Limitations of LLM Software Agents for Source code in Physical Planning and Control systems

Suzanna Sia

1 November 2024

Microsoft Program Synthesis Team (PROSE) Talk

Limitations of LLM Software Agents for Source code in Physical Planning and Control systems

Suzanna Sia

1 November 2024

Microsoft Program Synthesis Team (PROSE) Talk

Note: This talk highlights unsolved problems

A better title is “Challenges for a Software Agent …”

Summary

1. Troubleshooting and Code Generation for Systems and Applications is >10X more complicated than Academic Code generation benchmarks.

2. End-to-End generation is Impossible. Therefore, we need to study the human troubleshooting workflow, to decompose the problem into Tasks that CodeAgent LM can focus on.

3. "Isolating Functional Blocks" and "Understanding Source Code Logic" has dependencies on Understanding the Program's Design Principles, System Causal Graph and Domain Understanding.

4. "Formulating Plan for Code Improvement" generally works well at a high (and not very useful level).

5. Logic errors can be due to wrong high level plan, wrong implementation logic, wrong hyperparameters, or wrong input assumptions. Neither the code agent or Human has vision over what is happening.

6. Code Generation with Agentic self-corrective Feedback, and human feedback has irrecoverable failure.

7. Vision LMs can Providing Feedback (Labels) on the correctness of small functions, but we need to address the latency of Vision LMs for synthetic generation.

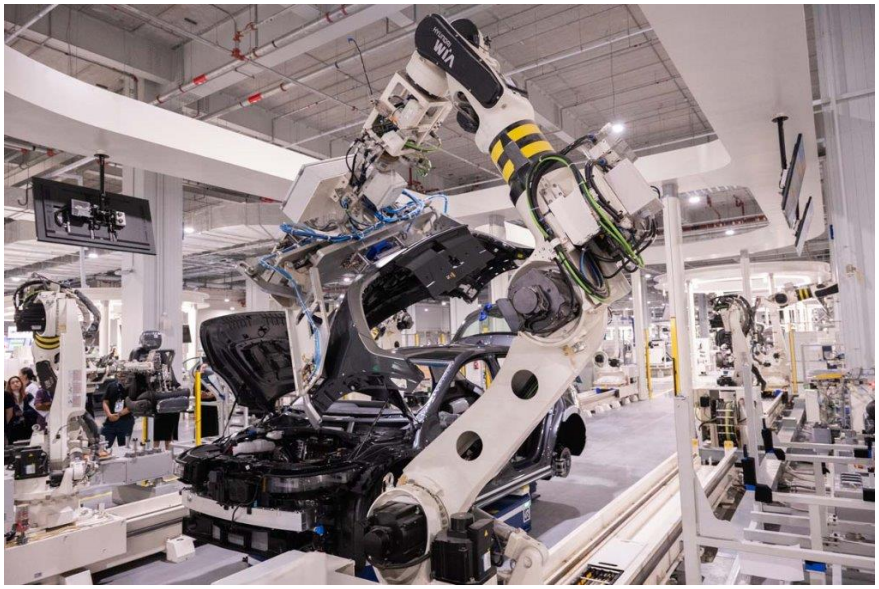
8. Vision LMs can potentially be used as a reward generator in RL for Code Generators, or distilled into small models which perform the desired functionality (instead of code).

9. Counterfactual Tests should be generated which check whether the code is faithful to the High level plan approved by the human.

Rule-based Planning and Control Systems

Q: What are Rule-based Planning and Control Systems?

A: Logistics and Assembly Related Systems.



Q: Aren't these "Solved by RL"?

A: "Low-level" path planning and navigation are handled by RL. But logistics and coordination are still handled by Rule-based Planning Systems

Media Images from https://carbuyer.com.sg/hyundai-motor-group-innovation-centre-singapore-opens/

Troubleshooting and Generation benchmarks vs ACTUAL production code

Academic Open-source Code Benchmarks	Real-world Complex Software Systems
Single file, single function 50 of lines of isolated functional code	100 files, function taking input from other modules. 100,000 lines of code
Well-specified function from the docstring. Description and range of expected behavior.	No comments, poorly commented, or function behavior described in other documentation (not always correct)
Only one isolated system, inputs well understood.	Interfacing with multiple systems, inputs dependent on the output of other complex systems.
Mostly Python	Legacy code and functions (C#, C++, Ladder Diagrams – PLC)
Output can be verified at the functional level	Output verified in simulation and then in production.
Direct Performance Gains from Training on Github Repositories, many of which contain answer to leetcode style questions	Not clear where to get raw performance gains from. Factory operational environment codes dissimilar to anything seen on the internet. End-to-end training seems impossible.

Correctness around 70-80%

Correctness < 0.1%?!

Code Benchmarks Vs Software Issues in the Factory

Academic Code Benchmarks

Error Type	#Errors	Error Cause	One-line Description
AssertionError	8171(63.64%)	Test Case Failure	Model-generated function fails pre-set test cases
		Empty Function	Model creates an empty function or uses pass statement
NameError	2916(22.71%)	Misremembered Name	Model defines a function but calls it with incorrect name
		Missing Content	Model doesn't generate any content, leading to missing entry_point
		Missing Import	Model doesn't import required modules or functions
SyntaxError	739(5.76%)	Unbalanced Delimiters	Imbalance in quotes or parentheses in generated code
		Function Overflow	Excessive function generation hits limit, causing incomplete output and SyntaxError.
ValueError	337(2.63%)	Empty Sequence	Function fails to handle empty input, causing ValueError
		Intentional Raise	Model includes raise statements for program robustness
		Inappropriate Argument	Function receives correct type but inappropriate value
IndexError	291(2.27%)	Out of Bounds	Attempting to access an index outside sequence range
TypeError	220(1.71%)	Incompatible Operation	Applying operation to object of inappropriate type
AttributeError	58(0.45%)	Non-existent Attribute	Accessing a non-existent attribute of an object
TimeoutError	50(0.39%)	Execution Timeout	Code execution exceeds set time limit
IndentationError	32(0.25%)	Inconsistent Indentation	Indentation levels in same code block are inconsistent

Fixing Code Generation Errors for Large Language Models,
Hao Wen et al., 2024 https://arxiv.org/pdf/2409.00676

Real-world Software Systems

Logistics System is assigning wrong mission orders and wrong destinations

Traffic Jam is occuring near the buffer (waiting area), too many AGVs assigned to the same buffer area

AGVs switching from Automatic to Manual mode on wake-sleep command

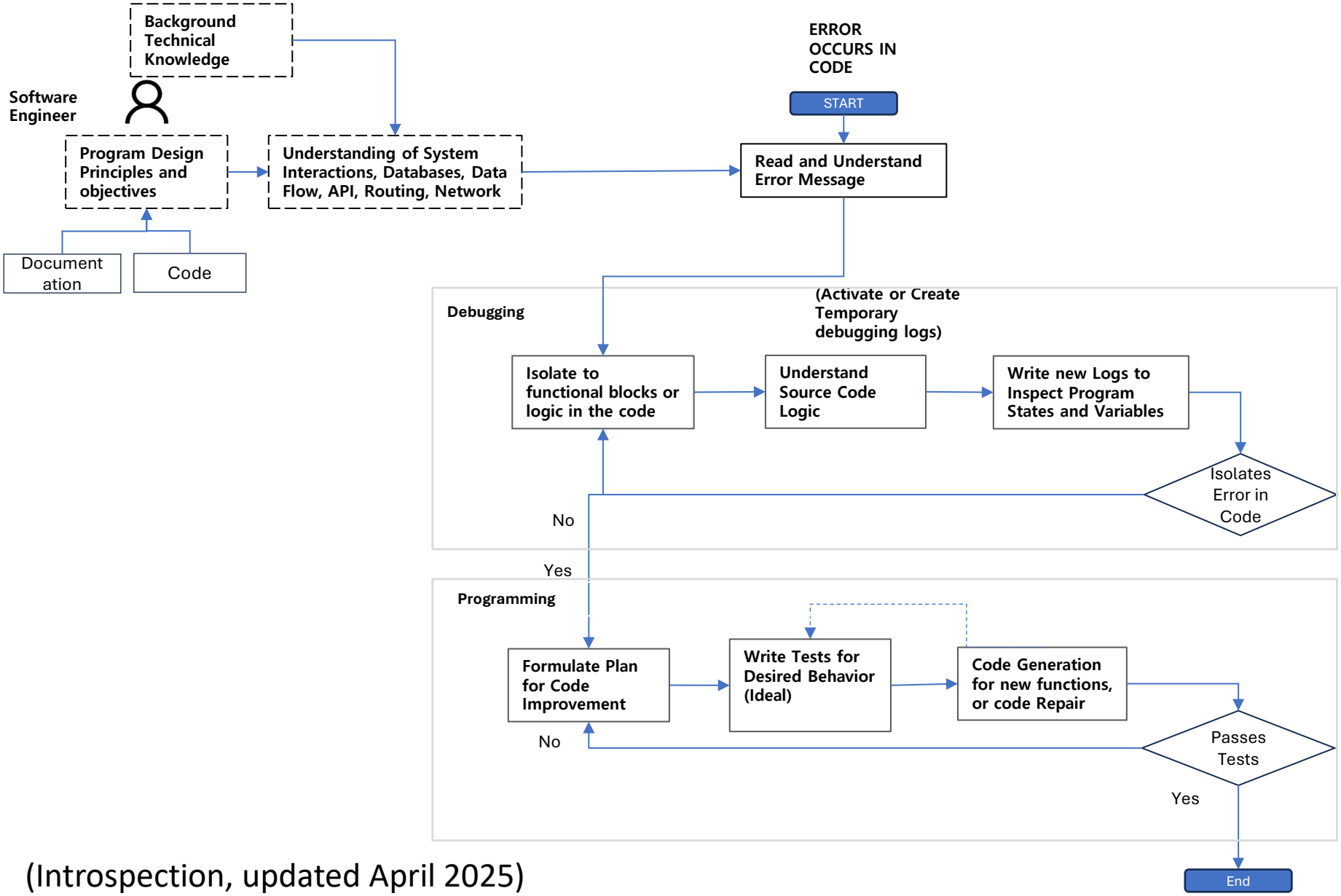
AGVs stop moving in a particular region of the factory

Summary

1. Troubleshooting and Code Generation for Systems and Applications is >10X more complicated than Academic Code generation benchmarks.

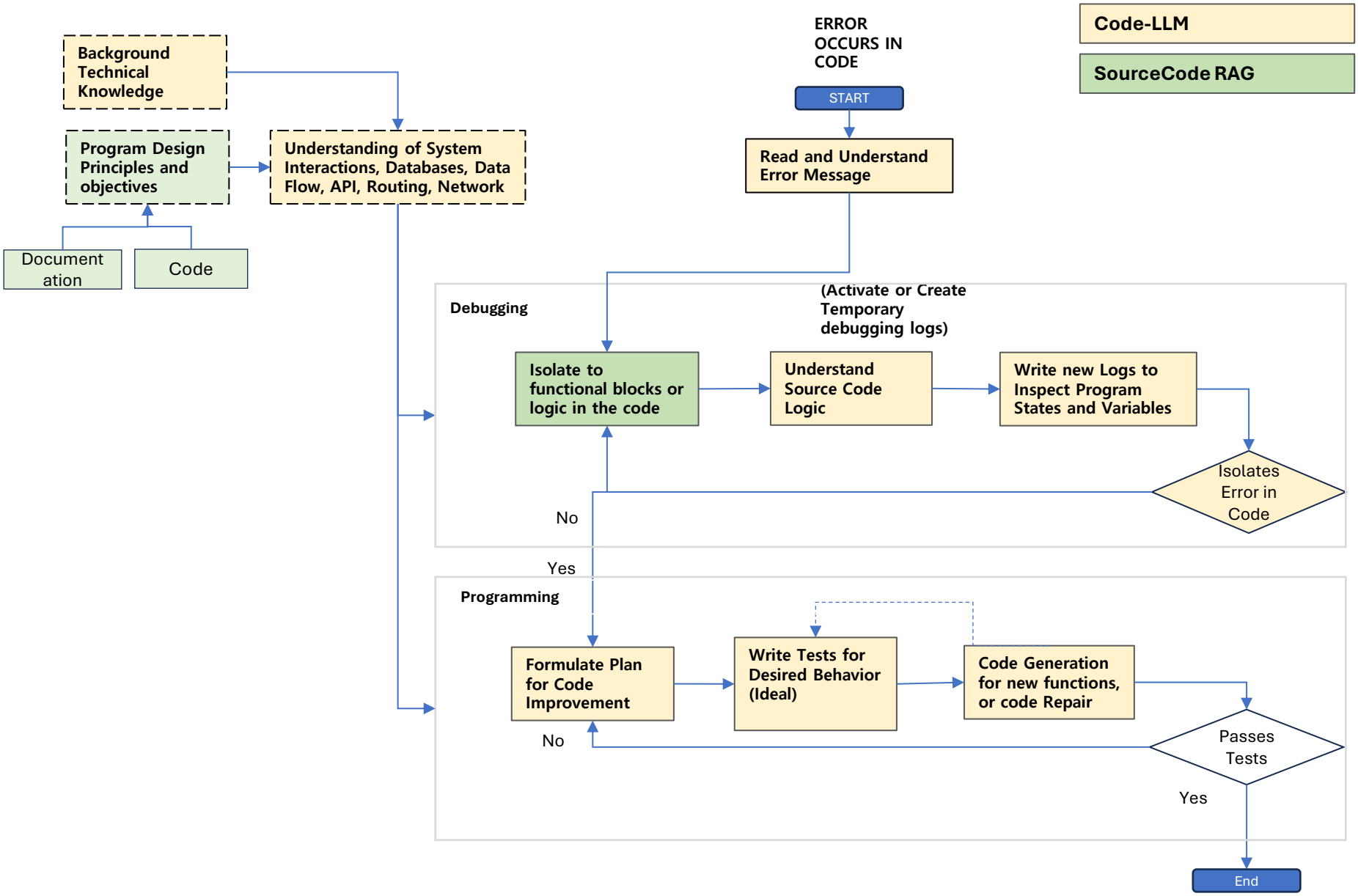
2. End-to-End generation is Impossible. Therefore, we need to study the human troubleshooting workflow, to decompose the problem into Functional Tasks that CodeAgent LM can assist the human on.

Troubleshooting and Code Generation workflow without LLMs



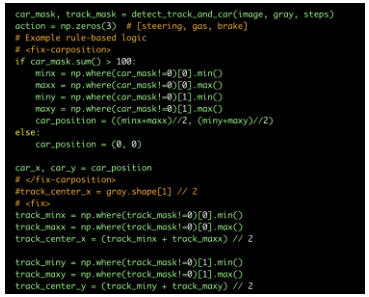
(Introspection, updated April 2025)

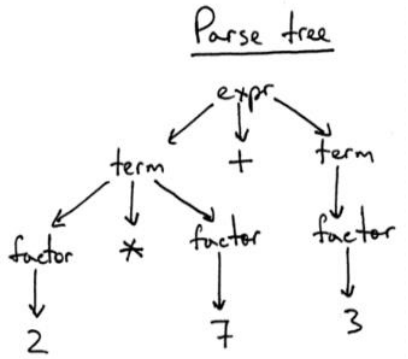
Troubleshooting and Code Generation workflow with LLMs



Code RAG

Strategy for Chunking: Naturally we want to utilise concrete syntax trees and to just have “functions” or classes as code blocks.

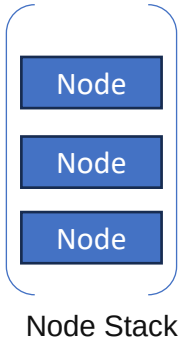


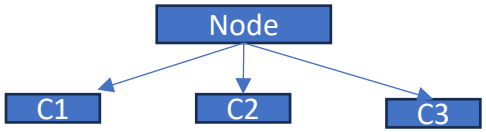


Tree with nodes represent expressions, statements, classes functions and other constructs.

Its not meaningful to retrieve a 1000 line function or 1000 line class, nor 5 lines of variable assignment.

Hence we want functional chunking **but with some meaningful Size Constraints which still allow *natural* syntax based chunking. (DFS + BFS)**






```
For each child_node:
    if size > max_threshold_size:
        recursive_chunk(child_node)
    else:
        if stack[-1] + child_node > min_threshold_size:
            result.append(stack.pop() + child_node)
        else:
            stack.append(child_node)
```

DFS

BFS

Open-source Programming Language Parsers: https://tree-sitter.github.io/tree-sitter
Chunking Code for Retrieval with Size Constraints: https://suzyahyah.github.io/code/2024/07/07/Chunking-code-for-RAG.html

Summary

1. Troubleshooting and Code Generation for Systems and Applications is >10X more complicated than Academic Code generation benchmarks.

2. End-to-End generation is Impossible. Therefore, we need to study the human troubleshooting workflow, to decompose the problem into Tasks that CodeAgent LM can focus on.

3. Steering LLMs towards Code Generation depends on the Programmer providing the relevant context to the Model, for understanding the Program's Design Principles, System Understanding and Domain Understanding.

However if we can provide ALL context to the LLM, then we should be able to replace programmers?

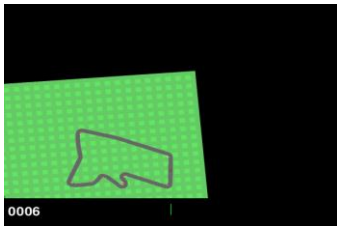
No, because debugging and generating new code in physical control systems is extremely difficult. It’s not fix patterns that you can rehash like in constructing certain types of games or UI.

Chat GPT Programmer Workflow

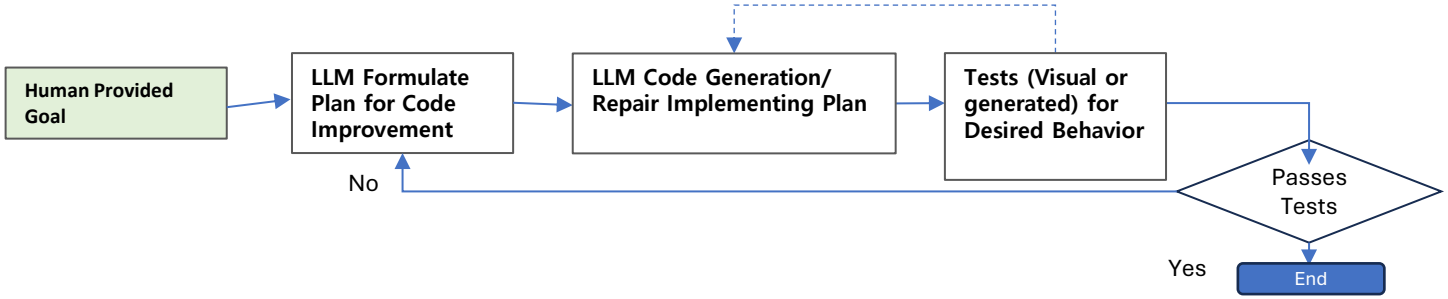
Gen Z
Chat-GPT
Programmer

Start

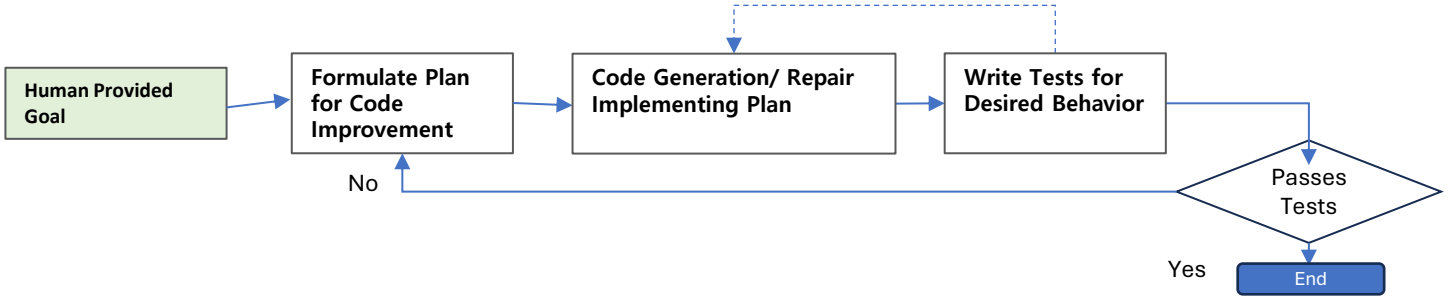
The Car is going off-track and not making the turn early enough



Top-down Image of the Car-racing Track



Formulating Plan for Code Improvement



Step 0: (Human provides Goal)

"Generate the high level logic for a program which will navigate a car on a track.

The track is surrounded by grass and there are no other obstacles.

The only goal is to stay on the track and reach the destination.

There are four possible actions, steer left, steer right, go forward, brake.

The only thing the program can observe is the pixel image of a top-down view of where the car is. Which might or might not be on the track. "

Summary

1. Troubleshooting and Code Generation for Systems and Applications is >10X more complicated than Academic Code generation benchmarks.

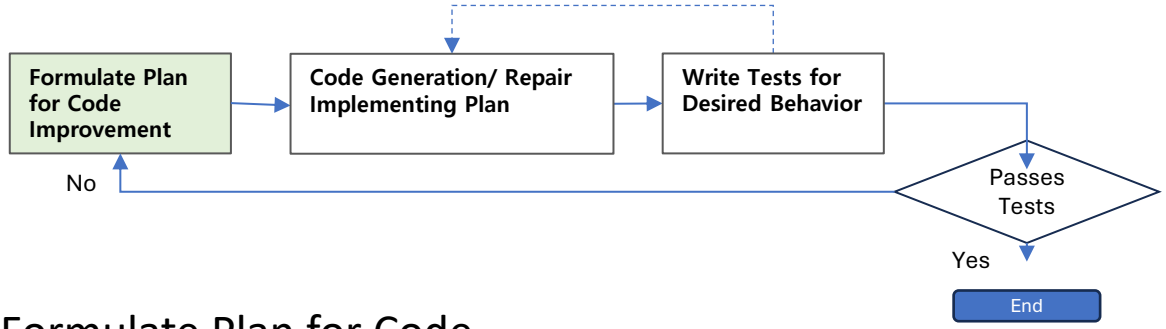
2. End-to-End generation is Impossible. Therefore, we need to study the human troubleshooting workflow, to decompose the problem into Tasks that CodeAgent LM can focus on.

3. "Isolating Functional Blocks" and "Understanding Source Code Logic" has dependencies on Understanding the Program's Design Principles, System Causal Graph and Domain Understanding.

4. Debugging and generating new code in physical control systems is extremely difficult. It’s not fix patterns that you can rehash like in constructing certain types of games or UI.

5. "Formulating Plan for Code Improvement" can generally works well at a high (and not very useful level).

Formulating Plan for Code Improvement

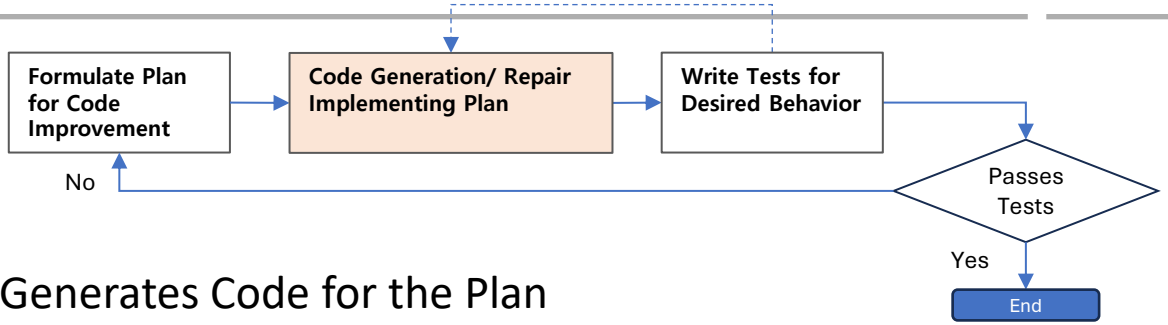


Step 1: Agent Formulate Plan for Code

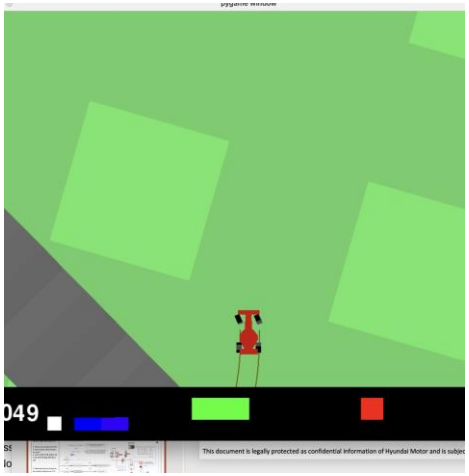
Main Control Loop

- While the car has not reached the destination:
  1. **Determine Position Relative to Track**
     - Assess the car's location:
       - **In Track:** If the car is on the track based on pixel analysis.
       - **Off Track:** If the car is off the track (i.e., on the grass).
  2. **Decision Making**
     - Based on the car's relative position:
       - **If On Track:**
         - Check for upcoming turns:
           - If a turn is detected, decide whether to:
             - **Brake:** If speed needs to be reduced.
             - **Steer Left/Right:** Adjust the steering to follow the track.
         - If no turns are imminent:
           - **Action:** Go forward.
       - **If Off Track:**
         - Determine the direction to steer based on pixel analysis:
           - If the car is off to the left, **steer right** to realign.
           - If the car is off to the right, **steer left** to realign.
  3. **Execute Action**
     - Perform the chosen action (steer left, steer right, go forward, brake).
     - Update the car's state (position and orientation) based on the action.

Formulating Plan for Code Improvement

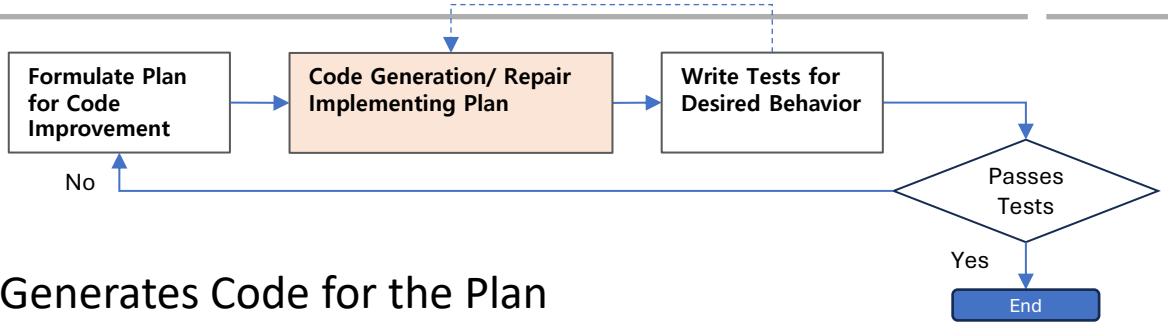


Step 2: Agent Generates Code for the Plan



Oops

Formulating Plan for Code Improvement



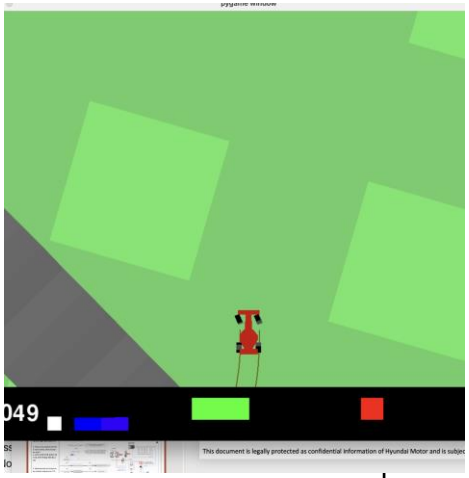
Step 2: Agent Generates Code for the Plan

Potential Errors

- Wrong High-level Plan
- Wrong Implementation of Logic
- Wrong assumptions over available inputs
- Wrong hyperparameters

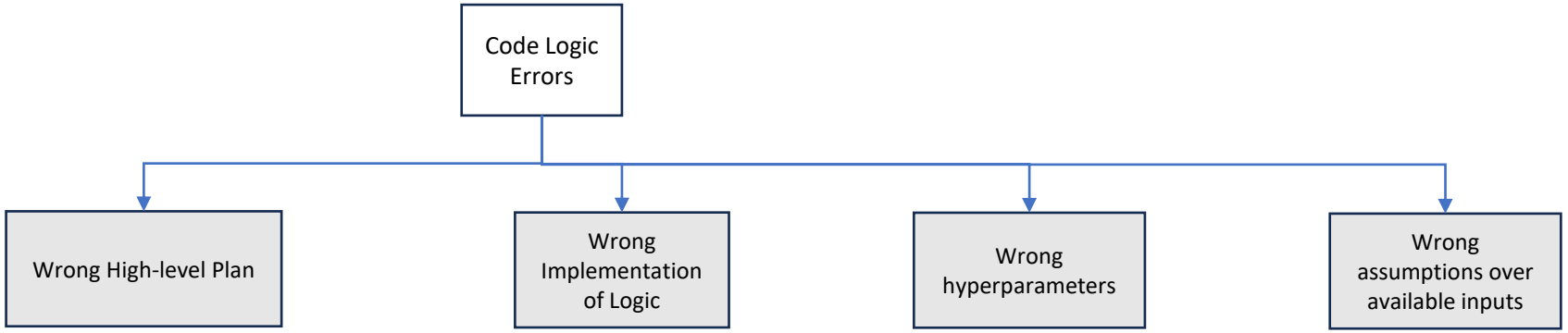
```
# Check if car is within the bounds of the track mask
if 0 <= x < track_width and 0 <= y < track_height:
    if track_mask[y][x] == 1:  # Assuming 1 represents track pixels
        on_track = True
    else:
        on_track = False
else:
    on_track = False

# Decision making based on the car's position
if on_track:
    # Check for upcoming turns (simplified logic)
    if detect_turn(track_mask, car_position):
        # Determine if braking is necessary
        if speed > desired_speed:  # Assuming desired_speed is defined elsewhere
            action = "brake"
        else:
            # Determine direction to steer based on upcoming turn
            action = determine_steering_direction(track_mask, car_position)
    else:
        action = "go forward"
```

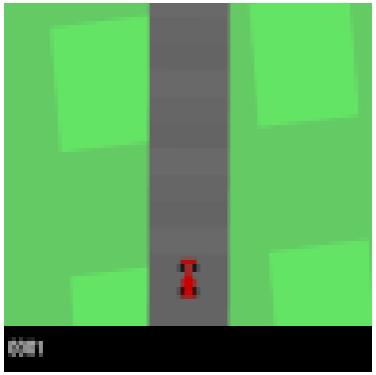


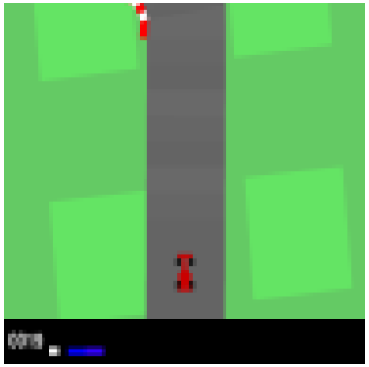
Oops

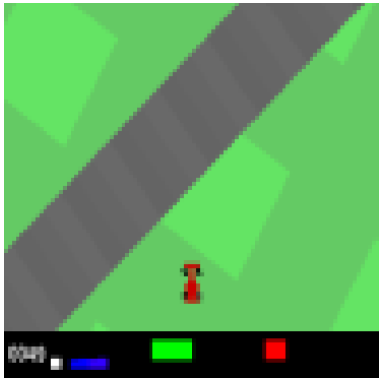
Step 2: Agent Generates Code for the Plan: Code Logic Errors




```
    # Check if the car is on the track
    if track_mask[int(car_y), int(car_x)] == 0:
        capture_screenshot(image, f"images_debug/off-track.step{steps}.png")
```



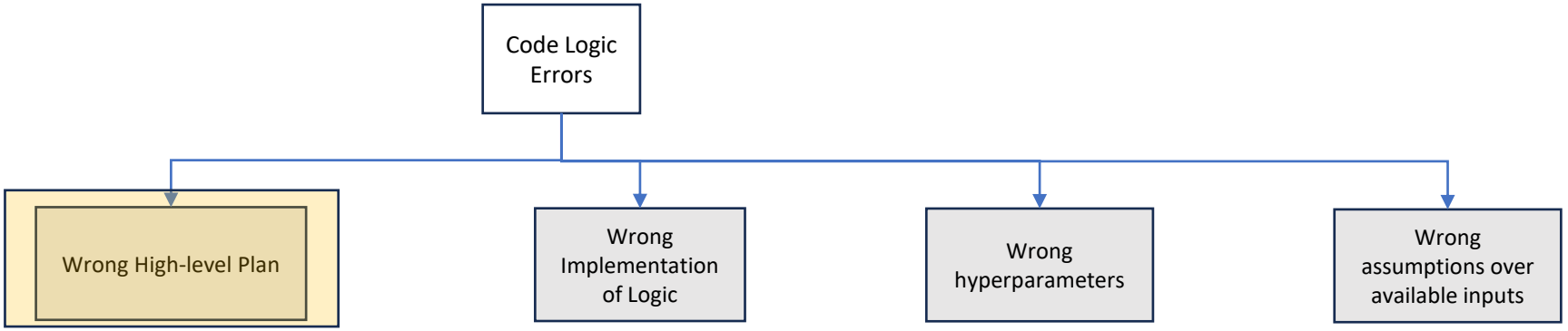




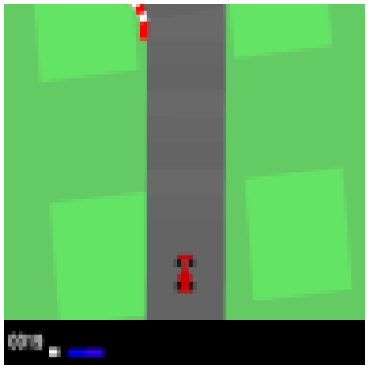
Is this line correctly written?

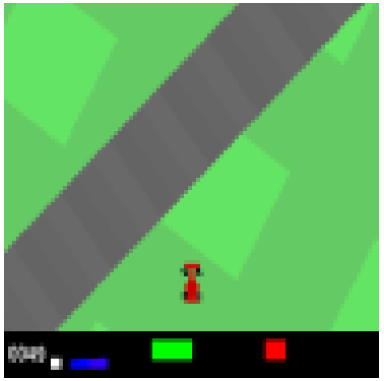
How can the model verify this line is as intended?

Step 2: Agent Generates Code for the Plan: Code Logic Errors




```
    # Check if the car is on the track
    if track_mask[int(car_y), int(car_x)] == 0:
        capture_screenshot(image, f"images_debug/off-track.step{steps}.png")
```

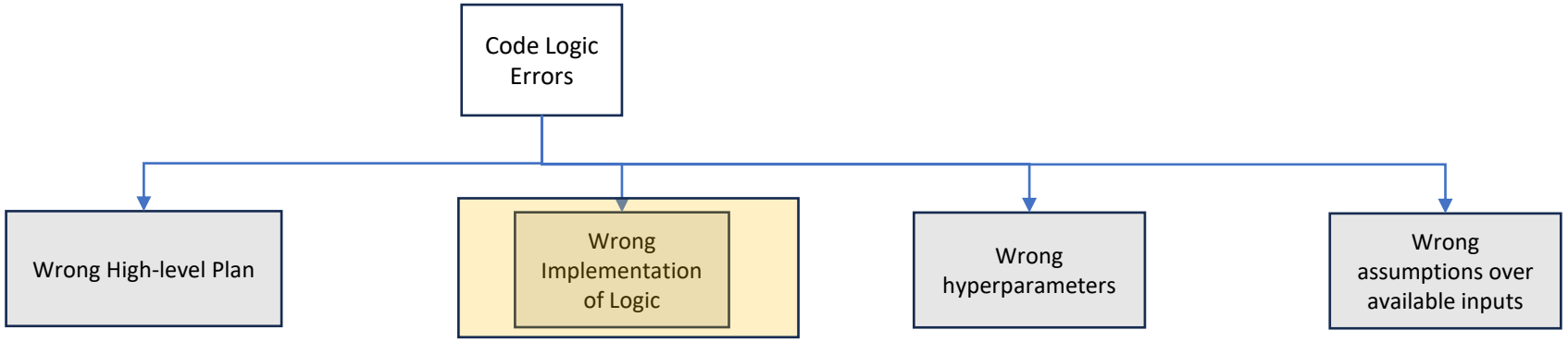





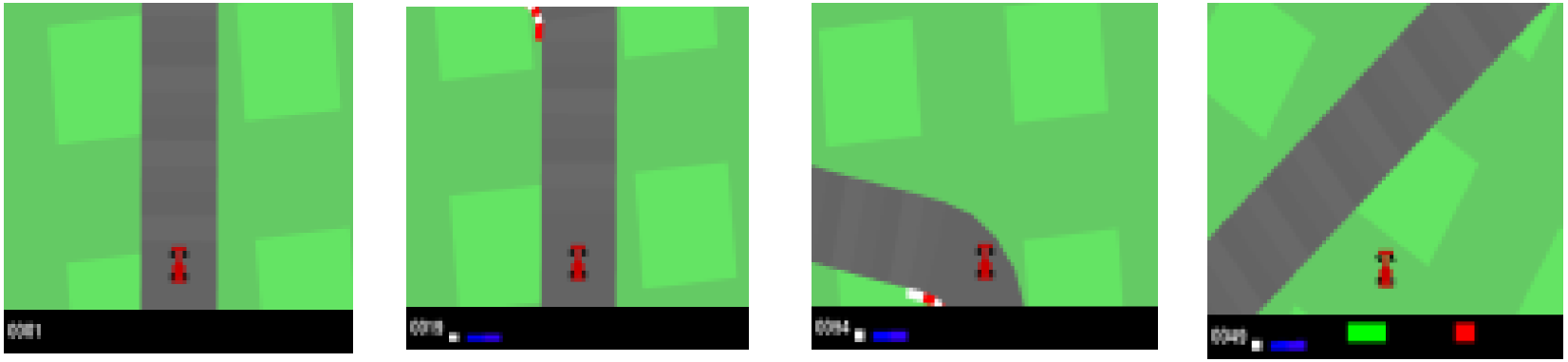
Is this line correctly written?

How can the model verify this line is as intended?

Step 2: Agent Generates Code for the Plan: Code Logic Errors



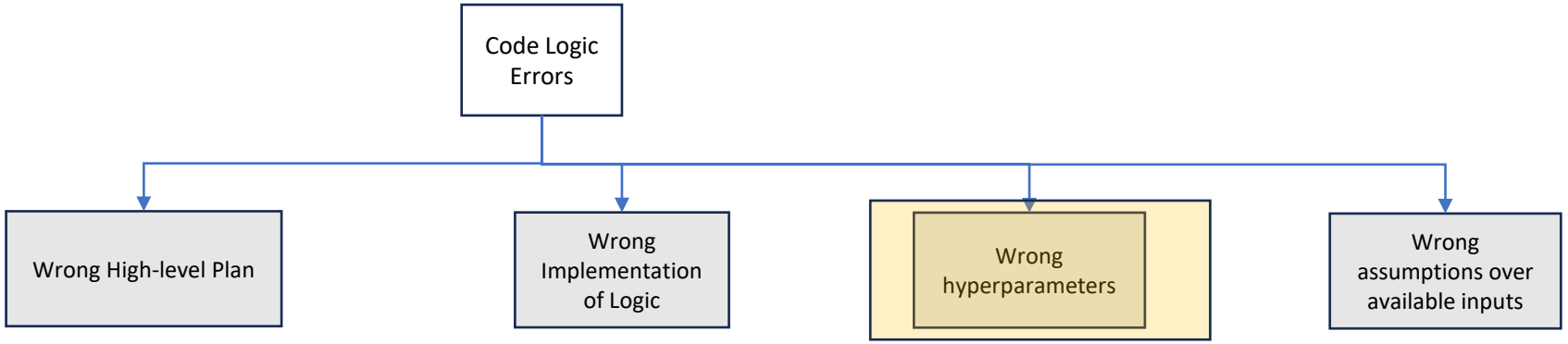

```
    # Check if the car is on the track
    if track_mask[int(car_y), int(car_x)] == 0:
        capture_screenshot(image, f"images_debug/off-track.step{steps}.png")
```



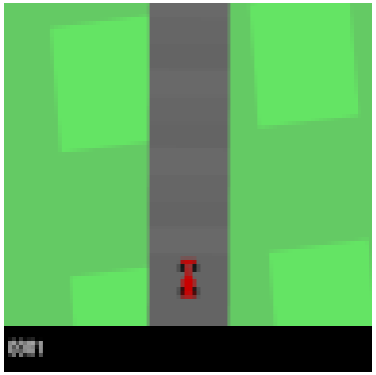
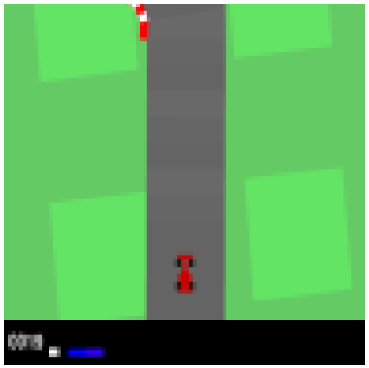
Is this line correctly written?

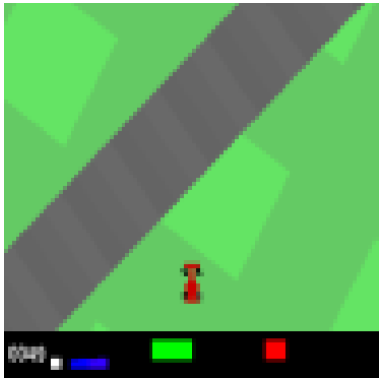
How can the model verify this line is as intended?

Step 2: Agent Generates Code for the Plan: Code Logic Errors




```
    # Check if the car is on the track
    if track_mask[int(car_y), int(car_x)] == 0:
        capture_screenshot(image, f"images_debug/off-track.step{steps}.png")
```

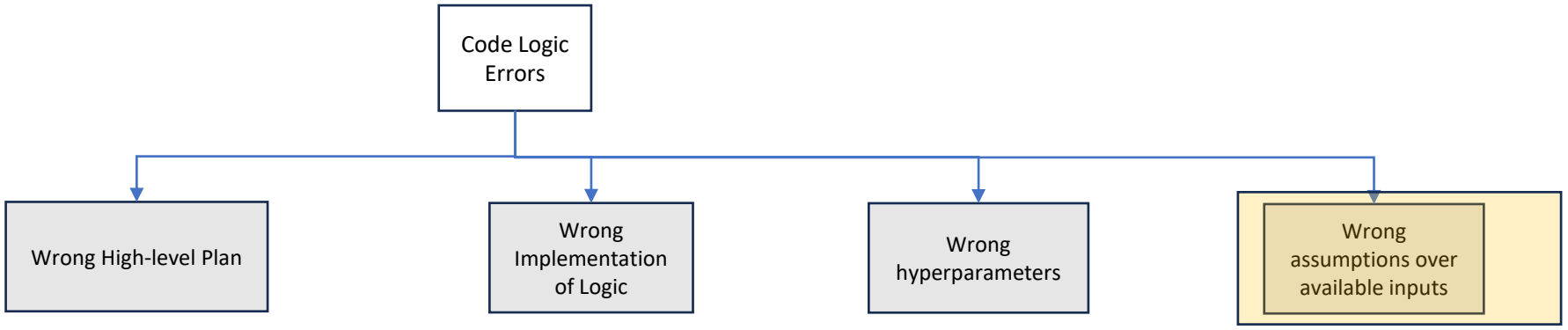





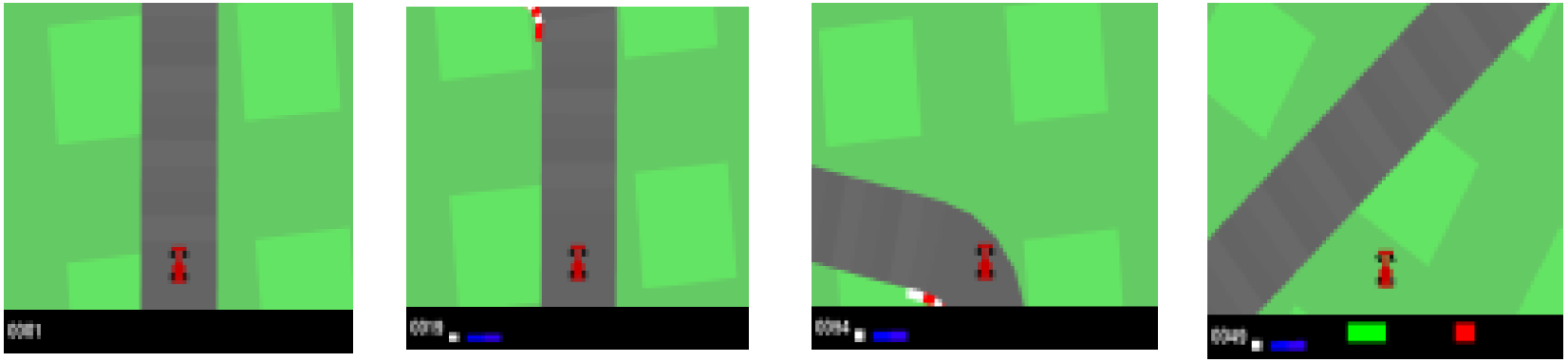
Is this line correctly written?

How can the model verify this line is as intended?

Step 2: Agent Generates Code for the Plan: Code Logic Errors



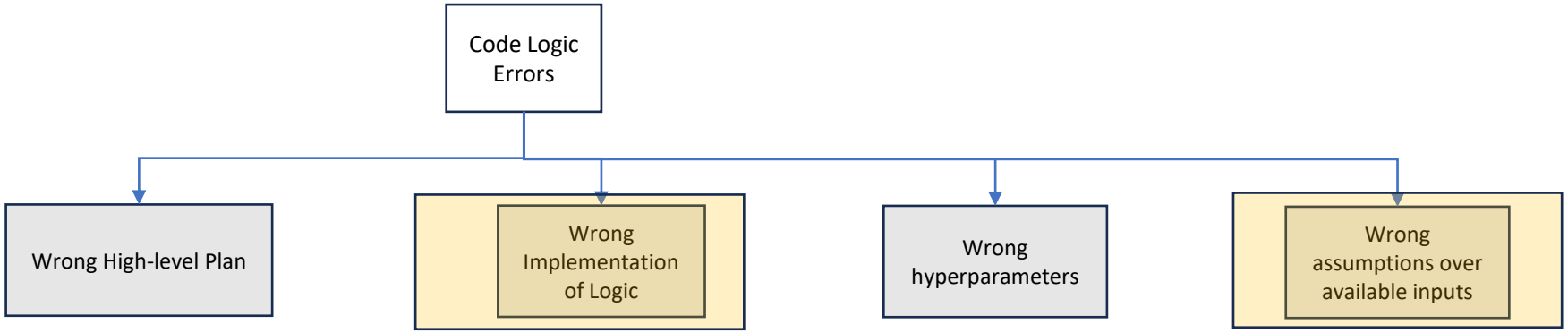

```
    # Check if the car is on the track
    if track_mas[int(car_y), int(car_x)] == 0:
        capture_screenshot(image, f"images_debug/off-track.step{steps}.png")
```



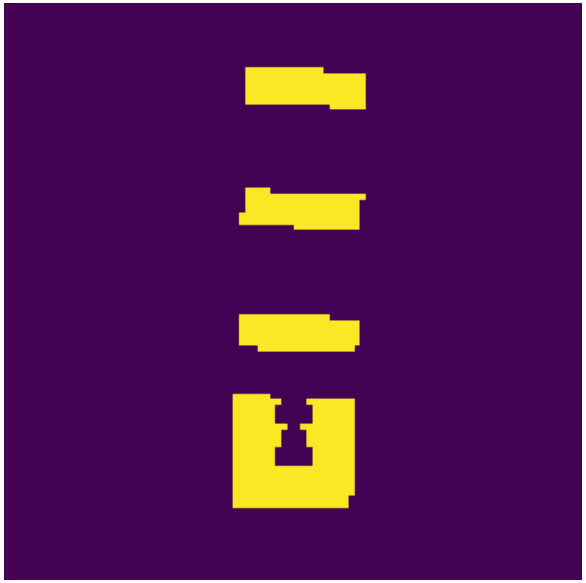
Is this line correctly written?

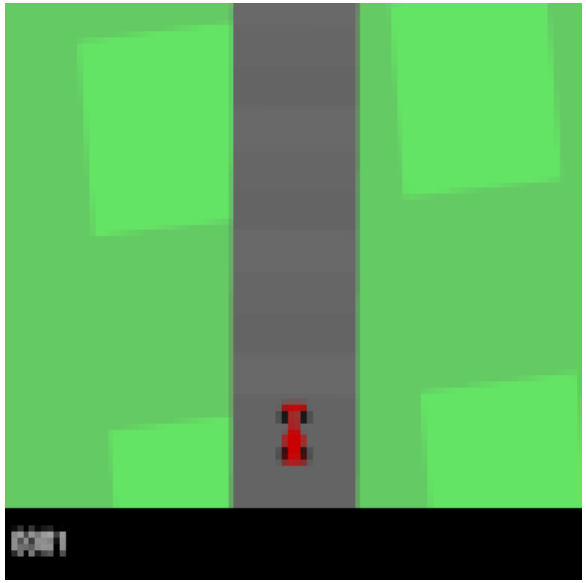
How can the model verify this line is as intended?

Step 2: Agent Generates Code for the Plan: Code Logic Errors



Using the track mask to detect if the car is on or off-track was a bad idea.





"Track Mask" which the Program uses, but the LLM cannot "see".

Summary

1. Troubleshooting and Code Generation for Systems and Applications is >10X more complicated than Academic Code generation benchmarks.

2. End-to-End generation is Impossible. Therefore, we need to study the human troubleshooting workflow, to decompose the problem into Tasks that CodeAgent LM can focus on.

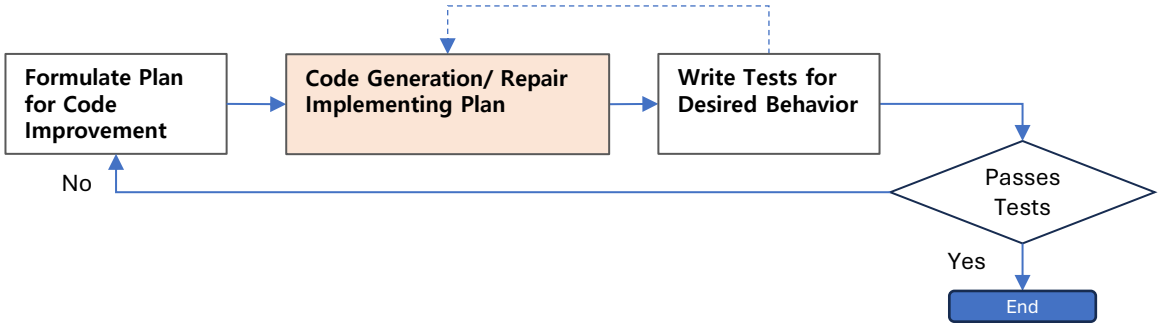
3. "Isolating Functional Blocks" and "Understanding Source Code Logic" has dependencies on Understanding the Program's Design Principles, System Causal Graph and Domain Understanding.

4. Debugging and generating new code in physical control systems is extremely difficult. It’s not fix patterns that you can rehash like in constructing certain types of games or UI.

5. "Formulating Plan for Code Improvement" can generally works well at a high (and not very useful level).

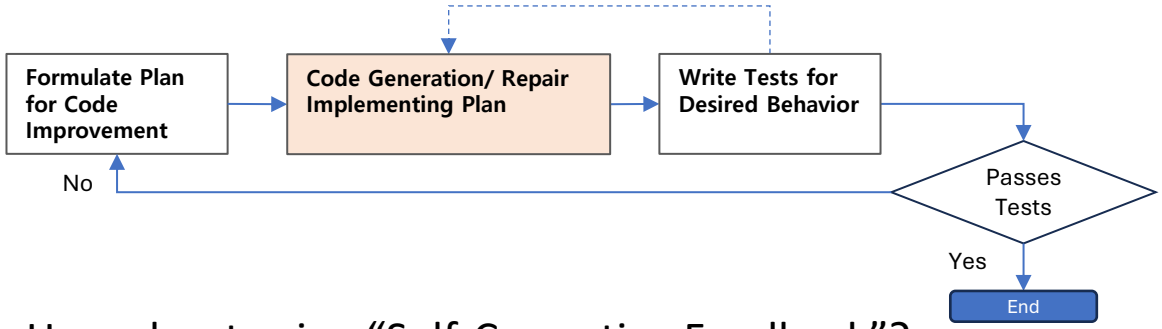
6. Logic errors can be due to wrong high level plan, wrong implementation logic, wrong hyperparameters, or wrong input assumptions. Neither the code agent or Human has vision over what is happening.

Step 2: Agent Generates Code for the Plan

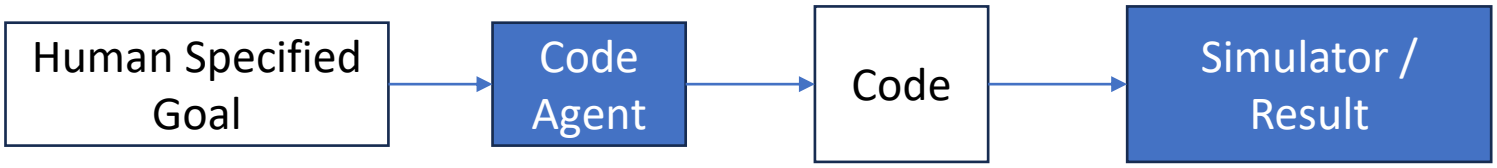


How about using “Self-Corrective Feedback”?

Limitations of Agentic-LLM corrective feedback



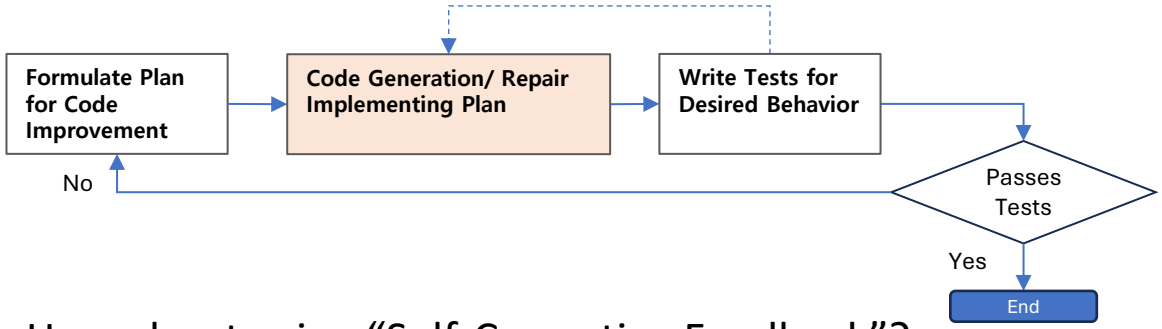
How about using “Self-Corrective Feedback”?



Code Agent sees

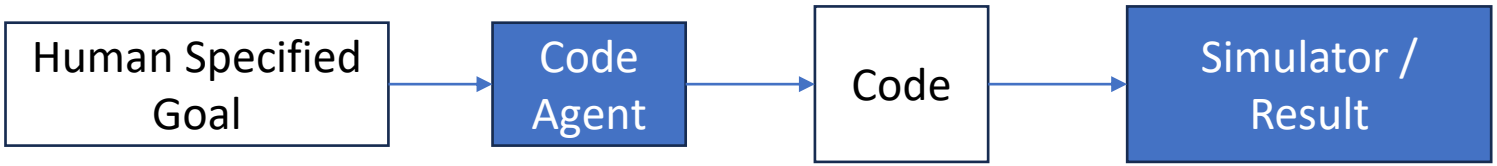
(1) Human Specified Goals

Limitations of Agentic-LLM corrective feedback



How about using “Self-Corrective Feedback”?

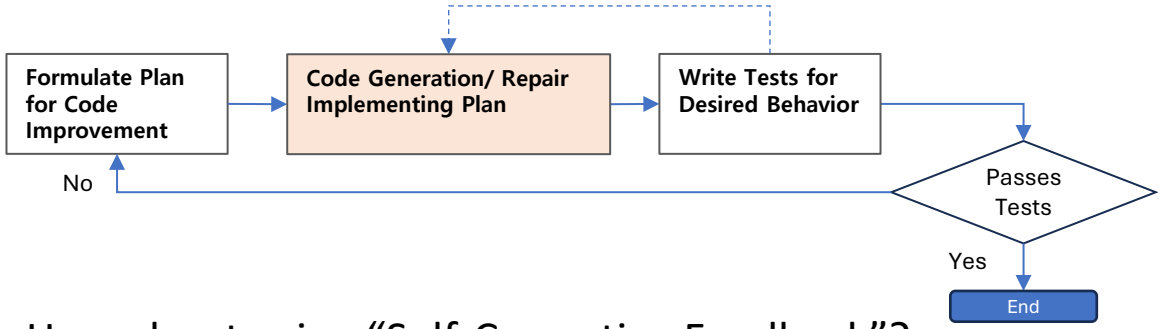
(1) Easy: Execution Error



Code Agent sees

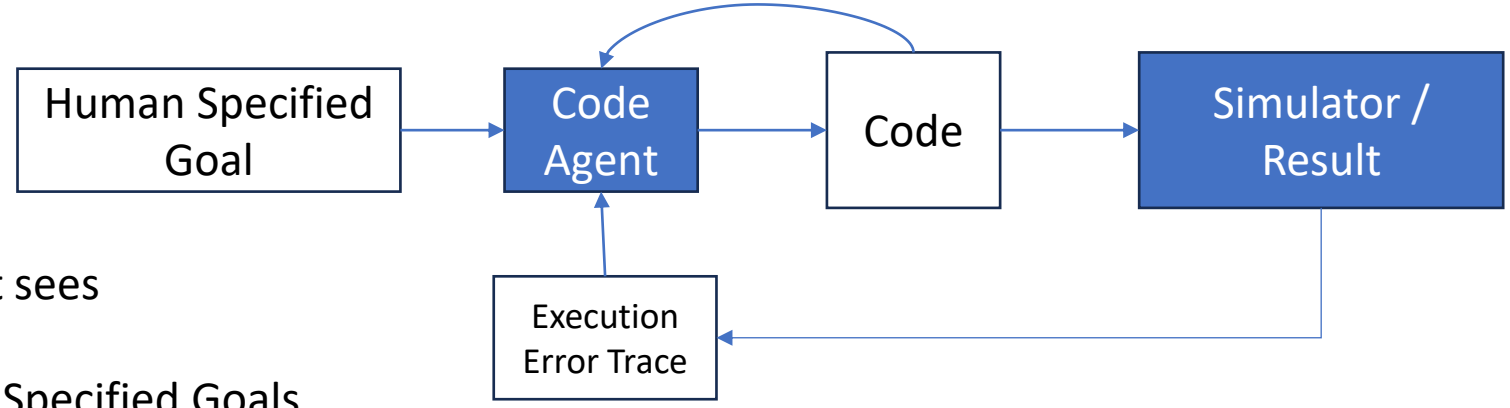
(1) Human Specified Goals

Limitations of Agentic-LLM corrective feedback



How about using “Self-Corrective Feedback”?

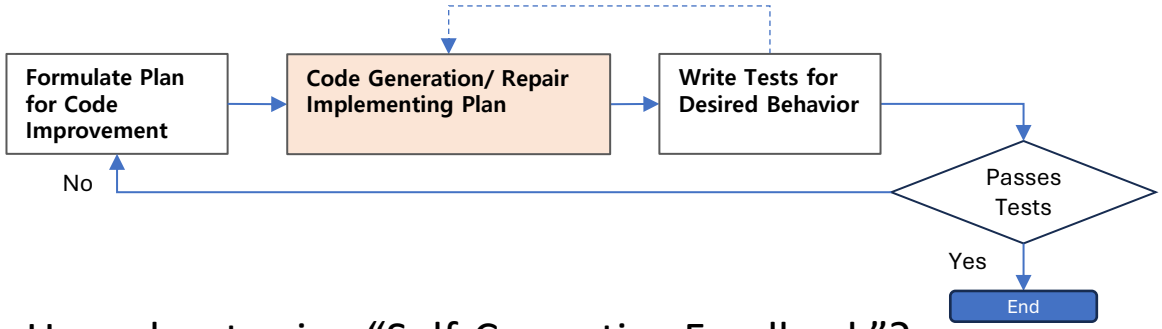
(1) Easy: Execution Error



Code Agent sees

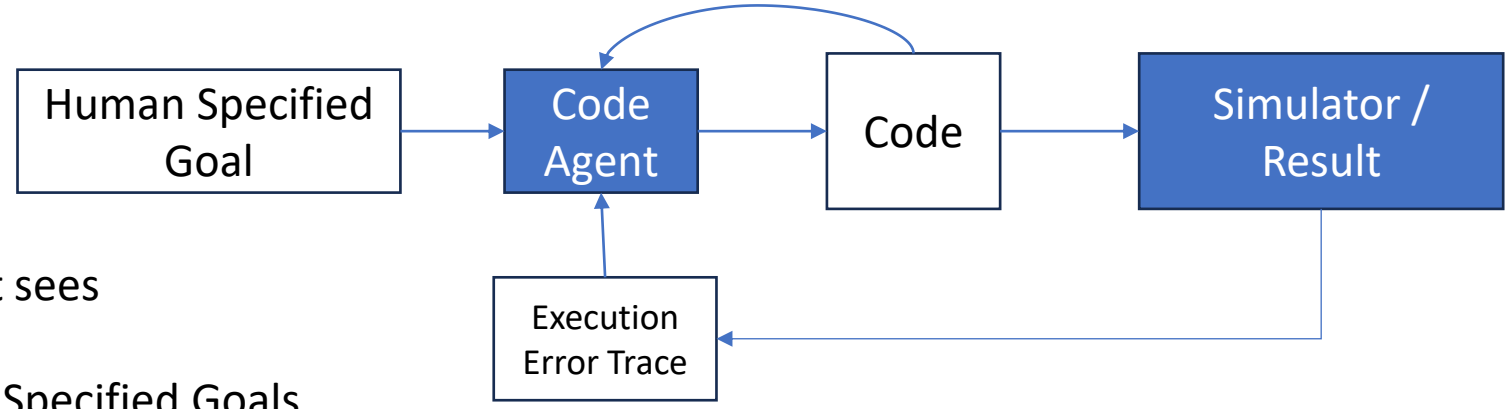
(1) Human Specified Goals,
(2) Execution error Trace
(3) Current Code

Limitations of Agentic-LLM corrective feedback



How about using “Self-Corrective Feedback”?

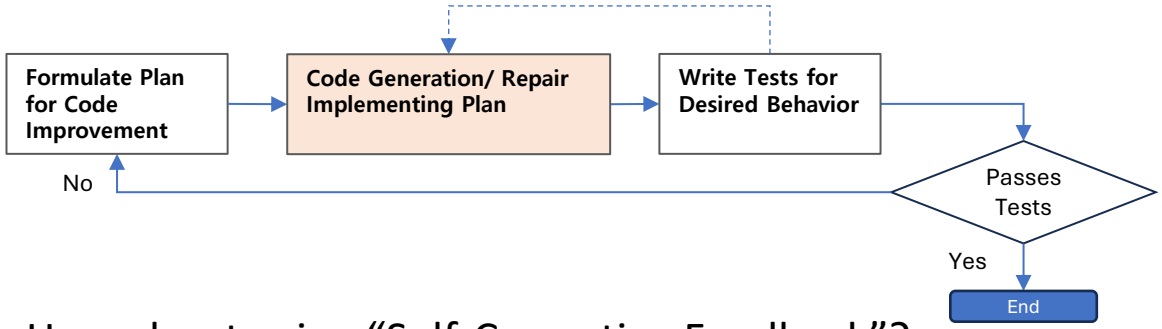
(2) Medium Stuck: No more explicit execution errors, but cannot reach human specified goal



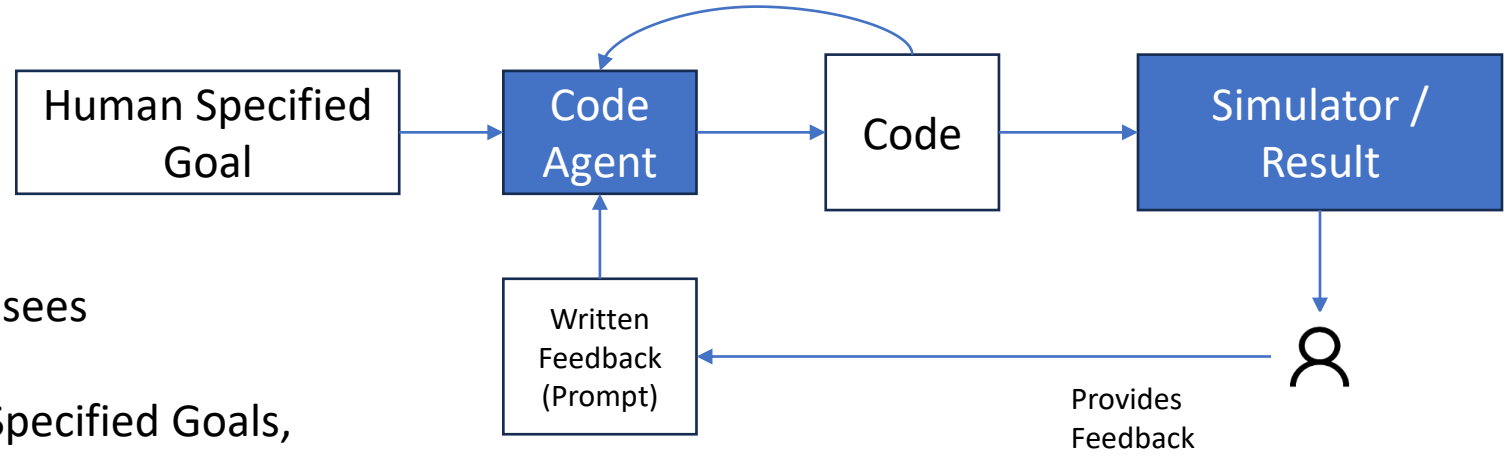
Code Agent sees

(1) Human Specified Goals,
(2) Execution error Trace
(3) Current Code

Limitations of Agentic-LLM corrective feedback



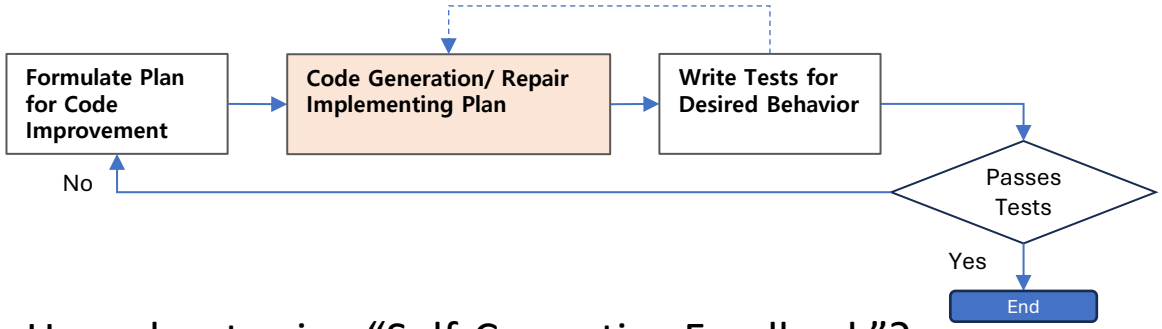
How about using “Self-Corrective Feedback”?



Code Agent sees

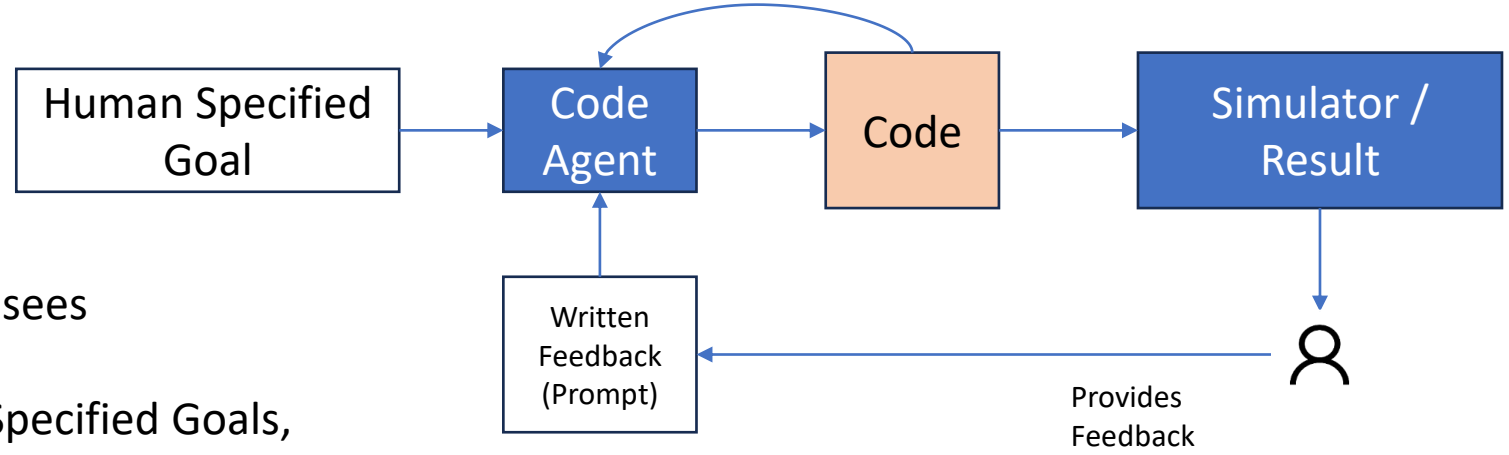
(1) Human Specified Goals,
(2) Current Code (no more execution Errors)
(3) Human Feedback of the simulator results

Limitations of Agentic-LLM corrective feedback



How about using “Self-Corrective Feedback”?

(3) Hard Stuck: The code keeps getting updated, but the simulation result is the same.. Irrecoverable Failure

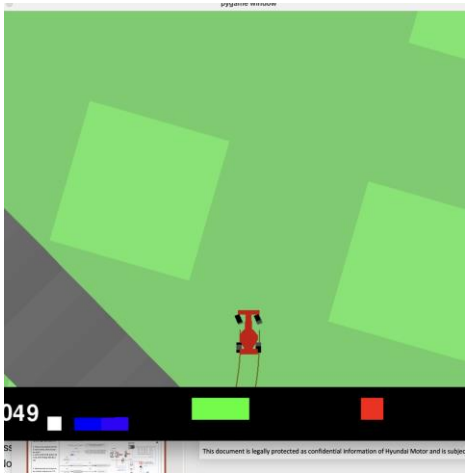


Code Agent sees

(1) Human Specified Goals,
(2) Current Code (no more execution Errors)
(3) Human Feedback of the simulator results

Self-Correction can fail miserably

What the user sees:



What the User instructs:

Rewrite the above function. The problem with this code is the car moves forward but failed to make a sharp turn.

It should look ahead alot more, and slow down alot more when it detects a turn upcoming

```
        else:
            # Slow down significantly before making a sharp turn
            near_forward_pixel = track_mask[int(car_y + direction_vector[1] * lookahead_distance / 4), int(car_x + direction_vector[0] * lookahead_distance / 4)]
            if near_forward_pixel == 0:
                return "brake"
            else:
                # Check further ahead to anticipate turns earlier and stop completely if necessary
                far_forward_pixel = track_mask[int(car_y + direction_vector[1] * lookahead_distance * 2), int(car_x + direction_vector[0] * lookahead_distance * 2)]
                if far_forward_pixel == 0:
                    return "brake"
                else:
                    # If a sharp turn is detected well in advance, stop completely to reassess
                    very_far_forward_pixel = track_mask[int(car_y + direction_vector[1] * lookahead_distance * 3), int(car_x + direction_vector[0] * lookahead_distance * 3)]
                    if very_far_forward_pixel == 0:
                        return "brake"
                    else:
                        return "gas"
```

Code LLM is trying Really hard. But the same exact error keeps happening.

→ The Key Problem: The code generation model has no idea about what it is doing in the actual environment. The only feedback is from the human.

But the human has no idea what's wrong with the code. It's a blind leading the blind situation.

Irrecoverable Failures that cannot be handled by “think step-by-step” or Self-correction

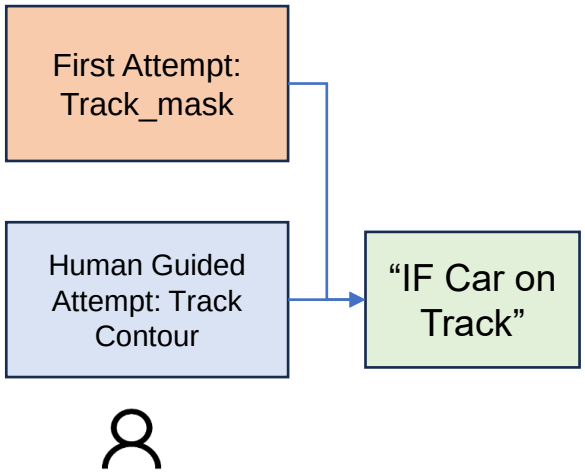
Generated Logic / Code Correctly

Irrecoverable Failure

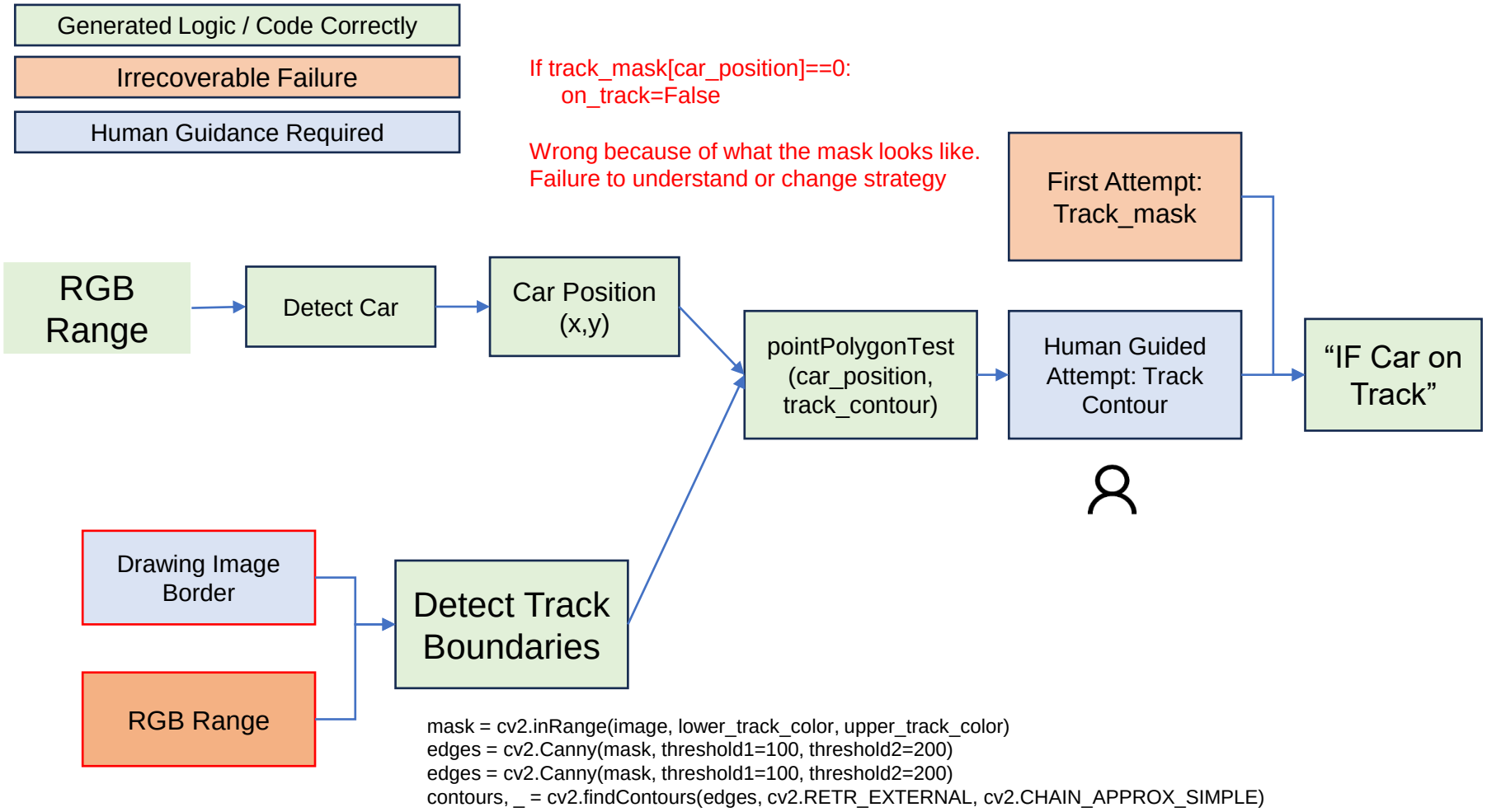
Human Guidance Required

```
If track_mask[car_position]==0:
    on_track=False
```

Wrong because of what the mask looks like.
Failure to understand or change strategy



Irrecoverable Failures that cannot be handled by “think step-by-step” or Self-correction



Irrecoverable Failures that cannot be handled by “think step-by-step” or Self-correction

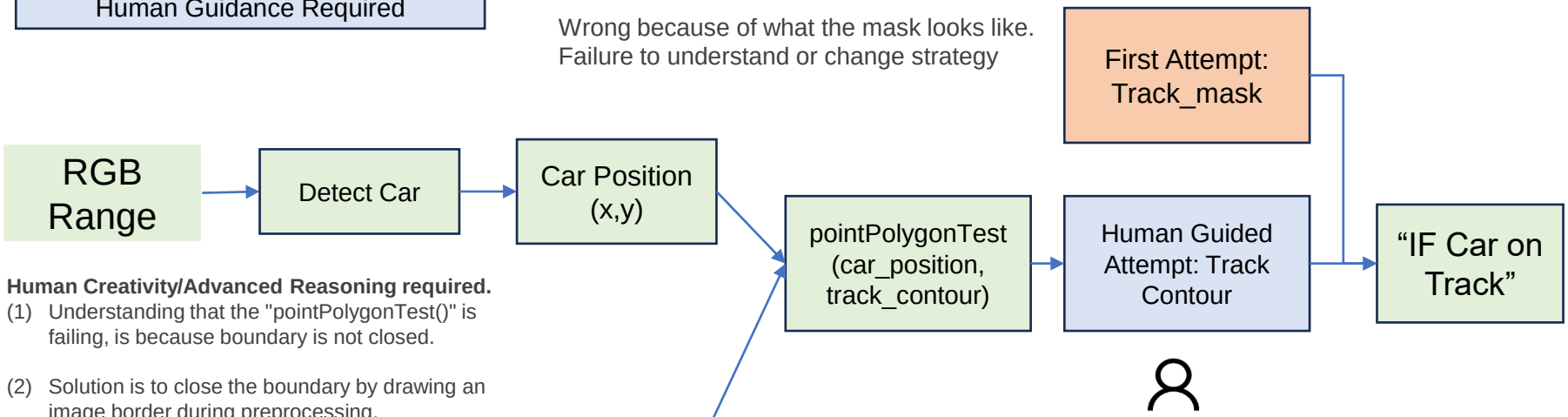
Generated Logic / Code Correctly

Irrecoverable Failure

Human Guidance Required

```
If track_mask[car_position]==0:
    on_track=False
```

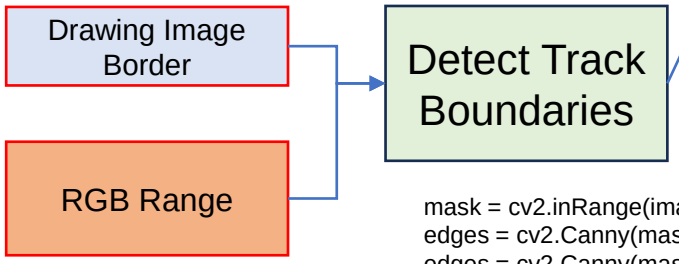
Wrong because of what the mask looks like.
Failure to understand or change strategy



Human Creativity/Advanced Reasoning required.

(1) Understanding that the "pointPolygonTest()" is failing, is because boundary is not closed.

(2) Solution is to close the boundary by drawing an image border during preprocessing.



Advanced Tool use required

```
mask = cv2.inRange(image, lower_track_color, upper_track_color)
edges = cv2.Canny(mask, threshold1=100, threshold2=200)
edges = cv2.Canny(mask, threshold1=100, threshold2=200)
contours, _ = cv2.findContours(edges, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
```

Summary

1. Troubleshooting and Code Generation for Systems and Applications is >10X more complicated than Academic Code generation benchmarks.

2. End-to-End generation is Impossible. Therefore, we need to study the human troubleshooting workflow, to decompose the problem into Tasks that CodeAgent LM can focus on.

3. "Isolating Functional Blocks" and "Understanding Source Code Logic" has dependencies on Understanding the Program's Design Principles, System Causal Graph and Domain Understanding.

4. Debugging and generating new code in physical control systems is extremely difficult. It’s not fix patterns that you can rehash like in constructing certain types of games or UI.

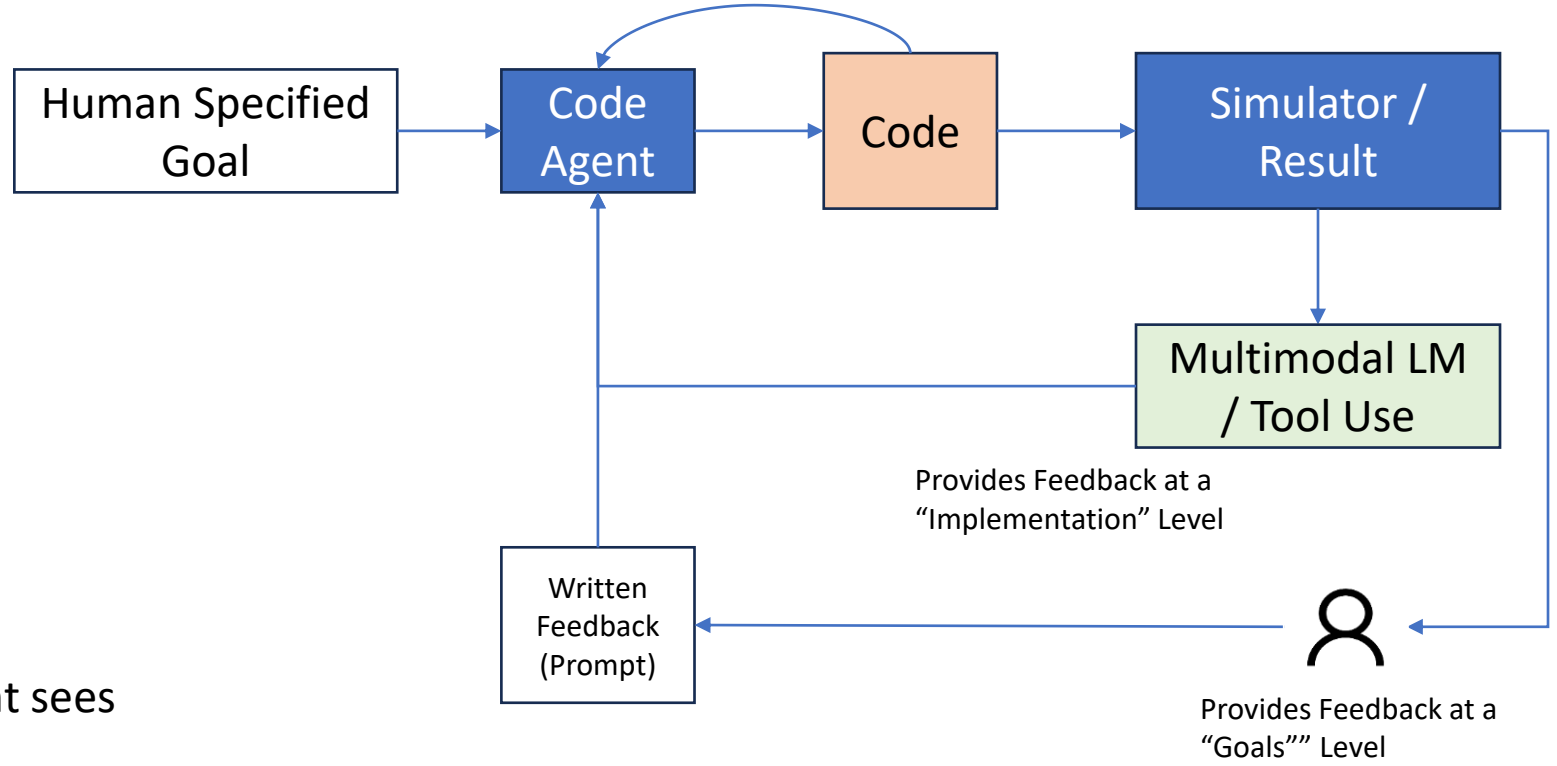
5. "Formulating Plan for Code Improvement" can generally works well at a high (and not very useful level).

6. Logic errors can be due to wrong high level plan, wrong implementation logic, wrong hyperparameters, or wrong input assumptions. Neither the code agent or Human has vision over what is happening.

7. Code Generation with Agentic self-corrective Feedback, and human feedback has irrecoverable failure.

Overcoming the Limits of Agentic-LLM corrective feedback

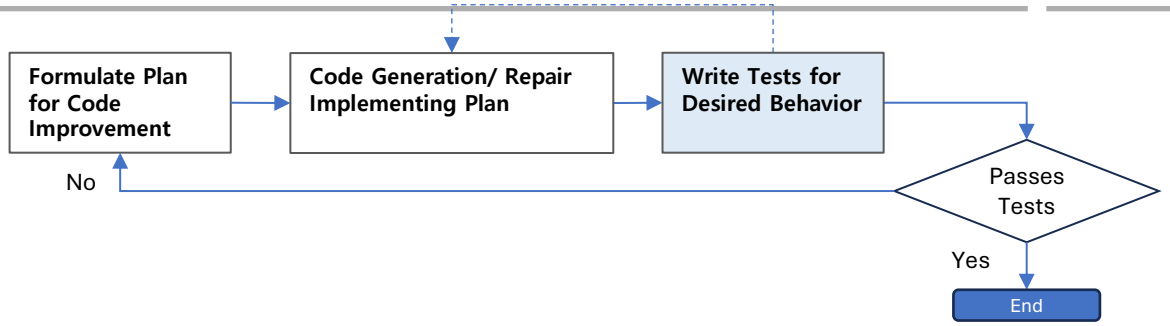
(3) Hard Stuck: The code keeps getting updated, but the simulation result is the same.. Irrecoverable Failure
SOLUTION: Introduce Corrective Feedback with External Help



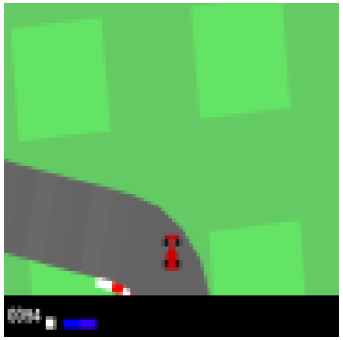
Code Agent sees

(1) Human Specified Goals,
(2) Current Code (no more execution Errors)
(3) Human Feedback of the simulator results
(4) Reconstructed "Image" of the Situation

Step 3: Writing Tests for Desired Behavior



Assumption: We can keep decomposing until task complexity is low enough, such that the Multimodal LM p(y|x) can handle it; potentially with prompt examples or zero-shot



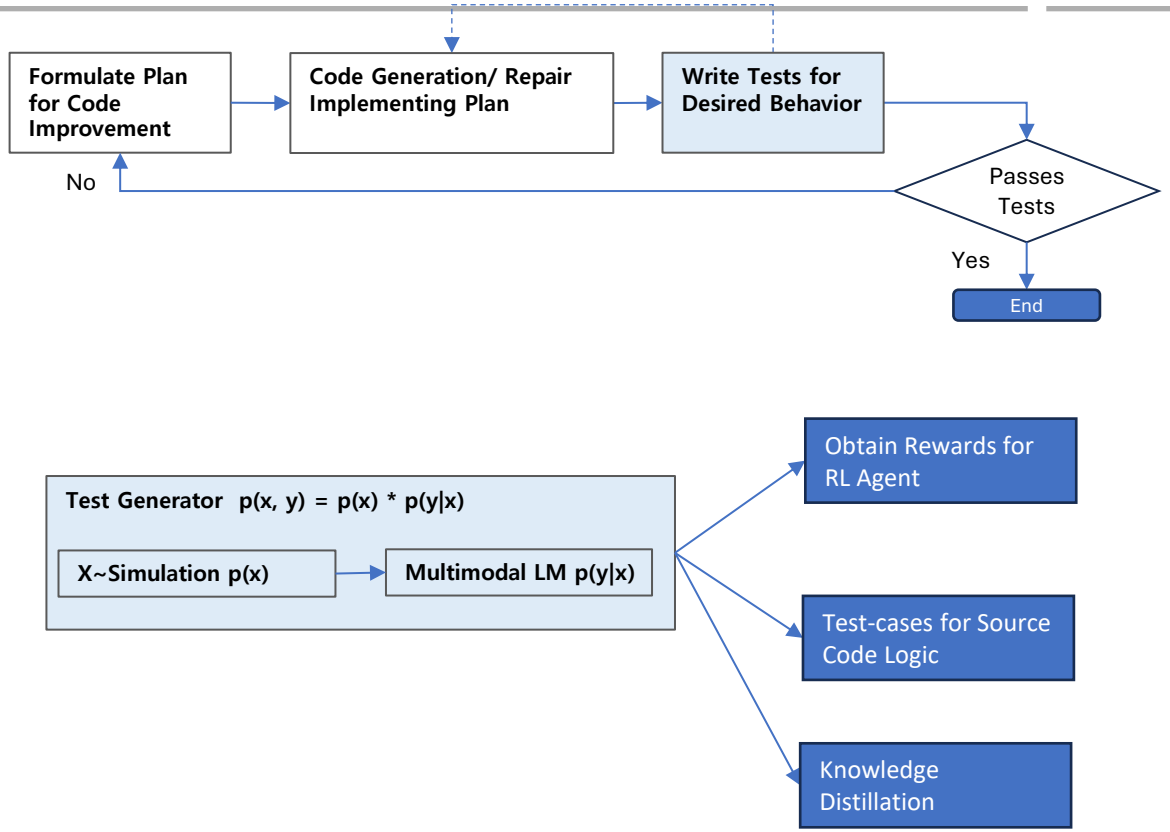
"Is the car on-track, off-track, or spiraling? Answer with only one of the three options."

Vision LM

{On-track,
close-to-edge,
Off-center-left,
Off-center-right,
Off-track,
Left-turn Upcoming
Right-turn Upcoming
Spiralling,}

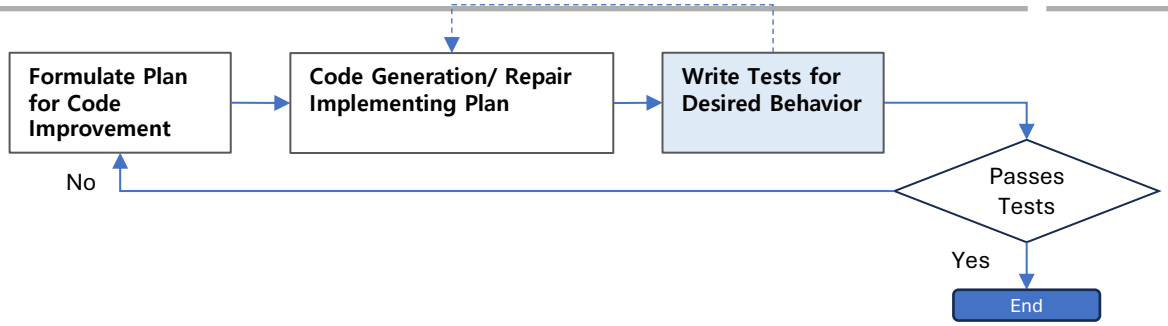
Why not just use a Computer Vision Model?
In the factory, spatio-temporal data from logs, positional coordinates, allows us to reconstruct the "image" of what happened and the sequence of events.

Step 3: Writing Tests for Desired Behavior



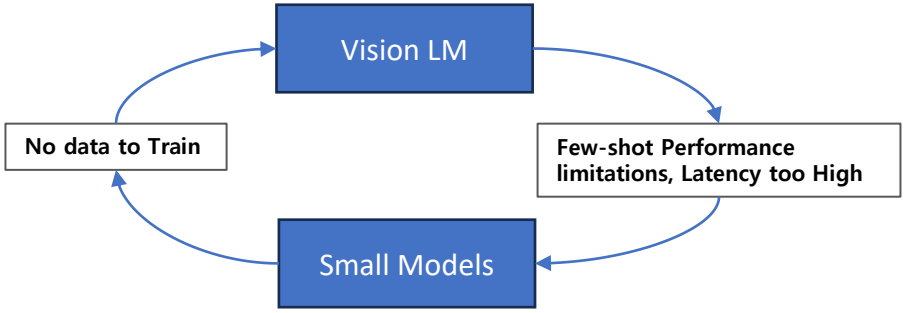
In theory, if we had a way to generate Test cases that cover the distribution of inputs and correct labels, that means we have the ability to generate synthetic data to do many things (RL rewards, training data, knowledge distillation…)….. Sounds too good to be true.

Step 3: Writing Tests for Desired Behavior



Assumption: We can keep decomposing until task complexity is low enough, such that the Multimodal LM p(y|x) can handle it; potentially with prompt examples or zero-shot

Problem: We also can't keep decomposing Tasks: Inference Speed, LLM costs.



We need to address the **latency of large models in generating synthetic Tests and Labels.**

(especially if we do very granular decomposition of tasks)

Summary

1. Troubleshooting and Code Generation for Systems and Applications is >10X more complicated than Academic Code generation benchmarks.

2. End-to-End generation is Impossible. Therefore, we need to study the human troubleshooting workflow, to decompose the problem into Tasks that CodeAgent LM can focus on.

3. "Isolating Functional Blocks" and "Understanding Source Code Logic" has dependencies on Understanding the Program's Design Principles, System Causal Graph and Domain Understanding.

4. Debugging and generating new code in physical control systems is extremely difficult. It’s not fix patterns that you can rehash like in constructing certain types of games or UI.

5. "Formulating Plan for Code Improvement" can generally works well at a high (and not very useful level).

6. Logic errors can be due to wrong high level plan, wrong implementation logic, wrong hyperparameters, or wrong input assumptions. Neither the code agent or Human has vision over what is happening.

7. Code Generation with Agentic self-corrective Feedback, and human feedback has irrecoverable failure.

8. Vision LMs can Providing Feedback (Labels) on the correctness of small functions, but we need to address the latency of Vision LMs for synthetic generation.

Addressing the latency of large models in generating synthetic data.

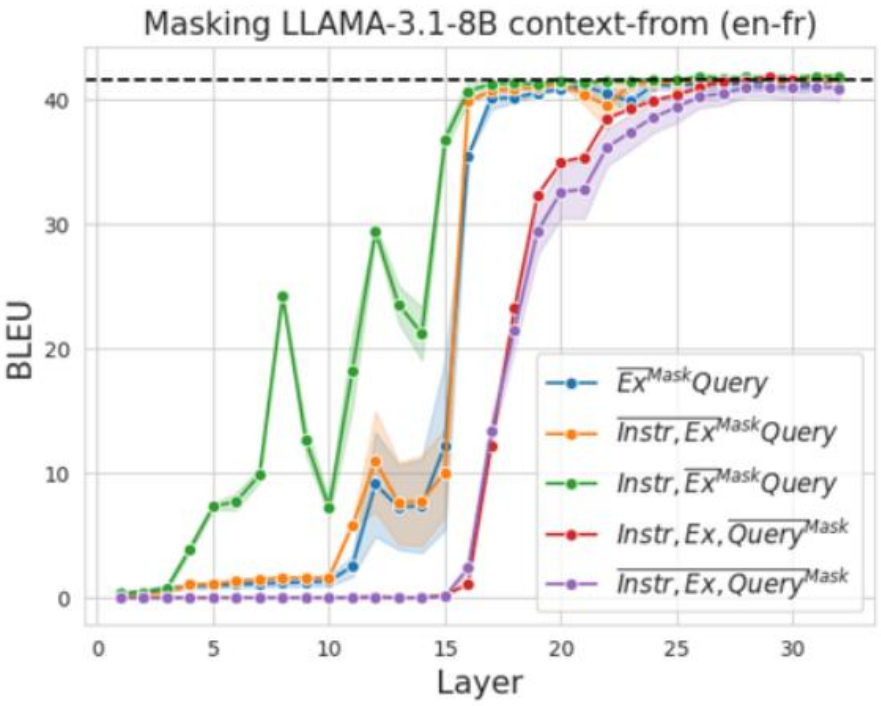
What if we had a way to **Increase Inference Speed by >50%**, without Sacrificing Performance?

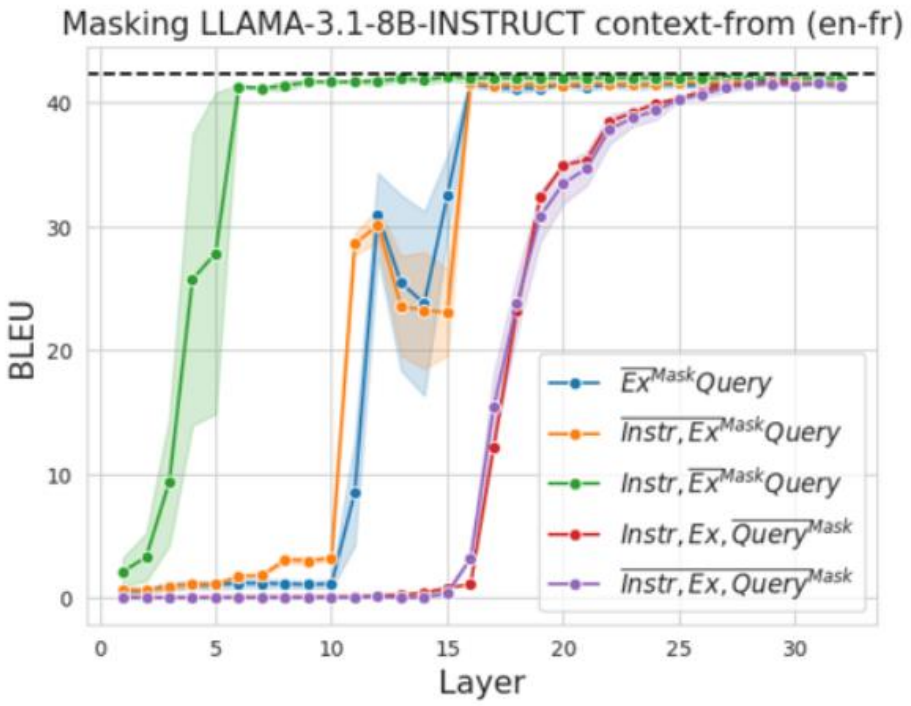
In LLMs, there is a "**Task Recognition Point"**, where the Model no longer needs to perform self-attention over the context (instructions and examples) to perform the Task.

"Where does In-context Learning happen in Large Language Models". Sia, Mueller, Duh., 2024 NeurIPS

Addressing the latency of large models in generating synthetic data.

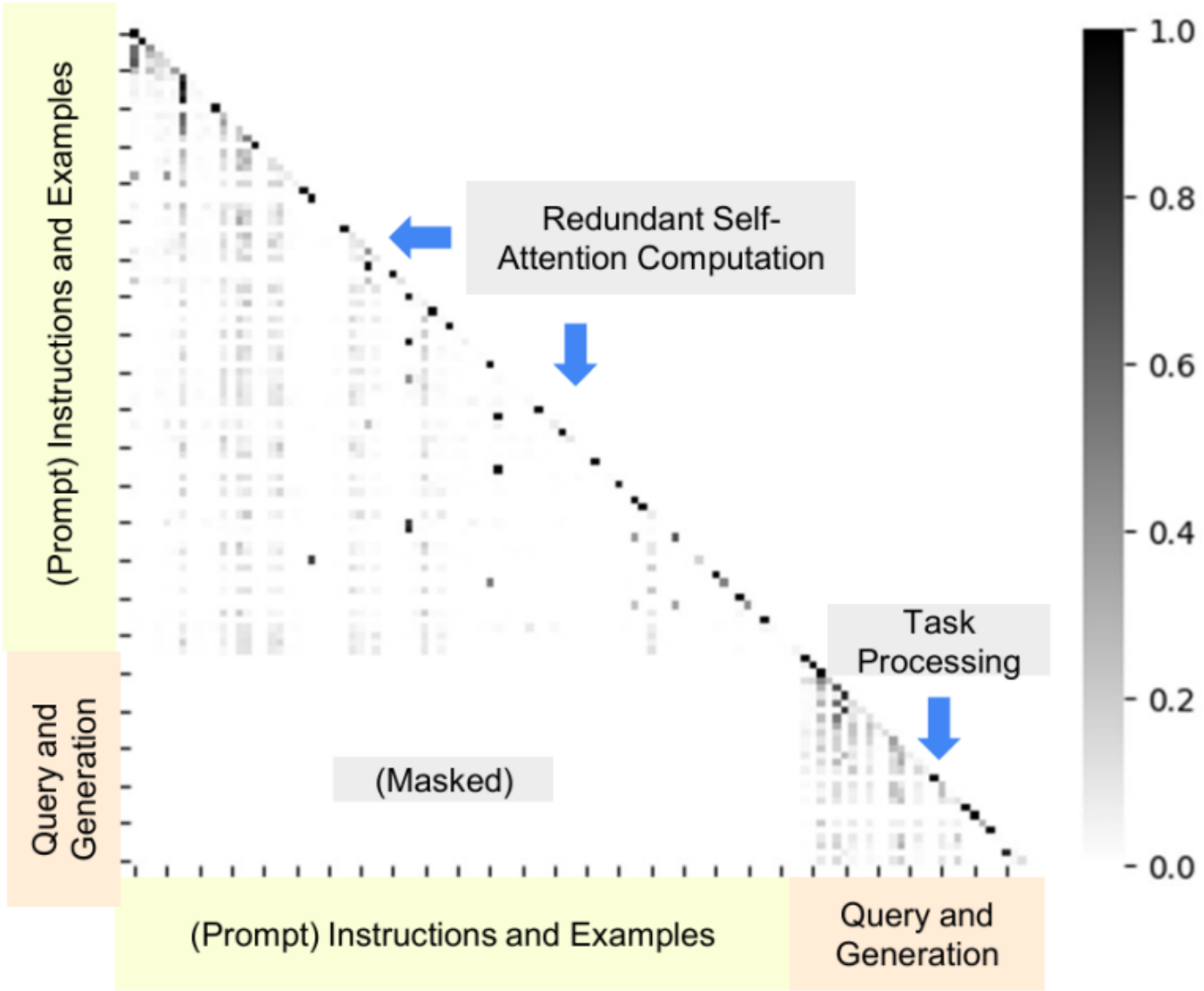
In LLMs, there is a “**Task Recognition Point”**, where the Model no longer needs to perform self-attention over the context (instructions and examples) to perform the Task.





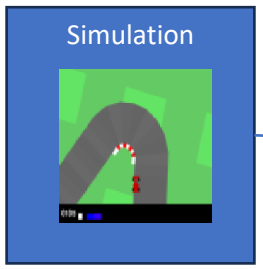
“Where does In-context Learning happen in Large Language Models”. Sia, Mueller, Duh., 2024 NeurIPS

Addressing the latency of large models in generating synthetic data.



"Where does In-context Learning happen in Large Language Models". Sia, Mueller, Duh., 2024 NeurIPS

Implication For Code Generation



Decomposed Tasks for
Over-kill Expensive Model

Multimodal LLM

Labels

{On-track,
close-to-edge,
Off-center-left,
Off-center-right,
Off-track,
Left-turn Upcoming
Right-turn Upcoming
Spiralling,}

"IF detect_curve():"

Option1: CodeLM generates the code for the function, using Multimodal LM as reward

CodeLM

Code: Write Logic for curve detection

Reward function in RL,
(Train for e.g., with PPO)

Option2: Lightweight model is distilled from Multimodal LM which we import as a blackbox function.

Knowledge Distillation

Curve Detector

Code: from curveDetector import detect_curve()

Summary

1. Troubleshooting and Code Generation for Systems and Applications is >10X more complicated than Academic Code generation benchmarks.

2. End-to-End generation is Impossible. Therefore, we need to study the human troubleshooting workflow, to decompose the problem into Tasks that CodeAgent LM can focus on.

3. "Isolating Functional Blocks" and "Understanding Source Code Logic" has dependencies on Understanding the Program's Design Principles, System Causal Graph and Domain Understanding.

4. Debugging and generating new code in physical control systems is extremely difficult. It's not fix patterns that you can rehash like in constructing certain types of games or UI.

5. "Formulating Plan for Code Improvement" can generally works well at a high (and not very useful level).

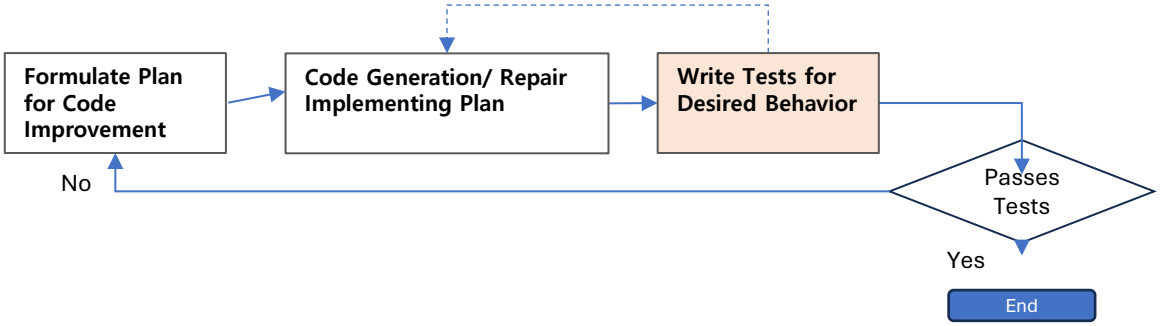
6. Logic errors can be due to wrong high level plan, wrong implementation logic, wrong hyperparameters, or wrong input assumptions. Neither the code agent or Human has vision over what is happening.

7. Code Generation with Agentic self-corrective Feedback, and human feedback has irrecoverable failure.

8. Vision LMs can Providing Feedback (Labels) on the correctness of small functions, but we need to address the latency of Vision LMs for synthetic generation.

9. Vision LMs can potentially be used as a reward generator in RL for Code Generators, or distilled into small models which perform the desired functionality (instead of code).

Tests Generation: Logic Robustness Check



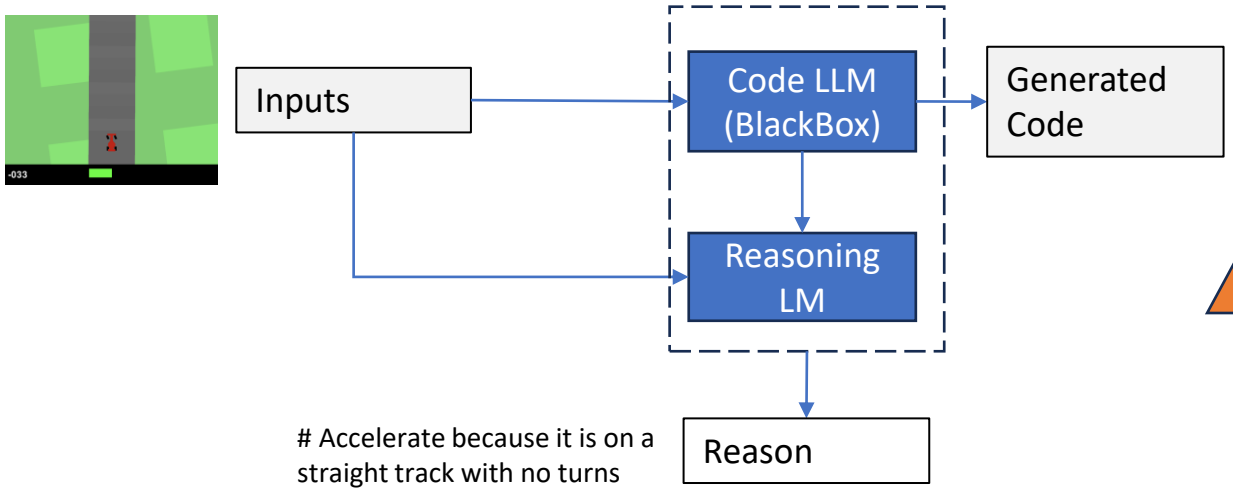
How to Generate Tests to Check expected Behavior based on Plans?

Problem: Code Generated is not Consistent with the Plans

"Intuitively, we would like the provided interpretation **to reflect the *true reasoning process* of the model when making a decision.**

-- Jacovi & Goldberg (2020) Towards Faithfully Interpretable NLP Systems: How Should We Define and Evaluate Faithfulness? *ACL*

Tests Generation: Logic Robustness Check

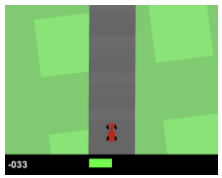


Explainer has its own strong prior as a LM and biases from its own training data.

How can we tell if the Code is Faithful to the Plans and Reasooning?

Logical satisfiability of Counterfactuals for Faithful Explanations. Sia Et al., AAAI 2023

Tests Generation: Logic Robustness Check



Inputs

Code LLM (BlackBox)

Generated Code

Reasoning LM

Explainer has its own strong prior as a LM and biases from its own training data.

How can we tell if the Code is Faithful to the Plans and Reasoning?

Accelerate because it is on a straight track with no turns

Reason

Inputs

CounterFactual Input and Output Generator

Counterfactual Inputs

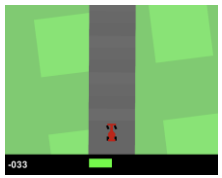
Counterfactual Output

Input: Track with turns

(No Acceleration)

Logical satisfiability of Counterfactuals for Faithful Explanations. Sia Et al., AAAI 2023

Tests Generation: Logic Robustness Check



Inputs

Code LLM (BlackBox)

Generated Code

Reasoning LM

Explainer has its own strong prior as a LM and biases from its own training data.

How can we tell if the Code is Faithful to the Plans and Reasoning?

Accelerate because it is on a straight track with no turns

Reason

Inputs

CounterFactual Input and Output Generator

Counterfactual Inputs

Counterfactual Output

Input: Track with turns

(No Acceleration)

Consistency Check

Counterfactual Input (Tests Cases)

Generated Code

(No Acceleration)

Logical satisfiability of Counterfactuals for Faithful Explanations. Sia Et al., AAAI 2023

Summary

1. Troubleshooting and Code Generation for Systems and Applications is >10X more complicated than Academic Code generation benchmarks.

2. End-to-End generation is Impossible. Therefore, we need to study the human troubleshooting workflow, to decompose the problem into Tasks that CodeAgent LM can focus on.

3. "Isolating Functional Blocks" and "Understanding Source Code Logic" has dependencies on Understanding the Program's Design Principles, System Causal Graph and Domain Understanding.

4. Debugging and generating new code in physical control systems is extremely difficult. It's not fix patterns that you can rehash like in constructing certain types of games or UI.

5. "Formulating Plan for Code Improvement" can generally works well at a high (and not very useful level).

6. Logic errors can be due to wrong high level plan, wrong implementation logic, wrong hyperparameters, or wrong input assumptions. Neither the code agent or Human has vision over what is happening.

7. Code Generation with Agentic self-corrective Feedback, and human feedback has irrecoverable failure.

8. Vision LMs can Providing Feedback (Labels) on the correctness of small functions, but we need to address the latency of Vision LMs for synthetic generation.

9. Vision LMs can potentially be used as a reward generator in RL for Code Generators, or distilled into small models which perform the desired functionality (instead of code).

10. Counterfactual Tests should be generated which check whether the code is faithful to the High level plan approved by the human.

“So can I use LLMs for Code Generation in my Organisation"

Code Generation (actually Code Repair) is Problem-solving in Software Systems. Whether the effort is successful depends on

1. Constraints that the Code needs to operate in.
2. Expectations of the end-users.
3. Quality of the current source code.
4. Availability of well written documentation or APIs.
5. Ability of Programmers themselves to steer the LLM based on strong technical knowledge.
6. Mature tech organization with controls over code quality of LLM generated code to be checked in.