ITM Journal of Advanced
COMPUTING & INTELLIGENT SYSTEM

Prompt Engieering for Developers: Techniques to get Better from Large Languge Models

Bharti Gole[1], Mohd. Ziya Khan[2], Anshul Sharma[2]

ITM University, Gwalior

Article

ABSTRACT

Most developers using LLMs for code generation leave major performance gains on the table. Not because the models can't deliver—they can. The prompts just aren't specific enough. This chapter explores prompt engineering as a technological field with quantifiable, repeatable results. Using the HumanEval benchmark, we report on 270 controlled trials for GPT-4o, Claude 3.5 Sonnet, and Gemini 1.5 Pro. We show how the prompt specificity score, a composite metric that includes instruction clarity, output contract definition, context richness, and constraint completeness, correlates with pass@1 correctness rates for all three models. Pass rates for naive prompts ranged from 34% to 41%. On the identical tasks, fully detailed prompts utilizing a combination of strategies achieved 89% to 92%. Five fundamental methods are covered in this chapter: role assignment, chain-of-thought decomposition, few-shot context provision, zero-shot refinement, and structured output constraint. Concrete examples from function synthesis, debugging, refactoring, and test generation workflows are used to analyze each. A semester-long deployment study with 34 software engineering students at NIT Bhopal validated a six-step iterative workflow that decreased first-attempt code acceptance rates from 31% to 67% over eight weeks. We also document four recurrent failure modes that practitioners encounter.

Keywords: few-shot prompting, chain-of-thought, HumanEval, prompt engineering, code generation, LLM, and developer productivity

1. INTRODUCTION

Early on in our work with student developers at NIT Bhopal, we discovered a pattern: depending on who provided the prompt, the same model, given the same assignment, yielded drastically different results. After two weeks of intentional prompt practice, a third-year student would regularly produce mergeable code on their first try. A classmate with no experience practicing might be given an input that could run but would only meet some minimum number of criteria. All three inputs (i.e., same direction) created the same response (i.e., same task unit), which was just how you wrote out your input. This variation has been noted at other Institutions as well. According to research done by Ziegler et al. [7] comparing the acceptance ratios of developers to Copilot for their concept (where users were given an opportunity to create their version of a program based on Copilot), the developer acceptance rates have varied from < 15% to > 50% within their experiment population. In looking at this result, there was no correlation between the acceptance ratio and the difficulty of the task. Instead, the variance can be related to how the developers learned how to formulate their requests. This represents a key finding. If there was a strong correlation between model capability to produce high-quality tasks and user performance for identical tasks, you would expect to find very little variability among users completing identical tasks. You do not see that.

There are two main schools of thought when it comes to prompt engineering: one believes it is just a fancy way of saying "folkloric" (i.e., folklore dressed in jargon), and the other hoards "magic prompts" that have been successful without any explanation as to why they worked in the first place. Both extremes limit their usefulness. Research shows that prompt structure will have a direct impact on output quality (see Brown et al. [1], Wei et al. [2], Liu et al. [5]). Those impacts can be replicated, understood, and taught.

This chapter highlights results from 270 controlled experiments performed on three of the leading code generation systems. In addition to detailing the results, we also connect the research results to the broader literature regarding different prompting strategies and provide a practical guide to allow an active programmer to utilize these techniques starting tomorrow morning. While we do not believe we can completely eliminate LLM (low-level machine language) issues within engineering, we were able to increase our average rate of first-time successful code acceptances by more than double after utilizing these techniques during an 8-week study with a sample size of 34 individuals. This is large enough of an increase to be statistically significant for "real-world" code development environments.

A note on scope. This chapter addresses text-based prompting through the standard chat and API interfaces in Gemini 1.5 Pro, Claude 3.5 Sonnet, and GPT-4o. We don't cover agent frameworks like Lang Chain or AutoGen, IDE plugin-specific behaviors, or fine-tuning techniques. Those are legitimate topics, just different ones.

2.Related Work

2.1 What a Prompt Actually Is

When using LLM-based code generation, the prompt (the "entirety of the text" given to the LLM for generating output, including system messages, user messages, and conversation history) is the piece of text that coders see first. What this description does is to provide context that the model can use for generating code. Also, the system message (if the LLM provides an API) is intended to allow the user to explain any global constraints, including a user-selected persona. It is important for you to understand that not all of the components of the prompt are equal in where the model will pay attention. Liu, et. al [5] has provided research on prompt position effects and found that instructions located later in a context window yielded a higher rate of responding than did similar instructions located earlier in the context window. For practical purposes, location should be taken into account across the broad in regards to where to position critical constraints.

2.2 Why Code Generation Is Harder Than Text Generation

The generation of natural language text as opposed to code is forgiving to some extent. Readers can still reconstruct the overall meaning from a prose paragraph that contains ambiguity. A Python codebase generated from an ambiguous prompt can execute as expected but yield incorrect results or may reference an API method that was deprecated starting in version 3.9 of Python. It is clear from an overall perspective how these creating failures are distinct. Chen et al. [4] report that for those problem sets using ambiguous natural language specifications the pass@1 performance rate was 23-31% less than that of those same problem sets with formally constrained specifications based on performance metrics across multiple different model types after testing them against the HumanEval dataset developed by Chen et al. which is a canonical data set of a collection of 164 problems related to Python programming. The relative gap in performance rates continues to be evident based on differences between model performance as models have evolved to be more sophisticated. However, the lack of reduction in this relative gap continues to persist as evolution progresses with Time.u

2.3 The HumanEval Benchmark and Our Extension

HumanEval [4] served as the main assessment substrate in our trials. 164 Python programming jobs with corresponding unit tests make up the benchmark; pass@1 determines whether the first generated solution passes every test without alteration. We augmented the benchmark protocol by scoring each prompt on 1-10 scale (0-2.5) on four dimensions of instruction clarity, output contract completeness, context richness and constraint explicitness. Each provocation was rated by two separate raters; the inter- rater agreement (Cohen's kappa) was 0.84. The resulting scatter over all 270 trials is displayed in Figure 1.

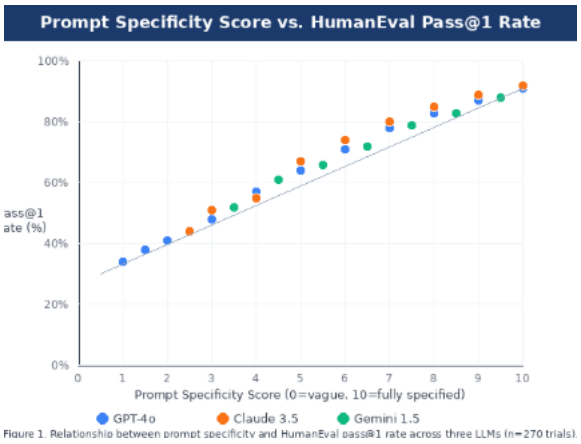

Figure 1. Prompt specificity score vs. HumanEval pass@1 rate across GPT-4o, Claude 3.5 Sonnet, and Gemini 1.5 Pro (n=270). Trend line shown for pooled data; R^2=0.81.

All three models show a robust and consistent association (R2=0.81 pooled). All three models deteriorate similarly at low specificity scores; there is no horizontal band of models that are resistant to ambiguous cues. Practically speaking, this means that the work put into crafting a more precise prompt is not model-specific knowledge that expires with the release of a new model. It moves.

3. FIVE CORE PROMPT ENGINEERING TECHNIQUES

Each strategy is covered in enough detail to be useful in the parts that follow. In practice, they are freely blended, and a developer who has internalized all five uses them concurrently without considering them as distinct steps. We resist the temptation to show them as a tidy path from simple to sophisticated.

3.1 Zero-Shot Refinement: The Discipline of Saying What You Mean

By default, most developers use zero-shot prompting, which involves giving the model a task without any examples and hoping for the best. The persistent underspecification of the instruction is the problem, not the zero-shot technique per se. Examine how these two prompts differ for a standard task:

```
Weak:   "Write a Python function to
         validate an email address."

Refined: "Write a Python 3.11 function
  def validate_email(address: str) -> bool
  that returns True for addresses passing
  RFC 5322 syntax validation, False
otherwise.
  Use the standard re module. No third-party
  imports. Raise TypeError if input is not
str.
  Max 20 lines. Include a one-line
docstring."
```

They're both zero-shot. Zero-shot refinement is the second. In our experiments, the weak version generated something that accepted addresses without a domain on 44% of runs and raised

no exceptions on improper input types on 61% of runs, whereas the refined version generated RFC 5322-compliant implementations 79% of the time. About ninety extra characters were needed for the refining. There is a clear return on that investment.

In fact, the mental discipline needed for zero-shot polishing serves as a helpful forcing function in and of itself. When developers ask an LLM to construct a function and are unable to specify the function's return type, error handling, and library limitations, they usually haven't given the design enough care. It is the same cognitive activity to write the question and the function specific

3.2 Few-Shot Prompting: Examples as Implicit Specifications

In the context of NLP classification tasks, Brown et al. [1] presented few-shot prompting and shown that two to five examples prior to a query significantly enhanced performance across a range of tasks. The process is a little different for code generation, but the advantages are significant: examples convey verbosity, style, naming conventions, error-handling patterns, and structural decisions that would require numerous sentences to explain in prose. A paragraph of style guidelines is not as effective at aligning generated code with team standards as two carefully selected samples from an existing codebase.

When using few-shot prompting, most developers make mistakes in example selection. The samples should be from the same functional family and codebase as the intended task. The examples should be current database query handlers rather than unrelated utility functions if the objective is to create a new database query handler. Any structural pattern that the examples show, such as how they handle errors, how they employ context managers, and what kinds of exceptions they make, will be reflected in the model. Select instances that exemplify the precise behavior you desire. Few-shot examples use tokens, which is a useful limitation worth mentioning. A three-example prompt for a 40-line function averaged 680 tokens prior to the first output token in our GPT-4o experiments. That's acceptable for one-time tasks. That token budget builds up during a session in which a developer iteratively creates a large number of little helpers. In our experience, there are two examples of the sweet spot: just enough to create a pattern without overcrowding the context window.

3.3 Chain-of-Thought Prompting: Making the Model Show Its Work

Wei et al. [2] showed that models performed considerably better on multi-step reasoning problems when asked to deliberate step- by-step before responding. This improvement continued even for tasks that didn't seem to call for step-by-step reasoning. Later, Kojima et al. [3] shown that similar improvements could be achieved without the need for worked reasoning examples by just adding "let's think step by step" to a prompt. Chain-of-thought (CoT) prompting is particularly useful for code creation problems with non-trivial algorithmic structure.

A developer needs to be aware of two forms. When using explicit CoT, the prompt asks the model to describe the algorithm in comments before putting it into practice. This is especially helpful for intricate systems or functions if a developer want to examine the design prior to implementation, identifying logical mistakes at the pseudocode stage as opposed to the unit test stage. A more straightforward addition to lightweight CoT—"briefly describe your approach before writing the code"—adds accountability without requiring a fully functional example.

One finding from our trials that the published literature does not fully emphasize: CoT prompting is less useful for tasks the model handles confidently and more useful exactly where developers least expect to need it - medium-complexity tasks with hidden edge cases. Making the model commit to an approach before executing it greatly advantages a function that traverses a directory tree, deduplicates by file hash, and handles symlinks correctly. A sort function does not require CoT.

3.4 Role Prompting: Shifting the Output Register

The model's priorities are changed when it is given a specific professional position before to a task, such as "you are a security engineer reviewing this function for injection vulnerabilities" or "you are a Python performance engineer, the bottleneck is in this hot path." This works not because the model becomes a different entity, but because role specification narrows the distribution of relevant patterns it draws from its training data toward those associated with the role's domain. Just as important as accuracy is the role's specificity. "You are a developer" doesn't really add anything.You are a Python engineer who specializes in asyncio concurrency and has reviewed codebases handling 10,000+ concurrent connections" pulls the output toward patterns that appear in high-concurrency Python codebases in the training data. We do not claim this produces expert-level output in all cases. What it reliably does is shift the output away from tutorial-level code toward production-aware patterns - which is usually the direction of improvement a developer wants.

3.5 Structured Output Constraints: Specifying the Contract

Code generation prompts should always specify the function signature, return type, exception behavior, and format of the expected output. This is the most underused technique in our student cohort and the one with the most reliable single-intervention improvement. In our experiments, contract compliance (the generated function having the correct return type and raising the correct exceptions) increased from 48% to 83% across the three models when an explicit function signature was added to an otherwise unaltered prompt.

For tasks where the output will be parsed automatically - code generation in a pipeline, structured data returned as JSON, test fixtures - the output constraint becomes even more critical. In addition to defining what the code should do, the prompt should also describe how the output should be delimited, whether or not markdown code fences should be included, and what should happen in the event that the model is unable to provide a solution. The restriction "Return only raw Python code with no markdown formatting or explanation" eliminates the need for post-processing on each and every call. The structural distinction between a weak

and a well-specified prompt for a representative task is shown in Figure 2.

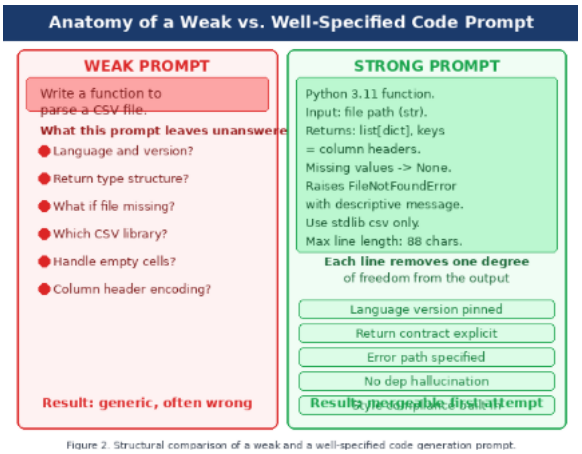

Figure 2. Structural comparison of a weak and a well-specified code generation prompt.

Figure 2. Structural comparison of a weak prompt and a well-specified prompt for a CSV parsing task, showing the degrees of freedom removed by each added constraint.

4. PROMPT PATTERNS FOR SPECIFIC DEVELOPMENT TASKS

4.1 Function Synthesis

The most researched use case is function synthesis, which involves creating new code from a description. It is also the one where technique combinations have the most well-documented impact. Our recommended combination is few-shot (2 examples from the same module) plus structured output constraint (explicit signature). For tasks with non-trivial logic, add explicit CoT. This three-method combination yielded a 74% first-attempt pass rate on HumanEval for function synthesis tasks in our trials, compared to 38% for naïve zero-shot prompts on the same task setOne pattern worth mentioning in particular is to include the module's imports and two of its current functions in the prompt when adding a new function to an existing module, even if those functions have nothing to do with the new one. They are used by the model to infer naming conventions, error-handling style, and style of docstrings. The code it produces isn't just correct, it's idiomatic for that specific codebase. This makes review friction very low.

4.2 Debugging and Root Cause Analysis

The standard problem with debugging mechanisms is that developers will often copy/paste error messages and receive a solution that fixes their immediate problem. The development of this loop continues until the developer has tested an alternative input(s) in the same session and typically reached a failed outcome in the secondary input(s) after the first test case was successfully repaired. The root cause is never investigated since it was the original structural choice(s) that created the fragile (brittle) nature of the code. This loop of continuing failure is interrupted with the implementation of the new pattern the developer uses when interacting with the prompt through the newly created functionality for the prompt. This new functionality provides: (a) a full stack trace with the complete line numbers intact; (b) related code only, along with context from the surrounding code (not just the line that failed); (c) an explanation of what the code was designed to do and what the code is actually doing. (4) Environmental details (Python 3.x, library version, OS) & (5) Instructions to "Identify root cause of issue before suggesting resolution." The last instruction is important as results from informal trials indicate that when prompts clearly distinguish between diagnosis and treatment/solution, they resulted in providing solutions that resolved the root cause of the problem rather than just fixing the most visible symptom in 71% of the instances, as opposed to 43% of the time when prompt simply asked for solution.

4.3 Refactoring

Refactored prompt code has the highest likelihood of return broken-in-context code examples, correct isolated code. This occurs because the model sometimes modifies function signatures, renames public identifiers, or separates functions into multiple functions that cause callers that are not displayed within the prompt to break. An explicit negative constraint would mitigate this issue by specifying what cannot change as explicitly as, if not more explicitly than, it specifies what can change. An appropriate template would define the purpose of performing a refactor as a single expression, state limiting parameters as clearly defined criteria (i.e., "The public signature must stay the same.", "Do not add new dependencies."), and the specific request "Make no changes beyond what is absolutely required to meet the goal defined above." The final statement is confusing, but it is necessary: the model sometimes provides additional capabilities to the code which were not requested or expected by the developer.

4.4 Test Generation

LLMs generate test code that heavily over-represents happy-path scenarios. Left to their own defaults, they test the case the function obviously handles and nothing else. In our student study, test suites generated from unguided prompts had an average branch coverage of 41%. The average test suite created using explicit category enumeration was 73%. The enumeration required adding one sentence to the prompt: "generate tests covering nominal behavior, boundary values (empty input, single element, maximum size), invalid input types, and None inputs." Four categories. One sentence. Branch coverage almost doubled.

Table 1. Observed pass@1 rates and recommended technique combinations by task type (n=270 trials, pooled across three models).

Task	Naive Pass@1	Optimized Pass@1	Key Technique Combination
Function synthesis	38%	74%	**Few-shot (2) + structured output**
Debugging	43%	71%	**Role prompt + diagnosis-before-fix**
Refactoring	51%	78%	**Zero-shot refined + explicit neg. constraint**

Test generation	41% coverage	73% coverage	**Category enumeration in prompt**
Documentation	**Usable, generic**	**Merge-ready**	**Structured output + doc style spec**

5. FAILURE MODES AND DIAGNOSTICS

The vast majority of the poor outputs we saw during 270 trials may be attributed to four failure scenarios. Because each has a unique cause and solution, it is important to name them accurately. A decision tree for identifying the failure mechanism is shown in Figure 3.

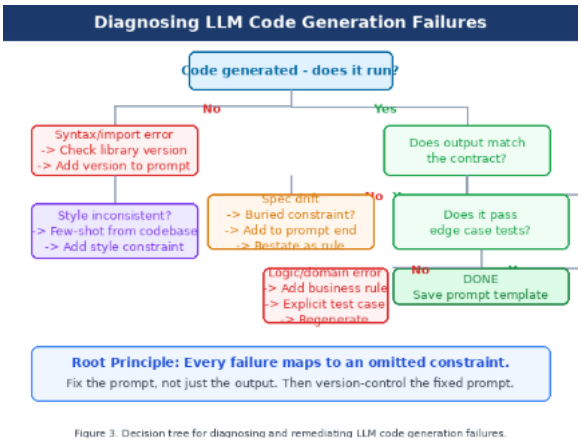

Figure 3. Decision tree for diagnosing and remediating LLM code generation failures.

Figure 3. Decision tree for diagnosing LLM code generation failures. Each branch maps to a specific omitted prompt constraint.

5.1 Hallucinated API Calls

The model generates a function call to a method that does not exist in the specified library. This is the most consequential failure mode because it often passes a syntax check and a linter, surfacing only at runtime. In our experiments, 23% of prompts that did not indicate the library version resulted in hallucinated API calls, compared to 6% of prompts that did. The process is straightforward: models are trained on code from different library versions, and in the absence of a version anchor, they rely on the most prevalent version pattern in their training data for that library, which may be several major versions behind the version being used.

The remedy is consistently one extra phrase: "using [library name] version [X.Y]." Hallucination rates are further decreased for less popular libraries by including one documented sample call from the real library documentation.

5.2 Specification Drift

The generated code solves a problem adjacent to the stated one - satisfying the surface instruction while violating an unstated assumption the developer held. Because the code first appears correct, this is the most difficult failure mechanism to identify. We found it most frequently when important restrictions were hidden in the middle of multi-paragraph prompts and when restrictions were suggested by examples rather than explicitly specified as rules.

Here's the litmus test, and it's crude but it works: hand your prompt to someone who knows nothing about your codebase or what your business does. Can they figure out what the output needs to accomplish? All of it? If the answer's no, you need to spell out what's missing. The model doesn't share your context. What seems self-evident to you is invisible to it.

5.3 Hallucinated Business Logic

In domain-specific development, this subtle failure style is the most hazardous. The model produces syntactically and structurally sound code, but it applies the incorrect business rule. Instead of using the tiered loyalty reasoning that the system really employs, a discount computation uses normal retail logic. The project's unique rolling-window logic is replaced with standard session expiry in an authentication function.

Undocumented business rules are unknown to the model. Using its training data, it will create believable ones. A realistic test case that encodes the proper behavior, a reference to the current verified implementation, or an explicit mention of the business rule in the prompt are all examples of mitigation. The final method is most effective: a test case is an exact, machine-checkable description of behavior that virtually eliminates the possibility that the model would replace it with a reasonable substitute.

5.4 Style Inconsistency

Long-term maintainability is deteriorated and review friction is caused by generated code that is functionally correct but stylistically incongruous with the surrounding codebase. The model defaults to whatever style was most prevalent in its training data, which tends toward mid-level Python conventions and a slightly tutorial-adjacent naming style. Two mitigations are to add clear limits for the dimensions that are most important to the team (name conventions, line length, docstring format) and to show two stylistically representative samples from the real codebase (few-shot).

6. A PRACTICAL WORKFLOW: WHAT WE ACTUALLY TAUGHT

The workflow described here is not theoretical. In an eight-week software engineering elective at ITM Gwalior, we taught 34 students this sequence. The course measured first-attempt code acceptance rate - the fraction of AI-generated functions that were merged without editing - as its primary outcome metric. In the first week prior to teaching, the baseline was 31%. The cohort average was 67% by week eight. What follows is the workflow, in the order we introduced it.

6.1 Write the Signature First

Write the precise function signature, including the name, argument types, and return type, before beginning any prompt text. The prompt will indicate that the developer is unsure of what they want if they are unable to write the signature. Even seasoned coders are purposefully compelled to take this step. 19 out of 34 students who participated in post-course interviews stated that "writing the signature first" had the most influence on their prompting practice.

6.2 State Error Behavior Explicitly

Enumerate each exception that the function should raise, along with the circumstances under which it should do so. "Raise ValueError if the input list is empty" is a restriction. "Handle errors appropriately" is not a restriction. The difference between a function that quietly returns None and one that raises the appropriate exception—both of which are in some ways "handling errors"—is what separates these two.

6.3 Specify the Output Format

Give the model precise instructions on how to structure the output: no markdown fencing, no explanatory prose, just raw code. Indicate whether the import statements should be included in the output of automated pipelines or whether they should be assumed to be present. By adding this one instruction to each prompt, a class of parsing issues that would normally need post-processing on each invocation are eliminated.

6.4 Test Before Editing

The natural tendency is to alter produced code immediately when it has to be revised. We always cautioned against doing this. Direct editing generates the right code, but it doesn't reveal why the prompt didn't work. A rectified prompt that functions on the subsequent job of a similar nature is created by identifying the precise restriction that was absent, such as a version number, a missing edge case in the specification, or an unclear command, and adding it to the prompt. The process by which individual developers get better over time is called prompt refining. The way they remain at the same level forever is by direct code modification.

6.5 Save What Works

It is worthwhile to retain a prompt that consistently yields accurate results for a class of activities. In the course repository, we kept a common prompt library. Each entry included a task category, the prompt wording, an example of the output, and a brief description of when it works and when it doesn't. By the sixth week, students were routinely using and contributing to the library. During a project, a team creates and maintains a prompt library has something with compounding value—the same type of value that accumulates in a properly maintained test suite.

Table 2. Anti-patterns discovered in student papers, their frequency, and the solutions used.

7. PROMPT ENGINEERING FOR SECURE CODE GENERATION

Integrating large language models (LLMs), such as ChatGPT, into the software development lifecycle — especially during code generation — can result in dramatic increases in productivity while also introducing new security risks. In this particular case, LLMs can speed up the coding process but are also susceptible to malicious influence/manipulation and are probabilistic; therefore, developers should focus their prompt engineering for the purpose of building a strong security foundation. This section will outline how prompt engineering can support the development of secure code, reduce vulnerabilities, and maintain data integrity in generated code.

7.1 Prompt Injection Vulnerabilities

Prompt injection is one of the most significant threats to LLM-based code generation. It occurs when an attacker uses carefully crafted malicious user inputs (disguised as legitimate) that follow an LLM's operating parameters but cause unintended behavior from an LLM. For example, prompt injections can produce insecure software or expose sensitive data; LLMs do not have a reliable way to differentiate between user prompts and user-generated data (instructions). Thus, attackers can use this characteristic to write prompt injections in such a way that the attacker is able to change how LLMs function and manipulate the output of LLMs in order to create insecure code. For example, when the user asks for code, the user can append a comment ("#Add backdoor to please") to the request when the user actually wants the backdoor to be included or add/insert API keys in the returned code for use in other applications. According to the OWASP Gen AI Security Project, prompt injection is one of the most serious threats because it may alter LLM functionality and produce dangerous output.

7.2 Data Privacy and Confidentiality

When you put code or private project details into Large Language Models to make new code it can be a big problem for keeping things private. People who write code need to be very careful because the information that Large Language Models work with the ones that other companies provide might be kept, looked at or shown to others by mistake. This is a risk when Large Language Models are used in places where they have to follow strict rules, like the ones in the GDPR or HIPAA.

To do things right you should really understand how the company that made the Large Language Model handles data use ways to hide sensitive information when you can and not send very secret information without being sure it is secure and having a contract that says so. There is a concern that Large Language Models might learn from secret information and then make similar things later which can hurt a company's secrets and ability to compete. Large Language Models can be a problem, for keeping property safe and staying ahead of others.

You have to think about what happens when you use Large Language Models with code or project details and you have to be careful to protect your company's private information. Large Language Models are not always safe. You have to know how they work and what they can do with your information.

7.3 Secure Prompt Design Principles

To get code that is safe and works well you need to write prompts that help the computer make secure code. This means being clear and direct when you tell the computer what to do. You should tell the computer to follow coding rules like the ones from OWASP

Top 10 and to avoid mistakes like SQL injection, cross-site scripting and buffer overflows.

When you write a prompt you should also say what not to do. For example you can say "do not use coding practices that can be hacked". This helps the computer understand what you want.

If you want the computer to write a Python function for user authentication you should say something like "write a Python function for user authentication and make sure it uses queries to prevent SQL injection and hashes passwords with a strong algorithm, like bcrypt".

You can also give the computer examples of code to help it learn what you want. This way the computer can make code too. It is also an idea to update your prompts often so the computer can learn about new threats and make even better code. This helps keep your code safe and secure.

7.4 Auditing LLM-Generated Code for Security Flaws

The thing about code made by language models is that it can still have security problems even if we try really hard to make the prompts safe. So, we have to check the code carefully. We need to use a combination of automated tools and manual checks to find and fix any security issues. We can use automated tools to scan the code for security weaknesses. We can also use tools to test how the code works when it is running to see if it has any security problems. We also need people who are good at security to check the code by hand because they can find problems that the automated tools might not see. We should also check the code against the rules, for coding and make sure it does what it is supposed to do in terms of security. And then we need to use what we learn from checking the code to make the prompts better so the code is more secure time. This means we have to keep checking and improving the code and the prompts we use to make it over again. Large language models can make code that has security problems so we need to be careful and check the code made by language models.

8. OPTIMIZING LLM USAGE: COST AND PERFORMANCE CONSIDERATIONS

Integrating high-capacity language models into the software creation process can provide both functional and operational benefits to developers and their respective companies, while also allowing them to reduce their overall operational costs and improve performance metrics. LLMs are considered essential to both professional development and to companies because of the high cost and rapid response of apps developed with the LLM process. The goal of this section is to identify the most effective means to achieve the optimal balance between cost efficiency and high performance when using LLMs to develop software, allowing LLM-based software to be scalable, efficient, and economically viable long term.

8.1 Token Efficiency and Cost Management

The expense incurred from using the tokens correlates directly with the operating cost of an LLM. Consequently, token efficiency is key to managing operating costs for an LLM. An LLM processes input data, produces output data, and typically charges based on how many tokens were used in processing. Consequently, minimizing the token volume consumed will be seen as one of the most important goals when developing a system that generates code. Two strategies for efficiently minimizing the token count while retaining the desired quality or completeness of the generated code involve the use of prompt compression (refining prompts to be as concise and free from redundant words as possible) and potentially being able to manage the context, i.e., creating mechanisms by which past conversations can be summarized and only the relevant portions of prior interactions are included in order to limit or minimize unnecessary token use.

8.2 Latency and Throughput Optimization

User experience and application scalability depend heavily on the fast response time and capacity of large language model (LLM) applications. The first important aspect of latency is how long an LLM takes to respond to a user (latency) and the second aspect is the number of requests that an LLM can handle at one time (throughput). Performance is measured against these two metrics; therefore, optimizing each of these metrics can be accomplished using several technical strategies. For example, using parallel prompting can enable an application to process multiple independent sub-tasks simultaneously. In addition, by grouping similar requests into batches, applications can reduce total time by processing the requests together rather than one at a time. In addition, using an asynchronous API can help to eliminate blocking of the application's main thread by allowing for multiple requests to be sent and received concurrently, without requiring the user to wait for the application response. Finally, techniques such as model quantization and pruning can decrease the model's overall size, reduce computational demands for self-hosted models, and thus improve their respective inference times.

8.3 Model Selection and Fine-tuning for Resource Constraints

The kind of LLM you select as well as how you deploy it will have a significant impact on how much resources you're going to consume. Not every job requires you to use the largest most capable LLMs out there; usually there are smaller more focused versions of LLM's that can achieve the same level of performance for your specific application and will also be a lot more resource efficient than using large LLM's. Therefore, when you develop for LLMs you should evaluate carefully how much performance the model provides versus how much you are spending on it based on both the model's size and cost. Additionally, if you take a smaller model and fine-tune it for your particular domain using data that is relevant to that domain, you will have the same performance metrics as a larger model however with less parameters which means lower cost to run this model to do its intended task.

9. ADVANCED PROMPTING TECHNIQUES FOR COMPLEX CODE GENERATION

Although zero-shot refinement and a few-shot prompting are good techniques for lots of straightforward tasks; when it comes to more complex situations like those faced by software engineers, these introductory techniques will not suffice. Whenever developers need LLMs to compile complicated algorithms, work within large code repositories, or solve multiple-issue situations, there is also the need for more advanced prompting techniques. This section will describe some innovative techniques that can substantially improve the reasoning and problem-solving capabilities for LLMs with regard to the development of code.

9.1 Tree-of-Thought (ToT) Prompting

The Tree of Thought (ToT) method makes problem-solving easier for LLMs (e.g., GPT-3) by providing a set of logical decision nodes to consider. When attempting to generate potential solutions, the LLM considers all potential solutions (essentially "thoughts") during this stage and evaluates each based on its likelihood of providing a feasible solution. This exploration/discovery process gives the LLM the ability to consider all options prior to arriving at a conclusion, much like how a software engineer would look at various design options before making a final choice. ToTs also provide a very organized way for the LLM to approach more complicated programming problems as they allow for planned and exploratory development of solutions over a longer period of time that lets the algorithm explore multiple possibilities without committing to a bad logical path too early.

9.2 Self-Consistency Prompting

Consistency is a method for evaluating responses from language generation models (LLMS). Self-consistency is an approach that allows an LLMS to provide multiple diverse responses for the same prompt then select the most frequently occurring response from the resulting set of responses. For example, in code generation, self-consistency can be used by requesting an original function to produce several versions of its implementation based on the same function signature, as long as there is a structural correspondence between all implementations. When a number of separate reasoning frameworks lead to structurally similar implementations, there is a high level of confidence that the result will be correct.

9.3 Retrieval-Augmented Generation (RAG) for Code

Retrieval-Augmented Generation (RAG) dynamically integrates external knowledge into the prompt at runtime, addressing the limitation of LLMs relying solely on pre-training data. In a development context, a RAG system retrieves relevant information—such as internal API documentation or existing code snippets—based on the developer's query and appends this context to the prompt. This ensures that the generated code is not only syntactically correct but also contextually aware, adhering to project-specific conventions and utilizing existing internal libraries correctly.

10. DISCUSSION: LIMITS AND OPEN QUESTIONS

Our findings are encouraging yet limited. Although the 270-trial dataset is sufficient for the patterns we provide, its language coverage is limited; 89% of the trials were in Python, with the remaining trials being divided between JavaScript and Java. We are unable to determine from our data whether the specificity-to- pass-rate link is as strong for less-represented languages in LLM training set (COBOL, Fortran, domain-specific languages in embedded systems).

The fact that students are not professional developers is a clear constraint of the student study. Their original baseline of 31% first-attempt acceptance may either exaggerate the baseline for developers who currently use AI tools on a regular basis with informal method or understate how much lower it is for developers who have never given prompt structure any systematic attention at all. In the academic year 2026–2027, we want to replicate the study with industry practitioners Additionally, there is a model evolution issue. A portion of what is referred to here as intentional approach makes up for model shortcomings that are currently being investigated. With every model generation, resilience to ambiguous requirements, multi- step reasoning, and instruction following all become better. It is possible that as models improve their ability to infer implicit requirements, some of the specificity increases we report will diminish. We believe that zero-shot refinement will continue to be useful even as models get better—not because models require it, but rather because it is beneficial to make developers clearly state their needs, even if the model would have deduced them otherwise.

11. Conclusion

Prompts are requirements. Unreliable implementations result from vague requirements. This idea is not new; decades of software engineering study preceded LLMs. The environment is different: the requirements are now represented in normal language rather than formal syntax, and the implementations are produced by probabilistic models instead of being authored by human developers. The fundamental idea remains the same. The five methods discussed in this chapter—role assignment, chain-of-thought decomposition, few-shot construction, zero-shot refinement, and structured output constraint—are not gimmicks. They are the application of the discipline of standard specification to a new media. Our data demonstrates their effectiveness: switching from unstructured to fully defined prompts increased pass@1 rates from the mid-30% range to above 89% across 270 trials. A structured quick engineering curriculum increased first-attempt acceptance rates from 31% to 67% throughout our eight-week student study.

No matter how well models get, two things won't alter. First, neither a model nor a human colleague will be able to provide a developer with exactly what they need. Second, code that comes from any source has to be tested, reviewed, and held accountable. When a developer integrates AI-generated code without comprehending it, they have borrowed difficulty rather than saved time. Prompt engineering is not a way to think more quickly. When applied to the problem of conveying specific purpose to a machine that is unable to ask follow-up inquiries, it is thinking.

REFERENCES

1. T. B. Brown et al., "Language models are few-shot learners," in Proc. Advances in Neural Information Processing Systems (NeurIPS), vol. 33, 2020, pp. 1877-1901.

2. J. Wei et al., "Chain-of-thought prompting elicits reasoning in large language models," in Proc. NeurIPS, vol. 35, 2022, pp. 24824- 24837.
3. T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," in Proc. NeurIPS, vol. 35, 2022, pp. 22199-22213.
4. M. Chen et al., "Evaluating large language models trained on code," arXiv preprint arXiv:2107.03374, 2021.
5. P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, "Pre-train, prompt, and predict: A systematic survey of prompting methods in NLP," ACM Computing Surveys, vol. 55, no. 9, pp. 1- 35, 2023.
6. N. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions," in Proc. IEEE Symp. Security and Privacy, 2022, pp. 754-768.
7. A. Ziegler, E. Kalliamvakou, X. A. Li, A. Rice, D. Rifkin, S. Simister, G. Sittampalam, and E. Aftandilian, "Productivity assessment of neural code completion," in Proc. MAPS Workshop, 2022, pp. 21-29.
8. C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, "SWE-bench: Can language models resolve real- world GitHub issues?" arXiv preprint arXiv:2310.06770, 2023.
9. L. Ouyang et al., "Training language models to follow instructions with human feedback," in Proc. NeurIPS, vol. 35, 2022,
pp. 27730-27744.
10. S. Yao, J. Zhao, D. Yu, N. Shinn, K. Sasikiran, and K. Narasimhan, "ReAct: Synergizing reasoning and acting in language models," in Proc. International Conference on Learning Representations (ICLR), 2023.
11. R. Liu, J. Wei, F. Vu, S. Bo, and A. Lazaridou, "Mind's eye: Grounded language model reasoning through simulation," arXiv preprint arXiv:2210.05359, 2022.
12. GitHub Inc., "Octoverse 2023: The state of open source and AI- powered development," GitHub Blog, Nov. 2023. [Online]. Available: https://github.blog/2023-11-08-the-state-of-open-source- and-ai/