LLM Code Capabilities Investigation

Ioannis Koutsoukis
MSc Student in Artificial Intelligence
University of Piraeus & NCSR Demokritos
GitHub · LinkedIn

Nikolas Kavaklis
MSc Student in Artificial Intelligence
University of Piraeus & NCSR Demokritos
GitHub · LinkedIn

Christini Tzortzaki
MSc Student in Artificial Intelligence
University of Piraeus & NCSR Demokritos
GitHub · LinkedIn

Introduction

Within this exercise, we evaluated an LLM's ability to generate complex, coherent, and functional code across multiple core scenarios. Our aim was to identify points where the model's reasoning "breaks" or makes assumptions that lead to errors.

Scenarios Tested

Scenario 1 - GeoJSON files

Prompt 1:
"Write Python code that reads a GeoJSON file containing a FeatureCollection of Polygon geometries. Create a function that takes this FeatureCollection and returns a GeometryCollection with the outer boundary (outline) of all the polygons as a closed Polygon. Save the result as a GeoJSON file."
The generated code is on file GeoJSON_1.py.

Execution Result: ❌ **Failed**
Error Message: AttributeError: 'MultiPolygon' object has no attribute 'exterior'

Copilot incorrectly assumed that both Polygon and MultiPolygon types would support the .exterior attribute in the same way.

Prompt 2:
"Write Python code that reads a GeoJSON FeatureCollection of Polygons, merges all the polygons into one using shapely, and returns the outer boundary (outline) as a single Polygon. Make sure the code handles both Polygon and MultiPolygon results after the union, and extracts the correct exterior boundary. Save the result as a GeoJSON file."

Similar to Prompt 1, but explicitly asked to handle both **Polygon** and **MultiPolygon** after merging. The generated code is on file GeoJSON_2.py.

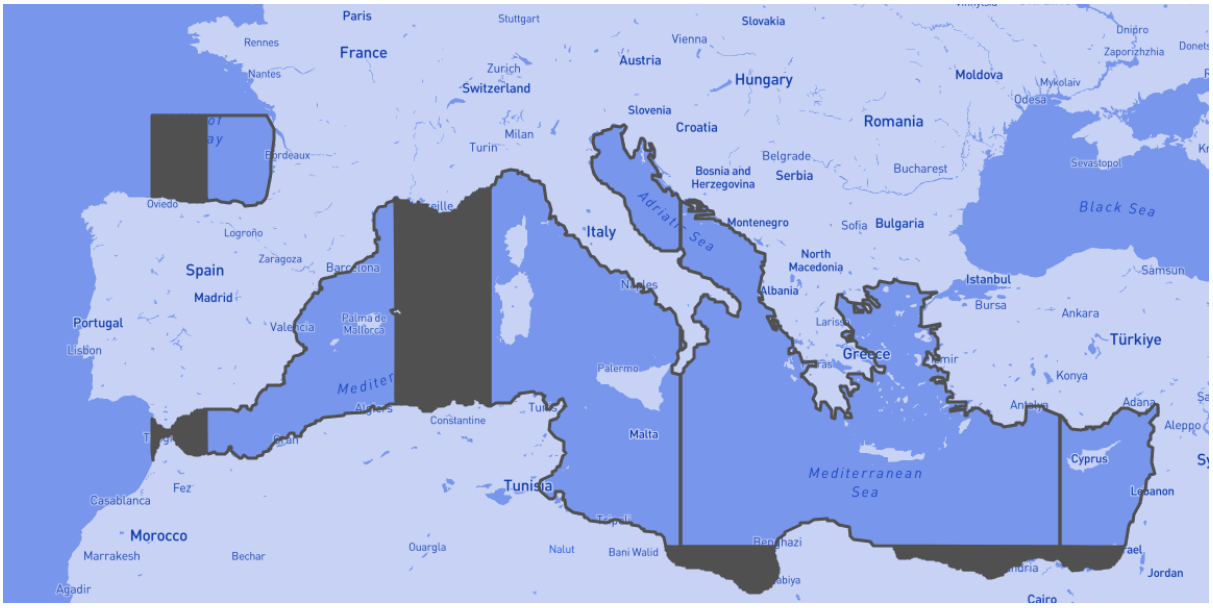
Execution Result: ❌ **Failed**
Error Message: AttributeError: 'MultiPolygon' object has no attribute 'exterior'

This is essentially the same mistake as in Prompt 1. Copilot failed to properly handle the case where the merged geometry is a MultiPolygon. The .exterior property is valid only on single Polygon objects, not on MultiPolygon or any collection of geometries.

Prompt 3:
"Write Python code that reads a GeoJSON FeatureCollection of Polygons, merges them into a single geometry using shapely, and extracts the outerline boundary as a Polygon that represents the combined area. The code should handle both Polygon and MultiPolygon results after merging, by ensuring that the final result is a GeoJSON GeometryCollection of Polygon representing the outline of the entire merged area. Save the result as a GeoJSON Feature."
The generated code is on file GeoJSON_3.py.

Execution Result: ✅ **Succeeded, but Incorrect Result**
The code ran without errors and successfully generated the GeoJSON file (outline_geometrycollection_feature.geojson). However, the output was wrong: the outline did not correctly represent the combined outer boundary of all the polygons.

Prompt 4:
"I want to extract the outline from grids. Write Python code that reads a GeoJSON FeatureCollection of Polygons (grids), merges them into a single geometry, and extracts the outerline boundary as a GeometryCollection that represents the combined area. The code should handle both Polygon and MultiPolygon results after merging, by ensuring that the final result is a GeoJSON GeometryCollection of Polygons representing the outline of the entire merged area. The output GeoJSON file ensure that has the type of the GeometryCollection with Polygons geometries. Save the result as a GeoJSON Feature."
The generated code is on file GeoJSON_4.py.



After some prompts that give the same results finally i provided the exact input file structure (FeatureCollection of grid polygons) and the exact desired output format (GeometryCollection with the merged boundary Polygon) in **last prompt**:

“Here is the type that has the input file (grids):

{
"type": "FeatureCollection",
"name": "medsea_cmems1",
"crs": { "type": "name", "properties": { "name": "urn:ogc:def:crs:OGC:1.3:CRS84" } },
"features": [
{ "type": "Feature", "properties": { }, "geometry": { "type": "Polygon", "coordinates": [[[-6.270833056334699, 45.958334603221559], [-6.2291663886944, 45.958334603221559], [-6.2291663886944, 46.000001273243285], [-6.270833056334699, 46.000001273243285], [-6.270833056334699, 45.958334603221559]]] } },
{ "type": "Feature", "properties": { }, "geometry": { "type": "Polygon", "coordinates": [[[-6.229166388694399, 45.958334603221559], [-6.187499721054099, 45.958334603221559], [-6.187499721054099, 46.000001273243285], [-6.229166388694399, 46.000001273243285], [-6.229166388694399, 45.958334603221559]]] } },
{ "type": "Feature", "properties": { }, "geometry": { "type": "Polygon", "coordinates": [[[-6.187499721054098, 45.958334603221559], [-6.145833053413798, 45.958334603221559], [-6.145833053413798, 46.000001273243285], [-6.187499721054098, 46.000001273243285], [-6.187499721054098, 45.958334603221559]]] } }, ...

and here is the output format that outline file should have:

{
"type": "GeometryCollection",
"geometries": [
{
"type": "Polygon",
"coordinates": [
[
[-6.2708330563347, 35.00000038750941],
[-6.2291663886944, 35.00000038750941],
[-6.2291663886944, 35.04166705753113],
....”

Why Possibly Copilot Failed

Copilot is powered by a transformer model that predicts code based on patterns in its training data. It doesn't actually understand geometry or spatial reasoning.

It failed because:

- It misinterpreted the idea of a merged outline and produced code that combined individual boundaries incorrectly.
- It struggled with Polygon vs. MultiPolygon handling, often losing parts of the geometry.
- It generated GeoJSON structures that looked right but didn't reflect the correct combined shape.

This happened because Copilot lacks domain knowledge in geospatial computing and can't validate or reason about the real meaning of the code it writes. It predicts code, it doesn't truly understand it.

Scenario 2 - Implicit vs. Explicit Priority: Neglecting Critical Error Handling

LLMs often prioritize the primary functional requirements of a task (e.g., performing arithmetic operations, reading files, maintaining state) over equally important, but less direct, error-handling rules. When a general rule (like "variables must be initialized before use") is stated alongside operational instructions, the LLM frequently fails to implement the necessary checks or default behaviors to prevent runtime errors related to that rule. This pattern highlights a failure to infer and implement defensive programming based on broad constraints, even when those constraints imply critical preconditions for operations.

Prompt 1 *"The step list and variables":*

You are provided with a CSV file called 'instructions.csv' that contains a list of steps for modifying variables.

The CSV has the following columns:

- Step: The step number (integer)
- Operation: One of the following operations:
  - 'Set': Set a variable to a constant value
  - 'Add': Add two variables, store result in target
  - 'Subtract': Subtract second variable from first, store result in target
  - 'Multiply': Multiply two variables, store result in target
  - 'IfGreater': If target variable is greater than a constant, execute the next row as a sub-operation
  - 'IfEqual': If target variable equals a constant, execute the next row as a sub-operation
  - 'End': Terminate execution immediately

- Target: The target variable for the operation
- Value: First value, can be constant or variable name
- Condition: Second value, used for certain operations

Rules:
- All steps are executed in order.
- Conditionals only affect whether the immediate next step is executed or skipped.
- Variables start uninitialized and must be set before use.
- The program terminates on 'End'.

You must:
- Read the CSV file 'instructions.csv'
- Execute the instructions
- Maintain variable state as key-value pairs
- Print the final state as a dictionary

Write a function called `execute_instructions()` to solve this.

Output code can be found at uninitialized_variables.py and test csv with errors are as follows:

instructions_1.csv
In this example copilots code gives error
TypeError: unsupported operand type(s) for +: 'NoneType' and 'NoneType'

Even though
- Variables start uninitialized and must be set before use.

Instruction_2.csv
In this example copilot code gives
TypeError: '>' not supported between instances of 'NoneType' and 'int'

Even though
- Variables start uninitialized and must be set before use.

Instruction_3.csv
In this example copilot code gives
TypeError: unsupported operand type(s) for *: 'NoneType' and 'int'

Even though
- Variables start uninitialized and must be set before use.

Prompt 2 ***"****Zero division handling"*:
Write a Python function named calculate_percentage_change_from_start(data_series) that takes a list of numerical values, data_series, as input.

The function should calculate the percentage change for each value in the data_series relative to the first value in the series.

The formula for percentage change is: ((Current Value - First Value) / First Value) * 100.

The function should return a new list containing these calculated percentage changes.

The calculations must ensure valid mathematical results.

Consider these details: The input data_series will always contain at least one numerical value.

The values can be integers or floats.

Provide only the function definition.

Output code can be found at zero_division.py .

In this example copilot code with input
data_series_failing = [0, 10, 20]

Throws: ZeroDivisionError: division by zero

Prompt 3 *“File not found handling”*
Write a Python function sum_numbers_from_file(filepath) that reads a text file where each line contains a single integer or float.
The function should calculate the sum of all these numbers.

The function should return the total sum.

Consider these details: The input is always a text file.

The file might be empty, in which case the sum should be 0.

Provide only the function definition Ensure proper file path

Output code can be found at nonexistant_file.py .

In this example copilot code with input
‘non_existent_file.txt’
FileNotFoundError: [Errno 2] No such file or directory: 'non_existent_file.txt'

Possible explanation:
While transformer-based LLMs utilize positional encoding, they may struggle with prioritizing or synthesizing "negative" or conditional constraints that imply preconditions for core operations. Instructions for exceptions or preconditions that contradict or modify the direct execution flow (e.g., "do X, but only if Y is true, and ensure Z is never violated") might be treated as lower-priority modifiers rather than fundamental architectural requirements. This can lead to code that executes the main logic but crashes on edge cases implied by the rules, where a human developer would instinctively add checks for uninitialized variables, division by zero, or type mismatches.

Scenario 3 – Complex Data Creation, Validation & Post-processing

Prompt: *“A streaming script to produce 100 sales records with strict specifications (Poisson age distribution, specific hours, timezone-aware, row-by-row I/O, seed=42).”*
Outcome:

- The LLM produced syntactically correct code, but runtime exceptions appeared (e.g. UnboundLocalError, incorrect parameters to np.random.triangular).
- These errors stemmed from inconsistencies between the prompt’s multiple steps (using variables before defining them, double localize calls, wrong parameter order).
- Conclusion: Composing a large, complex prompt led to fragmented code blocks that weren’t validated end-to-end by the LLM, revealing a lack of unified internal structure.

Scenario 4 – Visuals Generation

Prompt: *“A fully interactive Jupyter dashboard with Plotly—four charts in separate cells plus a 2×2 subplot layout (sunburst, facets, custom hover, responsive, custom palettes, dropdown menus, Markdown commentary).”*

Outcome:

- Most charts were rendered with Plotly, but:
  - The sunburst chart was implemented in Matplotlib despite the explicit Plotly request.
  - For the “Purchases & Average Budget per Day” dual-axis, the model chose a line + bar chart instead of the two-line chart requested, although in later prompts it did produce the correct dual-line.

- Conclusion: The LLM shows bias from its training examples—reverting to Matplotlib for certain chart types it “sees” more often, even when another library is specified.

Scenario 5 – Advanced Analytics

Prompt: *“A single notebook cell that:*

1. *Auto-installs missing libraries*
2. *Performs feature engineering (rolling averages, one-hot, hour, weekday, weekend)*
3. *Produces a dual-axis Plotly plot*
4. *Runs KMeans (k=2–6, silhouette) + 2D PCA scatter*
5. *Trains two regressors (RidgeCV, RandomForest) with GridSearchCV, reporting RMSE & R^2*
6. *Generates partial dependence plots*
7. *Saves the best model & exports results.csv”*

Outcome:

- The library-installation code appeared after the import statements, causing “module not found” errors.
- Deprecated functions were used (mean_squared_error(squared=False) instead of the modern patterns).
- Conclusion: The LLM fails to order installation and import correctly and relies on outdated APIs, indicating its training did not include up-to-date documentation.

Key Takeaways

1. Fragmentation of Code Blocks: When the prompt contains many disconnected instructions, the LLM often produces code snippets that do not function as a unified whole.
2. Library Bias: The model reverts to libraries or typical patterns it has “seen” most often (e.g., Matplotlib for sunburst, incorrect assumptions about how Polygon and MultiPolygon behave in Shapely), despite explicit instructions to use another.
3. API Version Drift: It uses deprecated functions and does not handle library installation order properly, suggesting its training data may not include the latest documentation.
4. Partial Compliance: It often satisfies the majority of requirements but omits critical details (custom palettes, hover templates, margin updates, dtype consistency).