Specifications: The missing link to making the development of LLM systems an engineering discipline

Ion Stoica UC Berkeley
Matei Zaharia UC Berkeley
Joseph Gonzalez UC Berkeley
Ken Goldberg UC Berkeley
Hao Zhang UC San Diego
Anastasios Angelopoulos UC Berkeley
Shishir G. Patil UC Berkeley
Lingjiao Chen Stanford University
Microsoft Research
Wei-Lin Chiang UC Berkeley
Jared Q. Davis Stanford University

December 10, 2024

Abstract

Despite the significant strides made by generative AI in just a few short years, its future progress is constrained by the challenge of building *modular* and *robust* systems. This capability has been a cornerstone of past technological revolutions, which relied on combining components to create increasingly sophisticated and reliable systems. Cars, airplanes, computers, and software consist of components—such as engines, wheels, CPUs, and libraries—that can be assembled, debugged, and replaced. A key tool for building such reliable and modular systems is *specification*: the precise description of the expected behavior, inputs, and outputs of each component. However, the generality of LLMs and the inherent ambiguity of natural language make defining specifications for LLM-based components (e.g., agents) both a challenging and urgent problem. In this paper, we discuss the progress the field has made so far—through advances like structured outputs, process supervision, and test-time compute—and outline several future directions for research to enable the development of modular and reliable LLM-based systems through improved specifications.

1 Introduction

Software has been one of the main drivers of economic growth over the past several decades, a trend famously captured by Andreessen in his influential blog post, “Why software is eating the world,” more than a decade ago [1]. More recently, AI—particularly Large Language Models (LLMs)—has emerged as the next revolution, poised to disrupt the very software that has been "eating the world" [2, 3, 4]. However, to fully realize this vision, we need to build LLM-based systems with the same level of reliability and rigor as those found in established engineering disciplines, such as control theory, mechanical engineering, and software engineering.

One key tool that has historically enabled the rapid growth of engineering disciplines is *specification*. Specifications describe the expected behavior, inputs, and outputs of a system. These specifications have different levels of precision and come in many forms, including formal specifications, product requirements documents (PRDs), and user manuals. Specifications help developers in several ways: (1) decomposing complex systems into smaller components, (2) reusing existing components when possible, (3) verifying that a system works as intended (i.e., it meets its specification), (4) fixing the system when it does not, and (5) creating systems capable of making decisions without human intervention. Figure 1 illustrates how specifications enable these five properties in the automotive industry.

In this paper, we distinguish between two types of specifications: (i) a description of what the task should do, which we call a **statement specification**, and (ii) a description that enables a user to verify task solutions (outputs), which we call a **solution specification**. Essentially, statement specifications answer the question, "What should the task accomplish?" while solution specifications answer the question, "How can one verify that the task’s solution adheres to the statement specification?" In traditional software systems, the statement specification can be captured in a Product Requirements Document (PRD), whereas the solution specification can be expressed as input-output tests (e.g., unit

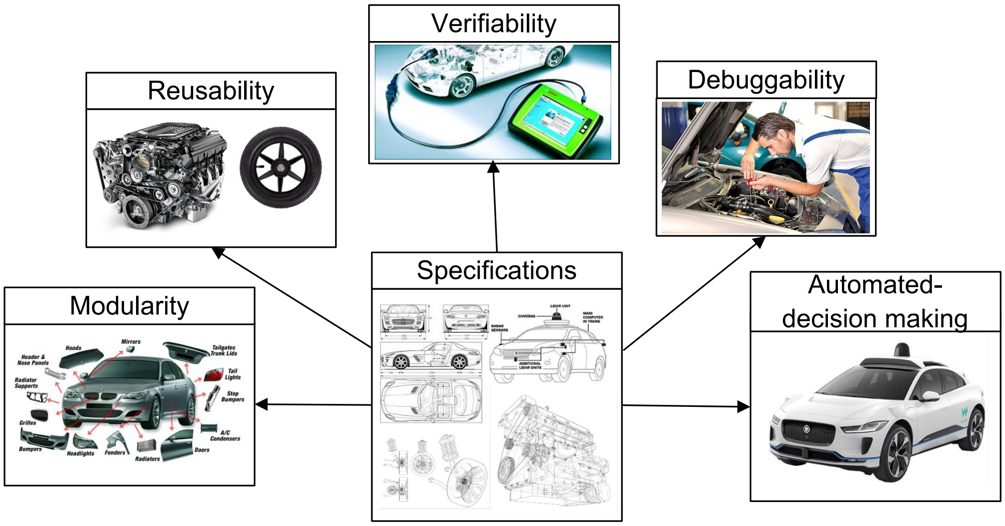


Figure 1: A car consists of many parts (modularity), where some parts are shared with other cars (reusability). When something goes wrong, the mechanic diagnoses the problem (verifiability) and then fixes it (debuggability). Furthermore, some functions of the car don't require human intervention (automated decision making). To enable these properties we need detailed schematics and detailed descriptions of how each of these components works (specifications).

tests). In formal frameworks like Coq/Gallina [5, 6], the statement specification is represented by a formal specification, and the solution specification is captured by a proof demonstrating that the generated code is correct. When building a radio, the statement specification is represented by the circuit schematics, while the solution specification is defined by the expected voltage and amperage at different points in the circuit. Finally, in some cases, the statement and solution specification coincide, e.g., the task, "solve $x^3 - x = 0$", specifies both what the task should do (i.e., solve the equation) and how a solution should be verified—by substituting the solution back into the equation to check the equality.

Defining good specifications is challenging. Apart from formal specifications, most specifications contain some degree of ambiguity. For example, it is notoriously difficult to describe and test for every possible behavior of a software system [7]. That said, LLMs are even more ambiguous than software systems, and this has significant implications. With LLMs, users provide specifications through natural language prompts. While natural language is highly general and flexible, it often encourages users to specify ambiguous, ill-defined tasks.[1] While encouraging ambiguity has been one of the main strengths of LLMs, this has also led to spectacular failures, such as the well-known "hallucinations", where LLMs generate incorrect answers to user input prompts [8, 9, 10].

As illustrated by "hallucinations," the lack of clear specifications makes it challenging to build *reliable systems*. If a task is not well-specified, how can one even determine whether the task is being performed correctly? Moreover, this ambiguity complicates the development of *modular systems* composed of multiple LLM components (e.g., agents). Without a clear understanding of what each component should do, how can one define the behavior of a system that integrates these components? As we will argue in Section 2, the challenge of building reliable, modular systems is significantly hindering the growth of the LLM ecosystem. Finally, in the absence of well-defined solution specifications, LLM-based applications (e.g., Q&A, customer support, copilot tools) often depend on humans to evaluate the quality of their outputs, as essential context is frequently missing from the task specifications. This dependence makes such applications unsuitable for fully automated decision-making without human intervention.

Table 1 provides examples of failures in today's LLM-based systems and suggests potential fixes. Unsurprisingly, these fixes revolve around improving task specifications, underscoring their importance in building modular and reliable LLM-based systems. Indeed, we are already seeing the drive for better task specifications for LLMs, and the evidence that better specifications help.

- **Prompt engineering:** This involves crafting and refining prompts to achieve the desired outputs. Techniques include creating detailed prompts[15, 16] (i.e., better specifications) to generate better results. Recently, DSPy [17] has emerged as an automated solution for prompt engineering that uses examples of desired prompts and outputs (i.e., solution specifications) along with additional hints.

[1] This should come as no surprise, as natural language was developed by humans to communicate with one another. Humans excel at handling ambiguity by relying on rich context to complement language, including facial expressions, tone of voice, and shared historical and cultural backgrounds. Additionally, humans often ask clarifying questions to resolve ambiguity.

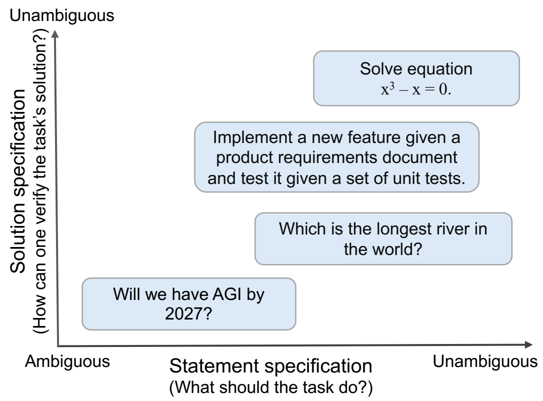
Failures	Description & Impact	Possible fixes
Wrong policy	An Air Canada customer service chatbot provided wrong policy information about a ticket being eligible for refund. Eventually, Air Canada was ordered to honor the erroneous refund policy [11].	Specify policies to verify the answers. Ask customers for clarifications to disambiguate their questions, if needed.
Wrong offer	A Chevrolet customer service chatbot agreed to sell to a customer a late model for $1 [12].	Specify the rules for a valid offer (e.g., price $\geq$ threshold), and monitor generated offers.
Wrong health advice	LLMs provided incorrect and harmful health advice [10].	Specify the rules for "safe" health advice, and use it to validate the advice.
Bad code	An LLM generated code that appears plausible but is functionally incorrect or references non-existent libraries and functions [9].	Provide or generate comprehensive tests to validate the generated code.
Data leakage	LLMs reproduced proprietary code or personal data present in their training datasets [8].	Check answers for possible proprietary info (e.g., SSNs, phone #, closed-source software licenses) and refuse to answer if info present.
Security vulnerability	LLM-generated code inadvertently introduced security flaws, such as SQL injection vulnerabilities and improper handling of user inputs.	Provide tests to expose vulnerabilities, e.g., fuzz testing for improper input handling [13], SQL injection attacks [14].
Legal misinformation	A lawyer submitted a ChatGPT-generate brief containing fictitious case citations leading to professional embarrassment and legal repercussions [8]	Check every citation in the brief against law case databases.

Table 1: Examples of LLM failures, their impact, and possible fixes. These fixes include better specifications: clear descriptions of what tasks should do, and clear criteria for evaluating tasks' solutions (e.g., answers). These specifications can lead to better, less ambiguous prompts and the implementation of guardrails for LLM outputs. Many LLM-based services, such as customer service chatbots, consist of multiple components, including knowledge retrieval, language understanding, and answer generation components.

- **Constitutional AI and guardrails:** Constitutional AI [18] establishes rules to avoid harmful outputs, enhancing task statement specifications to guide LLM answers. Guardrails [19] consist of code snippets that filter prompts and outputs, either avoiding certain topics or enforcing structured formats. These can address both statement and solution specifications.
- **Structured Outputs:** A growing number of LLMs support JSON-formatted inputs and outputs or other standardized formats [20, 21, 22]. For example, OpenAI recently introduced "structured outputs" [23], ensuring outputs adhere to schemas like JSON, as well as schemas described in Pydantic [24] and Zod [25], respectively. Additionally, Trace [26] allows users to specify a program using pseudocode and iteratively refine it. These constructs help define statement specifications (e.g., the desired output format) and, to a lesser degree, solution specifications (e.g., they typically describe the syntax of the outputs rather than their semantics).
- **Reward models:** Some of the most impressive results, such as those demonstrated by the recently released OpenAI's o1 [27], have been in the domains of problem-solving and reasoning—areas where it is easier to define clear task specifications. Specifically, to enhance reasoning capabilities, o1 employs reinforcement learning, which relies on an accurate reward function or model. A reward model can be thought of as a weak form of solution specification that enables the selection of the most likely correct solution among many candidates. To improve both training efficiency and scalability during testing, the reward function must be sufficiently precise to allow the system to evaluate solutions automatically.

This paper advocates for building on these early efforts to develop a coherent framework for improving task specifications. New methodologies are essential for task specifications in this new era of AI. These include clear and measurable performance metrics, as well as approaches to ensure that desired outcomes are reliably achieved. As illustrated in Figure 2, LLM tasks can have specifications with widely varying degrees of ambiguity. Our goal is to move task specifications toward the top-right corner of Figure 2(a). The less ambiguous the statement specification, the more likely the LLM is to generate the correct output. Similarly, the less ambiguous the solution specification, the easier it becomes to verify the LLM's output.

Reducing ambiguity will enable the application of engineering principles to LLMs, fostering the development of new solutions for an expanding range of use cases. We believe these properties will help unlock the potential of compound AI systems [28] and agentic systems [29], which are seen by some as the next revolution in the software industry. For example, verifiability—the ability to trust LLM outputs (which requires clear solution specifications)—and the ability


Unambiguous
Solution specification
(How can one verify the task's solution?)
Solve equation $x^3 - x = 0$.
Implement a new feature given a product requirements document and test it given a set of unit tests.
Which is the longest river in the world?
Will we have AGI by 2027?
Ambiguous
Statement specification
(What should the task do?)
Unambiguous

(a)

Task specification	Statement specification	Solution specification
“Will we have AGI by 2027?”	Ambiguous, as AGI is not well defined.	Ambiguous as solution specification is missing, and no clear way to evaluate the answer.
“Which is the longest river in the world?”	Unambiguous	Ambiguous, as the prompt doesn’t provide info on how to evaluate the answer
“Implement a new feature given a product requirements document and test it given a set of unit tests.”	Ambiguity depends on the PRD ambiguity.	Ambiguity depends on the level of coverage of unit tests.
“Solve equation $x^3 - x = 0$.“	Unambiguous: clear what solving an equation means.	Unambiguous: substitute solution and check question.

(b)

Figure 2: (a) A task specification describes (i) what the task should do (i.e., statement specification) and (ii) how should the task’s solution be verified (i.e., solution specification). (b) Some tasks, such as "Will AGI happen by 2027?" are ambiguous both in terms of statement and solution specifications (e.g., AGI is not well defined, and there is no clear way to evaluate how good an answer (solution) is.), while some tasks such as "Solve $x^3 - x = 0$" are unambiguous both in terms of statement and solution specifications (e.g., one can easily verify the correctness of the solution by substituting it in the equation).

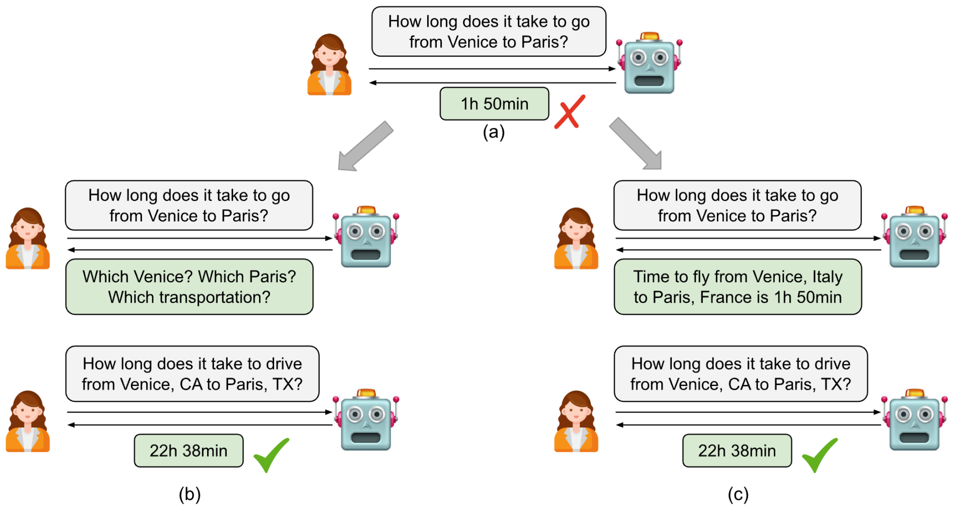

How long does it take to go from Venice to Paris?
1h 50min
(a)
How long does it take to go from Venice to Paris?
Which Venice? Which Paris? Which transportation?
How long does it take to drive from Venice, CA to Paris, TX?
22h 38min
(b)
How long does it take to go from Venice to Paris?
Time to fly from Venice, Italy to Paris, France is 1h 50min
How long does it take to drive from Venice, CA to Paris, TX?
22h 38min
(c)

Figure 3: A simple example of prompt (statement) disambiguation. (a) The user wants to learn how long it takes to drive from Venice, CA to Paris, TX, but doesn’t mention all details in her question. As a result, the agent gives the wrong answer, i.e., the flight time from Venice, Italy to Paris, France. Two simple methods to disambiguate: (b) the agent asks for clarification about which cities and the method of transportation; (c) the agent states all the assumptions in its answer, so the user sees how to improve the prompt.

to make decisions without human intervention are critical challenges for today’s agentic systems [30]. Furthermore, reducing ambiguity is valuable even for inherently ambiguous tasks (e.g., writing an email). Figure 3 illustrates an ambiguous prompt and two methods for disambiguating its statement specification (see Section 7.2).

The rest of the paper is organized as follows: Section 2 discusses the main challenges confronting today’s LLMs; Section 3 examines task specifications in the context of software engineering; Section 4 presents the five properties enabled by these specifications, which have been instrumental in driving the growth of many engineering disciplines; Section 5 contrasts software development with LLM development; Section 6 addresses the challenges of disambiguating LLM task specifications (e.g., prompts); Section 7 approaches for disambiguate task specifications; Section 8 demonstrates how better specifications can enable the application of the five engineering properties to building LLM systems; finally, Section 9 summarizes our findings.

2 Challenges with today's LLMs: building modular and reliable systems

Over the past two decades, and especially in the past two years since the introduction of ChatGPT, generative AI has made enormous strides. While the rate of progress has been astounding, most of this progress has come from building larger and better models—models that often take many months, even years, to develop and require staggering computational and financial resources. Today, the cost of developing state-of-the-art models is on the order of hundreds of millions of dollars and is soon expected to climb to billions.

This state of affairs raises two challenges. First, the prohibitive costs constrain the development of these models to only a few companies. Second, the monolithic nature of these models makes it difficult to address issues when they produce incorrect outputs. Even if we have access to all the model's parameters, it is far from trivial[2] to identify the root of the problem. This makes it challenging to fix hallucinations, arguably the biggest drawback of today's LLMs. These two challenges can significantly slow the rate of progress and ultimately hamper the growth of the AI industry.

In contrast, previous engineering-driven revolutions progressed not only by improving monolithic systems but even more so by building modular systems. Take a car, for example: it is built from a myriad of components, many of which are shared across different models (e.g., wheels, brakes, engines), and these components are produced by different teams within the same company or across companies (see Figure 1). Similarly, an operating system is built from numerous components (e.g., scheduler, file system, memory manager), each of which is composed of smaller components (e.g., device drivers), developed by hundreds of individuals.

Decomposing complex systems into simpler ones and reusing existing components has significantly lowered the barriers to building complex systems. This approach makes it easier to fix and improve these systems by isolating faulty components and repairing or replacing them. Furthermore, modularity enables a broader set of participants to contribute to value creation, as it lowers the barrier to building new, complex systems from existing ones rather than creating them from scratch.

Perhaps not surprisingly, given these historical patterns and advantages, we have recently seen efforts to build more *modular* AI systems [28, 29, 34]. However, despite these efforts, breakthroughs in AI have largely continued to come from a handful of frontier labs developing larger, more powerful models.

In an effort to turn this tide, in this paper, we draw from the rich body of work on building modular systems and apply these lessons to transitioning from monolithic to modular LLM systems. Such transformations have happened before. As engineering disciplines mature, they typically evolve from building large, monolithic artifacts to building modular ones, from using ad hoc processes to employing precise rules and standardized components.

One example is the automotive industry, which evolved from building bespoke cars—where much of the work was done by hand, and parts of the same car model were not interchangeable (e.g., a door of a Ford Model K [35] wouldn't fit another Model K car without adjustments)—to building millions of cars from thousands of standard, interchangeable components. Another example is the computer industry, which transitioned from creating very expensive, one-of-a-kind computers (e.g., ENIAC [36] to building affordable PCs using standardized components. These evolutions dramatically lowered the cost and barrier to building new artifacts.

In addition to building modular LLM-based systems, another significant challenge is building *reliable* systems. As illustrated in Table 1, LLM-based products and services are prone to high-profile failures across a wide variety of scenarios. To a large extent, this is due to their intrinsic flexibility: LLMs are often asked to perform ambiguous, ill-defined tasks. While embracing ambiguity has been a key to their success, deploying LLM-based products in practical scenarios often requires them to operate reliably, ideally without human supervision.

Once again, we can draw inspiration from engineering disciplines, which have developed processes and techniques to ensure reliability over time. For instance, the construction industry has implemented a variety of techniques, tests, and standards to build structures capable of withstanding earthquakes and hurricanes. Similarly, the software industry has adopted development methodologies (e.g., waterfall, agile development) and rigorous testing procedures (e.g., formal verification [37, 5, 38], fuzz testing [13]) to create systems that can operate error-free for extended periods and control critical systems such as airplanes, cars, and nuclear power plants.

Building modular systems is closely related to building reliable systems. Reliable systems are often built as modular systems by combining smaller, reliable components. For example, a reliable car consists of a reliable engine, reliable brakes, and so on. Similarly, a reliable backend service consists of a reliable web server, a reliable database, a reliable operating system, and so forth.

In general, there are five properties which allow us to build such systems:

[2] While there are efforts to improve the debuggability and interpretability of LLMs [31, 32, 33]

- **Verifiability**: ability to verify wether a system works correctly.
- **Debuggability**: The ability to debug a system if it doesn't work as expected. *Verifiability and debuggability are key to building reliable systems.*
- **Modularity**: The ability to decompose a system into multiple components and combine them together.
- **Resuability**: The ability to use existing components to build larger systems. Reusability complements modularity by leveraging existing components. *Modularity and reliability are key to building modular systems.*
- **Automated decision making**: The ability to build systems that can make decisions without human supervision, which typically implies reliability.

But is there anything that ties these properties together? Our answer is "yes." Our central thesis is that the key building block enabling these five properties is *specification*: the ability to describe what each component should do and how to verify its outputs. In the next section, we will illustrate how specifications are critical to enabling these properties by discussing them in the context of software engineering.

3 Specification, the foundation of software engineering

The goal of a software system is to perform a specified task. The specification provides a detailed description of what the task should achieve, including the required inputs and expected outputs. When clear from context, we will refer to "the specification of the task implemented by a system" simply as the "system's specification."

Ideally, specifications should be unambiguous, leaving no room for interpretation about the system's correct behavior or output for any given input. As shown in Figure 2, a task specification should describe both (i) what the task should do (statement specification) and (ii) how should one verify the task's solution/outputs (i.e., *solution specification*). A task specification is said to be unambiguous if both its statement and solution specifications are unambiguous; otherwise it is ambiguous. Note that these two specifications are complementary. The statement specification alone cannot guarantee the desired output, as there could be bugs in the task's implementation. Thus, we need a solution specification to be able to check wether the solution produced by the task's implementation satisfies the specification.

Figure 2 shows several simple examples of tasks with various degrees of ambiguity. Task "Solve equation $x^3 - x = 0$ has both unambiguous statement and solution specifications. In contrast, task "What is the longest river in the world?" has an unambiguous statement specification, but its solution specification is ambiguous, as it doesn't provide the user with enough information to verify an answer, and it is unlikely the user has this information either. In particular, it is unlikely for a user that asks this question to know that "Mississippi", for instance, is a wrong answer.

Specifications do not exist in isolation; they typically assume a user (or another system) that can interpret them. For example, a user using a formal specification is expected to understand the formalism; a developer using copilot for Rust is expected to be familiar with Rust and with the code base; a user submitting a task to solve a math equation is expected to be able to check wether the solution is correct or not. Throughout this paper, unless otherwise noted, we assume an "average" user, not an expert in the intended user group. For example, in a programming task, we assume an average programmer rather than an expert; for a user asking a factual question, we assume they may not know the answer and might be misled by incorrect answers, like "Mississippi" in our "What is the longest river in the world?" example.

When it comes to software systems, there are several frameworks for writing unambiguous specifications, including Coq/Gallina [5, 6], TLA+ [37], Alloy [38] and Lean [39]. For example, one could use Gallina to write unambiguous statement specifications, and Coq to generate unambiguous solution specifications as proofs that the generated programs indeed satisfy their statement specifications. Unfortunately, writing such specifications is far from trivial, so these frameworks end up covering a narrow set of use cases, e.g., use cases in which the cost of failure is very high. One example is chip design where flaws can lead to delays of months and huge financial losses (see here). Another example is a reactor control system in a nuclear plant, where a failure can lead to loss of lives.

However, most software systems do not incur such prohibitive costs on failures. These include the majority of software systems we are using daily such as text editors, chat applications, and even databases or operating systems. In those cases, a failure (e.g., crash) has relatively low cost, and it is relatively easy to recover (e.g., just restart the system). As a result, many such systems can get away with weaker forms of specifications, e.g., solution specifications that define the outputs for most common, but not all, inputs. A common example is a solution specification that leverages unit tests which don't cover all code paths. Another example is a specification that uses preconditions to filter incorrect outputs and postconditions to filter most, but not all, incorrect outputs [40] (e.g., a postcondition can specify that $a > b$, but not specify precisely the relation between a and b).

It is important to note that an unambiguous specification does *not* necessary imply a deterministic behavior, e.g., the outputs of a system are deterministic. Indeed, an unambiguous specification can describe a stochastic system, such as

Markov chains [41] or queueing systems [42]. For example, consider an M/M/1 queue [42] with mean arrival rate λ and mean service rate μ. This specification is unambiguous as it leaves no room for interpretation of how the system should behave, what are the expected inputs and outputs (e.g., system utilization $\rho = \lambda/\mu$ and mean serving time $1/(\mu - \lambda)$). Of course, to verify the outputs of such systems one would need to use statistical techniques, such as confidence intervals, to determine the probability these outputs meet the systems' specifications.

Finally, note that some stochastic systems lead to pseudo-deterministic outputs given an enough number of executions (trials), as the law of large numbers kicks in. Stochastic systems can even produce deterministic outputs. One example is using randomized search to find a solution to a problem, such as generating a program passing given tests [43].

4 The five properties of software engineering?

In this section, we introduce five desirable properties enabled by statement and solution specifications–verifiability, debuggability, modularity, reusability, and automated decision making–in the context of software systems. To illustrate these properties, we consider a running example using a database query engine as shown in Figure 4. A query engine consists of several components, (1) a parser which takes a query and produces an intermediate representation of the query such as an abstract syntactic tree (AST), (2) a query optimizer that takes the AST as input and applies various optimization techniques to generate an efficient query execution plan, and (3) a query execution component that executes the query plan to produce the final result. In addition, the query engine interacts with a data catalog that maintains metadata information (e.g., table schemas, location) needed by the query optimizer to produce the execution plan, and the database storing the actual data.

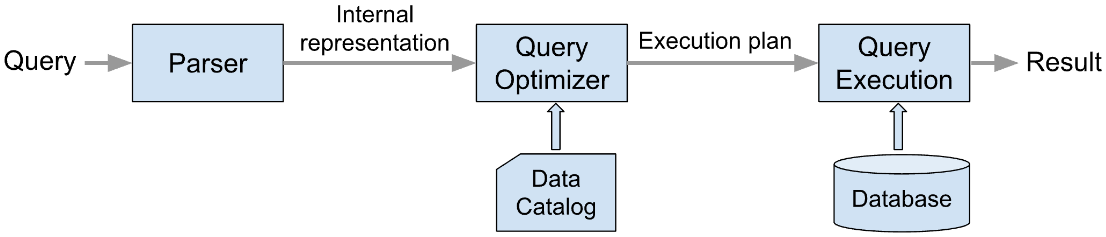


Figure 4: The main system components of a SQL query engine.

As illustrated in Figure 4, one of the properties of the query engine is **modularity**, i.e., the query engine consists of multiple software components. This modularity enables us to build these components in parallel and then assemble them together. A second related property is **reusability** which is the ability to use existing components when available. For example, instead of writing a query parser from scratch, we can use an existing library such as BlazingSQL [44] or JSQLParser [45], to speed up building the query engine.

Modularity and reusability are enabled by the fact that each component has a clear **specification** which, again, consists of the statement specification (e.g., PRD) and the solution specification (e.g., unit and integration tests). These specifications help ensure that the output of a component matches the expected input of the next component, and help us reason about the end-to-end behavior of the system. For instance, we expect the query optimizer to affect only the query performance, not the outputs produced by the query which should remain the same.

While modularity and reusability can help building new systems from existing components, in many cases we need to build components from scratch. Implementing a new component requires two other properties: **verifiability** and **debuggability**. Verifiability is enabled by the solution specification which describes the correct outputs of the task implemented by the component. In most cases the solution specification consists of a set of input-output tests (e.g., unit tests, integration tests, statistical tests, etc). Alternatively, it can come in the form of preconditions and postconditions which filter out invalid inputs and outputs [40, 46]. In our query optimizer example, a verifier would check that none of the optimizations (e.g., push down predicates, join reordering) change the query result.

Debuggability is the process of fixing the implementation if the verification fails. This involves locating the bug and fixing it. Debuggability is enabled by the hierarchical modularity of software systems. In our example, we can start with the query engine, and identify which component is buggy. Furthermore, within that component, we identify the module, function, and finally the sequence of instructions causing the bug. Then, we fix the bug. This top-down process of locating the bug is enabled by the fact that each component has a clear solution specification.

A final property of software systems is automated **decision-making**. This enables a system to make decisions without human supervision. This property is key to addressing use cases where humans are too slow or error-prone. One simple example is the query optimizer in our SQL engine, which computes an efficient query plan automatically. Other examples include mission-critical systems such as anti-lock brake systems for cars, automatic takeoff and landing systems for airplanes, and reactor control systems in nuclear power plants.

Unlike automated decision-making systems, the majority of existing software systems (e.g., chat apps, IDEs, games, etc.) assume a human in the loop. As such, these systems typically rely on humans to detect failures and provide remediations (e.g., restarting a program when it crashes or hangs). In contrast, automated decision-making systems do not rely on humans, which means they need to avoid failures in the first place. As a result, these systems tend to have less ambiguous (ideally, unambiguous) statement and solution specifications. Indeed, automotive and aerospace industries often use formal tools for specification and verification, such as SPARK [47] and the SCADE Suite [48].

5 Comparing LLM development and software development

The recent progress of LLMs has sparked the imagination of many with Andrej Karpathy famously stating that English is the hottest new programming language" [49], which suggests that one can "program" LLMs by simply describing tasks in natural language.

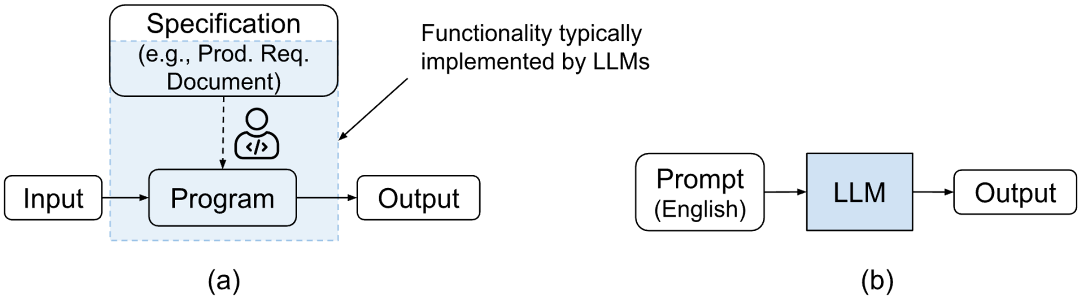

Specification
(e.g., Prod. Req. Document)
Functionality typically implemented by LLMs
Input
Program
Output
(a)
Prompt
(English)
LLM
Output
(b)

Figure 5: High-level architecture of (a) developing a program for a given task vs (b) using an LLM to accomplish the same task.The LLM obviates the need for writing the program and doesn't need a detailed specification. The LLM prompt contains both the description (specification) of the task and its input.

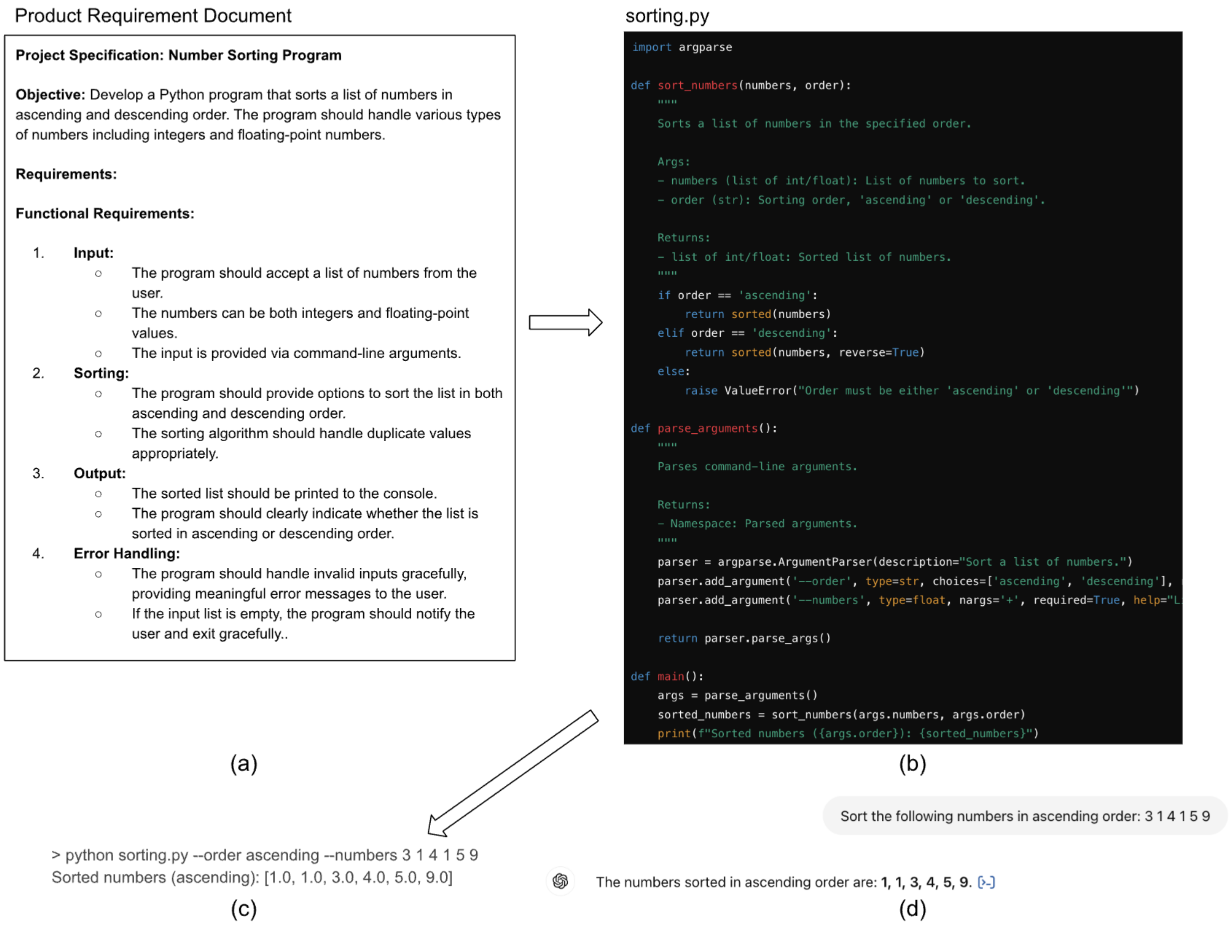
Figure 5 contrasts (a) developing a program for a given task using traditional software engineering with (b) accomplishing the same task using LLMs. With software engineering, a developer takes a task's statement specification and writes a program to implement that task. The program then takes an input and generates an output which can be verified using the solution specification. In contrast, an LLM takes a description (e.g., a high-level statement specification) of the task together with the task's input as a prompt and produces an output. Thus, LLMs have the potential to obviate the need for writing programs and for detailed specifications. When it comes to specifications, LLMs typically trade unambiguity for the ease of writing these specifications in natural language,

Figure 6 shows a simple task to sort a list of integers, implemented by using both a program and an LLM. Figure 6(a) shows the statement specification in the form of a PRD, Figure 6(b) shows a Python program implementing the PRD, and Figure 6(c) shows a call to the program with input list: 3, 1, 4, 1, 5, 9. In contrast, Figure 6(d) shows an equivalent LLM solution consisting of a prompt that contains both a high-level statement specification of the task (i.e., "Sort the following numbers in ascending order") and the same input. Note that none of these cases shows a solution specification.

While the example in Figure 6 works well and shows the full power of LLMs, unfortunately, in many cases the prompts are ambiguous and users don't get the desired answers. We discuss this challenge and approaches to address it next.

6 Task specifications in LLMs

When it comes to LLMs, we are faced with a conflict between (1) the accessibility and generality of the natural language (i.e., which allows anyone to specify any task), on one hand, and (2) its inherent ambiguity, on the other hand. Next, we examine approaches on how to resolve this conflict by making it easier to write clear (unambiguous) specifications for a growing number of LLM tasks. This will allow us to leverage engineering principles to implement these tasks and provide solutions for an increasing range of applications.



Product Requirement Document

Project Specification: Number Sorting Program

Objective: Develop a Python program that sorts a list of numbers in ascending and descending order. The program should handle various types of numbers including integers and floating-point numbers.

Requirements:

Functional Requirements:

1. **Input:**
   - The program should accept a list of numbers from the user.
   - The numbers can be both integers and floating-point values.
   - The input is provided via command-line arguments.
2. **Sorting:**
   - The program should provide options to sort the list in both ascending and descending order.
   - The sorting algorithm should handle duplicate values appropriately.
3. **Output:**
   - The sorted list should be printed to the console.
   - The program should clearly indicate whether the list is sorted in ascending or descending order.
4. **Error Handling:**
   - The program should handle invalid inputs gracefully, providing meaningful error messages to the user.
   - If the input list is empty, the program should notify the user and exit gracefully..

(a)

sorting.py

```python
import argparse

def sort_numbers(numbers, order):
    """
    Sorts a list of numbers in the specified order.

    Args:
    - numbers (list of int/float): List of numbers to sort.
    - order (str): Sorting order, 'ascending' or 'descending'.

    Returns:
    - list of int/float: Sorted list of numbers.
    """
    if order == 'ascending':
        return sorted(numbers)
    elif order == 'descending':
        return sorted(numbers, reverse=True)
    else:
        raise ValueError("Order must be either 'ascending' or 'descending'")

def parse_arguments():
    """
    Parses command-line arguments.

    Returns:
    - Namespace: Parsed arguments.
    """
    parser = argparse.ArgumentParser(description="Sort a list of numbers.")
    parser.add_argument('--order', type=str, choices=['ascending', 'descending'],
    parser.add_argument('--numbers', type=float, nargs='+', required=True, help="L

    return parser.parse_args()

def main():
    args = parse_arguments()
    sorted_numbers = sort_numbers(args.numbers, args.order)
    print(f"Sorted numbers ({args.order}): {sorted_numbers}")
```

(b)

Sort the following numbers in ascending order: 3 1 4 1 5 9

```
> python sorting.py --order ascending --numbers 3 1 4 1 5 9
Sorted numbers (ascending): [1.0, 1.0, 3.0, 4.0, 5.0, 9.0]
```

(c)

The numbers sorted in ascending order are: **1, 1, 3, 4, 5, 9**.

(d)

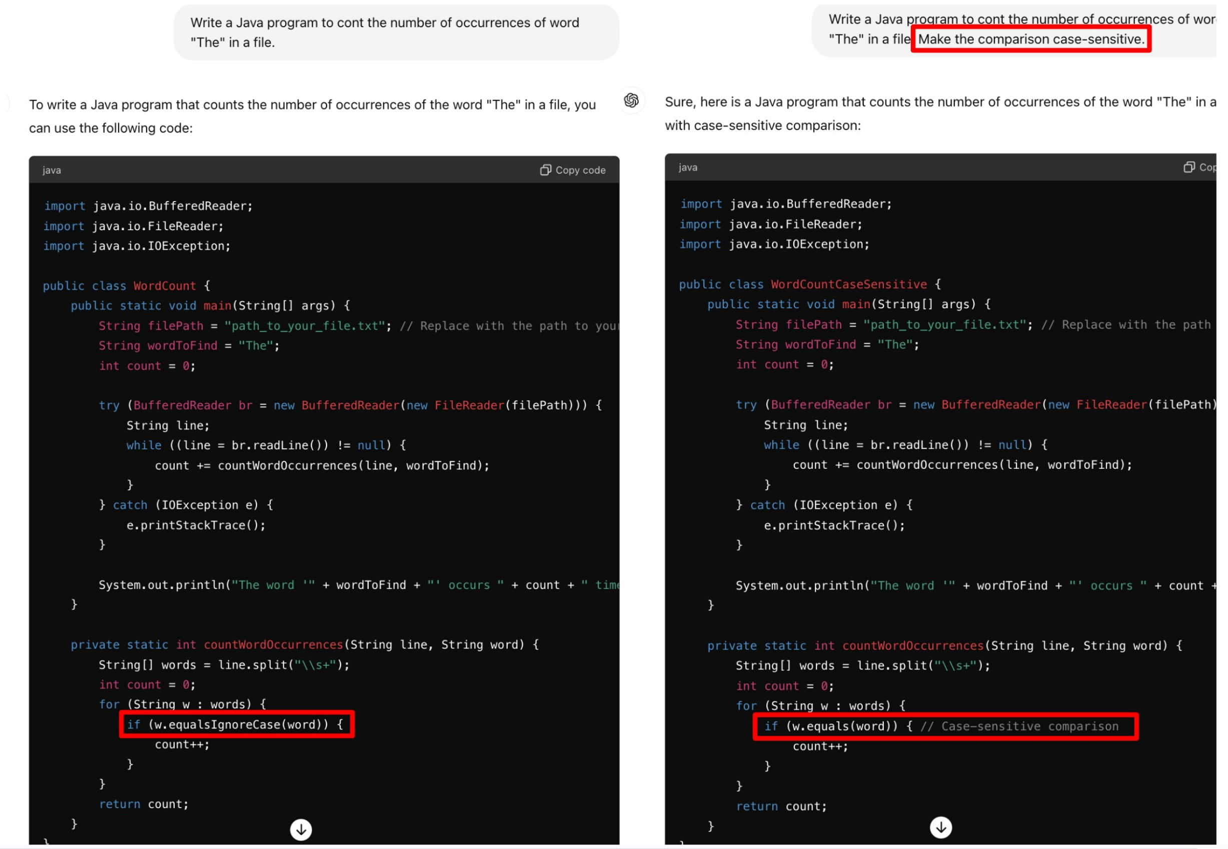
Figure 6: The task of ordering an array of numbers. (a) Specification; (b) Program implementation; (c) Executing the program on an example input. (d) A prompt asking ChatGPT to order the same example input.

6.1 Are LLM prompts inherently ambiguous?

With LLMs, we specify tasks using prompts. We say that a prompt is inherently ambiguous if it is very hard, or infeasible, to disambiguate it. As before, we need to differentiate between statement and solution specifications. An example of a prompt that is ambiguous both in terms of statement and solution is "Write a poem about a white horse in Shakespeare's style". Such prompt is inherently ambiguous as there is no clear way to disambiguate it. Indeed, the user might not even know what a good poem looks like before seeing one!

On the other hand, consider the prompt "How long does it take to go from Venice to Paris?". This prompt has an ambiguous statement specification since it is not clear which "Venice" and which "Paris" the user is referring to, i.e., is it the "Venice" in Italy or the one near Los Angeles? Is it the "Paris" in France or the one in Texas? Does the user intend to travel by car, train, or airplane? This prompt's statement specification is not inherently ambiguous as the user can easily disambiguate it by adding the missing information, as shown in Figure 3(b). However, note that the solution specification of this prompt is still ambiguous as it is not clear how to check whether the answer is correct from the information provided by the prompt alone.

In another example (see Figure 7(a)), consider the prompt, "Write a Java program that counts the number of occurrences of the word "The" in a file. Similarly, this prompt has an ambiguous statement specification as it doesn't say whether the match should be case sensitive or not (though in this case one could argue that the user's intent was to have a case-sensitive match since she capitalized the first letter in "The"). Again, the prompt's statement is not inherently ambiguous as a user can easily disambiguate it by adding "Make the comparison case-sensitive", and, by doing so, she gets the desired result (see Figure 7(b)). At the same time, this prompt doesn't provide a solution specification either. However, one could add such specification by providing some unit tests, e.g., provide files with a known number of "The" words and extend the prompt to provide the expected number of "The" words for each test file.



Write a Java program to cont the number of occurrences of word "The" in a file.

To write a Java program that counts the number of occurrences of the word "The" in a file, you can use the following code:

```java
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class WordCount {
    public static void main(String[] args) {
        String filePath = "path_to_your_file.txt"; // Replace with the path to you
        String wordToFind = "The";
        int count = 0;

        try (BufferedReader br = new BufferedReader(new FileReader(filePath))) {
            String line;
            while ((line = br.readLine()) != null) {
                count += countWordOccurrences(line, wordToFind);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }

        System.out.println("The word '" + wordToFind + "' occurs " + count + " tim
    }

    private static int countWordOccurrences(String line, String word) {
        String[] words = line.split("\\s+");
        int count = 0;
        for (String w : words) {
            if (w.equalsIgnoreCase(word)) {
                count++;
            }
        }
        return count;
    }
```

Write a Java program to cont the number of occurrences of wor "The" in a file Make the comparison case-sensitive.

Sure, here is a Java program that counts the number of occurrences of the word "The" in a with case-sensitive comparison:

```java
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class WordCountCaseSensitive {
    public static void main(String[] args) {
        String filePath = "path_to_your_file.txt"; // Replace with the path
        String wordToFind = "The";
        int count = 0;

        try (BufferedReader br = new BufferedReader(new FileReader(filePath)
            String line;
            while ((line = br.readLine()) != null) {
                count += countWordOccurrences(line, wordToFind);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }

        System.out.println("The word '" + wordToFind + "' occurs " + count +
    }

    private static int countWordOccurrences(String line, String word) {
        String[] words = line.split("\\s+");
        int count = 0;
        for (String w : words) {
            if (w.equals(word)) { // Case-sensitive comparison
                count++;
            }
        }
        return count;
    }
```

Figure 7: Consider the prompt, “Write a Java program that counts the number of occurrences of word “The” in a file”. (a) The generated program uses case-insensitive matching despite the fact that the user capitalized the first letter of “The” which might indicate she intended case-sensitive matching, (b) Eliminating the ambiguity by adding “Make the comparison case-sensitive” leads to the desired result.

7 Disambiguating LLM specifications (prompts)

As mentioned above, while most of LLM prompts are ambiguous, in many cases this ambiguity is not fundamental; it is merely a reflection of the fact that writing unambiguous specifications is often hard. In this section, we suggest several approaches to make it easier to disambiguate LLM prompts. We do so by drawing our inspiration from how humans deal with ambiguity when they communicate with each other.

7.1 How do humans deal with ambiguity?

People use natural language every day to describe tasks for others to perform, such as "Buy one gallon of milk," "Bring one more chair to the table," or "Let’s meet at the market hall at 7" (see Table 2). These examples provide both a statement specification (i.e., what to do) and a solution specification (i.e., enough information to verify whether the task was performed correctly). For example, one can check that "the milk was indeed bought," "a new chair was brought to the table," and "the person attended the meeting."

However, since natural language is inherently ambiguous, the tasks it describes are often ambiguous as well, including the examples above. So, how do people handle this ambiguity? They typically rely on three main approaches.

First, most people’s interactions are *multi-turn*, e.g., dialogs, discussions, etc. This enables people to refine their task’s description if ambiguous. If Alice asks Bill to do something, and Bill is confused, he will ask for clarifications. This will give Alice the opportunity to clarify the task’s description, and Bill the opportunity to build more context about the task. This pattern is very common. Table 2 shows some examples of clarification questions: "What kind of milk? What brand? By when do you need it?", or "Which chair?", or "Which market hall? Which stall in the market hall? 7am or 7pm?"

Task	Multi-turn	Task-aware context	People-aware context
"Buy a gallon of milk."	"What kind of milk?" "Which brand?" "By when?"	I am buying milk every week, so I already know the kind of milk and brand. I know you need the milk to bake a cake so I know when you need the milk by.	I know you like organic, reduced-fat milk.
"Bring a chair to the table."	Which chair?	I already know which chair to bring, as this is not the first time you asked me to do it.	I know you like firm chairs.
"Let's meet at the market hall at 7"	Which market hall? Which stall? 7am or 7pm?	This is a regular meeting point and it's the usual time we are meeting.	I know you are a late morning person, so it's likely 7pm.

Table 2: Example of everyday tasks, and approaches people use to disambiguate them.

Second, people assume a *task-aware context*. Such context is built from performing similar tasks in the past or extra information related to the given task. Referring again to the tasks in Table 2, if this is not the first time you are buying milk, you probably already know which kind of milk or which brand you need to buy, so no need to ask. Also, if you know that the milk is needed for baking a cake for dinner, you might already know by when you need to buy the milk.

Third, people often "model" other people they communicate with. They do so by learning from the repeated interactions with these people, what they like and don't like, what is the most effective way to ask these people to perform a task, etc. We call this, *people-aware context*. In the example, "Let's meet at the market hall at 7", if you know the person asking you this is a late morning person, you can easily infer this must be 7pm not 7am.

Interestingly, people leverage similar approaches when interacting with a chatbot like ChatGPT as well: they often use multi-turn interactions to refine their asks, assume the prior context from the conversation is available when asking the next question, and learn what the LLMs are good at or they are not got at (e.g., what kind of questions are more likely to produce hallucinations).

In summary, humans typically communicate iteratively by assuming some shared context. As such, they often leave details out of the conversation, as these details can be easily filled in via subsequent questions or can be inferred from shared context. Natural language has evolved over thousands of years to let humans communicate with each other efficiently. However, this does not necessarily make it the most effective way to communicate between an LLM and a human, two LLMs, or an LLM and an existing software system (e.g., function calling). Thus, it should come as no surprise that we need new techniques to improve communication in those cases.

7.2 Solutions for prompt disambiguation

In this section, we discuss several techniques to disambiguate LLM prompts drawing inspiration from how humans disambiguate their communication. Most of the techniques here focus on tasks' *statement* specifications.

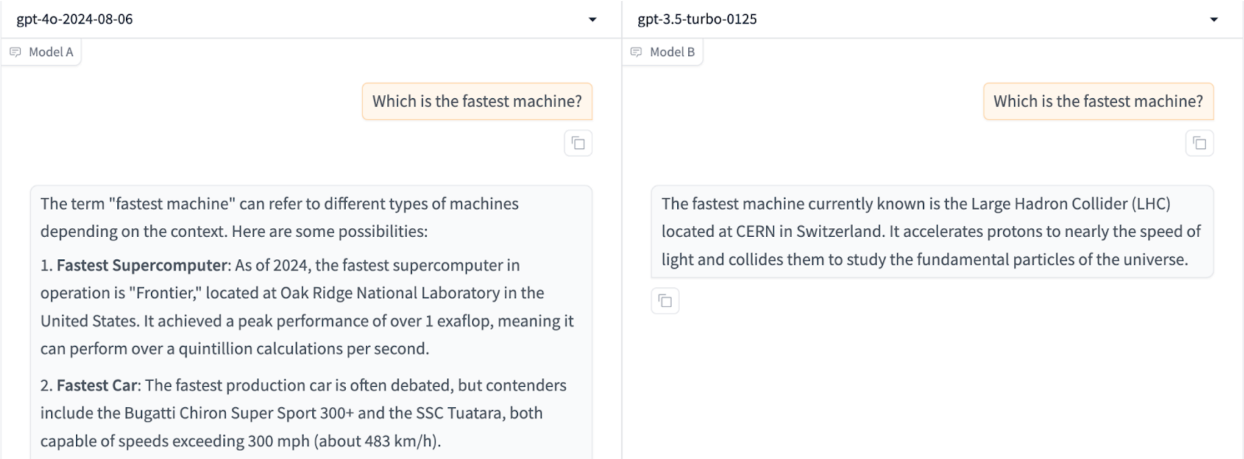
Iterative disambiguation: As mentioned, in real life, people often ask questions to clarify the description of a task provided by another person. One example is a product manager refining a product requirement document (PRD) based on the feedback from a developer: when a developer is confused about the PRD, she asks questions, which typically lead to the program manager adding clarifications or new information to the PRD with the goal of disambiguating it.

One natural approach would be to mimic this process by having an LLM ask a human questions to help her disambiguate her prompt (see Figure 3(b)). But how can an LLM figure out whether a prompt is ambiguous? One possibility is to leverage the observation that an ambiguous prompt is more likely to generate different outputs than an unambiguous one.[3] In particular, if an LLM generates multiple outputs for the same ambiguous prompt, the chances are some of these outputs will be different. Consider again our example in which we want to generate a program that counts the number of occurrences of the word "The" in a particular document. By generating many programs [4] we might end up with programs that provide different outputs depending on whether the comparison is case sensitive or not. By showing the differences between the programs to the user, we can help her realize that she needs to add the case sensitivity matching requirement to the prompt.

[3] While in many cases a prompt with different answers indicates ambiguity, this is not always the case. An unambiguous prompt can have different answers, all correct, e.g., "fastest driving route from A to B" can return two different routes that take the same time.

[4] To increase the chance of generating different programs for an ambiguous prompt, one could increase the temperature, use different LLMs, use different languages, rewrite the prompt, etc.

A more advanced approach would not only show the differences between programs to the user, but also suggest how to disambiguate the prompt, e.g., “Consider specifying whether the matching is case-sensitive or not?”. In another example, consider prompt “Which is the fastest machine?”. This prompt is also ambiguous. Is it the fastest computer? The fastest car? And, if it is the fastest car, is it the fastest road car? The fastest race car? Or maybe the fastest vehicle period, i.e., the kind of cars using jet engines that are purposely built for land speed records? Again, one way to disambiguate the original prompt is to use one or more LLMs to provide different outputs that expose the ambiguity (see Figure 8).



Which is the fastest machine?

The term "fastest machine" can refer to different types of machines depending on the context. Here are some possibilities:

1. **Fastest Supercomputer**: As of 2024, the fastest supercomputer in operation is "Frontier," located at Oak Ridge National Laboratory in the United States. It achieved a peak performance of over 1 exaflop, meaning it can perform over a quintillion calculations per second.

2. **Fastest Car**: The fastest production car is often debated, but contenders include the Bugatti Chiron Super Sport 300+ and the SSC Tuatara, both capable of speeds exceeding 300 mph (about 483 km/h).

Which is the fastest machine?

The fastest machine currently known is the Large Hadron Collider (LHC) located at CERN in Switzerland. It accelerates protons to nearly the speed of light and collides them to study the fundamental particles of the universe.

Figure 8: Two LLMs give two different answers to the prompt: “Which is the fastest machine?”. This suggests the prompt is ambiguous and the user should disambiguate it, e.g., “What is the fastest car?”.

Yet another approach would be for the LLM to give answers that clearly state the assumptions it made in response to ambiguous prompts. For example, given the question “How long does it take to go from Venice to Paris?”, an LLM might answer “The *flying* time between Venice, *Italy* and Paris, *France* is 1h and 50min”, (The words in italic capture the assumptions made by LLM in answering the question.) This will make it easy for the user to check whether this answers the question she had in mind and, if not, improve the prompt specification, e.g. “How long does it take to drive between Venice, CA to Paris, TX?” (see Figure 3(c)).

Note the last two approaches might require finetuning or post-training an LLM to either provide suggestions to disambiguate prompts or state the assumptions for its outputs to fill the gaps in the prompts’ specifications.

There is already an active area of related research in programming languages to identify the user’s intent [50, 51, 52]. This research can be a promising base to build on.

Domain-specific rules: People in the same profession or organization share context in the form of predefined rules or special semantics associated with words or phrases. For example, when a soldier hears from his superior that she should perform a task at 11:00, she knows this is 11am (not 11pm) since the military uses the 24h time format to tell the time. In another example, for software developers, words like “class”, “loop”, “variable” have very precise meanings; the same words are ambiguous in a colloquial conversation. Finally, a programming language adds another layer of syntax and semantics which further helps to disambiguate the meaning of statements written in that language.

In domains where there are strong rules or operating procedures we can use those rules and procedures to disambiguate the specifications. Examples are the civil or penal codes, standard-operating-procedure manuals in hospitals, labs, etc. Note that new rules are already being proposed specifically for LLMs, such as Constitutional AI [18]. Constitutional AI is a list of rules to align LLMs so that to minimize the harm they can cause. Some areas of research are (1) capturing these rules effectively in the prompt (e.g., select only the rules relevant to the task), (2) enforcing/verifying that these rules were followed by the output, i.e., solution specification. The latter is particularly challenging as most of these rules are not formal. Thus, formalizing these rules when possible is another area or research.

Finally, another important area of research is defining new domain-specific rules. One example is capturing the laws and constraints in the physical world, e.g., people do not walk upside down unless they are in space, they do not walk through walls, etc. This is related to the ongoing work on world models [53, 54].

Monitor, learn & build context: Humans often create models (context) about the external world, including people they are interacting with. For example, we communicate and explain things differently to children, to our friends, to colleagues, or to our family. In some of these cases, we do so by learning what is the most effective way to communicate with a particular person after repeatedly interacting with that person (see Section Section 7.1). One area of research is to identify the prompts a model is good at (e.g., it excels in generating python programs for data processing) and not good at (e.g., it does a mediocre job on world problems). Also, we can learn the style of the prompt that maximizes the accuracy of model's answer. Based on these learnings, we can change prompts or route prompts to the best model to handle it.

"The stranger test": If there is no shared context available when prompting an LLM, one approach is to provide as much as possible information in the prompt, e.g., just assume the prompt is intended for a stranger. To illustrate this approach, consider a text-to-image task. Even though such a task is inherently ambiguous, we can still disambiguate it substantially by thinking of the prompt as a one-shot specification for a stranger. Consider you want to generate an image of the "Golden Gate Bridge at sunset" (see Figure 9(c)). Then you should provide as much information as you can to increase the probability you get the picture you'd likely like from someone who knows nothing about your artistic preferences, e.g., the picture should use pastel colors, there should be realistic proportions between the bridge and buildings, the sun should set opposite to the city, etc. Basically, you will try to externalize the specification you have in mind about how the picture should look like.

Another way to externalize such a specification is by showing the user various pictures and asking her to say which picture she likes and which she does not. Based on this, we can infer some user's preferences (e.g., pastel colors vs bright colors, photograph-like pictures vs cartoon-like pictures, etc) and use these preferences to enhance the prompt. Note this is another way of building context.

Structured outputs: In an effort to improve reliability, many LLMs have started to provide support for describing their outputs and inputs in JSON format[20]. Furthermore, OpenAI has recently introduced "structured outputs" [23], a feature that not only ensures the output is in JSON format but that it adheres to a particular JSON schema. Furthermore, Trace [26] enables users to use pseudocode to specify a program and then use an iterative process to generate and refine the program. These constructs are one step closer to providing statement and solution specifications for programming tasks. In particular, a pseudocode provides a statement specification (i.e., it specifies what the task should do) while a schema provides a weak form of solution specification as it can be used to verify whether the output matches the given schema. However, the pseudocode can be still ambiguous, and an output schema captures mostly the syntax and not the output's semantics. As such, most of the disambiguation techniques discussed in this section apply to structured outputs as well. An interesting research direction is taking advantage of the additional structure and application domain (e.g., program synthesis) to develop more specific disambiguation techniques. Such techniques could borrow from and extend traditional formal specification research [55, 56]. Finally, while providing the pseudocde can help providing better solutions, writing such a pseudocode is not easy, especially for complex programs. One research direction here is to find the right balance between the level of detail of the pseudocode and the correctness of the generated program.

Instruction following: Even if a prompt is unambiguous, it is of little use if the LLM cannot provide a solution for it. As such, the ability of the LLM to follow the prompt's specification is critical to eventually find a solution. Thus, a future direction is to improve an LLM ability to follow a prompt's specification, an already active area of research. This might involve not only test-time compute, but also pre-training and post-training.

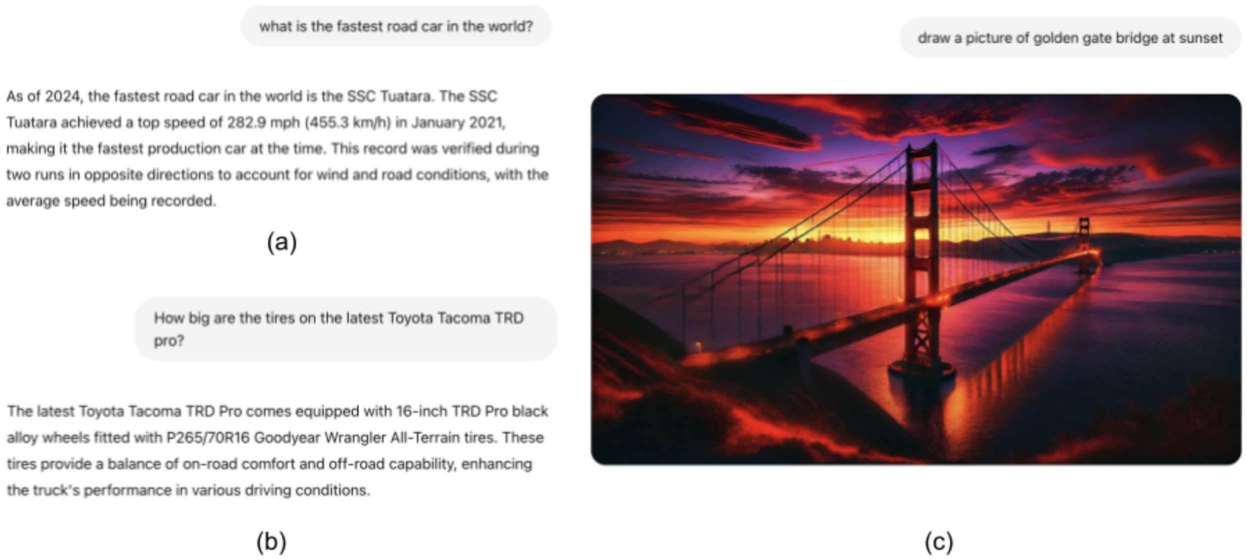
8 The five engineering properties in the LLM context

In this section, we show how we can leverage the five engineering properties in the context of LLMs, and in the process identify some key challenges and suggest several approaches to address them.

8.1 Verifiability: challenges and solutions

Verifiability refers to the ability to assess whether the task's implementation adheres to the statement specification (e.g., a program's PRD). This reduces to checking that the task's outputs satisfy the *solution* specification (e.g., unit tests). Consequently, the verification depends strongly on the solution specification. If the solution specification is ambiguous, verification may fail.

Figure 9 illustrates examples of prompts lacking clear solution specifications. In such cases, a user will likely fail to assess the correctness of the output: (a) the user may not know whether a faster car exists than the one mentioned in the response; (b) the user may be unaware of the size of the Toyota Tacoma's wheels; and (c) the user may not know where the sun sets in San Francisco. Note these are examples of "hallucinations", i.e., outputs that look correct but are not.



what is the fastest road car in the world?

As of 2024, the fastest road car in the world is the SSC Tuatara. The SSC Tuatara achieved a top speed of 282.9 mph (455.3 km/h) in January 2021, making it the fastest production car at the time. This record was verified during two runs in opposite directions to account for wind and road conditions, with the average speed being recorded.

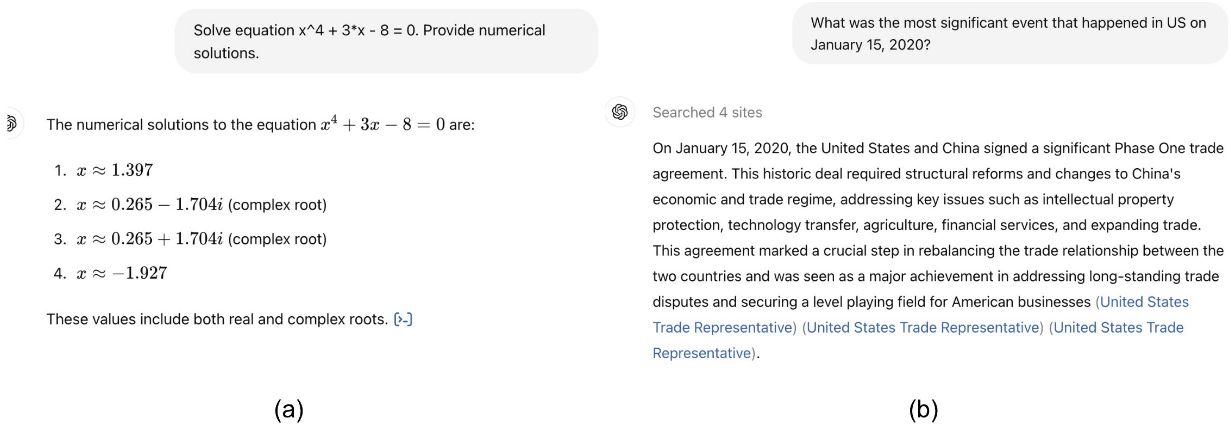
How big are the tires on the latest Toyota Tacoma TRD pro?

The latest Toyota Tacoma TRD Pro comes equipped with 16-inch TRD Pro black alloy wheels fitted with P265/70R16 Goodyear Wrangler All-Terrain tires. These tires provide a balance of on-road comfort and off-road capability, enhancing the truck's performance in various driving conditions.

draw a picture of golden gate bridge at sunset

Figure 9: Example of prompts without clear solution specifications: (a) SSC Tuatara is not the fastest road car in the word; Koenigsegg Jesko Absolut is (330 MPH); (b) Toyota Tacoma TRD pro doesn't use 16" wheels; it uses 18" wheels; (c) the picture doesn't show Golden Gate Bridge at sunset, as the sun sets on the opposite side of the city.

Even when a task has a clear solution specification, assessing whether its implementation works as intended is not always straightforward. This often requires additional context or tools. For instance, if we prompt an LLM to solve the equation "$x^4 + 3x - 8 = 0$" (note that here the equation serves as both the statement and the solution specification), a user may have to use a calculator to verify that the generated solution is correctnes (see Figure 10(a)). In another example, an LLM might generate references to reputable documents, which the user may have to cross-check to verify the accuracy of the answer (see Figure 10(b)).



Solve equation x^4 + 3*x - 8 = 0. Provide numerical solutions.

The numerical solutions to the equation $x^4 + 3x - 8 = 0$ are:

1. $x \approx 1.397$
2. $x \approx 0.265 - 1.704i$ (complex root)
3. $x \approx 0.265 + 1.704i$ (complex root)
4. $x \approx -1.927$

These values include both real and complex roots.

What was the most significant event that happened in US on January 15, 2020?

Searched 4 sites

On January 15, 2020, the United States and China signed a significant Phase One trade agreement. This historic deal required structural reforms and changes to China's economic and trade regime, addressing key issues such as intellectual property protection, technology transfer, agriculture, financial services, and expanding trade. This agreement marked a crucial step in rebalancing the trade relationship between the two countries and was seen as a major achievement in addressing long-standing trade disputes and securing a level playing field for American businesses (United States Trade Representative) (United States Trade Representative) (United States Trade Representative).

Figure 10: One can check the results provided by ChatGPT using (a) a calculator, and (b) taking a look at the documents linked in the results.

Next, we propose several directions for future work aimed at improving solution specifications, as opposed to the statement specifications discussed in Section 7.2. Additionally, we suggest exploring the use of external tools to verify a task's implementation effectively.

Proof-carrying-outputs: An unambiguous statement specification is usually insufficient to determine whether an implementation truly satisfies the specification. We also need a solution specification. Consider a code generation task with a formal (unambiguous) statement specification. How can we ensure that the generated code satisfies the specification for every possible input? One approach is to generate not only the code itself but also a proof that the code adheres to the specification, leveraging techniques such as *proof-carrying code* [57]. In this case, the proof serves as an unambiguous solution specification.

Step-by-step verification: A large body of work has demonstrated that asking the LLMs to provide step-by-step solutions (e.g., chain-of-thoughts [58], tree-of-thoughts [59], self-notes [60], scratchpad [61]) increases the accuracy of the solutions. Fortunately, this also makes the verification easier, as we can independently verify every step of the solution, and in many cases each of these steps is easier to verify than the end-to-end solution. Of course, each step being correct doesn't guarantee end-to-end correctness (e.g., there might be missing steps, or steps that are correct in isolation but inconsistent across them). However, early evidence shows that step-by-step verification can improve the verification accuracy, e.g., Process Supervision [62].

Execute-then-verify: Consider an output containing instructions. In many cases, it is easier to verify whether the output satisfies the task's solution specification by executing the instructions and observing the results. For example, it is often easier to verify the correctness of a program by running it and checking its outputs, than by analyzing the code alone. However, this approach poses a significant challenge: if the result is incorrect, it can lead to irreversible mistakes. For instance, imagine a program mistakenly deleting important files. To mitigate this risk, it is essential to ensure that execution is either reversible (i.e., can be undone) or does not alter the original state of the system. For reversibility, one approach is to checkpoint the system state before each operation. Another is to log each operation along with its corresponding undo operation. For non-altering execution, a viable strategy is to run the program in an isolated copy of the original environment, such as within a virtual machine. One project exploring these approaches is GoEx [63].

Preconditions & postconditions: One technique we can borrow from programming languages to disambiguate a task's solution specification is the use of preconditions for the task's input and postconditions for its output [40, 46]. For instance, when asking which is the fastest car, if the user knows that cars with a maximum speed exceeding 300 mph exist, they could specify this as a postcondition. Another promising direction is to leverage LLMs to generate postconditions directly from the prompt. For example, if the question is, "What is the tallest building west of the Mississippi?", the LLM might generate a postcondition requiring a check that the city where the building is located is within the United States and that its geographic coordinates are indeed west of the Mississippi River.

Statistical verification: While so far we have primarily focused on verifying whether a query answer satisfies its specification, in some cases, the focus shifts to the "aggregate performance" of an LLM-based system. This includes metrics such as its score on benchmarks like MMLU [64] or performance in real-world scenarios using platforms like ChatBot Arena [65]. A particular challenge in the latter setting is the lack of control over the workload (i.e., users' queries). Changes in the workload can significantly impact the system's performance. One approach to address this is to measure the LLM system's performance at deployment and continuously monitor it to detect any degradation over time. In this case, the performance metric, as measured at the deployment time, becomes the solution specification. A promising research direction is to develop statistically robust techniques for evaluating system performance in real-world conditions and detecting performance changes quickly and effectively.

8.2 Debuggability: challenges and solutions

When the verification of a task's output concludes that the output is incorrect, the next step is to debug (fix) it. Unfortunately, debugging an LLM-based system is inherently challenging. Unlike a traditional program, an LLM is virtually a black box. Even with full access to its weights, it is extremely difficult to correlate individual weight values with the semantics of the output. The traditional approach to debugging LLMs involves trial and error. This typically includes rewriting the prompt (prompt engineering) to influence the output or leveraging the stochastic nature of LLMs to generate alternative outputs for the same prompt. While these techniques can often improve output quality, they are neither straightforward to apply nor do they guarantee correctness. Debugging an LLM-based system is akin to debugging a huge program by looking at the output and changing a few lines of code and hoping for the best, without using any debugging tools, not even printfs!

Below, we propose several techniques for correcting the outputs of an LLM task, assuming the task has a clear specification. These techniques involve generating multiple prompts based on the original prompt and, in some cases, combining the outputs of these prompts to achieve the desired results.

Self-consistency: One research direction is to generate multiple outputs for the same task, outputs that are expected to be consistent, and then check whether they are indeed consistent. Assume a task with a statement specification, but no solution specification. One possibility would be to use the statement specification to generate not only (1) the task's solution, but also (2) the solution specification, and (3) then use the solution specification to check the actual task's solution. For example, given a PRD, we can generate both the program and the unit tests and check whether the generated program passes the unit tests. If yes, we conclude that there is a good chance the program is correct. Here we assume that there is a relatively low probability that both the generated program and the unit tests are incorrect, and the incorrect program passes the incorrect unite tests. Examples of works that already leverage this technique are AlphaCodium [66] and AlphaCode [43].

Mixture of outputs: A related and complementary technique is to generate many outputs for the same task and then select the best output, or derive a better answer than any output that has been generated so far. Examples are using majority voting, or another LLM acting as a “judge” to select the best (or the most correct) output [67, 68], or even to aggregate these outputs to obtain a better one [68, 69].

Process supervision: This is a recently proposed approach [62], and it is related to step-by-step verification. If a step fails during step-by-step verification, we can ask the same LLM or another LLM to fix the solution based on this feedback. We can employ this approach iteratively until there are no longer errors in the output. If we cannot fix the output (maybe after a predefined number of iterations) we provide the feedback to the user.

Automatic prompt generation: Most existing prompt engineering techniques are manual. However, given the huge number of possible prompts describing the same task, one possible direction is to generate these prompts automatically until we get the desired output. One project following this approach is DSPy [17].

Divide and conquer: As mentioned above, one reason LLMs are difficult to debug is because they are black boxes. One possibility to alleviate this challenge is to decompose a task into subtasks where each subtask implements a narrow functionality by eventually calling an LLM (which in turn might involve a function call). The intuition here is that it is easier to verify and debug a specialized sub-task than a general task. For example, assume that you want to book a vacation to Rome. Then, it is easier to verify a plane ticket reservation than the entire trip. Note this is related to planning and compound AI systems [28], where an LLM provides a plan and each step in the plan is implemented by another LLM call. Some of these LLM calls can implement function calling. This is also related to the recently proposed "structured outputs" [23] in which a developer can provide the pseudocode of a task which consists of multiple functions. This helps one verify and debug the implementation of each of these functions in isolation.

Statistical debugging: LLM-based systems often consist of multiple components [28, 29], such as knowledge retrieval, language understanding, solution generation, and solution selection. As discussed in Section 8.1 (see "Statistical Verification"), in some cases, we are interested in the system’s overall performance rather than the correctness of individual answers—for instance, in a customer support system, we may prioritize overall user satisfaction. How can we debug such a system? How can we pinpoint the issue if the system’s performance declines? Since many of these components (e.g., LLMs) are black-box, it is hard, even infeasible, to debug them in isolation. An alternative is to use a *grey-box* approach [70]: make changes to the system, monitor its performance, and attribute the issues to a particular component. In particular, we can replace a component with an equivalent one (e.g., substitute one model for another, or one vector store for another) or change a component’s configuration (e.g., modify an LLM’s temperature or the number of items retrieved by a vector store). We then monitor the performance of the modified system, compare it with the performance of the unmodified system, and use statistical methods to attribute performance regressions to specific components. An exciting research direction is developing statistical techniques that can accurately identify problematic components with minimal additional data and suggest potential changes to improve performance.

Appendix C provides several suggestions of projects that focus on verifiability and debuggability.

8.3 Modularity

To implement more sophisticated tasks, practitioners and researchers have developed frameworks that combine simpler components into larger, more complex systems. Examples of such frameworks include LangChain [71], LlamaIndex [72], AutoGPT [73], FrugalML [74], FrugalGPT [75], MemGPT [76], and others. Additionally, a recent blog post on compound AI systems demonstrates how composing LLM-based components can significantly enhance performance across various tasks [28]. All these efforts are closely related to the concept of modularity. Modularity involves breaking down a complex task into simpler subtasks, implementing and testing these subtasks independently, and then combining the components that implement them into a cohesive system.

Clear statement specifications are essential for modularity. Without each component having a clear statement specification, it becomes difficult, even infeasible, to define the behavior and outputs of the entire system. Therefore, the approaches to improving statement specifications, which we discussed in Section 7.2, will directly contribute to enhancing modularity.

Next, we propose two additional approaches for building more effective modular LLM-based systems.

Task decomposition: Given a complex task, how should we decompose it into smaller tasks to optimize performance, cost, or a combination of both? Early work in this area includes FrugalML [74] and FrugalGPT [75], which optimize cost and performance by selecting among similar AI services; RouteLLM [77], which chooses across multiple LLMs based on prompt difficulty; Mixture of Agents [78] and Network of Networks [68], which utilize multiple LLMs, where each LLM can play a different role (e.g., generator, judge, aggregator, majority voter) to boost performance; AutoGen [79] which decomposes tasks into sub-tasks and implements each sub-task as an agent. These are important lines of work that will help us better understand the behavior of compound AI and agent-based systems. This understanding can be used to devise new algorithms and techniques for architecting complex systems by defining the key components and how they connect to one another, similar to how we design and architect software systems today.

Furthermore, instead of the developer architecting the system, we could have another LLM-based solution do this. Such a solution could experiment with different decompositions and components to find the optimal configuration.

Increasing diversity: Task specification and verification are not enough—we also need to find an actual solution to the task. One approach to finding a solution is to generate a large number of proposals and check each one until we find one that passes the task's solution specification. To increase the chances of finding such a solution, we can maximize the diversity of the generated proposals. Many problem-solving LLM-based systems, such as AlphaCode [43] and AlphaGeometry [80], (implicitly) rely on this property. Diversity is also a key of widespread approaches such as Monte-Carlo Tree Search and Reinforcement Learning (e.g., AlphaGo [81], Alpha Go Zero [82]) which rely heavily on exploration to find new, better solutions. These approaches achieve diversity by carefully exploring the solution space. One research direction is to efficiently generate diverse proposals for general tasks.

8.4 Reusability

Reusability refers to the ability to leverage already *existing* components to build larger, more complex systems. It is closely tied to modularity, which specifies that a system is composed of multiple components. Reusability complements this concept by incorporating pre-existing components into modular systems. This is an area where LLMs excel. Foundation models, such as ChatGPT and Claude, are frequently offered as hosted services, and the majority of current LLM applications rely heavily on these services. However, there have been relatively few successes in combining these services to create larger, more powerful systems. We argue that this is primarily due to the lack of clear specifications, which complicates reasoning and debugging in such systems. This observation highlights that disambiguating statement specifications could significantly enhance the reusability of existing components.

Next, we explore two additional directions for future work.

Constraint-based decomposition: When decomposing a task into smaller, simpler components, we often want to reuse existing components that we have built or have access to, such as a specific LLM model or a particular agent. One question that arises is which of the existing components should be used, and how to best integrate these components into the overall system architecture.

Component substitution: Assume we have already built an LLM system and now want to substitute one component with another—perhaps we've changed the LLM provider, or a new, improved version of a model we are using has been released. Can we replace the LLM used by an application with another LLM without negatively impacting the application's performance? More generally, can we predict the performance of the application after replacing the LLM?

8.5 Automated decision making

In contrast to traditional software, nearly all of today's generative AI applications rely on a human in the loop—-whether in text-to-image and text-to-video generation, customer support, programming assistants, summarization, document search, or other domains. As discussed, this is because LLM prompts generally lack clear and precise solution specifications, making it challenging to assess the correctness of their outputs. However, even when tasks have unambiguous specifications, humans may still be required to assess their correctness if they have context that is missing from specifications (see discussion in Section 8.1).

Thus, to enable automated decision-making, we need not only clear solution specifications but also the ability to verify these specifications automatically. This requires codifying any human context needed for verifiability within the solution specification. Likewise, any tools required for verification——such as calculators or virtual machines to execute generated code——must be invoked automatically, with their results interpreted unambiguously.

Possible research directions include: (1) assessing when a solution specification can be verified automatically, (2) identifying gaps that hinder automatic verification of a solution specification, and (3) developing methods to address and fill these gaps.

9 Summary

Engineering disciplines, including software engineering, have been responsible for driving staggering economic progress since the start of the industrial revolution. In this paper, we contend that this progress was enabled by five key properties: verifiability, debuggability, modularity, reusability and automatic decision making. The first two properties enable developers to quickly build systems from scratch, the next two properties enable developers to build *reliable* LLM-based systems, and the last property enables developers to build systems that require no human intervention. Altogether, these properties have enabled an exponential growth of the ecosystem which ultimately fueled economic growth.

More importantly, we assert that the foundation for these engineering properties lies in having clear *specifications* that precisely describe what a task should accomplish (i.e., statement specifications) and provide all the information needed to verify whether the task's outputs are correct (i.e., solution specifications).

AI, and in particular LLMs, places us on the cusp of another economic and social revolution. However, we observe that the specifications—both statement and solution specifications—of most LLM tasks are ambiguous, in part because they use natural language. While natural language makes LLMs widely accessible and general (i.e., *anyone* can describe *any* task), its inherent ambiguity makes it difficult to build new, more powerful and reliable systems from (existing) simpler ones, which ultimately hampers the ecosystem and economic growth.

To address this limitation, we argue for developing new techniques that make it easier to write clear statement specifications and solution specifications to verify that an LLM system's outputs are correct, and debug the LLM system if they are not. By doing so, we can support all five software engineering properties, and enable developers to accelerate the development of new LLM solutions, solutions that are more powerful and reliable, and which can address a growing number of use cases.

Acknowledgements: We would like to thank everyone who reviewed earlier versions of this paper and provided valuable feedback, which significantly improved its quality: Lakshya A. Agrawal, Ali Ghodsi, Zhuohan Li, Robert Nishihara, Ying Sheng and Lianmin Zheng.

References

[1] M. Andreessen, "Why software is eating the world," 2011. https://a16z.com/why-software-is-eating-the-world/.

[2] T. Simonite, "Nvidia ceo: Software is eating the world, but ai is going to eat software," 2017. https://www.technologyreview.com/2017/05/12/151722/nvidia-ceo-software-is-eating-the-world-but-ai-is-going-to-eat-software.

[3] A. Kumar, "Ai is eating the software that is eating the world," 2023. https://www.linkedin.com/pulse/ai-eating-software-world-kumar-aakash-aacash-eth-/.

[4] D. Miessler, "How ai is eating the software world," 2023. https://danielmiessler.com/p/ai-is-eating-the-software-world/.

[5] "The coq proof assistant." https://coq.inria.fr/.

[6] "The gallina specification language." https://coq.inria.fr/doc/v8.9/refman/language/gallina-specification-language.html.

[7] F. P. B. Jr., *The Mythical Man-Month: Essays on Software Engineering*. Reading, Massachusetts: Addison-Wesley, 1st ed., 1975.

[8] "Hallucinations (atificial intelligence)." https://en.wikipedia.org/wiki/Hallucination_%28artificial_intelligence%29.

[9] Y. Tian, W. Yan, Q. Yang, X. Zhao, Q. Chen, W. Wang, Z. Luo, L. Ma, and D. Song, "Codehalu: Investigating code hallucinations in llms via execution-based verification," 2024.

[10] Jerome Goddard, "Hallucinations in ChatGPT: A cautionary tale for biomedical researchers," in *The American Journal of Medicine*, Elsevier Inc, 2023. https://www.amjmed.com/action/showPdf?pii=S0002-9343%2823%2900401-1.

[11] "What Air Canada lost in 'remarkable' lying AI chatbot case." https://www.forbes.com/sites/marisagarcia/2024/02/19/what-air-canada-lost-in-remarkable-lying-ai-chatbot-case/.

[12] "LLM hallucinations and failures: lessons from 4 examples." https://www.evidentlyai.com/blog/llm-hallucination-examples.

[13] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang, "Fuzzing: A survey for roadmap," *ACM Comput. Surv.*, vol. 54, Sept. 2022.

[14] W. B. Demilie and F. G. Deriba, "Detection and prevention of sqli attacks and developing compressive framework using machine learning and hybrid techniques," *Journal of Big Data*, vol. 9, no. 1, p. 124, 2022.

[15] E. Mollick, "Getting started with AI: Good enough prompting," 2024. https://www.oneusefulthing.org/p/getting-started-with-ai-good-enough.

[16] J. S. Damji, "Best prompt techniques for best llm responses," 2024. https://medium.com/the-modern-scientist/best-prompt-techniques-for-best-llm-responses-24d2ff4f6bca.

[17] O. Khattab, A. Singhvi, P. Maheshwari, Z. Zhang, K. Santhanam, S. Vardhamanan, S. Haq, A. Sharma, T. T. Joshi, H. Moazam, H. Miller, M. Zaharia, and C. Potts, "Dspy: Compiling declarative language model calls into self-improving pipelines," 2023.

[18] Y. Bai, S. Kadavath, S. Kundu, A. Askell, J. Kernion, A. Jones, A. Chen, A. Goldie, A. Mirhoseini, C. McKinnon, C. Chen, C. Olsson, C. Olah, D. Hernandez, D. Drain, D. Ganguli, D. Li, E. Tran-Johnson, E. Perez, J. Kerr, J. Mueller, J. Ladish, J. Landau, K. Ndousse, K. Lukosuite, L. Lovitt, M. Sellitto, N. Elhage, N. Schiefer, N. Mercado, N. DasSarma, R. Lasenby, R. Larson, S. Ringer, S. Johnston, S. Kravec, S. E. Showk, S. Fort, T. Lanham, T. Telleen-Lawton, T. Conerly, T. Henighan, T. Hume, S. R. Bowman, Z. Hatfield-Dodds, B. Mann, D. Amodei, N. Joseph, S. McCandlish, T. Brown, and J. Kaplan, "Constitutional ai: Harmlessness from ai feedback," 2022.

[19] "NVIDIA NeMo Guardrails." https://docs.nvidia.com/nemo/guardrails/.

[20] B. T. Willard and R. Louf, "Efficient guided generation for large language models," 2023.

[21] "Guidance." https://github.com/guidance-ai/guidance.

[22] L. Zheng, L. Yin, Z. Xie, C. Sun, J. Huang, C. H. Yu, S. Cao, C. Kozyrakis, I. Stoica, J. E. Gonzalez, C. Barrett, and Y. Sheng, "Sglang: Efficient execution of structured language model programs," 2024.

[23] "Structured Outputs." https://platform.openai.com/docs/guides/structured-outputs.

[24] "Pydantic." https://docs.pydantic.dev/latest/.

[25] "Zod." https://zod.dev/.

[26] "Trace." https://microsoft.github.io/Trace/.

[27] A. El-Kishky, F. S. Daniel Selsam, G. Parascandolo, H. Ren, H. Lightman, H. W. Chung, I. Akkaya, I. Sutskever, J. Wei, J. Gordon, K. Cobbe, K. Yu, L. Kondraciuk, M. Schwarzer, M. Rohaninejad, N. Brown, S. Zhao, T. Bansal, and W. Z. Vineet Kosaraju, "Learning to reason with llms," 2024. https://openai.com/index/learning-to-reason-with-llms/.

[28] M. Zaharia, O. Khattab, L. Chen, J. Q. Davis, H. Miller, C. Potts, J. Zou, M. Carbin, J. Frankle, N. Rao, and A. Ghodsi, "The shift from models to compound ai systems," 2024. https://bair.berkeley.edu/blog/2024/02/18/compound-ai-systems/.

[29] P. P. Singh, "Agentic systems : A business perspective," 2024. https://www.linkedin.com/pulse/agentic-systems-business-perspective-prabhdeep-pd-singh-wy0oc/.

[30] B. Morris, "Why building ai-powered agents is so challenging. for now.," 2024. https://www.ben-morris.com/why-building-ai-powered-agents-is-so-challenging-for-now/.

[31] H. Luo and L. Specia, "From understanding to utilization: A survey on explainability for large language models," 2024.

[32] "Transformer debugger," 2024. https://github.com/openai/transformer-debugger.

[33] "Awesome interpretability in large language models," 2024. https://github.com/ruizheliUOA/Awesome-Interpretability-in-Large-Language-Models.

[34] K. Goldberg, "Is data all you need?: Large robot action models and good old fashioned engineering," 2024. https://docs.google.com/document/d/1WZwS4ZAWFNTwusFmW6UKYrs9QqYy7nc2qLPZbeuCmDE/edit.

[35] "For Model K." https://en.wikipedia.org/wiki/Ford_Model_K.

[36] "ENIAC computer." https://www.britannica.com/technology/ENIAC.

[37] L. Lamport, "TLA+," 2024. https://lamport.azurewebsites.net/tla/tla.html.

[38] "Tutorial for alloy analyzer 4.0." https://alloytools.org/tutorials/online/.

[39] "Lean." https://lean-lang.org/about/.

[40] "Design by contract." https://en.wikipedia.org/wiki/Design_by_contract.

[41] E. Çınlar, *Probability and Stochastics*. Graduate Texts in Mathematics, Springer New York, 2011.

[42] L. Kleinrock, *Theory, Volume 1, Queueing Systems*. USA: Wiley-Interscience, 1975.

[43] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago, T. Hubert, P. Choy, C. de Masson d'Autume, I. Babuschkin, X. Chen, P.-S. Huang, J. Welbl, S. Gowal, A. Cherepanov, J. Molloy, D. J. Mankowitz, E. Sutherland Robson, P. Kohli, N. de Freitas, K. Kavukcuoglu, and O. Vinyals, "Competition-level code generation with alphacode," *Science*, vol. 378, p. 1092–1097, Dec. 2022.

[44] "BlazingSQL." https://github.com/BlazingDB.

[45] "JsqlParser." https://github.com/JSQLParser/JSqlParser.

[46] "Eiffel." https://en.wikipedia.org/wiki/Eiffel_(programming_language).

[47] "SPARK." https://www.adacore.com/about-spark.

[48] "Introduction to Ansys SCADE suite." https://resources.randsim.com/brochures/intro-to-ansys-scade-suite.

[49] A. Karpathy. https://x.com/karpathy/status/1617979122625712128?lang=en.

[50] S. K. Lahiri, S. Fakhoury, A. Naik, G. Sakkas, S. Chakraborty, M. Musuvathi, P. Choudhury, C. von Veh, J. P. Inala, C. Wang, and J. Gao, "Interactive code generation via test-driven user-intent formalization," 2023.

[51] M. Endres, S. Fakhoury, S. Chakraborty, and S. K. Lahiri, "Can large language models transform natural language intent into formal method postconditions?," 2024.

[52] S. K. Lahiri, "Evaluating llm-driven user-intent formalization for verification-aware languages," 2024.

[53] Y. LeCunn, "A path towards autonomous machine intelligence," 2022. https://openreview.net/pdf?id=BZ5a1r-kVsf.

[54] V. Pallagani, K. Roy, B. Muppasani, F. Fabiano, A. Loreggia, K. Murugesan, B. Srivastava1, F. Rossi, L. Horesh, and A. Sheth1, "On the prospects of incorporating large language models (LLMs) in automated planning and scheduling (APS)," 2024. https://arxiv.org/html/2401.02500v2.

[55] P. Tolmach, Y. Li, S.-W. Lin, Y. Liu, and Z. Li, "A survey of smart contract formal specification and verification," 2021.

[56] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Lüttgen, A. J. H. Simons, S. Vilkomir, M. R. Woodward, and H. Zedan, "Using formal specifications to support testing," 2009. https://dl.acm.org/doi/10.1145/1459352.1459354.

[57] G. C. Necula, "Proof-carrying code," in *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1997.

[58] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," 2023.

[59] S. Yao, D. Yu, J. Zhao, I. Shafran, T. L. Griffiths, Y. Cao, and K. Narasimhan, "Tree of thoughts: Deliberate problem solving with large language models," 2023.

[60] J. Lanchantin, S. Toshniwal, J. Weston, A. Szlam, and S. Sukhbaatar, "Learning to reason and memorize with self-notes," 2023.

[61] M. Nye, A. J. Andreassen, G. Gur-Ari, H. Michalewski, J. Austin, D. Bieber, D. Dohan, A. Lewkowycz, M. Bosma, D. Luan, C. Sutton, and A. Odena, "Show your work: Scratchpads for intermediate computation with language models," 2021.

[62] L. Luo, Y. Liu, R. Liu, S. Phatale, H. Lara, Y. Li, L. Shu, Y. Zhu, L. Meng, J. Sun, and A. Rastogi, "Improve mathematical reasoning in language models by automated process supervision," 2024.

[63] S. G. Patil, T. Zhang, V. Fang, R. Huang, A. Hao, M. Casado, J. E. Gonzalez, R. A. Popa, I. Stoica, *et al.*, "Goex: Perspectives and designs towards a runtime for autonomous llm applications," *arXiv preprint arXiv:2404.06921*, 2024.

[64] D. Hendrycks, C. Burns, S. Basart, A. Zou, M. Mazeika, D. Song, and J. Steinhardt, "Measuring massive multitask language understanding," 2021.

[65] W.-L. Chiang, L. Zheng, Y. Sheng, A. N. Angelopoulos, T. Li, D. Li, H. Zhang, B. Zhu, M. Jordan, J. E. Gonzalez, and I. Stoica, "Chatbot arena: An open platform for evaluating llms by human preference," 2024.

[66] T. Ridnik, D. Kredo, and I. Friedman, "Code generation with alphacodium: From prompt engineering to flow engineering," 2024.

[67] L. Zheng, W.-L. Chiang, Y. Sheng, S. Zhuang, Z. Wu, Y. Zhuang, Z. Lin, Z. Li, D. Li, E. P. Xing, H. Zhang, J. E. Gonzalez, and I. Stoica, "Judging llm-as-a-judge with mt-bench and chatbot arena," 2023.

[68] J. Q. Davis, B. Hanin, L. Chen, P. Bailis, I. Stoica, and M. Zaharia, "Networks of networks: Complexity class principles applied to compound ai systems design," 2024.

[69] P. Verga, S. Hofstatter, S. Althammer, Y. Su, A. Piktus, A. Arkhangorodsky, M. Xu, N. White, and P. Lewis, "Replacing judges with juries: Evaluating llm generations with a panel of diverse models," 2024.

[70] "Gery-Box Testing." `https://en.wikipedia.org/wiki/Gray-box_testing`.

[71] "Langchain." `https://pypi.org/project/langchain/`.

[72] "Llamaindex." `https://github.com/run-llama/llama_index`.

[73] H. Yang, S. Yue, and Y. He, "Auto-gpt for online decision making: Benchmarks and additional opinions," 2023.

[74] L. Chen, M. Zaharia, and J. Zou, "Frugalml: How to use ml prediction apis more accurately and cheaply," 2020.

[75] L. Chen, M. Zaharia, and J. Zou, "Frugalgpt: How to use large language models while reducing cost and improving performance," 2023.

[76] C. Packer, S. Wooders, K. Lin, V. Fang, S. G. Patil, I. Stoica, and J. E. Gonzalez, "Memgpt: Towards llms as operating systems," 2024.

[77] I. Ong, A. Almahairi, V. Wu, W.-L. Chiang, T. Wu, J. E. Gonzalez, M. W. Kadous, and I. Stoica, "Routellm: Learning to route llms with preference data," 2024.

[78] "Mixture-of-agents enhances large language model capabilities," 2024.

[79] Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. Zhu, L. Jiang, X. Zhang, S. Zhang, J. Liu, A. H. Awadallah, R. W. White, D. Burger, and C. Wang, "Autogen: Enabling next-gen llm applications via multi-agent conversation," 2023.

[80] T. H. Trinh, Y. Wu, Q. V. Le, H. He, and T. Luong, "Solving olympiad geometry without human demonstrations," 2024.

[81] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of go with deep neural networks and tree search," 2016.

[82] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," 2017.

[83] "Carl Benz's first car." `https://group.mercedes-benz.com/company/tradition/company-history/1885-1886.html`.

[84] "Ford Model T." `https://en.wikipedia.org/wiki/Ford_Model_T`.

Appendix A: The five engineering properties: Software Systems vs LLMs

Table 3 compares how software systems and LLM systems satisfy the five properties of software engineering (see section 4).

Property	**Description**	**Software Systems (mostly unambiguous, but less accessible (need to learn a programming language) and less general (only support computation tasks))**	**LLM Systems (mostly ambiguous, but more accessible (just describe the task in english), and more general (support any task one can describe in english))**
Verifiability	If output meets the specification, the output is valid; otherwise it is invalid	Specifications enable output validation, e.g., via validation tests, proof that the solution satisfies specification.	If a task is ambiguous (e.g., natural language), it is very hard to validate the output given an input. Typically humans verify the output is valid by using extra context or/and additional tools.
Debuggability	If output is not correct, modify the system to generate the correct output (i.e., output to meet program's specification)	Programs are recursively built from smaller components each with a unambiguous specification. This enables debugging by recursively narrowing down the segment of code responsible for the incorrect output and fixing the code.	LLMs are black-boxes, which makes it very hard to debug: you can only change the prompt and there is no easy way to reason on how the input can change the output.
Modularity	Build bigger systems from smaller ones.	Speed up building larger apps from smaller components which are implemented and tested in parallel (e.g., microservices, workflows, multi-module program).	Enable developers building more complex apps by composing multiple LLMs. Unfortunately, if these LLMs are ambiguous, the outputs of the composed app will also be ambiguous.
Reusability	Reuse existing components to build new ones.	Speed up building new apps by leveraging existing components (e.g., libraries, databases, web servers, load balancers).	Enable building new apps leveraging existing components (e.g., Vector Stores) and LLM service (e.g, ChatGPT).
Automated decision making	Build systems that make decisions without human intervention.	Enable building automated decision-making systems by leveraging unambiguous and complete specifications to guarantee the correctness of the output, e.g., car, plane, factory software.	Most popular LLM apps have a human in the loop. This is because, in general, specifications are ambiguous so cannot guarantee output correctness.

Table 3: Comparison between traditional software development and LLM prompting.

Appendix B: Why modular AI systems?

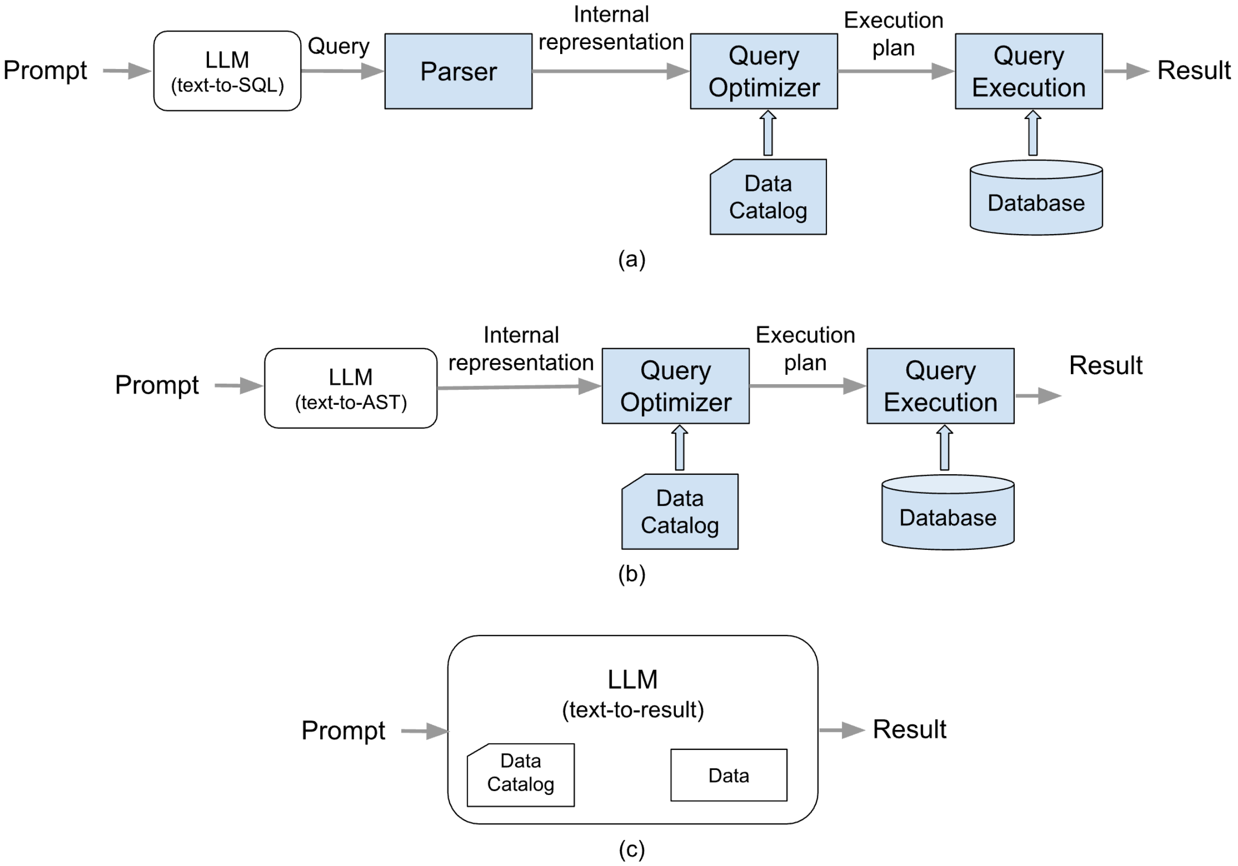
So far we have implicitly assumed that it is a good idea to build AI systems by combining many sub-systems (components). But why is this? After all, the generality of LLMs enables anyone to specify any task. So, why not just describe that task carefully in a prompt and use a single LLM call to solve it? Here are few reasons (many of them have been well articulated in a recent blog on Compound AI Systems [28]) of why we might want to build modular AI systems:

- **Better accuracy**: Given a particular domain, it is possible to use specific data to train or fine-tune a model to optimize its performance for that domain. Furthermore, recent research has shown that it is possible to provide better accuracy by aggregating multiple calls to different LLMs or the same LLM (see here). In particular, recent research has shown that one can get more reliable results by decomposing a complex task into smaller tasks, solve each of these tasks independently, and aggregate the results.

- **Lower cost**: By fine-tuning models on domain specific data it is possible to achieve the same performance or even exceed the performance of a generic frontier model such as ChatGPT at a fraction of cost. This also lowers the development cost by allowing one to stitch together existing LLMs and create more powerful systems.

- **Faster improvements**: Decomposing a complex task into subtasks, and having a component implementing each subtask, makes it easier to debug and improve the results. This is because it is easier to (1) identify which component doesn't perform well, (2) fix a component that implements a specialized rather than a general task, and (3) reuse existing components that already implement the desired subtask.

- **Integration with software systems**: A modular system can much easier to mix LLM components with existing software components using the best component for the subtask at hand. For example, for arithmetic operations a simple calculator can be much faster and far more accurate than the most powerful LLM. Figure 11 shows several possibilities of extending our query engine from Figure 4 to take the query description in natural language.



(a)

(b)

(c)

Figure 11: Different options to implement a query optimizer that takes as input the query description in natural language: (a) translate text to SQL and then use an existing query engine; (b) translate text to intermediate representation and then use existing query optimizer and execution components; (c) implement everything in an LLM (here we assume the LLM was trained on date and metadata or these are part of its context/prompt)

Appendix C: Research projects

Table 4 shows several research projects focussed on verifiability and debuggability.

Project	Description	Properties	Comments
Iterative disambiguation	Iteratively check for inputs that generate different outputs and add rules to disambiguate these outputs, as needed.	Disambiguate specification	
Domain-specific rules	Explicitly extract or define the rules for the domain and use these rules to disambiguate the specification and verify the implementation.	Verifiability: use rules to check the implementation	A rule is akin to a unit test.
formal specification → code + proofs	E.g., (1) the user uses Coq/Gallina (or similar) to write formal specifications; (2) the LLM generates both programs and proofs that the program meets the certification.	Verifiability: verify proof & Debuggability: If the proof verification fails, the way it fails might help the users identify bugs	
Tests → code	Given a comprehensive set of tests, generate the code to pass these tests.	Verifiability: via testing & Debuggability: If a test fails, it might help the user identify bugs	Similar to DSPy for code generation, and generate code from input/output examples in software systems
Fact-based validation	Output provides references to reputable sources	Verifiability: verify the information in the output is correct. Debuggability: if the output answers the prompt but the references are inconsistent with the output, we can identify the problems with the output.	Verifying the information in the output is correct, doesn't necessarily mean the answer is correct (the output might have no relation with the input).
Consistency checking (generate both code and tests)	Generate the solution (program) and tests from the same prompt. If solution passes the tests, we assume the solution is correct,	Verifiability: Improves results, but no guarantees. Debuggability: if tests are correct. In this case, the failed tests can point us to bugs in the code.	Code might pass tests, but both code and tests can be wrong.
(Automatic) prompt engineering	Chain-of-X, self-notes, memGPT, DSPy (automatic), etc	Debuggability: Improve results.	Orthogonal to other projects.
Hybrid Systems	Compare Monolithic LLM systems with systems that use modular LLM components, and Hybrid systems that combine LLM modules with "Good Old Fashioned Engineering (GOFE)" modules such as search engines (RAG), FEM simulations, FFTs, linear and non-linear regression, etc.	Verifiability: GOFE components can be used in parallel to verify outputs. Debuggability: GOFE modules from libraries have been extensively debugged.	

Table 4: Several projects to help LLMs provide better support for the five properties in Table 3.

Appendix D: Frequently asked questions

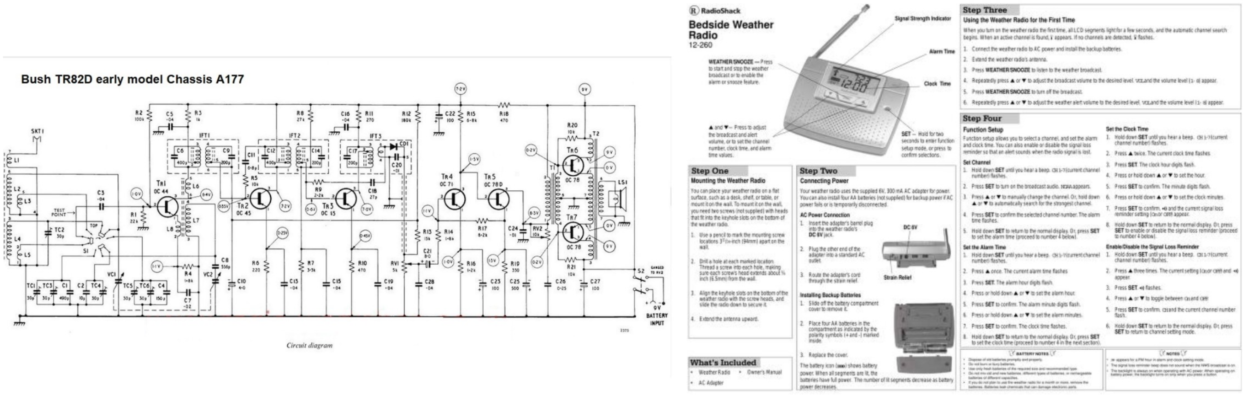
LLMs are still making rapid progress so is modularity necessary? Indeed, for the past decade every year or so we have witnessed the emergence of new models that significantly outperforms and replaces the previous models. For example, consider the sequence of revolutionary models released by OpenAI over the past six years: GPT 1 in 2018, GPT 2 in 2019, GPT 3 in 2020, GPT 3.5 in 2022, and GPT 4 in 2023. So wouldn't this process be enough to lead to exponential growth? While this might be the case, it does not change the fact that without modularity and reusability LLMs will be deprived of a huge vector of growth which is enabled by modularity/reusability.

Car industry might provide a good analogy. There have been huge advancements from the introduction of the first car [83] in 1886 until the introduction of Ford Model T [84] in 1908. During this time period new cars were manually built from the ground up similarly with today's LLMs, and each generation was markedly better then the previous one. But only after the introduction of the assembly line (which basically leverages modularity) to produce Ford Model T did the car industry explode.

Aren't humans effective in collaborating despite their ambiguous interactions? Humans are using natural language to communicate, which is inherently ambiguous. Despite this, they are still able to work effectively in organizations, so isn't this a form of modularity? Why wouldn't we be able to successfully compose LLMs in a similar way? While this is true to some extent, scaling human organizations has been an exercise in removing the ambiguity rather than tolerating it. Indeed, as a company scales, one needs to add more structure in the form of clearly defining different organizations (i.e., product, engineering, marketing, sales, finance, etc) and processes (e.g., financial reports, product planning, tracking key initiatives using objectives and key performance indicators, etc). All this added structure has one role: remove the ambiguity. In fact, the most scalable human organization is arguably the army. But the army is also the organization that defines the most unambiguous communication standards and how everyone should behave and act given her position in the command chain. Thus, the history shows that reducing ambiguity is key to scaling human organizations, which is the main point of this paper.

Aren't the ambiguous specifications just a reflection of the task people are using the LLMs for rather than the LLMs themselves? We believe the ambiguity is directly related to the difficulty of specifying a task, rather than an inherent property of the task, i.e., with enough effort most tasks can be significantly disambiguated. In the end, for any given prompt, the user still needs to make a decision of whether the response is good or not. The problem is that the user makes the decision on internal context not available to the LLM. However, this does not mean this context cannot be externalized. To see this, consider that you want to generate an image based on a text prompt, e.g., "Golden Gate Bridge at sunrise". Now, imagine that you cannot choose the picture yourself; instead the picture is picked by someone else. What would you do then? Well, probably you are going to give that person as much information as you can to increase the probability of that person will pick a picture you will like, e.g., the picture should use pastel colors, the sun should raise from the city, there should be realistic proportions between the bridge and buildings, etc. All this information helps to disambiguate the task.

What are some examples of specifications in engineering? In general, there are two types of specifications. The first type targets system's builders. These specifications should be detailed enough so one can actually build a system using its specification (see Figure 11(a)). The second type targets the system's users. These specifications should be detailed enough so one can operate the system. For physical systems such as cars, TVs, phones, microprocessors, storage systems, such specifications often come in the form of user manuals. These manuals contain instructions like setting up the system, powering it on, switching it off, a description of what each button (if any) does, and what each display (if any) should show (see Figure 11(b)). These manuals also describe the expected behavior of the system, possible errors, and troubleshooting steps. Irrespective of the type, a specification defines the outputs a system should produce given the input, which is necessary to achieve the five key properties of software systems.



RadioShack

Bedside Weather Radio
12-260

WEATHER/SNOOZE — Press to start and stop the weather broadcast or to enable the alarm or snooze feature.

▲ and ▼ — Press to adjust the broadcast and alert volume, or to set the channel number, clock time, and alarm time values.

SET — Hold for two seconds to enter function setup mode, or press to confirm selections.

Step One
Mounting the Weather Radio

You can place your weather radio on a flat surface, such as a desk, shelf, or table, or mount it on the wall. To mount it on the wall, you need two screws (not supplied) with heads that fit into the keyhole slots on the bottom of the weather radio.

1. Use a pencil to mark the mounting screw locations 3⁷/₁₆-inch (94mm) apart on the wall.
2. Drill a hole at each marked location. Thread a screw into each hole, making sure each screw's head extends about ¼ inch (6.5mm) from the wall.
3. Align the keyhole slots on the bottom of the weather radio with the screw heads, and slide the radio down to secure it.
4. Extend the antenna upward.

What's Included
- Weather Radio
- Owner's Manual
- AC Adapter

Step Two
Connecting Power

Your weather radio uses the supplied 6V, 300 mA AC adapter for power. You can also install four AA batteries (not supplied) for backup power if AC power fails or is temporarily disconnected.

AC Power Connection

1. Insert the adapter's barrel plug into the weather radio's **DC 6V** jack.
2. Plug the other end of the adapter into a standard AC outlet.
3. Route the adapter's cord through the strain relief.

Installing Backup Batteries

1. Slide off the battery compartment cover to remove it.
2. Place four AA batteries in the compartment as indicated by the polarity symbols (+ and -) marked inside.
3. Replace the cover.

The battery icon (▭) shows battery power. When all segments are lit, the batteries have full power. The number of lit segments decrease as battery power decreases.

Step Three
Using the Weather Radio for the First Time

When you turn on the weather radio the first time, all LCD segments light for a few seconds, and the automatic channel search begins. When an active channel is found, Ỹ appears. If no channels are detected, Ỹ flashes.

1. Connect the weather radio to AC power and install the backup batteries.
2. Extend the weather radio's antenna.
3. Press **WEATHER/SNOOZE** to listen to the weather broadcast.
4. Repeatedly press ▲ or ▼ to adjust the broadcast volume to the desired level. VOL and the volume level (1- 8) appear.
5. Press **WEATHER/SNOOZE** to turn off the broadcast.
6. Repeatedly press ▲ or ▼ to adjust the weather alert volume to the desired level. VOL and the volume level (1- 8) appear.

Step Four
Function Setup

Function setup allows you to select a channel, and set the alarm and clock time. You can also enable or disable the signal loss reminder so that an alert sounds when the radio signal is lost.

Set Channel

1. Hold down **SET** until you hear a beep. CH 1-7 (current channel number) flashes.
2. Press **SET** to turn on the broadcast audio. NOAA appears.
3. Press ▲ or ▼ to manually change the channel. Or, hold down ▲ or ▼ to automatically search for the strongest channel.
4. Press **SET** to confirm the selected channel number. The alarm time flashes.
5. Hold down **SET** to return to the normal display. Or, press **SET** to set the alarm time (proceed to number 4 below).

Set the Alarm Time

1. Hold down **SET** until you hear a beep. CH 1-7 (current channel number) flashes.
2. Press ▲ once. The current alarm time flashes.
3. Press **SET**. The alarm hour digits flash.
4. Press or hold down ▲ or ▼ to set the alarm hour.
5. Press **SET** to confirm. The alarm minute digits flash.
6. Press or hold down ▲ or ▼ to set the alarm minutes.
7. Press **SET** to confirm. The clock time flashes.
8. Hold down **SET** to return to the normal display. Or, press **SET** to set the clock time (proceed to number 4 in the next section).

Set the Clock Time

1. Hold down **SET** until you hear a beep. CH 1-7 (current channel number) flashes.
2. Press ▲ twice. The current clock time flashes.
3. Press **SET**. The clock hour digits flash.
4. Press or hold down ▲ or ▼ to set the hour.
5. Press **SET** to confirm. The minute digits flash.
6. Press or hold down ▲ or ▼ to set the clock minutes.
7. Press **SET** to confirm. ◂) and the current signal loss reminder setting (ON or OFF) appear.
8. Hold down **SET** to return to the normal display. Or, press **SET** to enable or disable the signal loss reminder (proceed to number 4 below).

Enable/Disable the Signal Loss Reminder

1. Hold down **SET** until you hear a beep. CH 1-7 (current channel number) flashes.
2. Press ▲ three times. The current setting (ON or OFF) and ◂) appear.
3. Press **SET**. ◂) flashes.
4. Press ▲ or ▼ to toggle between ON and OFF.
5. Press **SET** to confirm. CH and the current channel number flash.
6. Hold down **SET** to return to the normal display. Or, press **SET** to return to channel setting mode.

BATTERY NOTES

- Dispose of old batteries promptly and properly.
- Do not burn or bury batteries.
- Use only fresh batteries of the required size and recommended type.
- Do not mix old and new batteries, different types of batteries, or rechargeable batteries of different capacities.
- If you do not plan to use the weather radio for a month or more, remove the batteries. Batteries leak chemicals that can damage electronic parts.

NOTES

- PM appears for a PM hour in alarm and clock setting mode.
- The signal loss reminder beep does not sound when the NWS broadcast is on.
- The backlight is always on when operating with AC power. When operating on battery power, the backlight turns on only when you press a button.

(a) (b)

Figure 12: Two radio specifications, (a) for builders and (b) users, respectively