



Exploring and Characterizing Large Language Models for Embedded System Development and Debugging

Zachary Enghardt
zacharye@cs.washington.edu
University of Washington
Seattle, Washington, USA

Richard Li
University of Washington
Seattle, Washington, USA

Dilini Nissanka
University of Washington
Seattle, Washington, USA

Zhihan Zhang
University of Washington
Seattle, Washington, USA

Girish Narayanswamy
University of Washington
Seattle, Washington, USA

Joseph Breda
University of Washington
Seattle, Washington, USA

Xin Liu
University of Washington
Seattle, Washington, USA

Shwetak Patel
University of Washington
Seattle, Washington, USA

Vikram Iyer
vsiyer@uw.edu
University of Washington
Seattle, Washington, USA

ABSTRACT

Large language models (LLMs) have shown remarkable abilities to generate code. However, their ability to develop software for physical computing and embedded systems, which requires cross-domain hardware and software knowledge, has not been thoroughly studied. We observe through our experiments and a 15-user pilot study that even when LLMs fail to produce working code, they can generate helpful reasoning about embedded design tasks, as well as specific debugging suggestions for both novice and expert developers. These results highlight the potential to develop AI assistants to dramatically lower the barrier to entry for working with hardware. To evaluate the capabilities and limitations of LLMs, we develop an automated testbench to quantify LLM performance on embedded programming tasks and perform 450 trials. We leverage these findings to analyze how programmers interact with these tools including their productivity and sense of fulfillment and outline a human-AI collaborative workflow for developing and debugging embedded systems.

CCS CONCEPTS

• **Human-centered computing** → **Empirical studies in HCI**; **Natural language interfaces**; • **Hardware** → *Hardware validation*; • **Computer systems organization** → **Sensors and actuators**.

KEYWORDS

Large Language Models, GPT, Embedded Systems Development

ACM Reference Format:

Zachary Enghardt, Richard Li, Dilini Nissanka, Zhihan Zhang, Girish Narayanswamy, Joseph Breda, Xin Liu, Shwetak Patel, and Vikram Iyer.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

CHI EA '24, May 11–16, 2024, Honolulu, HI, USA

© 2024 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-0331-7/24/05

<https://doi.org/10.1145/3613905.3650764>

2024. Exploring and Characterizing Large Language Models for Embedded System Development and Debugging. In *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems (CHI EA '24)*, May 11–16, 2024, Honolulu, HI, USA. ACM, New York, NY, USA, 9 pages. <https://doi.org/10.1145/3613905.3650764>

1 INTRODUCTION

Large language models (LLMs) such as GPT-3.5, GPT-4 and PaLM 2 have recently made significant strides in code generation for a variety of software development tasks. These LLMs, which have been trained on many code samples, can generate syntactically correct and semantically meaningful code from high-level task descriptions, and rival human experts on certain tasks [5]. Researchers have also begun exploring optimizations such as encoding user intent [24], self-repair for LLMs to correct their mistakes [35], and collaboration between models [13]. Some integrated development environments (IDEs) have already begun to incorporate LLMs into their code editors [31], suggesting AI assistants will play a growing role in software development workflows in years to come.

Despite these efforts, there has been little to no study of using LLMs for physical computing and embedded systems development. While tools like Github Co-Pilot can auto-complete code, they have no link to the hardware, no context about the interconnections of components in the system, or what outputs they would produce. Embedded systems design and programming straddles the hardware-software interface and requires a cross-domain understanding of how devices *interact with the physical world* [25, 51]. Development of embedded systems requires a cyclical process of continuous development and verification, entailing numerous software-hardware integration trials. This iterative approach differs from conventional programming, where outputs are often deterministic and predictable. Embedded system development demands complex, multi-step reasoning and verification to ensure system reliability and functionality. For example, designing a simple wireless temperature sensor requires a physical understanding of the chips and their interfaces, network connectivity, and the sensed signals. These challenges impose a high barrier even for human programmers without hardware experience and have inspired projects

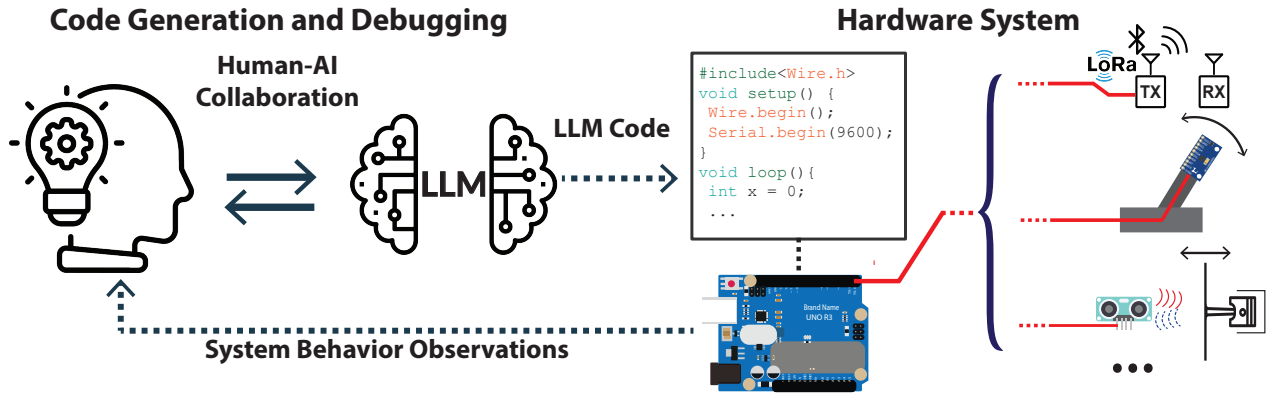


Figure 1: Illustration of human AI co-design for embedded system development where human developers collaborate with LLM-based AI agents to generate code and iteratively debug system behavior.

such as the MakeCode platform and others focused on improving accessibility to embedded systems development [11, 20]. General-purpose LLMs may have a robust representation of these topics or the cross-domain reasoning ability to solve problems in this space. This raises a key question: can LLMs generate reasoning about these concepts, and how can they play an effective role in embedded system development?

In this work, we aim to gain a better understanding of the different strategies employed by developers when using LLMs as part of an embedded system design workflow as well as the limitations of the LLMs themselves on embedded development tasks. We synthesize these findings into a generalizable workflow to enable human-AI embedded system co-development. We find that LLMs help bridge knowledge gaps of low level hardware implementation and offer personalized debugging advice that allowed first time embedded programmers to design fully functional systems. This human-AI approach has the potential to transform embedded systems education and reduce the barrier for researchers across the HCI community to work on hardware and physical computing topics. We summarize our contributions below:

- **Exploration of LLM usage for embedded development.** We conduct an exploratory user study with 15 individuals across experience levels. We find access to GPT-4 in addition to existing internet resources improves productivity for users across all experience levels. In some cases, this improvement was shown in success on complex tasks increasing from 25% to 100%. This allowed users with zero hardware or C/C++ experience to build fully functional two node wireless environmental sensing system in 40 minutes, showing the potential of this approach for education and lowering barriers for working with hardware. We observe that GPT-4 rarely produces fully-working code on the first attempt; however, we find that GPT-4 can provide specific and actionable hardware debugging advice, supporting iterative debugging alongside human developers.
- **Physical Testbench.** We build an open-source, end-to-end evaluation pipeline for real-time LLM-powered embedded code validation with sensor-actuator pairs. We support state-of-the-art LLMs in our framework, including PaLM 2, GPT-3.5, and GPT-4. We use

this setup to execute a comprehensive suite of 450 real-world evaluations involving integrated software-hardware interactions. The data and code are released in this GitHub repository¹. We envision our framework will serve as a valuable asset for the research community, fostering further exploration in developing AI assistants and validation of LLMs for the development of sensor-driven software-hardware systems.

2 RELATED WORK

Capabilities of Large Language Models. State-of-the-art LLMs [9, 36, 45] make use of transformer-based neural architectures consisting of billions to trillions of parameters [46]. These models, trained on Internet-scale text corpora, have shown the ability to perform logic-based reasoning, solve complex problems, have scored highly on language-task benchmarks, and produce text outputs indistinguishable from that of human authors [5, 22, 47]. These LLMs are being used as foundational models in domains spanning consumer health, medicine, education, finance, chip design, and environmental science, among others [4, 22, 27, 28, 41, 44, 53]. Such versatility highlights the potential of LLMs to serve as integral components within interdisciplinary research and application development.

Prompt Engineering. While these LLMs may be powerful tools, research has found that high-quality prompting is required to receive consistently high-quality model outputs [5]. Frameworks and workflows have been developed to optimize LLM responses [48–50, 55, 56, 58]. In large part these prompting techniques include using highly detailed, unambiguous prompts, providing additional context, and asking the model to self-evaluate.

LLMs for Code Generation and Embedded Systems. As LLMs are predominantly trained on Internet-scraped text, including open-source software projects (GitHub, etc.), these models encode a rich understanding of programming languages and code. Prior work has shown language models and AI assistants (e.g., Github Copilot) to generate software, with recent models performing on par with competent software developers [5, 7, 12, 16, 19, 26, 32, 54, 59]. Recent work has also fine-tuned LLMs for code in specific languages showing improvement over the base models [6, 33, 34]. Researchers have also developed code generation prompting strategies such as

¹<https://github.com/anonimwut/llm-embedded-testbench>

self-debugging [8, 43], task decomposition [21, 56], and prompting frameworks [38]. Prior work has evaluated LLM-based code generation in C/C++ (commonly used for embedded software) but has not explored uses in embedded systems with hardware interfaces.

Existing work on hardware-platform code-generation primarily involves IDEs, frameworks and devices to make hardware programming more approachable for novices but have not used LLMs [2, 3, 10, 23, 37]. Other systems have focused on generating hardware description language (HDL) code for field programmable gate arrays (FPGAs), from higher-level languages or design representations [17, 29, 30]. Although some tools [42] exist for LLM-based embedded code generation, and a handful of blog posts have used LLMs for embedded development [40, 52], these resources do not conduct a rigorous systematic evaluation of state of the art language models for embedded development and debugging or methods of interfacing directly with hardware.

3 USER STUDY

We conduct an exploratory user study to understand how developers of different skill levels interact with LLMs during embedded development. We also investigate changes in productivity and their impressions of the technology.

3.1 Methods

Participants. We recruit 15 users (age $M = 23.3$ years, $SD = 2.93$ years) comprised of undergraduates and PhD students, faculty, and professional software engineers. While all participants had an electrical engineering or computer science background, their embedded systems experience was evenly distributed from 0 to 4+ years as seen in Fig. 2a.

Setup. Studies were conducted in a quiet lab environment. Participants were given a laptop with the Arduino IDE and all necessary device drivers installed. A web browser with ChatGPT using the GPT-4 0314 model was open by default, and participants were told they could open additional tabs and use a search engine of their choice. Participants were also provided a kit of components with microcontrollers, sensors, actuators, wireless transceivers and jumper wires. The specific components are enumerated as part of the procedure.

Procedure. We ask participants demographic questions to gauge their experience with programming, C/C++, and embedded systems. Participants were assigned a unique ID number, used to select a row in a counter-balanced partial Latin square to determine the order of the tasks and conditions. Each participant completed a simple warmup task and four graded tasks. We recorded the laptop screen, a top-down video of participants' interactions with the hardware components, their chat history with ChatGPT, and their final submitted code. After completion, we conducted semi-structured interviews with quantitative questions based on the NASA Task Load Index (TLX) [18] and SPACE Framework [14]. Participants were encouraged to elaborate on their responses to the quantitative questionnaires.

Study Design. The study was a 5x2 between-subjects design. Participants were timed and completed all four tasks in a random order and with varying levels of GPT-4: no access, access to GPT-4, and access to GPT-4 with some preliminary workflow suggestions based

on our initial experience using GPT-4 for programming tasks. The suggestions provided to participants included guidelines to write detailed prompts and instructions to provide information about errors in follow-up prompts if the initial code did not function correctly. The guidelines can be found in Appendix A. We used a subset of the Latin square such that participants never had access to GPT-4 with our workflow guidelines before having access to GPT-4 alone. We designed the tasks to be similar to assignments from an introduction to embedded systems class, and they included:

- (1) **Warm-Up** (untimed and ungraded, ~5 minutes): Print "Hello, World!" to the Serial console and blink an LED.
- (2) **Environmental Sensor** (40 minutes): Connect the BME280 environmental sensor to the Arduino Uno (with already attached RFM9x LoRa radio) and transmit sensor readings to a second Arduino Uno with LoRa radio.
- (3) **Heart Rate Monitor** (40 minutes): Connect the analog photoplethysmography (PPG) sensor to the nRF52 (Arduino Nano 33 BLE) and transmit sensor readings to a second nRF52 over Bluetooth Low Energy.
- (4) **Timer Interrupt** (20 minutes): Connect the piezoelectric buzzer to the Arduino Uno and generate a tone of 440 Hz using built-in hardware timers.
- (5) **RTC Wakeup** (20 minutes): Configure a PCF8523 RTC (real-time clock) to wake an Arduino Uno from sleep mode to flash the LED before going back to sleep.

Analysis. We reviewed users' code and graded it using the rubric shown in Appendix B to award partial credit for completing each sub-task. This rubric was not provided to participants. We summarized users' experience with the study through descriptive statistics derived from these scores and participants' responses to the survey instruments, using a Student's t-test to determine significance. We reviewed interview transcripts for thematic analysis and extracted quotes to substantiate those themes.

Ethics. Participants were recruited through fliers placed around a university campus and through word-of-mouth. Participants were offered a \$20 USD gift card for participation. Prior to recording, participants were given an information sheet explaining the types of tasks they would perform and the types of data to be collected, including overhead video, screen recording of the development computer, and transcripts of post-study interviews. The protocol was approved by our Institutional Review Board.

3.2 Task Performance Results

Participants scored highest on the Timer Interrupt task ($M = 66.67\%$, $SD = 43.46$), and worst on the Heart Rate Monitor task ($M = 36.67\%$, $SD = 23.92$); the variance suggests the Heart Rate Monitor task was difficult across experience levels, while the Timer task was easier for experienced users. The Heart Rate Monitor task required multiple components that added points of failure. P10 cited how this complexity made it challenging to work with GPT: "*The first task [Bluetooth]... I'm not sure if it would have been easier with GPT or not... I had trouble like with everything on that... it was hard to explain the system... that you're building here to GPT.*"

Participants performed best with access to GPT-4 and our workflow guidelines, significantly better than without GPT-4 ($M = 65.48$ vs $M = 34.21$, $t(38) = 2.73$, $p < 0.01$), and better than using

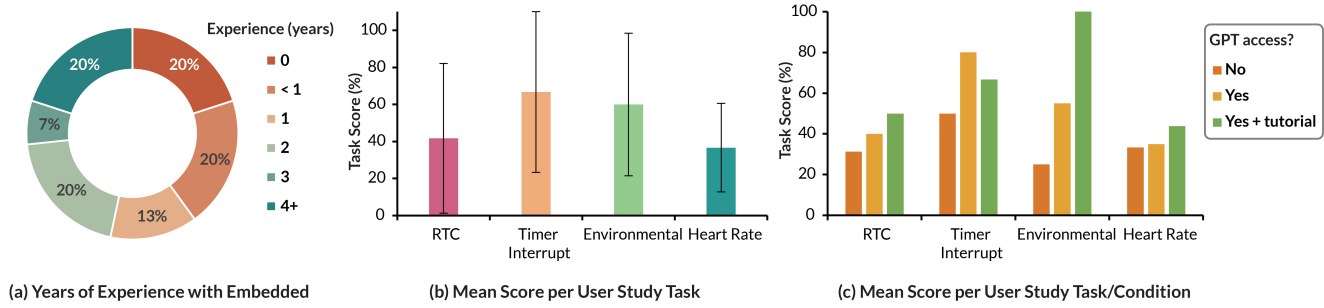


Figure 2: Left: Distribution of user study participants' experience with embedded systems. Middle: Mean scores obtained by participants on each task. Right: Mean scores obtained by participants on each task split by condition.

only access to GPT-4 with no significance, as seen in Fig 2c. In the Environmental Sensor task all 5 participants with GPT-4 and our workflow scored 100%, and half had little to no embedded systems experience. Participants without GPT-4 scored 25% on average. We attribute much of this success to our tutorial. For example, using clues from P7's detailed description of the Arduino output, ChatGPT explained Arduino's `sprintf()` does not support formatting floats. ChatGPT then corrected its code, using `dtostrf()`, enabling P7 to complete the task when otherwise "I don't think I would have been able to do the LoRa sensor in 40 minutes."

3.3 Interaction Observations

(1) **Many participants commented on how GPT-4 was useful for generating starter code.** Similar online examples as P9 put it, "Instead of searching for example code and then modifying that... you know it'll generate this code [that's] more customized to what you want." From an efficiency standpoint, P6 suggested that, "...the one good thing about ChatGPT – because when you search on google there are a lot of things you cannot use, and you have to click on them one by one, and you don't know what you want – maybe he [chatgpt] provides failing code, but you know it's what you want, and you can modify it based on that." The ability to get programmers started, and especially those that are inexperienced, with bootstrapped code is extremely powerful and motivating.

(2) **Detailed error feedback improved LLM self-correction.** While code generated by GPT-4 often compiled successfully, the system behavior was often incorrect correct on the first attempt. For example, on the buzzer task, P11 observed that the initial program produced a tone at a lower frequency than desired. They relayed this information back to GPT-4, inputting "The tone desired is "A 440Hz", the tone given [in the program] resulted in a lower tone". GPT-4 then identified an error in the prescaler configuration for the hardware timer used to generate the frequency signal, and updated the program to produce the correct tone. We notice a pattern where clearly stating difference between the current system behavior and the expected result enables more effective model self-correction, even when the participant may not be aware of the exact source of the issue in the software.

(3) **Experienced programmers asked GPT-4 for smaller code segments.** For example, a function to read a sensor, a function to send wireless transmissions, and then synthesized them together. P7, an experienced programmer with zero hardware experience, provided this context: "Hello Chatgpt I want help writing code for

a big task, we will break it down to two tasks. Let's first do the first task then we will do the next task which I will ask you to incorporate the first task along with the results of the second. Does this sound good? ... so our first task will be to read the pressure, temperature and humidity data from our BME280 sensor which is connected to our arduino uno board... great the first task is done and we have correctly got the output we wanted now lets start the second task which we will use the first task to get started. For this second task we are going to have another arduino uno board which has a LoRa radio RFM9x which has the corresponding ports..." By working with smaller tasks, GPT-4 was able to more effectively debug and iterate.

(4) **GPT-4 offered specific and actionable hardware debugging advice.** This included checking pin connections and wiring and was helpful to all participants, especially those with no hardware experience that struggled to interpret advice found online: "I saw a lot of the words [online] like, you have to connect to a ground I didn't really know what that meant... I figured I'm assuming that the colors of the cables don't mean anything" (P7).

In one particular case, ChatGPT went beyond the suggestion to simply check the connections and even provided a custom debugging script to scan for I2C addresses and verify that the sensor could be found.

Person: "I tried to run the code with both 0x76 and 0x77, but both print "Could not find a valid BME280 sensor, check wiring!""

ChatGPT: "If your Arduino says it cannot find a valid BME280 sensor, it might be due to one of the following reasons:

- (1) **Wiring issue:** Double-check your wiring connections as explained previously. Make sure the SDA and SCL connections are correct. The SDA pin should connect to A4 and the SCL pin to A5 on the Arduino Uno. Also, ensure the sensor is correctly powered with 3.3V and its ground is connected to the Arduino's ground.
- (2) **Incorrect I2C address:** The I2C address of the sensor can typically be 0x76 or 0x77. If neither of these addresses is working, your sensor might have a different address. You can use an "I2C scanner" sketch to find out the exact address of your sensor. Here is an example of an I2C scanner code: [I2C scanner program]"

(5) **Context switching between code editor and LLM.** Context switching is an inherently significant portion of any workflow, and it is certainly a large part of coding: programmers will very frequently switch between their code editor and resources on the Internet, such as documentation or forum discussions. Our study

was designed to mimic the workflow of a programmer switching between their code editor and their Internet resource. LLM integration into code editors is an active area of research and development [39]; however, switching between windows currently remains a common practice.

3.4 Exit Interview Feedback

We evaluate participants' responses to our subjective questions. Through our TLX questionnaire, we learned about participants' perceived load versus GPT-4 access condition. The average perceived load with access to GPT-4 and our workflow was lower than the perceived load with GPT-4 only (mean load 3.25 vs 3.83, $t(38) = 1.49$, $p = 0.1$), and also lower than no GPT-4 (mean load 3.25 vs 4.67, $t(35) = 4.35$, $p < 0.01$). Responses to the SPACE framework questionnaire also sparked interesting open-ended discussions. Users were largely split: some suggested that ChatGPT could aid learning, while others hinted that it could hinder learning. P4 did not feel very fulfilled *"...because, like, I feel like I didn't [do any] work. You know, like it wasn't something that I did. It's something that ChatGPT did for me... it was cool to see that it was working, but not nearly as cool as if I had done it personally."* On the other hand, P8 pointed out *"...it's fulfilling in different ways: ...instead of understanding the domain, you become more fulfilled in that you get the task to work... GPT handles some tedious things really well, and allows you to better understand some of the other aspects of the task."*

4 TOWARD AN AUTOMATED PHYSICAL TESTBENCH

The first step toward developing robust AI assistants for physical computing is a method to interface the physical hardware and development environments with LLM models and evaluate their outputs. We develop an evaluation framework for benchmarking LLM performance on hardware-in-the-loop embedded systems development tasks and implement an end-to-end pipeline for physical verification and fully automated, real-world testing of LLM embedded code generation. This is crucial because it enables programmatic testing of specific development tasks to identify the limits of what these models are capable of and presents a platform for iterative design. **Experiment Setup.** To develop an interface with LLMs we leverage the key observation that given an example format within prompts, all 3 of these language models can *very reliably* generate code to print data over serial. This capability allows us to create programs that produce text debugging output we can verify. We perform initial tests on a number of prompts and find this method works consistently. To verify the code, we compare these serial outputs to outputs from a separate device running verified human-written programs. We construct physical hardware with sensor-actuator pairs described in detail below and physically attach two sensors next to each other. We evaluate quality by uploading human code on both boards characterize noise and inter-device variance. We compute the Euclidean distance d between the signals: $d = \sqrt{\sum_{i=1}^n (x_{2i} - x_{1i})^2}$. We perform $N=1000$ measurements to determine a threshold for "correct" outputs (e.g. within 10% of the

reference). We use dynamic time warping (DTW) [1], a signal processing technique that corrects for time shifts, to account for variability in serial timing.

Our fully automated testing script is compatible with the Google Vertex and OpenAI APIs and uses them to automatically prompt models and parse the code output by removing explanatory text and merging code blocks into a single file. The script then attempts to compile the generated code, logs failures, and uploads it to the microcontroller. It then samples both the human and LLM code output over a serial connection and compares the signals as described above.

We run 450 experiments on three LLMs. We note automated testing is critical for fine-tuning models. This system can be extended to new devices and generalized using a logic analyzer or oscilloscope to analyze waveforms. This hardware-in-the-loop setup also extends the idea of LLMs using virtual tools and API calls [5] to *physical tools* for future cyberphysical systems where LLMs to query the real world through sensors. We describe our specific hardware configurations and discuss results below. In these experiments we provide *zero* additional context other than the prompt and used GPT-4 version 0314. The specific prompts and hardware configurations used can be found in Appendix C.

Benchmark Results. In the simple photodiode task PaLM 2 produces many compile errors, while the GPT models produce code that compiles on every iteration as seen in Fig. 3a. All programs that compiled were fully functional and correct. As the complexity of tasks increases, we see a steady drop off in the performance of all three LLMs, with all models failing to always produce correct code for the ultrasonic rangefinder task in 3b and GPT-4 producing fully functional code only 16% of the time on the 6-axis IMU task as shown in Fig. 3c.

5 DISCUSSION AND CONCLUSION

We demonstrate LLMs have capabilities far beyond code auto-complete utilities - they show a promising ability to work across hardware and software domains as well as rationalizing how code effects the *physical behavior* of real-world systems. We summarize key takeaways below and outline directions for future work.

LLMs enhance human developer productivity. Our benchmarks show current LLMs cannot reliably generate code for end-to-end systems on their own. However, LLMs can significantly increase productivity and satisfaction by providing automation and a level of abstraction for experienced developers who provide context-rich prompts. These tools can also empower novice developers and consistently assisted users in debugging across both hardware and software. In particular, GPT-4 provided useful hardware related debugging steps and identified sources of incorrect behavior (e.g. incorrect physical connections, packet structure, or I2C addresses) for errors not caught at compile time. These results highlight the potential of LLMs to dramatically reduce the barriers to physical computing through specific debugging advice. In our teaching experiences many embedded systems students with only software experience struggle to apply their programming skills to hardware. Future work to develop these findings into a teaching tool has significant potential to increase accessibility to physical computing and fabrication research.

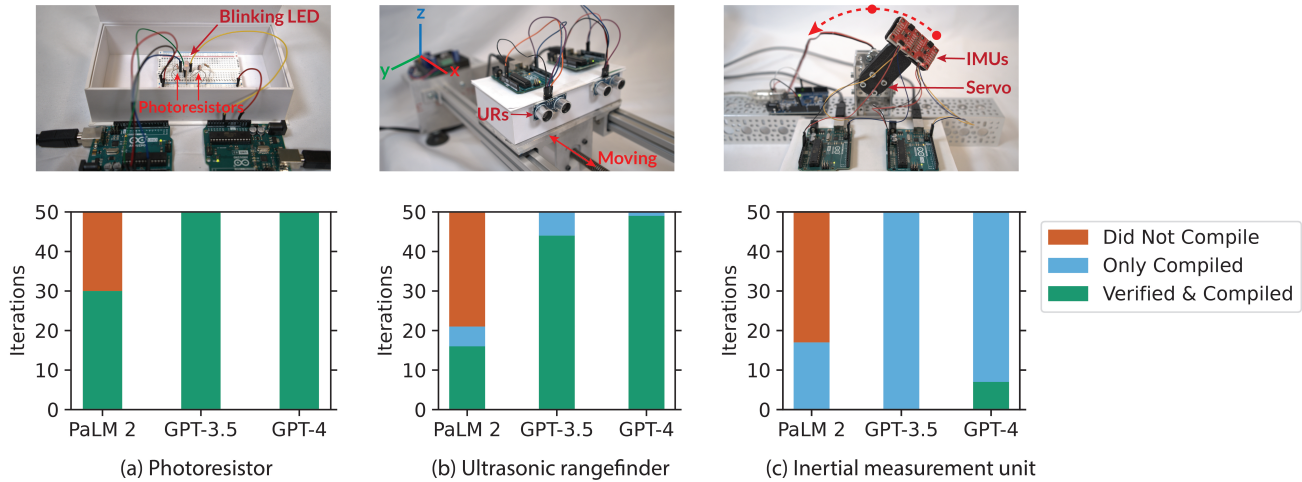


Figure 3: Evaluation of sensing tasks with our hardware-in-the-loop testing pipeline, $N = 50$ iterations on PaLM-2, GPT-3.5, and GPT-4.

Prompting is critical. Our user study and exploration of LLM capabilities confirm that effective prompting is crucial. Prompts must unambiguously encapsulate key system information for LLMs to engage with the technical nuances of both the physical hardware and the intended behaviors. There remains much room for future work to optimize collaboration between human developers and LLMs on embedded development tasks. Fine-tuning Large Language Models (LLMs) on specialized embedded repositories, datasets, and specific Software Development Kits (SDKs) could improve performance. Additionally, developing new ways of encoding the full system state to provide context to the model could improve performance. We see our framework for automated hardware-in-the-loop testing that would allow the models to query the system state like a software plugin as a powerful tool to support future work in these areas.

Benefits of hardware-in-the-loop. In this work, we consider the case of a human developer interacting with an LLM through a text interface. However, there are other interface techniques that could be explored. By integrating LLMs into future Integrated Development Environments (IDEs) and leveraging hardware-in-the-loop test benches, LLMs could programmatically overcome the limitations highlighted earlier. This integration could allow for automated and iterative design enhancements in real-time during development. By harnessing recent advancements in visual [57] and sensor [15] embeddings, upcoming research could unveil methods to represent hardware configurations and setups through consistent and unambiguous embeddings. Such developments would enable language models to generate more informed software solutions grounded in specific hardware requirements.

Role of humans and responsible development. Our user-study feedback regarding fulfillment, though generally positive, was at times ambiguous. Some participants expressed that LLM assistance made them feel less responsible for the results. As the capabilities of LLM-based tools continue to grow, it will be critical to define the role of human developers. This is particularly important when embedded systems are in mission-critical subsystems, where ownership and extensive verification has significant safety implications.

Large language models have the potential to re-imagine how embedded system development and debugging function, but creating direct interfaces between these systems and the physical world simultaneously introduce profound risks. It is imperative that future work proceeds responsibly to address these concerns and does not lose sight of our responsibility to build ethical and safe systems.

ACKNOWLEDGMENTS

This research was partially supported by an Amazon Research Award, a Google Research Scholar Award, the Washington NASA Space Grant Consortium Fellowship Program, and the Pastry-Powered T(uring Machine Endowed Fellowship.

REFERENCES

- [1] 2007. Dynamic Time Warping. In *Information Retrieval for Music and Motion*. Springer Berlin Heidelberg, Berlin, Heidelberg, 69–84. https://doi.org/10.1007/978-3-540-74048-3_4
- [2] Thomas Ball, Abhijith Chatra, Peli de Halleux, Steve Hodges, Michal Moskal, and Jacqueline Russell. 2019. Microsoft MakeCode: embedded programming for education, in blocks and TypeScript. In *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E*. 7–12.
- [3] Robert W Brennan and Jonathan Lesage. 2022. Exploring the Implications of OpenAI Codex on Education for Industry 4.0. In *International Workshop on Service Orientation in Holonic and Multi-Agent Manufacturing*. Springer, 254–266.
- [4] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.
- [5] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, et al. 2023. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712* (2023).
- [6] Angelica Chen, Jérémy Scheurer, Tomasz Korbak, Jon Ander Campos, Jun Shern Chan, Samuel R Bowman, Kyunghyun Cho, and Ethan Perez. 2023. Improving code generation by training with natural language feedback. *arXiv preprint arXiv:2303.16749* (2023).
- [7] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).
- [8] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128* (2023).
- [9] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2022. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311* (2022).

- [10] James Devine, Joe Finney, Peli De Halleux, Michal Moskal, Thomas Ball, and Steve Hodges. 2018. MakeCode and CODAL: intuitive and efficient embedded systems programming for education. *ACM SIGPLAN Notices* 53, 6 (2018), 19–30.
- [11] James Devine, Joe Finney, Peli De Halleux, Michal Moskal, Thomas Ball, and Steve Hodges. 2019. MakeCode and CODAL: Intuitive and efficient embedded systems programming for education. *Journal of Systems Architecture* 98 (2019), 468–483.
- [12] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. 2017. Robustfill: Neural program learning under noisy i/o. In *International conference on machine learning*. PMLR, 990–998.
- [13] Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2023. Self-collaboration Code Generation via ChatGPT. *arXiv:2304.07590* [cs.SE]
- [14] Nicole Forsgren, Margaret-Anne Storey, Chandra Maddila, Thomas Zimmermann, Brian Houck, and Jenna Butler. 2021. The SPACE of Developer Productivity: There's more to it than you think. *Queue* 19, 1 (2021), 20–48.
- [15] Rohit Girdhar, Alaaeldin El-Nouby, Zhuang Liu, Mannat Singh, Kalyan Vasudev Alwala, Armand Joulin, and Ishan Misra. 2023. ImageBind: One Embedding Space To Bind Them All. *arXiv:2305.05665* [cs.CV]
- [16] Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Harkirat Singh Behl, Xin Wang, Sébastien Bubeck, Ronen Eldan, Adam Tauman Kalai, Yin Tat Lee, and Yuanzhi Li. 2023. Textbooks Are All You Need. *arXiv:2306.11644* [cs.CL]
- [17] Zhi Guo, Walid Najjar, and Betül Buyukkurt. 2008. Efficient hardware code generation for FPGAs. *ACM Transactions on Architecture and Code Optimization (TACO)* 5, 1 (2008), 1–26.
- [18] Sandra G Hart. 1986. NASA task load index (TLX). (1986).
- [19] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. 2022. Jigsaw: Large language models meet program synthesis. In *Proceedings of the 44th International Conference on Software Engineering*. 1219–1231.
- [20] Kimmo Karvinen et al. 2019. Lowering barriers on embedded system design—Turning innovations into prototypes. (2019).
- [21] Geunwoo Kim, Pierre Baldi, and Stephen McAleer. 2023. Language models can solve computer tasks. *arXiv preprint arXiv:2303.17491* (2023).
- [22] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. [n.d.]. Large language models are zero-shot reasoners, 2022. *URL https://arxiv.org/abs/2205.11916* ([n.d.]).
- [23] Philip Koopman, Howie Choset, Rajeev Gandhi, Bruce Krogh, Diana Marculescu, Priya Narasimhan, Joann M Paul, Ragnathan Rajkumar, Daniel Siewiorek, Asim Smailagic, et al. 2005. Undergraduate embedded system education at Carnegie Mellon. *ACM Transactions on Embedded Computing Systems (TECS)* 4, 3 (2005), 500–528.
- [24] Shuvendu K Lahiri, Aaditya Naik, Georgios Sakkas, Piali Choudhury, Curtis von Vech, Madanlal Musuvathi, Jeevana Priya Inala, Chenglong Wang, and Jianfeng Gao. 2022. Interactive Code Generation via Test-Driven User-Intent Formalization. *arXiv preprint arXiv:2208.05950* (2022).
- [25] Edward A Lee, Sanjit A Seshia, et al. 2011. Introduction to embedded systems. *A cyber-physical systems approach* 499 (2011).
- [26] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *arXiv preprint arXiv:2305.01210* (2023).
- [27] Mingjie Liu, Teo Ene, Robert Kirby, Chris Cheng, Nathaniel Pinckney, Rongjian Liang, Jonah Alben, Himyanshu Anand, Sanmitra Banerjee, Ismet Bayraktaroglu, et al. 2023. ChipNeMo: Domain-Adapted LLMs for Chip Design. *arXiv preprint arXiv:2311.00176* (2023).
- [28] Xin Liu, Daniel McDuff, Geza Kovacs, Isaac Galatzer-Levy, Jacob Sunshine, Jiening Zhan, Ming-Zher Poh, Shun Liao, Paolo Di Achille, and Shwetak Patel. 2023. Large Language Models are Few-Shot Health Learners. *arXiv preprint arXiv:2305.15525* (2023).
- [29] Zhiqiang Liu, Yong Dou, Jingfei Jiang, and Jinwei Xu. 2016. Automatic code generation of convolutional neural networks in FPGA implementation. In *2016 International conference on field-programmable technology (FPT)*. IEEE, 61–68.
- [30] Tomas G Moreira, Marco A Wehrmeister, Carlos E Pereira, Jean-Francois Petin, and Eric Levrat. 2010. Automatic code generation for embedded systems: From UML specifications to VHDL code. In *2010 8th IEEE International Conference on Industrial Informatics*. IEEE, 1085–1090.
- [31] Nhan Nguyen and Sarah Nadi. 2022. An empirical evaluation of GitHub copilot's code suggestions. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 1–5.
- [32] Nhan Nguyen and Sarah Nadi. 2022. An empirical evaluation of GitHub copilot's code suggestions. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 1–5.
- [33] Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023. Codegen2: Lessons for training llms on programming and natural languages. *arXiv preprint arXiv:2305.02309* (2023).
- [34] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).
- [35] Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2023. Demystifying GPT Self-Repair for Code Generation. *arXiv:2306.09896* [cs.CL]
- [36] OpenAI. 2023. GPT-4 Technical Report. *arXiv:2303.08774* [cs.CL]
- [37] Sudeep Pasricha. 2022. Embedded Systems Education in the 2020s: Challenges, Reflections, and Future Directions. In *Proceedings of the Great Lakes Symposium on VLSI 2022*. 519–524.
- [38] Gabriel Poesia, Oleksandr Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. SynchroMesh: Reliable code generation from pre-trained language models. *arXiv preprint arXiv:2201.11227* (2022).
- [39] James Prather, Brent N Reeves, Paul Denny, Brett A Becker, Juho Leinonen, Andrew Luxton-Reilly, Garrett Powell, James Finnie-Ansley, and Eddie Antonio Santos. 2023. "It's Weird That it Knows What I Want": Usability and Interactions with Copilot for Novice Programmers. *ACM Transactions on Computer-Human Interaction* 31, 1 (2023), 1–31.
- [40] Puneeth. 2023. *I tried ChatGPT for Arduino - It's Surprising*. <https://blog.wokwi.com/learn-arduino-using-ai-chatgpt/>
- [41] Matthias C Rillig, Marlene Ågerstrand, Mohan Bi, Kenneth A Gould, and Uli Sauerland. 2023. Risks and benefits of large language models for the environment. *Environmental Science & Technology* 57, 9 (2023), 3464–3466.
- [42] CJS Robotics. 2023. *Duino Code Generator*. <https://www.duinocodegenerator.com/>
- [43] Noah Shinn, Beck Labash, and Ashwin Gopinath. 2023. Reflexion: an autonomous agent with dynamic memory and self-reflection. *arXiv preprint arXiv:2303.11366* (2023).
- [44] Karan Singhal, Tao Tu, Juraj Gottweis, Rory Sayres, Ellery Wulczyn, Le Hou, Kevin Clark, Stephen Pfohl, Heather Cole-Lewis, Darlene Neal, et al. 2023. Towards expert-level medical question answering with large language models. *arXiv preprint arXiv:2305.09617* (2023).
- [45] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).
- [46] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).
- [47] Alex Wang, Yada Pruksachatkun, Nikita Nangia, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel Bowman. 2019. SuperGlue: A stickier benchmark for general-purpose language understanding systems. *Advances in neural information processing systems* 32 (2019).
- [48] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2022. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171* (2022).
- [49] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. 2022. Chain of thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903* (2022).
- [50] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C Schmidt. 2023. A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv preprint arXiv:2302.11382* (2023).
- [51] Wayne Wolf. 2010. *High-performance embedded computing: architectures, applications, and methodologies*. Elsevier.
- [52] DroneBot Workshop. 2023. *Using ChatGPT to Write Code for Arduino and ESP32*. <https://dronebotworkshop.com/chatgpt/>
- [53] Shijie Wu, Ozan Irsoy, Steven Lu, Vadim Dabravolski, Mark Dredze, Sebastian Gehrmann, Prabhajan Kambadur, David Rosenberg, and Gideon Mann. 2023. Bloomberggpt: A large language model for finance. *arXiv preprint arXiv:2303.17564* (2023).
- [54] Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. 1–10.
- [55] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. *arXiv preprint arXiv:2305.10601* (2023).
- [56] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629* (2022).
- [57] Hang Zhang, Xin Li, and Lidong Bing. 2023. Video-LLaMA: An Instruction-tuned Audio-Visual Language Model for Video Understanding. *arXiv:2306.02858* [cs.CL]
- [58] Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. 2022. Large language models are human-level prompt engineers. *arXiv preprint arXiv:2211.01910* (2022).
- [59] Albert Ziegler, Eirini Kalliamvakou, X Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. 2022. Productivity assessment of neural code completion. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. 21–29.

A LLM USAGE SUGGESTIONS

A.1 Detailed GPT Prompt

Providing GPT with ample context is important to generating useful code. Make sure to provide a complete scope of what you are trying to accomplish. If you have example code snippets, include them. If you know what libraries to use, specify them. If you have a multi-device system, explain what the other devices will be doing and/or provide GPT with their code.

Bad prompt: *“Make the LED blink and print hello world”*

Good prompt: *“I have an Arduino Uno with an LED attached to Pin 13. I want to print the message “Hello World” over serial and blink the LED on and off every second. First explain how an expert programmer would approach this task and then provide a program to accomplish it.”*

A.2 Copy error message to GPT

If compilation fails and you don’t understand what is wrong with the generated code, copy and paste the error message back into GPT.

Prompt: *“When I try to compile this code, I get the following error: ***copied error here***”*

A.3 Flash and observe behavior

Just because code compiles and uploads does not mean that it does exactly what you want. Observe the behavior of the board and identify what aspects of the behavior are correct and incorrect. If everything is correct, you are done!

A.4 Describe incorrect behavior and ask for fix

Provide gpt with a detailed explanation of what exactly is wrong with the behavior of the system and ask it to fix the code.

Bad prompt: *“This doesn’t work”*

Good prompt: *“This code compiles and uploads, and I see the correct serial prints on my computer every second. The LED should be flashing once a second, but right now once it turns off it never turns back on again. Please identify the problem and suggest how to fix this.”*

A.5 Other tips

GPT often struggles with numbers. If there are calculations in the code, it can often be more efficient to manually fix numerical errors instead of prompting GPT to fix them.

Don’t forget that you are a programmer too! If you find a bug that you know how to fix, fix it yourself. If you know what is wrong but don’t know exactly how to fix it, use this to provide a more precise explanation to GPT. For example: You notice that you asked to flash an LED on Pin 13, but GPT responded with a program that flashes Pin 12. Manually update the pin number instead of going straight back to GPT.

“I asked to use the Sparkfun IMU library for this, but I see there are some calls to the Adafruit IMU library in the setup() function. Please rewrite this to use only the Sparkfun library to interface with the IMU.”

Keep the context up-to-date: If you make changes to the code, copy back your updated code to GPT so future responses can incorporate your changes. For longer programs, you can just copy the segments of the code that you updated.

Start with example code and explain what you want changed. Just like a human programmer, GPT can often be more successful when you give example code that is *close* to what you want. Make sure you are explicit about what you want changed and what you want to stay the same, otherwise GPT is prone to make unprompted changes to other parts of the program as well. For example:

“I have a program for the Arduino Uno that reads data from a LSM6DSO IMU using the Adafruit IMU library and prints the data over serial. Instead of printing to serial, I want to analyze this data and flash an LED on Pin 13 every time the total magnitude of acceleration is higher than 2 Gs. Please replace the portion of code that is printing data to serial with code to do this calculation, leaving as much of the original program unchanged as possible”

If GPT fails multiple times to fix an issue, it is often best to start over with a new chat. Break things down into modules. GPT is best at generating short to medium length programs. If GPT fails multiple times to generate complete working code, ask it to develop modular functions individually and integrate them into a complete program yourself.

B GRADING RUBRIC

B.1 Timer Interrupt

Criteria	Points
Speaker on digital pin and ground	25
Made sound	25
Made sound with timer	25
Right sound or close to it (within 15 Hz)	25

B.2 RTC

Criteria	Points
Doesn’t work	0
Makes RTC generate square wave	25
Blinks from RTC interrupt	25
Blinks from interrupt at 1s	25
Sleep mode + square wave interrupt	25

B.3 Heart Rate Sensor

Criteria	Points
Reads from analog HR sensor	25
Bluetooth connection made	25
Transmits some data over BLE	25
Visible waveform	25

B.4 Environmental Sensor

Criteria	Points
Reads BME sensor	25
Connection between RX and TX (any string is fine)	25
Sends some amount of BME data from RX to TX	25
Pressure, temp, and humidity are all visible on RX side	25

C PHYSICAL TESTBENCH SETUP

C.1 Photoresistor

Our first benchmark asks the model to read a photoresistor with the Arduino's onboard ADC.

Prompt: *I have an Arduino Uno with a photoresistor attached to pin A0. Write a program that reads the value on pin attached to the photoresistor and prints it on a new line to Serial at a baud rate of 9600 every 100ms. Example serial output: 233\n512\n555\n*

We connect a photoresistor and pull-up resistor to each microcontroller and orient them toward a blinking LED inside an enclosed box without external light. Figure 3a shows the results of these experiments.

We observe PaLM-2 produces many compile errors, while the GPT models produce code that compiles on every iteration. The most common compilation errors result from an incorrect program format lacking `setup()` and `loop()` functions. We found that all programs that compiled were also fully functional and correct. This shows that LLMs can repeatably produce the correct program structure and generate functional code with a single prompt. We note the high success rate on this task is likely due to a combination of its simplicity and many similar open-source code examples.

C.2 Ultrasonic Rangefinder

Next we ask the LLMs to continuously read distance from an ultrasonic rangefinder.

Prompt: *Write me an Arduino Uno program for the HC-SR04 ultrasonic ranging module that prints out the measured distance in cm without units every 100ms over Serial on a new line at baud rate 9600. The 'trig' output is connected to pin 12 and the 'echo' output is connected to pin 11. Do not use any external libraries. Example serial output: 23.273\n23.419\n23.366\n*

This sensor is more complex than a photoresistor and must be queried by setting the trigger pin high. The sensor sends an ultrasonic burst and sets the echo pin high until remains high until the sensor receives an echo. The code must read how long the echo pin is high, calculate the time difference, and then perform unit conversion using the speed of sound.

To test the sensor, we use a lead screw driven linear actuator (Phidget Wantai Mini Stepper) to continuously move a platform back and forth across 5 cm as shown in Fig 3b. We mount two ultrasonic distance sensors (Sparkfun HC-SR04) and place a screen at the end of the moving platform to reflect the sensor's transmitted waves at a consistent distance. One of the sensors is programmed with human code and the other is running on code generated by the LLM.

C.3 6-axis IMU

For our most complex task we test the ability of the three LLMs to generate code for a 6-axis IMU (LSM6DSO).

Our physical testbench consists of a 3D-printed servo horn that holds the two sensors side by side at an angle of 55° to the vertical axis. This angle ensures that when the servo (goBILDA 2000 Series Dual Mode Servo 25-3, Speed) moves through a set of positions, the IMU produces non-zero outputs on all axes. We perform verification experiments by using a third Arduino to control the servo. The testing script sends commands to move the servo to 3 angles (0, 90, and 180 degrees). At each angle the testing script logs data from the X, Y, and Z axes from LLM and user-written code for a 10s duration and checks they are within 10%. The test script proceeds to the next angle on a correct output and only considers the code fully correct if all three tests pass.

Prompt: *Write me a program for the Arduino Uno that interfaces with an LSM6DSO over I2C using only the Wire library. In my hardware implementation, the SDO/SA0 pin of the LSM6DSO is connected to GND (ground). Print the acceleration readings in gs every 100ms over Serial at baud rate 9600. Example serial output: A_X = -0.426\n A_Y = -0.023\n A_Z = 0.913\n*