

# Lab 4: Hierarchical Hardware Generation with GenAI

A Comparative Analysis of LLMs for Automated Chip Design

## Objective

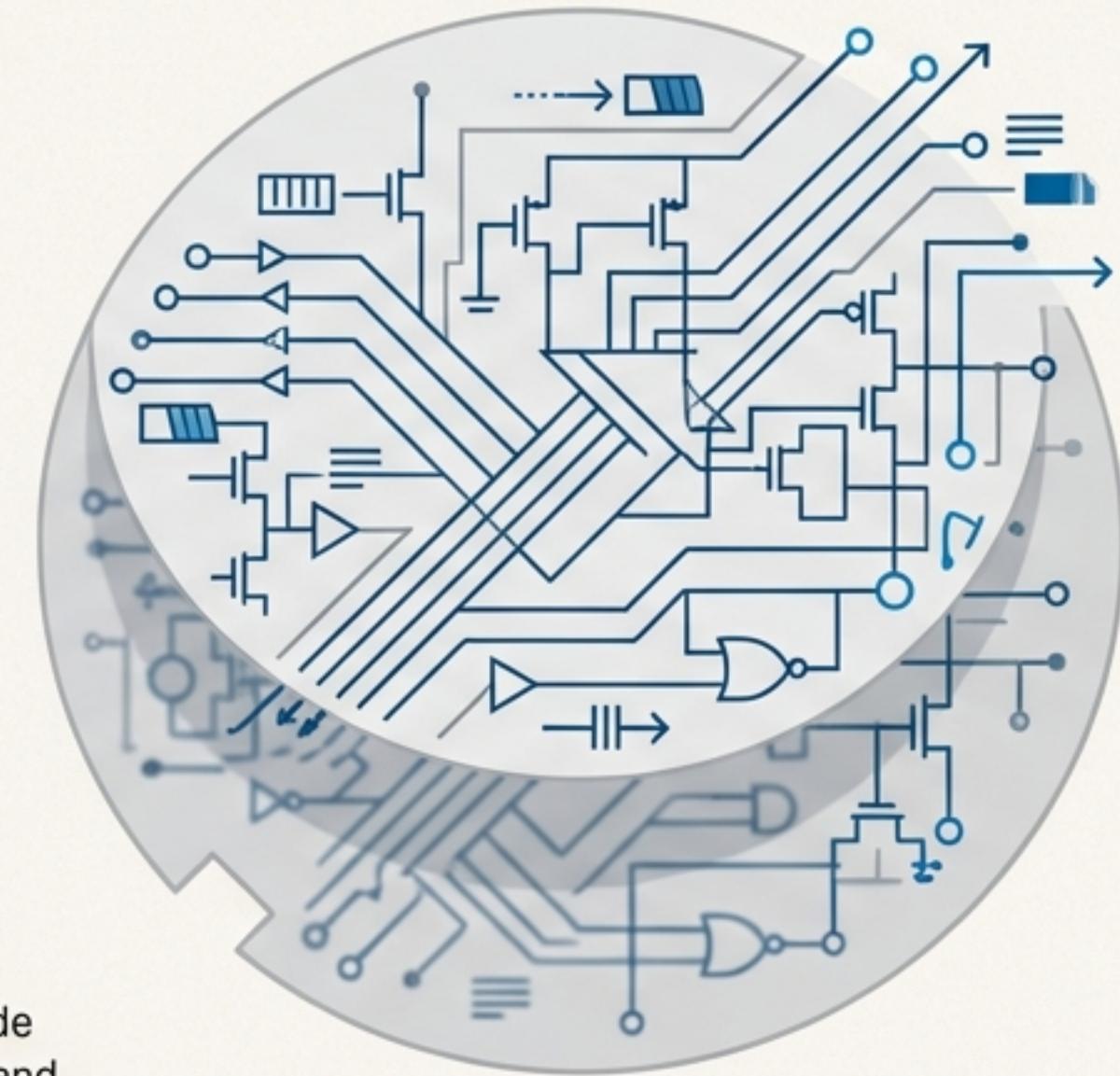
To explore hierarchical prompting techniques for generating hardware description code.

## Core Task

Systematically compare the performance of different Large Language Models (LLMs) on a hierarchical multiplexer design task.

## Methodology

Evaluate selected LLMs on code correctness, generation time, and operational robustness through automated compilation.



Source: Based on the NYU GenAI-based Chip Design course lab environment.

# The Problem: Scaling AI for Complex Chip Design

## The Baseline

Generating code for simple, “flat” hardware modules is a well-understood capability of modern LLMs.

## The Next Frontier

The real challenge is automating complex, hierarchical designs composed of multiple interconnected submodules.

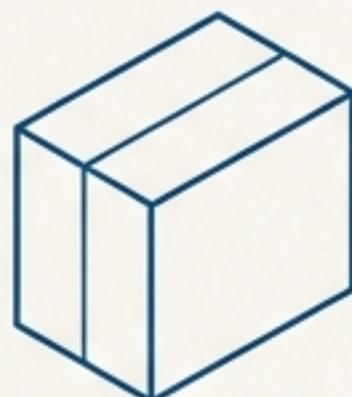
## The Core Question

Is a hierarchical prompting strategy—building modules upon each other—more effective than a single, monolithic prompt for complex systems?

## Experimental Focus

This lab directly tests the viability and performance of hierarchical prompting for a multi-level multiplexer design.

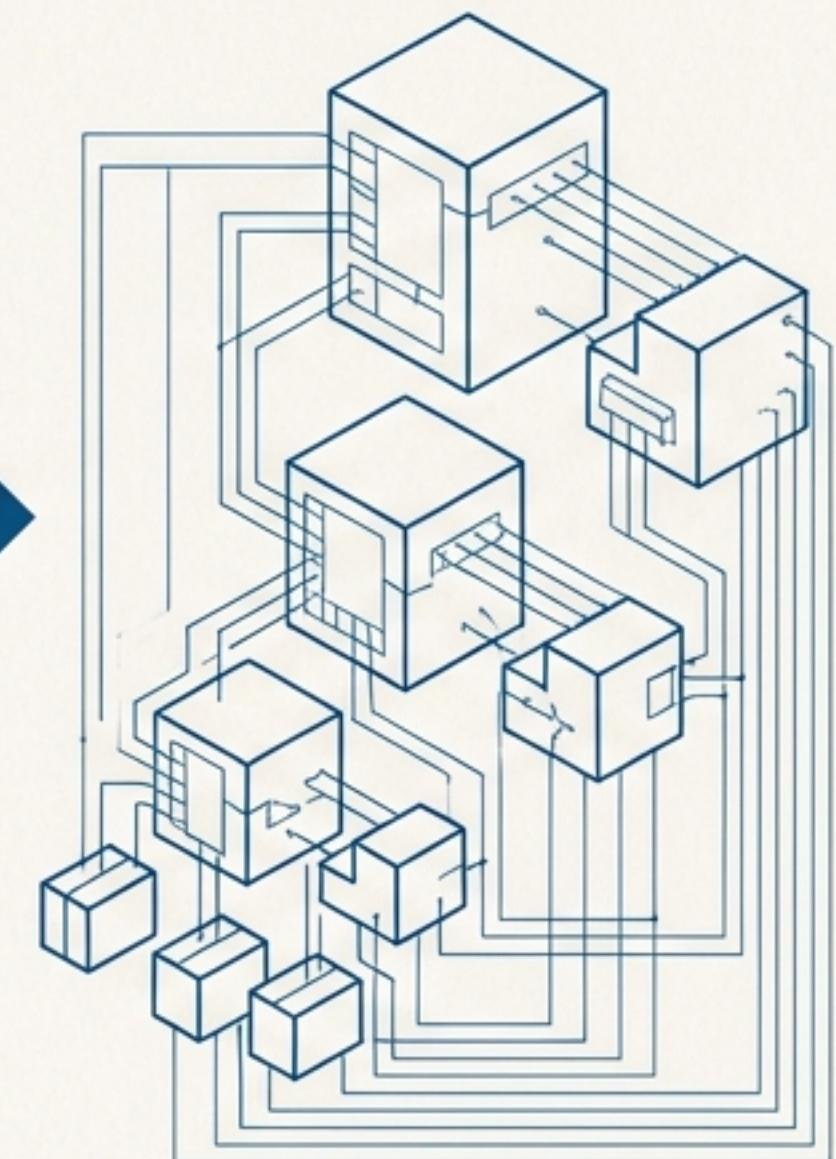
### The Baseline



Flat Module

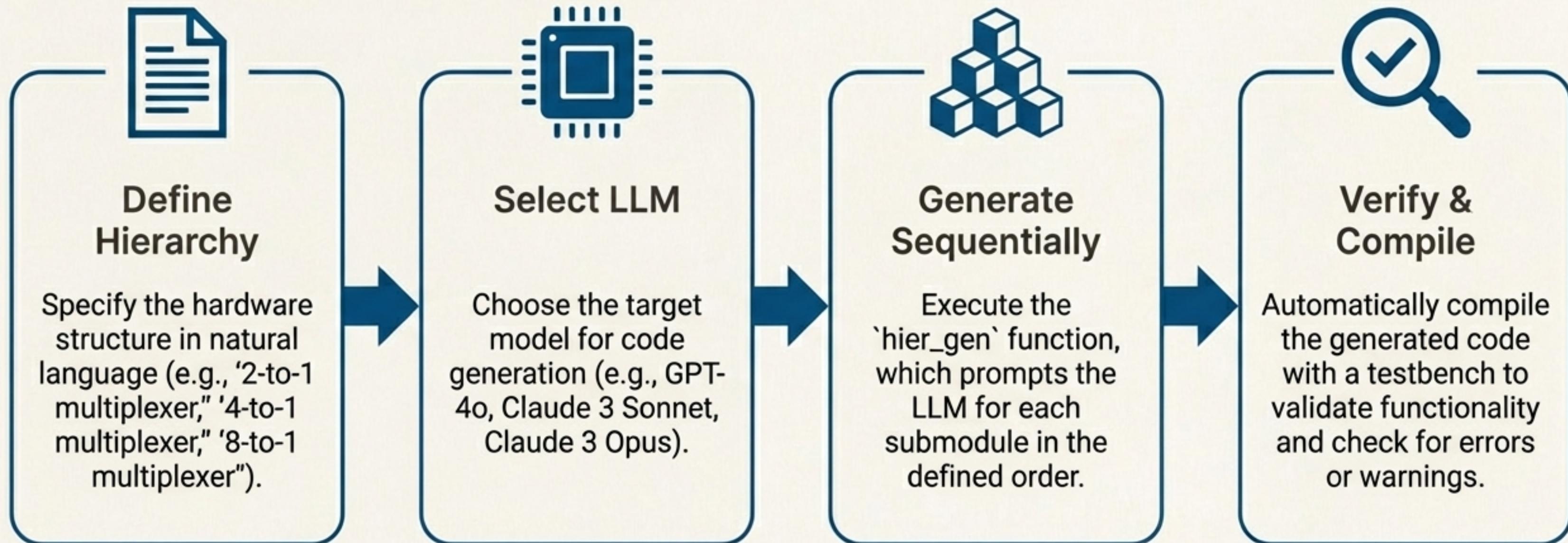


### The Real Challenge



# Our Automated Design Workflow

A Colab notebook with Python and APIs for various LLMs.



# Methodology: A Controlled Comparison

## Experimental Constants

- **The Constant Task**

Generate a series of multiplexers (mux2to1, mux4to1, mux8to1) using a consistent hierarchical structure.

- **The Independent Variable**

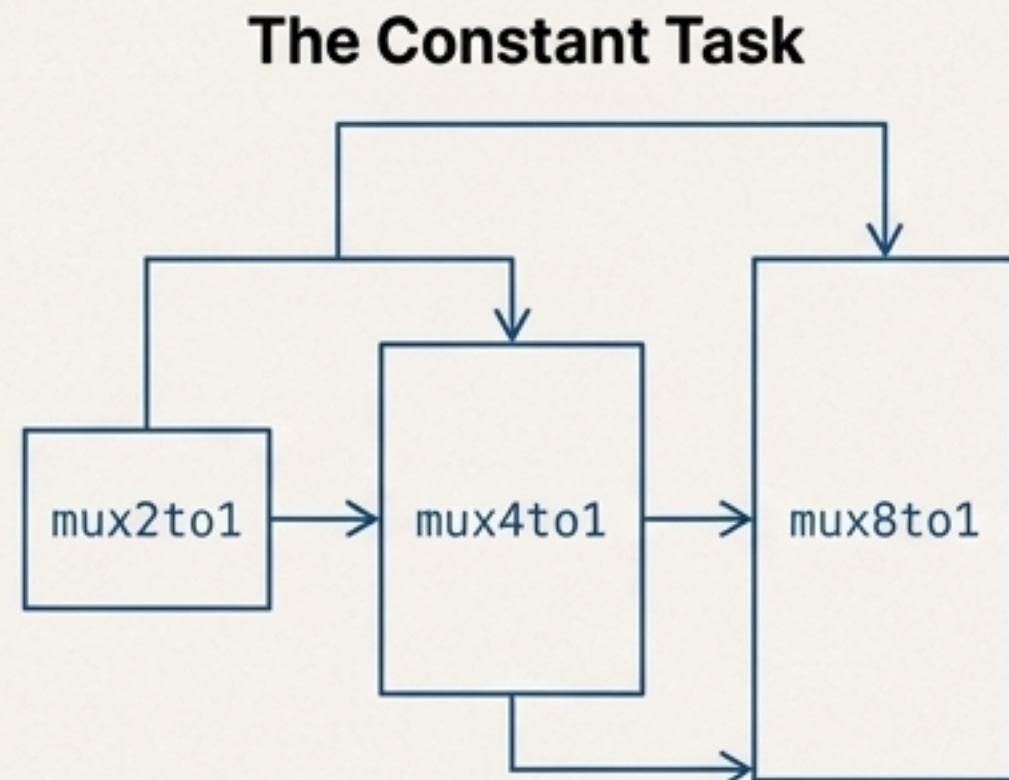
The Large Language Model used for code generation.

- **Key Metrics Recorded**

‘Overall Total time’: Total execution time for the entire hierarchy.

‘Overall Generation Time’: Time spent purely on LLM inference.

**Qualitative output:** Compiler warnings and critical API errors.



## Models Tested

Claude 3  
Sonnet  
(Baseline)

GPT-4o

Claude 3  
Opus

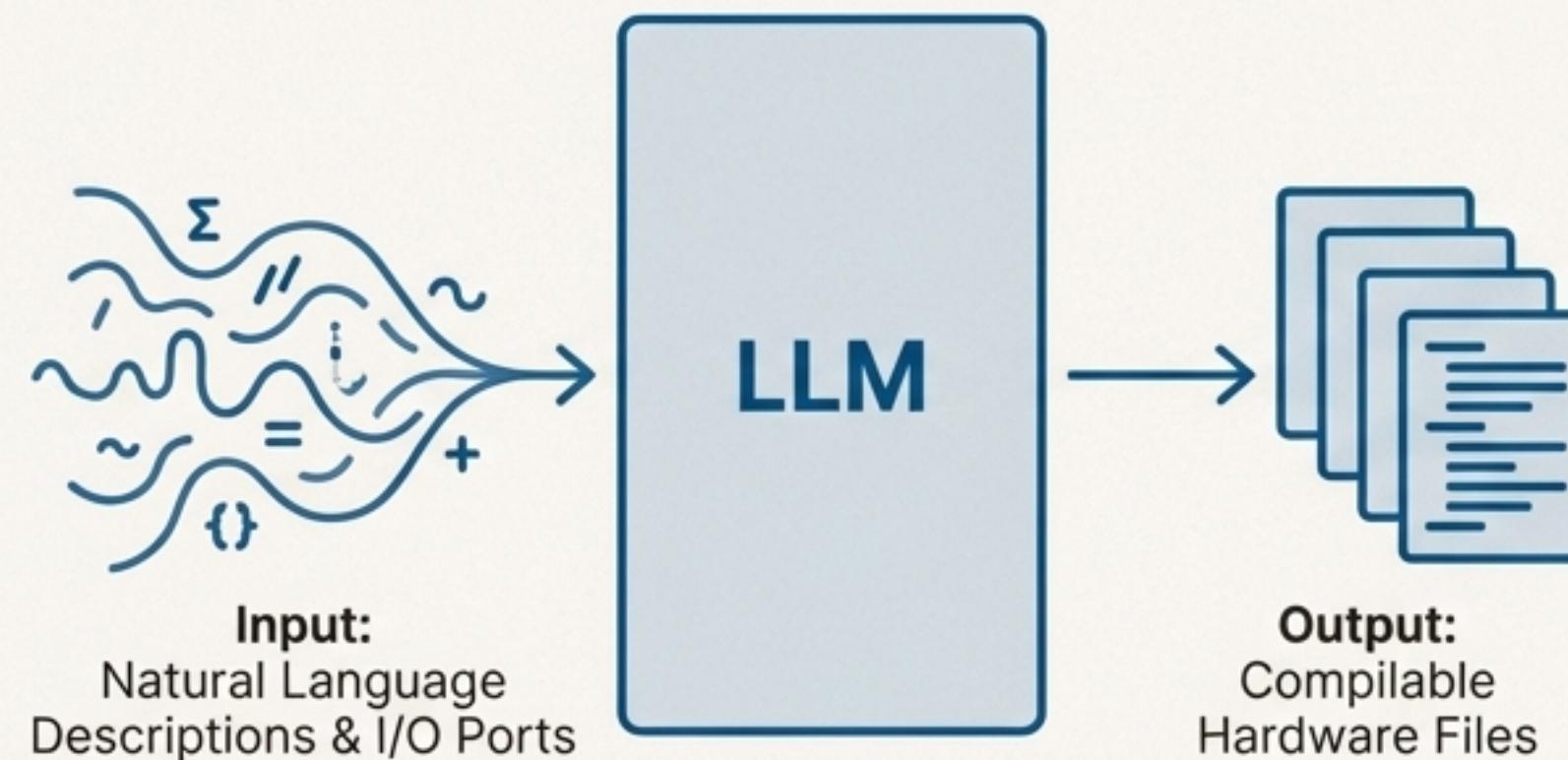
# The Role of GenAI: Translating Language to Logic

## Core Function

The LLM acts as an automated hardware engineer, translating natural language specifications into functional code.

## Input Structure

The function receives a list where each item contains the module name, a natural language description, and its I/O port definitions.



## Output

A complete set of compilable hardware description files for each component in the hierarchy.

## Prompting Strategy

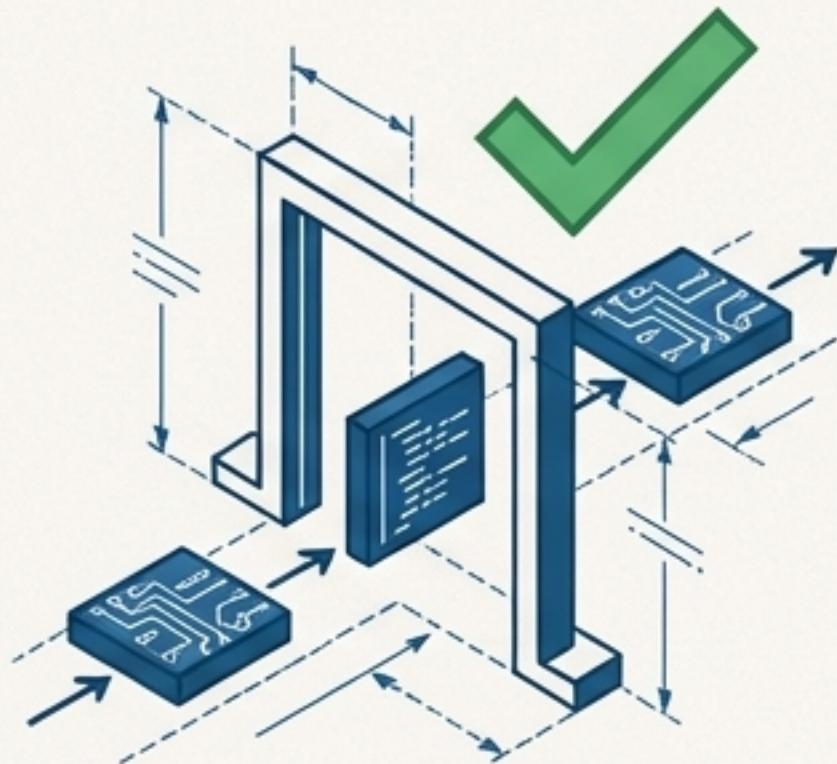
The `hier\_gen` function implements a sequential prompting method, building the design piece by piece.

## Process

The LLM is called iteratively for each submodule, using its description to generate the corresponding hardware code block.

# Verification: How We Measure Success

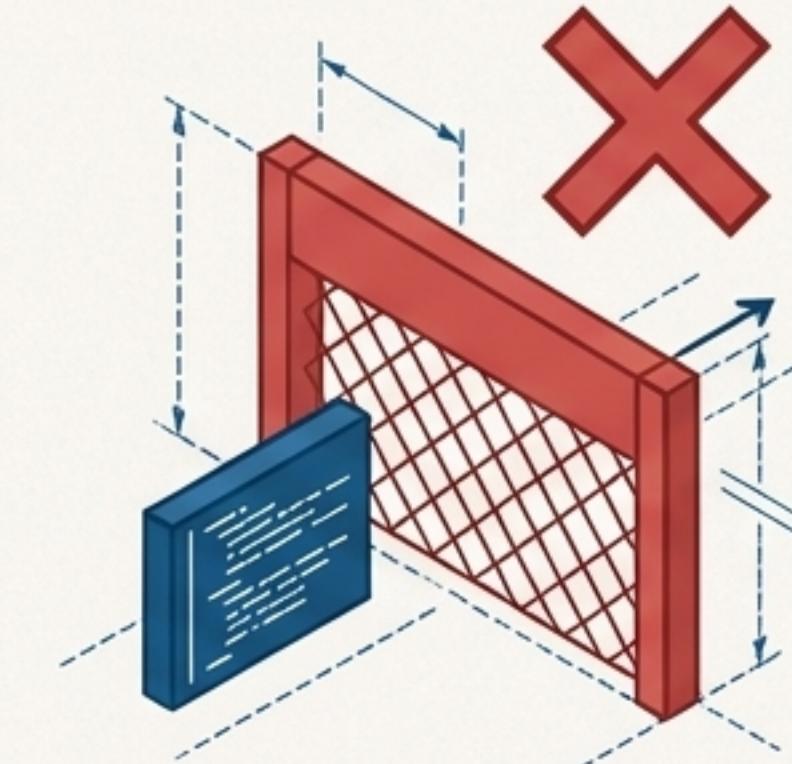
The primary method is the automated compilation of a testbench against each generated module.



- Success Indicator: The process completes the entire hierarchy without fatal compilation errors.



- Quality Indicator: The presence of "Warnings compiling testbench" suggests functional but potentially suboptimal or non-standard code practices.



- Failure Condition: The process is halted by unrecoverable issues, such as protocol-level 'BadRequestError' responses from the LLM service API.

# Performance Showdown: Speed, Success, and a Critical Failure

## Winner: GPT-4o

**Outcome:** Successful generation of all modules.

**Overall Total Time:** ~70 seconds.

## Runner-Up: Claude 3 Sonnet

**Outcome:** Successful generation of all modules.

**Overall Total Time:** ~95 seconds.

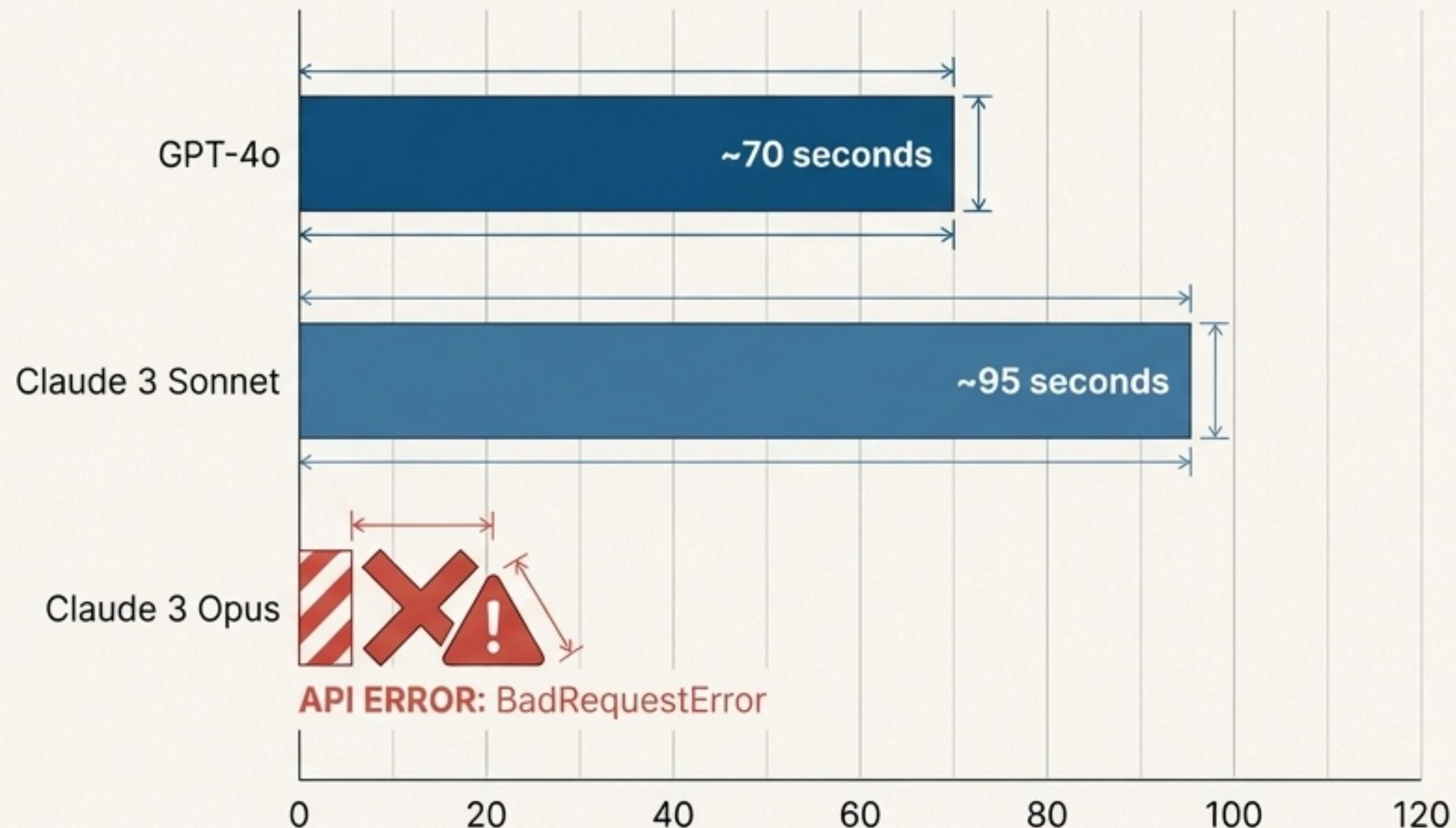
## Failure: Claude 3 Opus

**Outcome:** Failed to generate code.

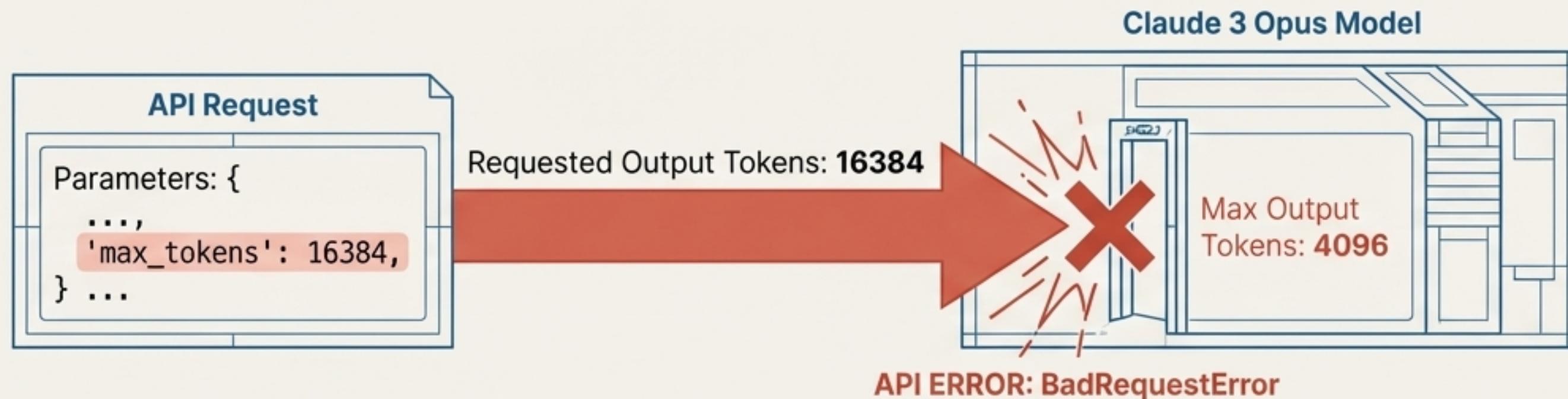
**Error:** Terminated due to a 'BadRequestError' from the API.

**Common Finding:** All successful runs produced recurring 'Warnings compiling testbench,' indicating a shared code quality characteristic.

## Overall Total Time (seconds)



# Anatomy of a Failure: When API Parameters Clash



## The Event

The `hier\_gen` process for Claude 3 Opus failed immediately.

## The Error

A `BadRequestError` with code 400 was returned by the Anthropic API.

```
'message': 'max_tokens: 16384 > 4096, which is the maximum allowed number of output tokens for claude-3-opus-20240229'
```

## Root Cause

The API call requested a maximum of 16,384 tokens, but this specific model has a hard limit of 4,096 output tokens.

## Implication

The failure was not due to the model's reasoning ability, but a fundamental incompatibility in API parameter limits.

# Lessons Learned From the Experiment

## Model Choice Is Critical



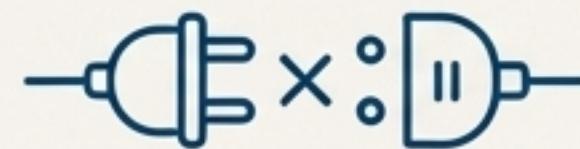
LLMs exhibit significant differences in speed (GPT-4o was ~26% faster than Sonnet) and API limitations.

## Hierarchical Prompting Is Viable



The technique successfully produced a multi-level hardware design, demonstrating its potential for complex tasks.

## APIs Are a Primary Failure Point



The Claude 3 Opus result proves that API parameter compatibility is as crucial as the model's underlying intelligence.

## Verification Is Non-Negotiable



The consistent compiler warnings show that automated testing is essential, as LLM-generated code is not guaranteed to be production-perfect.

# Conclusion: The Path Forward for Hierarchical Generation



## Key Finding

Hierarchical prompting is a powerful technique for complex hardware automation, but its practical use depends on careful model selection and robust, model-aware tooling.



## The Challenge

Performance is not uniform across LLMs. A “one-size-fits-all” workflow is brittle; API differences must be accounted for.



## Future Work

- Investigate automated methods to resolve compiler warnings and improve code quality.
- Develop an adaptive generation framework that dynamically adjusts API parameters based on the selected model's constraints.
- Expand testing to more complex hardware designs to further probe the limits of this technique.