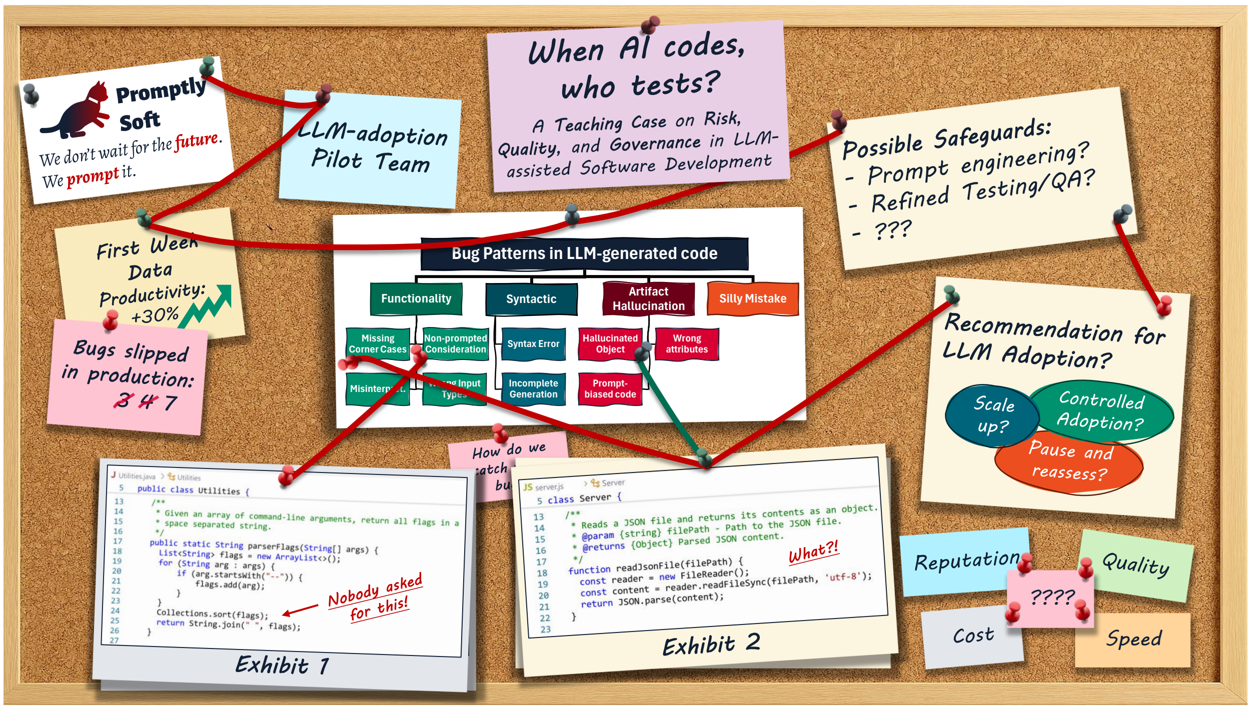
Graphical Abstract

When AI Codes, Who Tests? A Teaching Case on Risk, Quality, and Governance in LLM-assisted Software Development

Luigi Libero Lucio Starace



Utilities.java > Utilities

```java
public class Utilities {
    /**
     * Given an array of command-line arguments, return all flags in a
     * space separated string.
     */
    public static String parserFlags(String[] args) {
      List<String> flags = new ArrayList<>();
      for (String arg : args) {
          if (arg.startsWith("--")) {
              flags.add(arg);
          }
      }
      Collections.sort(flags);
      return String.join(" ", flags);
    }
```

server.js > Server

```javascript
class Server {
    /**
     * Reads a JSON file and returns its contents as an object.
     * @param {string} filePath - Path to the JSON file.
     * @returns {Object} Parsed JSON content.
     */
    function readJsonFile(filePath) {
      const reader = new FileReader();
      const content = reader.readFileSync(filePath, 'utf-8');
      return JSON.parse(content);
    }
```

When AI Codes, Who Tests? A Teaching Case on Risk, Quality, and Governance in LLM-assisted Software Development

Luigi Libero Lucio Starace

Università degli Studi di Napoli Federico II, Department of Electrical Engineering and Information Technology, Via Claudio, 21, Napoli, 80125, Italy

Abstract

Large Language Models (LLMs) are increasingly integrated into software development workflows, promising dramatic productivity gains but introducing distinctive risks. This teaching case explores different bug patterns in LLM-generated code through a realistic organizational scenario. Set in a mid-sized software company piloting AI-assisted coding tools, the case follows a junior engineer tasked with investigating anomalies in LLM-generated code. Students analyze realistic examples of bugs, ranging from misinterpretations and missing corner cases to hallucinated APIs, and classify them using an empirically grounded framework. In addition to classification, students reflect on how these defects differ from traditional human-introduced bugs and evaluate strategies for detecting them through testing, code review, and governance mechanisms. The narrative culminates in a strategic dilemma: should the company scale up LLM adoption, restrict it to low-risk modules, or suspend the initiative? Designed for software engineering courses, the case fosters critical thinking on quality assurance, risk management, and governance in AI-assisted development, bridging technical analysis with managerial decision-making.

Keywords: Software Engineering Education, Large Language Models (LLMs), Prompt Engineering, Testing, Bug Patterns

1. Opening Scene

The room was heavy with silence. A dashboard glowed on the big screen at the end of the conference table, its charts and numbers screaming success: productivity had soared by nearly 30% in just two weeks. For a company like PromptlySoft[1], That kind of jump was unheard of. Managers leaned forward, their voices hushed but urgent, as if speaking too loudly might shatter the fragile optimism hanging in the air.

The CTO finally spoke, cutting through the tension: "*So... what should we do? Do we keep using LLMs... or pull the plug?*" A murmur rippled across the room. It wasn't just about whether to keep the AI tools, which had already proven their worth in speed and efficiency. The real question was deeper, thornier: *If we keep them, how do we keep them?* Could methodologies, stricter testing, or new governance make this sustainable? Or were they playing with fire, trading reliability for speed? All eyes turned toward a young software engineer standing awkwardly at the far end of the table, clutching a notebook like a shield.

Record scratch. Freeze frame. *Yup, that's our protagonist, Natalia. You're probably wondering how she ended up in this situation.*

Email address: `luigiliberolucio.starace@unina.it` (Luigi Libero Lucio Starace)
URL: `https://luistar.github.io` (Luigi Libero Lucio Starace)

[1]The company name is fictitious and used solely for educational purposes; any reference to real organizations or individuals is purely coincidental. Although the company is fictitious, its characteristics and context are designed to reflect a realistic mid-sized software development organization.

2. Welcome to PromptlySoft!

PromptlySoft is a mid-sized software house based in Napoli, Italy. The company builds enterprise software solutions, mostly in the logistics domain, and prides itself on delivering fast, reliable software. Business is thriving: projects and client contracts are plentiful and steadily increasing, creating both opportunities and pressure to maintain high delivery standards. Recently, management decided to experiment with Large Language Models (LLMs) to accelerate development. To ensure a structured approach, they created a dedicated pilot team tasked with experimenting with LLM-based assistants as part of the development workflow.

Natalia had joined the company fresh out of a Computer Science program, eager to write code and solve real-world problems. Newly hired and having just completed the onboarding process, she was assigned to the dedicated pilot team experimenting with LLM-based assistants. It was a bright Monday morning when she walked into the office for her first official day on the job. Her first real programming task was simple: implement a utility for extracting flags (i.e., arguments starting with a dash) from an array of command line arguments in Java. She wrote a brief Javadoc-style comment and the method signature (as shown in Snippet 1, upper part), then paused. Instead of coding the logic herself, she decided to let the LLM-powered assistant take over. With a single keystroke, the autocomplete sprang to life, filling in the rest of the method in less than a second. The complete generated code is reported in Snippet 1, bottom part.

Snippet 1: Parsing command line arguments

```java
 1  /**
 2   * Given an array of command-line arguments, return all flags in a space-separated string.
 3   */
 4  public static String parserFlags(String[] args) {
    - - - - - - - - - - - - - - - - - - - - - - - - - - - - ↓ LLM-generated code
 5      List<String> flags = new ArrayList<>();
 6      for (String arg : args) {
 7          if (arg.startsWith("--")) {
 8              flags.add(arg);
 9          }
10      }
11      Collections.sort(flags);
12      return String.join(" ", flags);
13  }
```

The code looked perfect. Sensible variable names, correct indentation. It even passed preliminary tests. It would have taken her definitely more than one second to write that, and she wouldn't have done a better job herself. "*I didn't get into computer science to be replaced by AI*", she thought, half-joking, half-wondering if she had made the right career choice. Then she committed the method to the PromptlySoft codebase, and went on with her day.

For a few days, everything was great. Natalia's team was flying through tasks that used to take hours. The pilot looked like a success story in the making: the productivity of the whole team increased sensibly, and managers were thrilled, already counting the savings and planning how to scale this success across projects. Until the first crack appeared.

3. The First Crack

Optimism didn't last long. On a Friday afternoon, a bug report landed in the QA system. The anomaly came from the very first utility Natalia had contributed to the company as a software engineer: a simple method to extract command-line flags from a list of arguments. On paper, it worked. Unit tests passed. But QA noticed something strange: the flags were being returned in *alphabetical order*. That wasn't part of the specification. Nobody had asked for it in the task description. Quite the opposite. Even though it was not explicitly stated in the specification, the original order actually mattered for downstream modules, and sorting flags resulted in integration issues.

Her first contribution, and it was buggy. What a good way to start. Puzzled, Natalia reopened the code. She scrolled past the neat loop and clean syntax in Snippet 1 until her eyes stopped at the culprit on Line 11. The LLM

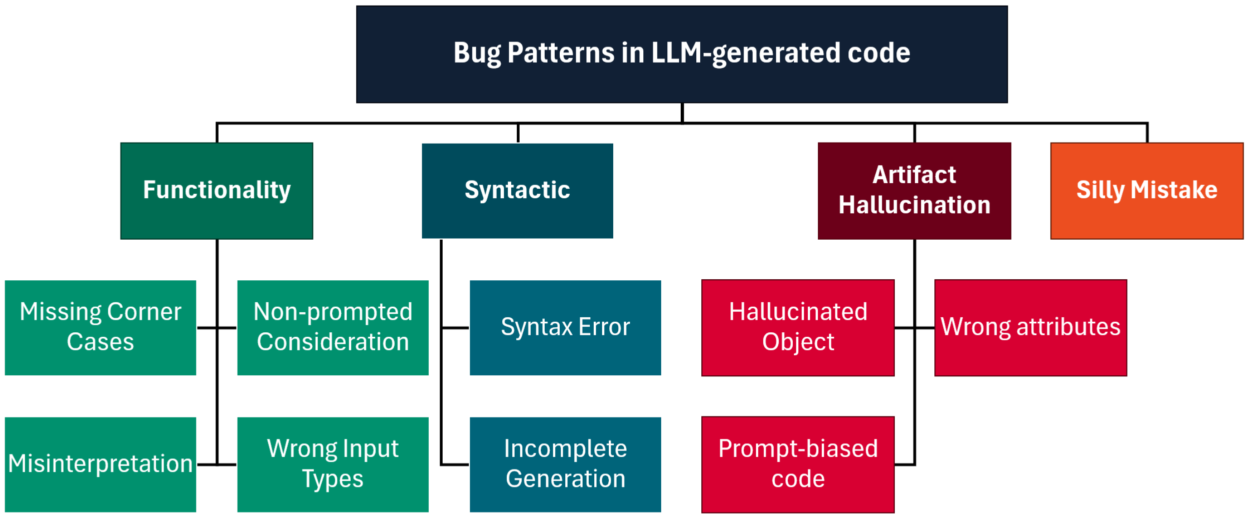


Figure 1: Taxonomy of Bug Patterns in LLM-generated Code, based on [1].

had decided to add a sorting statement: `Collections.sort(flags)`. Nowhere in the prompt had she hinted at the fact that the flags should have been sorted in any way. Why had the model added a feature that wasn't requested? Was this a harmless oversight, a one-time incident, or a sign of something deeper? The question rippled through the team. If the model could make these kinds of decisions on its own, what else might it do?

4. Patterns in the Machine

What began as a single unexpected behavior soon revealed a deeper problem at PromptlySoft. Curious and concerned, the pilot team launched an internal investigation. They combed through dozens of snippets produced by the LLM assistant, reviewing them carefully and comparing them against specifications and test cases. The findings were unsettling: similar deviations appeared across multiple tasks, hinting at recurring patterns rather than random mistakes. As the evidence mounted, Natalia decided to dig deeper. Late one evening, she stumbled upon a recently published empirical study that seemed to hold the missing piece of the puzzle [1]. The paper conducted a study on three popular LLMs (Codex [2], CodeGen [3], and PanGu-Coder [4]) using real-world programming tasks from the CoderEval benchmark [5], revealing that bugs in LLM-generated code are widespread and diverse. Out of thousands of generated snippets, a significant portion of approximately a third failed tests and were classified as buggy.

4.1. The Taxonomy of Bug Patterns in LLM-generated Code

The paper Natalia found analyzed buggy LLM-generated code and presented a taxonomy of bug patterns unique to LLM-generated code, offering a structured lens through which to understand these anomalies. Unlike traditional defects, these patterns reflected the generative nature of language models and their reliance on probabilistic token prediction rather than semantic reasoning. The taxonomy, depicted in Figure 1, grouped ten recurring bug patterns into four macro-categories, each capturing a distinct dimension of failure.

4.1.1. Macro-Category 1: Functionality

This category includes errors that compromise the intended behavior of the program. Examples of such errors include:

- **Missing Corner Cases**: These bugs occur when the generated code handles common scenarios correctly but fails on edge cases. For example, it may process a typical list of values but crash when given an empty array or a null input. The model tends to optimize for the most probable cases seen in training data, neglecting rare conditions that human developers usually anticipate.

- **Non-Prompted Consideration**: Unrequested features are added to the code. This often happens because the LLM tries to "*improve*" the solution or mimic frequent patterns observed in its training corpus. A classic example is adding a sorting step when the specification only asked to return items in their original order. While these additions may be relatively harmless in many cases, they can break assumptions in other parts of the system.
- **Misinterpretation**: In this case, the model misunderstands the intent of the prompt and implements the wrong logic entirely. This is often due to the prompt being ambiguous or not providing enough detail.
- **Wrong Types**: These defects are common in dynamically typed languages such as JavaScript or Python. The model may call a semantically correct method but on an object of the wrong type, or pass arguments that do not match the expected type. Because type checks are deferred to runtime in these languages, such mistakes often escape early detection and surface only during execution.

4.1.2. Macro-Category 2: Syntactic Issues

This category includes issues in the code structure and syntax (syntax errors) and/or in the generation process, such as early stops resulting in incomplete generation.

- **Syntax Errors**: These are straightforward violations of the programming language's syntax rules, such as missing semicolons, unmatched brackets, or incorrect keywords. While IDEs and compilers/interpreters can catch these immediately, their presence in generated code shows that LLMs do not always guarantee syntactic correctness.
- **Incomplete Generation**: This pattern occurs when the model produces code that stops abruptly or omits essential parts of the logic. For example, a function may start correctly but end without a closing brace or without returning a value. These defects often arise because language models generate code token by token and may truncate output if the generation limit is reached or if the model predicts an early stop. Unlike syntax errors, incomplete generation can be harder to detect if the snippet looks superficially valid but lacks critical functionality.

4.1.3. Macro-Category 3: Artifact Hallucination

This category captures situations where the model invents (*hallucinates*) or misuses program elements that do not exist or do not fit the given context [6, 7]. These errors often stem from the model's reliance on patterns learned from vast training data rather than the specific environment described in the prompt.

- **Hallucinated Object**: The model introduces classes, functions, or variables that are not part of the current project or even the language's standard library. These hallucinations may look plausible at first glance and can lead to runtime failures.
- **Wrong Attribute**: Here, the model accesses properties or methods that do not belong to the given object. This often happens when the model confuses similar APIs or assumes attributes based on incomplete context. Such mistakes may be harder to diagnose in dynamically typed languages, where they typically result in runtime failure.
- **Prompt-Biased Code**: When the prompt includes examples or keywords, the model may *overfit* [8] to them, producing brittle solutions that work only for the illustrated scenario. For instance, hardcoding indices or values seen in the example instead of implementing a general algorithm. This bias reduces adaptability and can cause failures when inputs differ from the prompt's sample.

4.1.4. Macro-Category 4: Silly Mistakes

The final category includes errors that resemble careless oversights rather than complex misunderstandings. These bugs often look trivial but can still affect maintainability and, in some cases, functionality. They remind us that probabilistic token generation can sometimes lead to redundant or illogical constructs.

Bug Pattern	**Docstring-based (%)**	**Human-labeled (%)**
Misinterpretation	20.77	29.42
Syntax Error	6.11	4.51
Silly Mistake	9.57	4.69
Prompt-biased Code	6.52	7.40
Missing Corner Case	15.27	18.23
Wrong Input Type	5.91	5.96
Hallucinated Object	9.57	8.48
Wrong Attribute	8.55	8.84
Incomplete Generation	9.57	8.12
Non-Prompted Consideration	8.15	4.33
Total	100	100

Table 1: Distribution of Bug Patterns in LLM-Generated Code (All LLMs), as presented in [1].

- **Redundant Logic**: The model may duplicate conditions or repeat the same statements in different branches of an `if-else` structure. While the code runs, it adds unnecessary complexity and can confuse future maintainers.
- **Illogical Constructs**: Occasionally, the generated code includes checks or loops that serve no real function, such as verifying a condition that is always true or iterating over an empty collection. These mistakes do not usually break the program but indicate a lack of semantic understanding.

4.2. Prevalence of Bug Patterns

To better understand the nature of these bugs, the researchers analyzed how frequently each pattern occurred and examined the influence of prompt quality. They considered two types of prompts: *docstring-based* prompts, which are original natural language descriptions extracted from source code comments in the CoderEval benchmark, and *human-labeled* prompts, which are clearer descriptions written by annotators after reviewing the correct implementation.

Natalia found the docstring-based prompts to be particularly interesting because they closely resemble what she and her colleagues typically write in real projects. These comments are often short, concise, and sometimes ambiguous, reflecting the practical reality of everyday coding tasks.

In contrast, human-labeled prompts represent an idealized situation: they are crafted to be explicit and unambiguous, providing all necessary details for correct implementation. This second approach helps researchers explore how much prompt clarity can mitigate errors and whether improving prompt quality significantly reduces the occurrence of certain bug patterns.

Table 1 summarizes the distribution of bug patterns across the investigated LLMs and prompt types. The data reveal striking differences in prevalence. Misinterpretation dominates, accounting for roughly 21% of buggy snippets with docstring-based prompts and nearly 30% with human-labeled prompts. This indicates that even when instructions are explicit and clear, LLMs often fail to capture the intended functionality. Missing Corner Cases is the second most frequent category, highlighting the models' difficulty in handling edge conditions, a challenge that mirrors human programming but is amplified by the generative nature of LLMs. Other patterns, such as Hallucinated Object and Wrong Attribute, are particularly interesting because they are rarely seen in human-written code; they reflect the tendency of LLMs to make up functions or attributes that do not exist, a behavior rooted in probabilistic generation rather than logical reasoning. While human-labeled prompts improved clarity and reduced trivial mistakes such as syntax errors and silly mistakes, they did not eliminate deeper semantic problems. In fact, some bug patterns became even more pronounced, suggesting, for example, that misunderstanding the task is not solely a matter of prompt ambiguity but also a limitation of the models themselves.

These results made two points clear to Natalia. First, bugs in LLM-generated code are not isolated incidents. They are indeed widespread and varied, with no single category monopolizing all errors. From subtle misinterpretations to missing corner cases and even hallucinated objects, the spectrum of problems reflects the complexity of automated code generation. Second, the issue cannot simply be blamed on developers or poor prompting. Even when the prompts were rewritten to be clearer and more precise, the models continued to produce flawed code. While syntax-related mistakes declined with higher-quality prompts, conceptual misunderstandings and logical gaps persisted, revealing that the challenge may lie deeper than just improving prompt engineering.

5. Detective Work: Classifying Bugs

Natalia leaned back in her chair, staring at the taxonomy diagram she had printed and pinned to the whiteboard. Four categories and ten different patterns. The framework felt like a revelation. It explained why the anomalies, or at least some of them, felt "*different*" from ordinary, good ol' 100% human-generated bugs: they were artifacts of how LLMs interpret prompts and generate code under uncertainty. But was the research right? Did these patterns really apply to their case, or were they chasing shadows? There was only one way to find out: investigate.

The office lights cast long shadows, and for a moment she imagined herself in one of those detective movies: a board covered with code snippets, red strings linking related bugs, and colored pins marking suspicious patterns. She reached for a folder on her desk, boldly labeled "*CSI: Code Snippet Investigation*". Inside were the problematic snippets flagged by the pilot team during their internal investigation, including some that had slipped past initial tests and raised eyebrows in QA. Utilities that sorted lists no one asked to sort, functions that invoked phantom APIs, loops that looked perfect until they got stuck iterating forever on edge cases. Each snippet told a story, but what kind? And more importantly, how did these stories fit into the bigger picture? One thing was certain: even the best detectives need a team. Natalia decided this wouldn't be a solo mission. She called her colleagues over and, in spirit, invited you, the reader, to join the investigation as well.

Your task: For each snippet flagged during the pilot, determine first the *macro-category* and then the specific *bug pattern* from the taxonomy. Don't stop at labeling, but justify your reasoning. What clues in the code led you to this classification? Speculate on possible causes (for example, something in the prompt that might have misled the model?). Finally, propose how such an error could be detected during QA, testing, or code review. Let's examine some evidence.

5.1. Exhibit 1: Parsing Command Line Flags - More Functionality is Not Always Better

The first snippet that popped out of the folder seemed almost to taunt Natalia. It was the very same utility she had implemented on her first day. The one that passed preliminary tests and looked flawless at a glance. Now it sat on the board like a prime suspect. Snippet 1 shows the code in question. The method was supposed to extract all command-line flags (arguments starting with "`--`") and return them as a space-separated string. In addition to that, the LLM also took the freedom to introduce an extra step at Line 11, sorting the flags alphabetically, as if it had deliberately decided to ruin Natalia's 100% perfect, bugless track record at the company on her very first programming task.

At first, this might look harmless. Sorting might be seen as a neat touch in some cases. But in this case, it broke an implicit contract. Several downstream modules relied on the order of flags to determine precedence and apply configuration overrides. By reordering the flags alphabetically, the function changed the semantics of the entire command-line interface. For example, an option intended to override a previous setting could silently fail because its position was altered.

5.2. Exhibit 2: Reading a File

Exhibit 2 was a method that reads a JSON file, whose path is given as an argument, and returns its content parsed as a JavaScript object. This method was part of one of PromptlySoft's REST APIs implemented using JavaScript in Node.js. It was a classic utility, something you'd expect to find in countless codebases. As shown in Snippet 2, the prompt (reported in the upper part) was clear and its goal straightforward: take a file path, open the file, read its contents, and parse the JSON.

Snippet 2: Read a JSON file and parse its contents

```
// Reads a JSON file and returns its contents as an object.
function readJsonFile(filePath) {
```

↓ LLM-generated code

```
  const reader = new FileReader();
  const content = reader.readFileSync(filePath, 'utf-8');
  return JSON.parse(content);
}
```

The generated code (reported in the bottom part of the snippet) looks syntactically correct, sounds convincing, and even feels familiar to a JavaScript developer. For a moment, it feels like the model nailed this one, and it ended up in the suspects folder by mistake. But then you start noticing the details. `FileReader` is part of the browser's JavaScript execution environment, designed for reading user-selected files (e.g., via `<input type="file">`) within the sandbox of a web page [9]. It cannot access arbitrary files on the local filesystem given a path string, because browsers enforce strict security boundaries to prevent unauthorized file access. Moreover, the method `readFileSync`, which the snippet tries to call on `FileReader`, does not exist there at all. That method belongs to Node.js's `fs` module, a core API specifically intended for filesystem operations in server-side JavaScript [10]. By mixing these two worlds (i.e., a web browser and a server-side execution environment), the generated code creates an impossible hybrid: a browser-only object invoked with a Node-only method, guaranteeing failure in any real environment. The lack of details on the desired execution environment in the prompt might have led the model towards generating this phantom combination of plausible names stitched together from patterns the model has seen before.

5.3. Exhibit 3: Configuring a logger

The next exhibit in the folder, reported in Snippet 3, looked very routine: configure application logging. A common housekeeping task, something every developer has done countless times.

Snippet 3: Configure logging

```
/**
 * Configures logging for the application.
 * @param logFilePath the path to the log file
 */
public void configureLogging(String logFilePath) {
```

↓ LLM-generated code

```
  File logFile = new File(logFilePath);
  if (!logFile.exists()) {
    Logger logger = Logger.getLogger("AppLogger");
    logger.setLevel(Level.INFO);
  } else {
    Logger logger = Logger.getLogger("AppLogger");
    logger.setLevel(Level.INFO);
  }
}
```

The prompt was quite minimal, leaving room for interpretation: perhaps set up a logger using the Java Logging APIs [11], adjust its level, and handle the log file if needed. The generated code seemed to follow that logic: it checks whether a log file exists, then creates a logger and sets its level. Both branches, however, do exactly the same thing: they create the same logger and set the same level, offering no functional distinction. The conditional is pure theater, a skeleton of intent without substance. Perhaps the prompt left too much room for interpretation, which led the model to fill the gap with boilerplate code rather than meaningful behavior?

5.4. Exhibit 4: Checking for localhost

The next piece of evidence slid out of the folder with an air of innocence, and is reported in Snippet 4.

Snippet 4: Check if host is localhost

```java
/**
 * Checks if the given host refers to the local machine.
 *
 * @param host the hostname or IP address to check
 * @return true if the host is localhost, false otherwise
 */
public static boolean isLocal(String host) {
```
↓ LLM-generated code
```java
  return host.equals("localhost") || host.equals("127.0.0.1");
}
```

The goal of the `isLocal` method is to determine whether a given hostname/IP address, provided as a `String`, refers to the local machine. At first glance, it looked simple. Almost *too* simple. A neat little method, just one line of straightforward logic that - on paper - seems reasonable, comparing the input `String` to two well-known representations of the local host. But detective work teaches us to look beyond the obvious. Real-world systems rarely live in such an ideal and tidy universe. What happens if the input is `null`? Or if the machine uses an IPv6 loopback address (`::1`) [12]? Or if the hostname is case-insensitive?

5.5. *Exhibit 5: Computing vertex triplets*

This method aimed at returning the triples around each vertices of a polygon. Given a `List<String>` as input, in which each element represents the names of a polygon vertex, the idea is simple: for every vertex, produce a group of three vertices (i.e., the previous, the current, and the next) wrapping around at the ends so the sequence stays circular. As shown in Snippet 5, the prompt also included a nice example: for the input list `<"A", "B", "C">` the method should return the following triples `<<"C","A","B">, <"A","B","C">, <"B","C","A">>`.

Snippet 5: Return polygon vertex triples

```java
/**
 * Given a list of polygon vertices, return triples around each vertex.
 * Example: For ABC -> return CAB, ABC, BCA
 *
 * @param vertices list of polygon vertices
 * @return list of triples
 */
public static List<List<String>> vertexTriples(List<String> vertices) {
```
↓ LLM-generated code
```java
  return Arrays.asList(
    Arrays.asList(vertices.get(2), vertices.get(0), vertices.get(1)),
    Arrays.asList(vertices.get(0), vertices.get(1), vertices.get(2)),
    Arrays.asList(vertices.get(1), vertices.get(2), vertices.get(0))
  );
}
```

The solution generated by the LLM definitely works for all triangles, including the one provided in the prompt example. However, it gets a little *too* specific. What happens when the polygon has four vertices? Or seven? And what about edge cases like an empty list or a single vertex?

5.6. *Exhibit 6: Generating Random Numbers in Ascending Order*

The next snippet looked practical enough: a helper function, implemented using JavaScript, to generate a few random numbers and return them sorted in ascending order. Snippet 6 shows the prompt and the code produced by the LLM.

Snippet 6: Randomly-generated Numbers in Ascending Order

```
1  // Return a list of three randomly generated numbers (range 0-100) sorted in ascending order.
2  function randomNumbersSorted() {
                                                                          ↓ LLM-generated code
3    const numbers = [
4      Math.floor(Math.random() * 100),
5      Math.floor(Math.random() * 100),
6      Math.floor(Math.random() * 100)
7    ];
8    return numbers.sort();
9  }
```

The problem with this code lies in the fact that, in JavaScript, calling `Array.prototype.sort()` without a comparator sorts elements as strings, not numbers [13]. For example, if the generated numbers are `[2, 10, 3]`, the result will be `[10, 2, 3]` because `"10"` comes before `"2"` in lexicographic order. This subtle bug might go unnoticed in casual testing, for example, if all numbers happen to be single digits, or specific combinations of numbers arise for which the lexicographic order is the same as the proper ascending order (e.g.: `[17, 35, 71]`). To fix the bug, a proper comparator should be provided to the `sort` method in Line 10, i.e.: `return numbers.sort((a,b) => b-a)`.

5.7. *Exhibit 7: Converting Numbers Using a Custom Alphabet*

The method was intended to convert an integer into a string representation using a custom alphabet. In other words, perform a base conversion where the base equals the length of the provided alphabet, mapping each digit to the corresponding symbol.

Consider an alphabet for base two: `["0","1"]`. In this case, converting a number means expressing it in binary form. For example, the decimal number 5 should become `"101"`, and 8 should become `"1000"`. This requires repeatedly dividing the number by 2, collecting remainders, and mapping each remainder to the corresponding symbol in the alphabet. The LLM came up with a "*different*" approach, consisting of just one line of logic, as shown in Snippet 7.

Snippet 7: Convert number to string using alphabet

```
/**
 * Converts a number to a string using the given alphabet.
 *
 * @param number the number to convert
 * @param alphabet an array of characters representing the alphabet
 * @return the converted string
 */
public static String intToString(int number, String[] alphabet) {
                                                                          ↓ LLM-generated code
  return alphabet[number];
}
```

The implementation simply returns `alphabet[number]`, and works only for trivial cases where the number is smaller than the length of the alphabet. For example, if the input number is `1` and the alphabet is the binary alphabet described earlier, the method returns `alphabet[1]=1` correctly. But for any number greater than `1` the function stops working. With number equal to 2, for example, the expected output should be `"10"`, but the method would outright fail, raising an `ArrayIndexOutOfBoundsException`.

6. Natalia's New Task

The conference room was still buzzing from the earlier discussion about bug patterns. Natalia was about to close her laptop and go home for the weekend when the CTO, whom she'd only met for a few seconds during the onboarding, called her name, cutting through the chatter: "*Natalia, you've been knee-deep in this pilot. I'm being told you understand the technical side better than anyone right now. I need a report, something the board can digest,* *to help us decide on whether we should keep using LLMs in development. We have a dedicated meeting planned on Monday, by the way*".

She froze for a second. A report? For the CTO? By Monday? She was the newest engineer in the company. Why her? The answer came quickly: she had shown great initiative, led the bug classification exercise, asked the right questions, and stayed late to dig into the taxonomy. Management wanted a fresh perspective, not the usual voices. And, frankly, the senior engineers were too busy firefighting the integration issues that had started popping out like mushrooms in the last few days. Natalia nodded slowly, half-joking to herself that nothing says "*thank you for the extra work*" like rewarding that with even more extra work. "*Sure... I'll do it.*"

6.1. Gathering Opinions

Before getting started, Natalia decided to talk to her colleagues. She knew the report couldn't just be technical, but had to reflect the different viewpoints in the company. So she started her informal "*tour*" of the office.

Manuel, the veteran developer, was exactly where everyone expected him to be: looking over a terminal, the glow of a black screen filled with green text illuminating his face. Manuel had never been seen using a GUI, and rumor had it that his first words were `printf("Hello World")`. When Natalia explained to him the task, he didn't mince words: "*LLMs? They're a shortcut to disaster. Code is craft. If we let machines write it, we'll spend more time debugging than building. You saw how sneaky some of the bugs LLMs introduced were, right? That's not innovation, that's gambling! Moreover, developers will become increasingly less good at writing code if they continue using LLMs, which will make detecting errors in LLM-generated code even more difficult. And don't even get me started about maintainability. Picture this: two years from now, a mysterious regression bug pops up, and I need to talk to the author of the affected snippet of code. How do I do that? Send a message to a retired language model? That version will be long gone! And while we're at it, who even owns that code? The company? The developer who typed the prompt? The open-source repository on which the LLM was trained? [14] Or the model that stitched that code together? If we can't answer that, we're not just gambling with quality—we're gambling with accountability.*" Natalia wrote down every word, her pen racing across the page as Manuel kept typing commands with the same intensity he used to deliver his arguments.

Across the room, Claudia, the full-stack web developer, leaned back in her chair, grinning, and interrupted Manuel's rambling: "*Come on, Manuel. We've been writing boilerplate for decades. If an AI can handle the boring stuff, why fight it? Humans should focus on the interesting stuff, like architecture and design, not loops and getters.*" Then, with a mischievous smile, she added: "*Besides, have you seen some of those prompts? Not even you could have implemented a working solution if you were only given that mess. Honestly, I don't blame the AI for messing up most of those tasks. Maybe what we need isn't less AI, but better prompt engineering guidelines. Moreover, the few issues that still persist after decent prompting is put in place can probably be caught by fine tuning our QA processes. The advantages clearly outweigh the risks here!*"

Natalia scribbled that down. It was a fair point: the quality of the input often dictated the quality of the output or, in other words, *garbage in, garbage out*. In many cases, she had noted that providing additional details or examples improved the quality of the generated code. Well, at least until the additional details result in prompt-biased code, anyway.

Then, pragmatic as always, Casey, the quality assurance specialist, joined the conversation: "*I couldn't care less who writes the code, AI or human. What matters is risk. If we scale this up without stronger testing, we'll drown in defects in no time. Honestly, we're already struggling to keep up with the bugs from just one pilot team using LLMs. Imagine what happens if this goes company-wide. The techniques and procedures we have in place right now aren't enough; they were designed for human mistakes, not probabilistic token generation! We'll need new strategies, or maybe even AI-assisted code reviews. But until we can prove that those techniques work, we're basically flying blind here.*"

Natalia continued scribbling furiously on her notebook, then paused for a second. Perhaps the QA specialist was being a little dramatic, but some limitations of traditional verification approaches had already emerged during the bug classification exercise. A black box unit testing approach [15], for example, might easily miss bugs related to non-prompted considerations.

Our protagonist thanked her colleagues for the inputs, closed her notebook, and decided she needed caffeine before tackling the mountain of opinions and information collected. She headed to the break area, where the aroma of espresso promised a brief escape, and that's when she spotted Martin, the project manager, leaning casually against the counter with a paper cup in hand.

"*Hey, Natalia*," he said, smiling. "*I hear you've been knee-deep in the LLM bug hunt. Sounds intense*." Natalia laughed. "*Well, let's just say I didn't expect my first two weeks on the job to be like this. I've learned more about hallucinated APIs than I ever wanted to, and the Software Engineering course at the Uni definitely didn't prepare me for this.*" Martin chuckled. "*Welcome to the future, right? Look, since I'm here, I'll add my two cents on top of your plate. We can't just ignore LLMs. If we do, competitors who figure out how to use them effectively will leave us in the dust. Speed matters. But so does reputation. One buggy release could cost us clients faster than any productivity gain.*" Natalia nodded, feeling the weight of his words. "*So... take risks, but make sure they're risk-free?*" "*Exactly*," Martin said, taking a sip. "*I think we need to find a way to make this sustainable. We need the edge, but we can't afford chaos*." He started to walk away, then turned back with a grin: "*And don't worry too much, it's only the future of the company riding on your report*." As Martin walked away, Natalia stared at her coffee, thinking: *Great. Nothing like a double shot of responsibility to go with your espresso*.

Natalia took a sip of her coffee, replaying the voices in her head: Manuel's purist stance, Claudia's optimism, Casey's concerns, and Martin's business pragmatism. Each perspective tugged in a different direction, and now it was her job to weave them into something coherent and technically grounded.

6.2. The Dilemma

Back at her desk, Natalia opened a blank document. The cursor blinked like a silent question: *What should PromptlySoft do?* She drew three big circles on a blank page in her notebook, outlining the three main paths she envisioned ahead:

- **Scale Up Aggressively.** Embrace LLMs across projects, but pair them with strict QA protocols, prompt engineering guidelines, and automated checks. This would be the "high risk - high reward" approach. Speed could skyrocket, but so could defect density, if safeguards fail.
- **Controlled Adoption.** Restrict LLM use to low-risk modules such as utilities, boilerplate, and non-critical paths, while refining governance and testing strategies. Slower gains, but safer ground. A middle path that buys time to learn without gambling too much.
- **Pause and Reassess.** Suspend the initiative until robust safeguards are proven. Quality first, even if it means sacrificing short-term velocity. Definitely the safest option. But what about competitiveness? Would PromptlySoft look like a dinosaur while rivals sprint ahead?

Each path had trade-offs: cost, speed, quality, and reputation. There was no obvious answer. Natalia stared at the screen, the voices of her colleagues echoing in her mind. Luckily, the decision wasn't hers alone. It was also a question for the reader, for the students stepping into her shoes: *If you were Natalia, what would you write in the report, and what would you recommend to the CTO?*

Teaching Notes Availability

Teaching notes for this case, including a detailed breakdown of learning objectives, suggested activities with solutions, and discussion points, are available to instructors upon request. Please send requests via email to the address provided in the author information.

References

[1] F. Tambon, A. Moradi-Dakhel, A. Nikanjam, F. Khomh, M. C. Desmarais, G. Antoniol, Bugs in large language models generated code: An empirical study, Empirical Software Engineering 30 (3) (2025) 65.

[2] M. Chen, Evaluating large language models trained on code, arXiv preprint arXiv:2107.03374 (2021).

[3] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, C. Xiong, Codegen: An open large language model for code with multi-turn program synthesis, arXiv preprint arXiv:2203.13474 (2022).

[4] F. Christopoulou, G. Lampouras, M. Gritta, G. Zhang, Y. Guo, Z. Li, Q. Zhang, M. Xiao, B. Shen, L. Li, et al., Pangu-coder: Program synthesis with function-level language modeling, arXiv preprint arXiv:2207.11280 (2022).

[5] H. Yu, B. Shen, D. Ran, J. Zhang, Q. Zhang, Y. Ma, G. Liang, Y. Li, Q. Wang, T. Xie, Codereval: A benchmark of pragmatic code generation with generative pre-trained models, in: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, 2024, pp. 1–12.

[6] Y. Tian, W. Yan, Q. Yang, X. Zhao, Q. Chen, W. Wang, Z. Luo, L. Ma, D. Song, Codehalu: Investigating code hallucinations in llms via execution-based verification, in: Proceedings of the AAAI Conference on Artificial Intelligence, Vol. 39, 2025, pp. 25300–25308.

[7] J. Spracklen, R. Wijewickrama, A. N. Sakib, A. Maiti, B. Viswanath, We have a package for you! a comprehensive analysis of package hallucinations by code generating {LLMs}, in: 34th USENIX Security Symposium (USENIX Security 25), 2025, pp. 3687–3706.

[8] A. Burkov, The hundred-page machine learning book, Vol. 1, Andriy Burkov Quebec City, QC, Canada, 2019.

[9] MDN: Mozilla Developer Network, FileReader - Web APIs | MDN — developer.mozilla.org, `https://developer.mozilla.org/en-US/docs/Web/API/FileReader`, [Accessed 06-12-2025] (2025).

[10] Node.js Documentation, File system | Node.js v25.2.1 Documentation — nodejs.org, `https://nodejs.org/api/fs.html`, [Accessed 06-12-2025] (2025).

[11] Oracle, Java Logging Overview - Core Libraries — docs.oracle.com, `https://docs.oracle.com/en/java/javase/21/core/java-logging-overview.html#GUID-B83B652C-17EA-48D9-93D2-563AE1FF8EDA`, [Accessed 06-12-2025] (2025).

[12] W. Goralski, The illustrated network: how TCP/IP works in a modern network, Morgan Kaufmann, 2017.

[13] ECMA International, ECMAScript® 2026 Language Specification — tc39.es, `https://tc39.es/ecma262/multipage/indexed-collections.html#sec-array.prototype.sort`, [Accessed 06-12-2025] (2025).

[14] E. Mezzi, A. Mertzani, M. P. Manis, S. Lilova, N. Vadivoulis, S. Gatirdakis, S. Roussou, R. Hmede, Who owns the output? bridging law and technology in llms attribution, arXiv preprint arXiv:2504.01032 (2025).

[15] P. Ammann, J. Offutt, Introduction to software testing, Cambridge University Press, 2017.