tool Use

The function-calling architecture described above requires explicit tool schemas provided by the developer and, in most cases, models that have been fine-tuned to generate structured tool calls. Toolformer, introduced by Schick et al. [Schick et al., 2023], demonstrated a more autonomous approach: a language model that teaches itself when and how to use tools through a self-supervised process, without any human-annotated tool-use data.

The Toolformer pipeline operates in four stages. In the first stage, the system takes a pre-trained language model and a set of tool APIs – a calculator, a search engine, a calendar, a translation service. In the second stage, the system prompts the model to insert API calls into existing text at positions where a tool would be helpful. For example, given the sentence "The Eiffel Tower, completed in [YEAR], stands 330 meters tall," the model might propose inserting a search API call to retrieve the completion year. In the third stage – the filtering step that makes Toolformer distinctive – the system executes each proposed API call and evaluates whether the tool's result actually improves the model's ability to predict subsequent tokens. Specifically, it compares the perplexity of the text following the API call with and without the tool result: if the tool result

reduces perplexity (meaning the model can better predict what comes next when it has the tool's information), the API call is retained. If the tool result does not help or hurts, the call is discarded. This filtering is the self-supervision signal: the model learns which tool calls are useful by measuring their effect on its own predictive performance. In the fourth stage, the model is fine-tuned on the filtered text – the original text with successful API calls inserted inline. After fine-tuning, the model generates tool calls naturally as part of its text generation, calling tools when and where they are needed without explicit prompting.

Toolformer improved factual accuracy by twenty to thirty percent on knowledge-intensive tasks, demonstrating that self-taught tool use is both feasible and effective. The approach is significant because it moves tool use from an externally engineered capability (where developers define schemas and fine-tune on human-annotated data) to an internally learned one (where the model discovers for itself which tools to use and when). However, a common misconception should be corrected: Toolformer requires a fine-tuning step on self-generated tool-use data. It is not a pure prompting technique. At inference time, the fine-tuned model generates tool calls seamlessly, but the ability to do so was instilled through training, not through prompt design alone. Toolformer represents a bridge between the prompting world of this chapter and the agentic world of Chapter 14: it shows that language models can learn to augment their own capabilities by incorporating external tools into their generation process, a capability that agents exploit at a much larger scale.

13.5 The Limits of Prompting

We have spent this chapter building an impressive toolkit. Now we must be honest about where it breaks.

In-context learning, prompt engineering, chain-of-thought reasoning, and tool use constitute a powerful set of techniques for eliciting sophisticated behavior from pre-trained language models. But every technique has boundaries, and a responsible treatment of prompting must include an unflinching assessment of where those boundaries lie. Practitioners who understand these limits will make better design decisions; those who do not will build systems that fail in predictable but costly ways. This section catalogs the failure modes of prompting, presents a decision framework for choosing between prompting, fine-tuning, and retrieval-augmented generation (RAG), and introduces the cost-quality-latency triangle that governs deployment trade-offs.

13.5.1 Where Prompting Fails

Prompting fails in four systematic ways, each corresponding to a fundamental limitation of the in-context learning mechanism. Understanding these failure modes is not pessimism – it is engineering maturity.

The first failure mode is *domain knowledge gaps*. When the task requires knowledge that is not in the model's training data, no amount of prompt engineering can bridge the gap. A company that wants to answer questions about its internal policies, proprietary processes, or confidential documents cannot rely on prompting alone, because the model has never seen those documents. Few-shot demonstrations cannot inject sufficient domain knowledge for complex tasks: showing the model three examples of correct answers about internal policies does not give it the underlying policy knowledge needed to answer a novel question. The solution for this failure mode is retrieval-augmented generation (RAG), which we cover in Chapter 14. The second failure mode is *consistency*

at scale. In a data pipeline that processes one hundred thousand documents in the same JSON format, prompting will produce formatting errors in one to five percent of cases – missing fields, incorrect types, extra explanatory text outside the JSON structure. A fine-tuned model, trained specifically on the target format, achieves 99.5% or higher format consistency. The difference is manageable for ten documents but catastrophic for a production pipeline processing millions. The third failure mode is *latency sensitivity*. Every token in the prompt increases inference time: adding eight few-shot exemplars to a prompt might add two thousand to four thousand tokens, which at current inference speeds adds one hundred to two hundred milliseconds of latency. For applications with strict latency budgets – real-time chat, autocomplete, high-frequency trading signals – this overhead may be unacceptable. A fine-tuned model that has internalized the task performs inference with only the user's input in the prompt, eliminating the overhead of demonstrations entirely.

Failure Mode	Description	Example	Severity	Mitigation
Domain knowledge gaps	Model lacks specialized knowledge	Medical terminology errors	High	RAG augmentation
Consistency at scale	Contradicts itself in long outputs	Different answers for same question	Medium	Fine-tuning
Latency overhead	Long prompts slow inference	4K-token prompt vs 100-token	Medium	Fine-tuning / smaller models
Capability gaps	Task beyond model ability	Complex multi-step math	High	Fine-tuning
Prompt sensitivity	Small changes lead to different results	Reordering examples changes output	Medium	Automated optimization

Figure 5: Figure 13.5 – Failure Modes of In-Context Learning

The fourth failure mode is *capability gaps* – tasks where the model's base capability, even with optimal prompting, falls substantially short of what fine-tuning can achieve. Named entity recognition on specialized domains (medical entities, legal citations, financial instruments) is a canonical example: fine-tuned models outperform few-shot prompting by fifteen to twenty-five F1 points on domain-specific NER tasks, because fine-tuning allows the model to learn entity boundaries, typing conventions, and domain-specific patterns that cannot be adequately conveyed through a handful of demonstrations. The pattern across all four failure modes is consistent: prompting excels at flexibility, rapid prototyping, and low-volume tasks where the cost of fine-tuning is not justified. It fails at high-volume, high-consistency, domain-specific, or latency-critical applications where fine-tuning's upfront investment pays for itself through superior per-query performance.

13.5.2 The Prompt vs. Fine-Tune vs. RAG Decision

Given the failure modes cataloged above, practitioners need a systematic framework for deciding when to prompt, when to fine-tune, and when to deploy RAG. These three approaches are not mutually exclusive – the most capable production systems often combine all three – but understanding when each is most appropriate prevents both over-engineering (fine-tuning when prompting would suffice) and under-engineering (prompting when the task demands fine-tuning).

The decision framework follows three questions. The first question is: "Does the task require knowledge that is not in the model's training data?" If the answer is yes – the task involves proprietary documents, recent information, or domain-specific data that the model has never seen – then RAG is necessary. RAG retrieves relevant documents from an external knowledge base and includes them in the prompt, grounding the model's generation in verified information rather than relying on its potentially outdated or incomplete parametric knowledge. RAG can be combined with prompting (retrieve-then-prompt) or with fine-tuning (retrieve-then-generate with a fine-tuned model) depending on the remaining requirements. The second question is: "Does the task require high consistency and throughput across thousands or millions of examples?" If yes, fine-tuning is almost certainly necessary. Fine-tuning bakes the task specification into the model's parameters, producing consistent output formats without the variance introduced by prompt interpretation. A fine-tuned model also requires a shorter prompt at inference time (no demonstrations needed), reducing both latency and cost per query. The third question is: "Is this a prototype, an exploratory analysis, or a low-volume task where flexibility matters more than peak performance?" If yes, prompting is the right starting point. Prompting requires no labeled training data (beyond the few exemplars in the prompt), no training infrastructure, and no model deployment pipeline. It can be iterated in minutes rather than hours or days, and it can be adapted to new tasks by simply changing the prompt.

In practice, these three approaches form a continuum rather than discrete categories. A typical development workflow begins with prompting as a rapid prototype, evaluates whether the prototype meets performance requirements, transitions to RAG if external knowledge is needed, and transitions to fine-tuning if consistency, latency, or per-query cost requirements demand it. Many production systems use all three simultaneously: RAG for knowledge grounding, fine-tuning for format consistency and domain adaptation, and prompting (through the system prompt and user prompt) for task-specific instructions and flexibility. The key insight is that prompting, fine-tuning, and RAG address different axes of the problem: prompting addresses task specification, fine-tuning addresses task performance and consistency, and RAG addresses knowledge coverage. Understanding which axes matter most for a given application is the core skill of the applied NLP engineer.

13.5.3 The Cost-Quality-Latency Triangle

Every deployment of a language model, whether it relies on prompting, fine-tuning, RAG, or some combination, faces a three-way trade-off among cost, quality, and latency. Understanding this triangle is essential for making informed engineering decisions, because optimizing along one axis invariably comes at the expense of another.

Cost includes both the per-query cost of inference (proportional to the number of input and output tokens for API-based models, or to GPU-hours for self-hosted models) and the amortized setup cost of fine-tuning, data labeling, and infrastructure. A zero-shot prompt costs approximately one hundred input tokens. A few-shot prompt with five exemplars costs five hundred to two thousand input tokens – five to twenty times more expensive per query. A fine-tuned model costs nothing extra per query (the prompt is shorter) but carries a fixed setup cost that must be amortized over the number of queries: fine-tuning a model may cost anywhere from ten to several thousand dollars depending on model size and dataset size. RAG adds the cost of embedding computation and vector search for each query, plus the storage cost of the knowledge base. *Quality* encompasses accuracy, consistency, and reliability. Prompting offers good but variable quality: accuracy depends on prompt design, and output format consistency is lower than fine-tuning. Fine-tuning offers the highest quality for well-defined tasks with sufficient training data. RAG improves quality

on knowledge-intensive tasks by grounding generation in retrieved evidence. *Latency* is the time from receiving a query to producing a complete response. Longer prompts (more exemplars, more instructions) increase input processing time. RAG adds retrieval latency (typically fifty to two hundred milliseconds for a vector search). Fine-tuned models have the lowest inference latency because their prompts are shortest.

The optimal operating point depends on the application's requirements. A customer-facing chatbot might prioritize latency (users expect sub-second responses) and quality (incorrect answers erode trust), accepting higher cost for a fine-tuned model with RAG. A batch data-extraction pipeline might prioritize cost (processing millions of documents) and quality (consistent formatting), accepting higher latency for a fine-tuned model that processes documents sequentially overnight. An internal prototyping tool might prioritize cost (minimal infrastructure) and flexibility (rapid iteration on new tasks), accepting lower quality from a prompting-only approach. There is no universally optimal point in the triangle: the right choice depends on the specific constraints and priorities of each deployment. The framework's value is in making these trade-offs explicit and quantifiable rather than implicit and intuited.

The trajectory of the field is toward making all three axes cheaper and better simultaneously: faster inference engines reduce latency, more efficient fine-tuning techniques reduce setup costs, improved base models raise the quality floor of prompting, and better retrieval systems reduce RAG overhead. But the triangle itself persists: no matter how much the absolute values improve, the relative trade-offs between cost, quality, and latency remain, and practitioners will always need to navigate them. This chapter has equipped readers with the tools to do so: ICL and prompting for flexibility and rapid iteration, CoT for reasoning-intensive tasks, tool use for capabilities beyond text, and the decision framework for knowing when these techniques suffice and when stronger interventions are needed. In Chapter 14, we extend this toolkit in three directions: retrieval-augmented generation grounds the model in external knowledge, agents chain multiple reasoning and tool-use steps into autonomous workflows, and multimodal models extend prediction from text to images and beyond.

The capabilities and limitations of prompting provide a natural transition to Chapter 14, where we explore how retrieval augmentation, tool use, and agentic architectures extend language models beyond their training data.

Exercises

Exercise 13.1 (Theory – Basic). Explain why in-context learning does *not* update the model's parameters. What does it mean to say the model "learns" from demonstrations if its weights are frozen? Distinguish functional learning (behavior changes with the prompt) from parametric learning (weight changes through gradient descent). Reference Equation 13.1 in your answer.

Exercise 13.2 (Theory – Intermediate). Analyze why exemplar ordering affects ICL performance. Given a model with a context window of 2,048 tokens and a recency bias, design an exemplar ordering strategy for a five-class classification task with three exemplars per class (fifteen exemplars total). Specify which exemplar goes last and why. How would you interleave the classes to prevent the model from developing a label bias?

Exercise 13.3 (Theory – Intermediate). Explain under what conditions self-consistency (Equation 13.2) improves over single-sample chain-of-thought. When does self-consistency fail to help? Hint: consider the Condorcet jury theorem. What is the minimum accuracy of individual chains needed

for majority voting to improve over a single chain? What happens when errors across chains are correlated rather than independent?

Exercise 13.4 (Theory – Basic). Compare the information-theoretic capacity of zero-shot, one-shot, and five-shot prompts. If each exemplar adds approximately 100 tokens of task-specification information, how does this compare to the information stored in the model's parameters (billions of parameters, each storing roughly one bit of task-relevant information)? Why does ICL work despite the enormous asymmetry between the information in the prompt and the information in the parameters?

Exercise 13.5 (Theory – Advanced). Critically evaluate the "ICL as implicit gradient descent" hypothesis [Akyurek et al., 2023]. What does it mean for a Transformer to "implement" gradient descent? What evidence supports this claim? What are its limitations? Is the gradient-descent metaphor literally true, approximately true, or merely a useful analogy? Justify your answer with reference to the types of tasks for which the equivalence has been demonstrated.

Exercise 13.6 (Programming – Basic). Build a few-shot sentiment classifier using only prompting (no fine-tuning). Use a pre-trained language model and construct prompts with 0, 1, 3, and 5 exemplars drawn from a sentiment dataset (e.g., SST-2). Measure accuracy on 100 test examples and plot accuracy as a function of the number of exemplars. Report the mean and standard deviation across five different random draws of exemplars.

Exercise 13.7 (Programming – Intermediate). Implement chain-of-thought prompting with self-consistency on a math reasoning dataset (e.g., GSM8K or a manually constructed set of twenty arithmetic word problems). Sample $K = 10$ reasoning chains at temperature 0.7 for each problem. Report single-chain accuracy versus self-consistency accuracy. Plot accuracy as a function of K for $K \in \{1, 3, 5, 10, 20\}$ and compare to the Condorcet jury theorem's theoretical prediction.

Exercise 13.8 (Programming – Intermediate). Implement a simple function-calling loop with a calculator tool. Given fifty arithmetic word problems (e.g., "A store sells 47 shirts at \$23 each. What is the total revenue?"), have the model generate a calculator tool call, execute it, and incorporate the result. Compare accuracy with direct prompting (no tool use). Measure the fraction of problems where tool use produces the correct answer versus direct generation.

Exercise 13.9 (Programming – Intermediate). Compare prompt-only versus fine-tuned performance on a text classification task (e.g., topic classification on the AG News dataset). Use the same base model for both approaches. Fine-tune on 500 labeled examples. Evaluate both on 200 test examples. Report accuracy, output format consistency (fraction of outputs that are valid single-word labels), and average inference time per example. Discuss which approach you would recommend for a production pipeline processing 100,000 documents per day.

References

Akyurek, E., Schuurmans, D., Andreas, J., Ma, T., and Zhou, D. (2023). What learning algorithm is in-context learning? Investigations with linear models. *Proceedings of ICLR*.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). Language models are few-shot learners. *Advances in Neural Information Processing Systems (NeurIPS)*.

Khattab, O., Singhvi, A., Maheshwari, P., Zhang, Z., Santhanam, K., Vardhamanan, S., Haq, S., Sharma, A., Joshi, T. T., Mober, H., et al. (2023). DSPy: Compiling declarative language model calls into self-improving pipelines. *arXiv preprint arXiv:2310.03714*.

Kojima, T., Gu, S. S., Reid, M., Matsuo, Y., and Iwasawa, Y. (2022). Large language models are zero-shot reasoners. *Advances in Neural Information Processing Systems (NeurIPS)*.

Lanham, T., Chen, A., Radhakrishnan, A., Steiner, B., Denison, C., Hernandez, D., Li, D., Durmus, E., Hubinger, E., Kernion, J., et al. (2023). Measuring faithfulness in chain-of-thought reasoning. *arXiv preprint arXiv:2307.13702*.

Liu, J., Shen, D., Zhang, Y., Dolan, B., Carin, L., and Chen, W. (2022). What makes good in-context examples for GPT-3? *Proceedings of DeepLearning Inside Out (DeeLIO), ACL Workshop*.

Schick, T., Dwivedi-Yu, J., Dessi, R., Raileanu, R., Lomeli, M., Zettlemoyer, L., Cancedda, N., and Scialom, T. (2023). Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems (NeurIPS)*.

Turpin, M., Michael, J., Perez, E., and Bowman, S. (2023). Language models do not always say what they think: Unfaithful explanations in chain-of-thought prompting. *Advances in Neural Information Processing Systems (NeurIPS)*.

von Oswald, J., Niklasson, E., Randazzo, E., Sacramento, J., Mordvintsev, A., Zhmoginov, A., and Vladymyrov, M. (2023). Transformers learn in-context by gradient descent. *Proceedings of ICML*.

Wang, X., Wei, J., Schuurmans, D., Le, Q., Chi, E., Narang, S., Chowdhery, A., and Zhou, D. (2023). Self-consistency improves chain of thought reasoning in language models. *Proceedings of ICLR*.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., and Zhou, D. (2022). Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems (NeurIPS)*.

Xie, S. M., Raghunathan, A., Liang, P., and Ma, T. (2022). An explanation of in-context learning as implicit Bayesian inference. *Proceedings of ICLR*.

Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T., Cao, Y., and Narasimhan, K. (2024). Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems (NeurIPS)*.

Zhou, Y., Muresanu, A. I., Han, Z., Paster, K., Pitis, S., Chan, H., and Ba, J. (2023). Large language models are human-level prompt engineers. *Proceedings of ICLR*.