DaisenBot: Human-AI Collaboration in GPU Performance Analysis with Multi-Modal AI Assistant

Enze Xu
exu03@wm.edu
William & Mary
Williamsburg, VA, USA

Jeremy Coonley
jmcoonley@wm.edu
William & Mary
Williamsburg, VA, USA

Daoxuan Xu
dxu05@wm.edu
William & Mary
Williamsburg, VA, USA

Yifan Sun
ysun25@wm.edu
William & Mary
Williamsburg, VA, USA

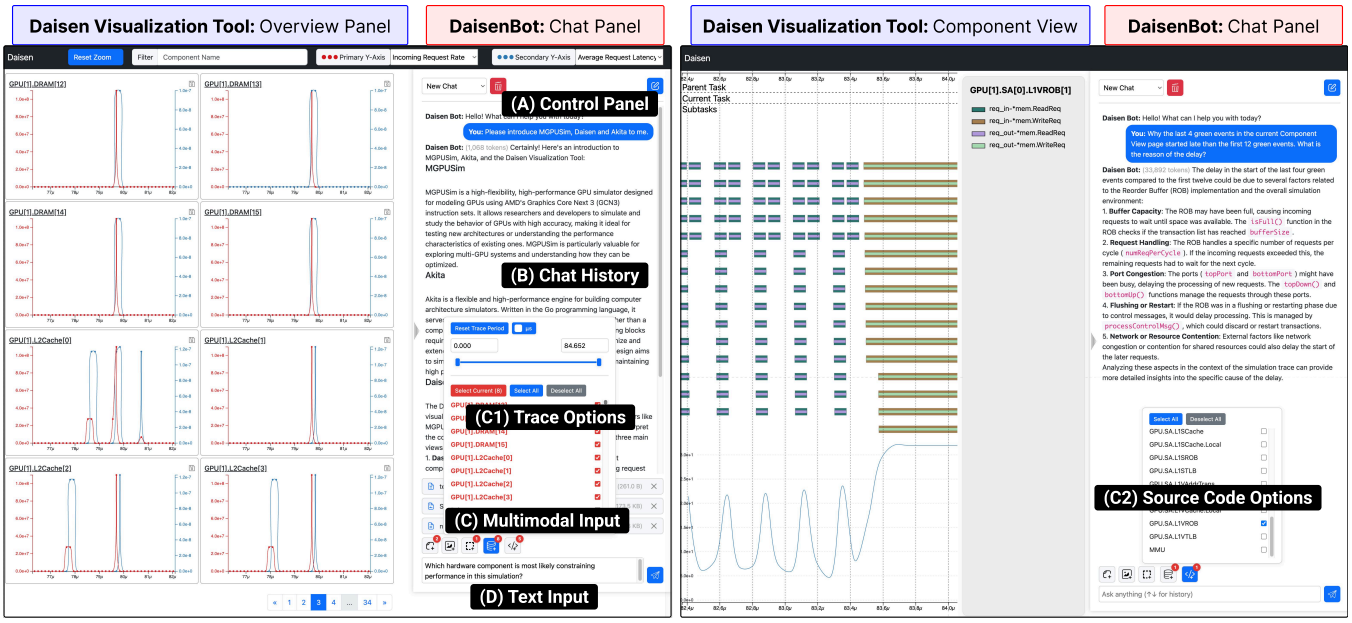

Figure 1: DaisenBot is a web-based interactive AI assistant that helps users address questions when using Daisen, the GPU simulator visualization tool. It generates accurate, context-specific answers from multimodal user inputs. (A) The control panel provides options for starting a new chat, deleting a chat, and switching between chats. (B) Chat history is displayed in the center of the panel for easy reference. (C) DaisenBot supports multimodal inputs, including file and image uploads, screenshot attachments, simulation trace attachments (C1), and source code attachments (C2). (D) Text input serves as the primary interaction channel, where users can enter questions and additional information.

Abstract

Graphics Processing Units (GPUs) play a critical role in accelerating applications across artificial intelligence, physical simulations, and medical imaging. Analyzing simulator-generated execution traces is essential for understanding GPU behavior and identifying performance bottlenecks, but users often face challenges interpreting complex visualizations and hierarchical data structures. To address this, we present **DaisenBot**, an interactive AI assistant that leverages large language models to provide accurate, context-specific answers from multimodal inputs, including text, images, simulation traces, and source code. DaisenBot helps users clarify questions, navigate relevant simulation data and subpages, and better understand and analyze simulation results, offering practical support for both novice and experienced users.

Keywords

GPU Simulator, GPU Performance Analysis, Large Language Models, Computer Architecture

1 Introduction

GPUs have been widely used to accelerate applications in artificial intelligence [7, 20], physical simulations [8, 24], medical imaging [5, 29], and information visualization [9, 19]. To improve GPU performance, hardware designers need to examine massive amounts

of simulator-generated traces to identify performance bottlenecks. Visualizing execution traces can reduce users' cognitive load and help them better understand the behavior of GPU hardware components [2, 23, 30].

However, for users relying on visualized GPU execution traces, understanding the visualization itself and understanding how it connects to the simulated hardware mechanisms and specific simulator implementations remains a major challenge—a well-recognized concern in the field of data visualization [3, 11].

Traditionally, users turn to simulator manuals or direct communication with developers for assistance. Manuals are often time-consuming to consult and limited by documentation quality [4, 14–16], while contacting developers depends on their responsiveness and is frequently hindered by communication inefficiencies, especially in small open-source teams. These limitations highlight the need for more effective, scalable support mechanisms.

Recent advances in generative AI, particularly multimodal large language models (LLMs), provide a promising new direction. Unlike conventional tools, multimodal LLMs can process heterogeneous inputs—such as natural language, screenshots, code snippets, and trace files—and generate context-aware answers that integrate knowledge across these modalities [17, 26, 27]. This capability makes them well-suited to assist users of GPU simulator visualization tools, where questions often require linking visual traces to underlying documentation and source code.

These challenges motivate our research question: *How can generative AI effectively support computer architects in the context of a visualization-based performance analysis tool?* A central obstacle in developing such support is the lack of paired multimodal training data, where user queries (e.g., screenshots, trace files) are aligned with the appropriate instructions or answers.

To address this, we propose DaisenBot, an interactive web-based assistant that leverages pre-trained LLMs. By integrating simulator documentation, images, trace files, and source code, DaisenBot can generate accurate and context-specific answers to users' questions when using the Daisen visualization tool.

In summary, this paper makes the following contributions:

- DaisenBot, an interactive AI chat assistant designed to help users address questions that arise when using the GPU simulator visualization tool, Daisen. DaisenBot generates accurate, context-specific answers from multimodal user inputs, without requiring paired training data.
- Practical support for users of the Daisen visualization tool, including clarifying questions, providing organized simulation data, guiding users to relevant subpages and simulation settings, and assisting in understanding and analyzing simulation results.

2 A Primer for Daisen

Overview. *Daisen* is a web-based framework that visualizes detailed GPU execution traces so architects can inspect behavior, identify bottlenecks, and validate design changes [23]. Daisen provides three coordinated views—Overview Panel, Component View, and Task View (see Figure 2)—to move fluidly from global patterns to per-component timelines and task hierarchies. In this paper, we use Daisen as the visualization substrate that our *DaisenBot* assistant augments to explain what users see and to guide analysis steps within the same interface.

Daisen trace data format. Daisen consumes detailed execution traces collected from MGPUSim [21]. Daisen's trace format models execution as hierarchical *tasks* with fields `ID`, `Parent ID`, `Category`, `Action`, `Location`, and `Start`/`End`. The streamlined yet expressive schema allows Daisen to reconstruct execution details and makes the data uniform and machine-readable, which is helpful for GenAI parsing and grounding.

Overview Panel. The Overview Panel (see Figure 2 (A)) is the entry point for locating time intervals that merit closer inspection. It renders small multiples—one time series per hardware component—split across pages when components are numerous. A regex filter (e.g., `(CU|L1|L2)`) narrows components for side-by-side comparison. Users can plot a primary and secondary y-axis metric (six metrics are available in the current implementation), and zooming or panning any chart synchronizes the time window across all charts. Clicking a component name jumps to its Component View for the same interval.

Component View. Selecting a component opens a hierarchical, Gantt-like view of all tasks executed on that component (see Figure 2 (C)). An *up-floating* row-assignment packs bars tightly from the top without overlap, so vertical stacking at the same x-position directly encodes instantaneous parallelism (how busy the component is). Tasks are color-coded by `Category–Action`; the interactive legend supports search/highlighting and reveals task metadata on hover.

Task View. The Task View (see Figure 2 (B)) complements the Component View by showing the *parent* task above the *current* task and all *subtasks* below it. Its time axis is locked to the Component View, so panning/zooming either view updates both. This coupling lets users trace a long-latency operation back to the component and understand why the component cannot finish the task earlier.

Why Daisen. We choose Daisen because it is widely used in GPU-architecture research and combines expressive analysis with easy deployment. As a web-based client–server system, it is straightforward to extend (e.g., adding analysis panels or APIs) and integrates cleanly with existing simulators—its lightweight instrumentation is already part of MGPUSim releases. Most importantly, Daisen enforces a uniform, task-centric trace schema that preserves hierarchy and causality across components. This consistency makes it a natural substrate for GenAI: models can reason over stable fields (IDs, parent/child links, categories/actions, timestamps), follow end-to-end *Request Out*/*Request In* chains, and align explanations with exactly what the UI displays.

Why a ChatBot. Daisen is an expert tool. While its *Overview* → *Component* → *Task* workflow greatly reduces the burden of navigating massive traces, effective use still leans on component knowledge (e.g., caches, compute units, memory controllers) and experience selecting representative intervals. Users must choose a time window, pick metrics, and inspect a handful of tasks—steps where newcomers may struggle to identify the "best" slice or task chain. A chatbot mitigates this by answering "what am I looking at?", proposing filters and time ranges, and suggesting next clicks that connect hypotheses to the corresponding task chains.

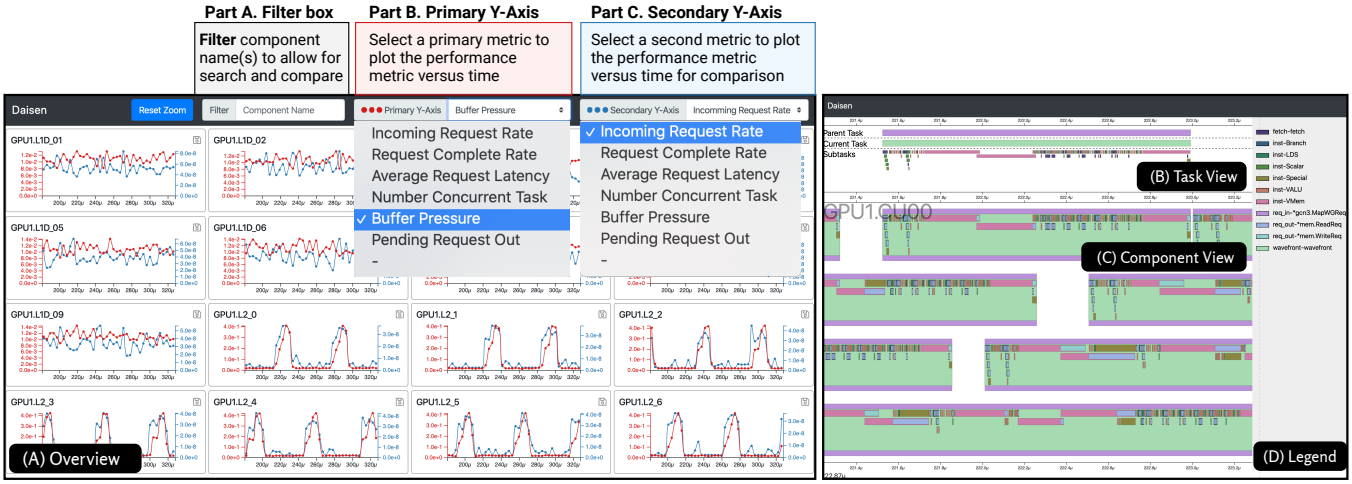

Figure 2: The original Daisen interface includes (A) the Overview Panel, (B) the Task View, and (C) the Component View.

3 DaisenBot

3.1 System Overview

DaisenBot is an extension of the Daisen Visualization Tool [23], residing in the right sidebar of the Daisen interface. Operating within the same system facilitates the convenient transfer of simulation traces and component view screenshots already available in Daisen directly to DaisenBot. The primary function of DaisenBot is to answer users' questions when examining Daisen visualizations in the main Daisen interface.

DaisenBot currently supports OpenAI APIs, with planned support for other chatbot APIs and self-hosted models in the future. Users may configure a *.env* file to specify their API credentials, including the API link, model name, and key.

DaisenBot's interface layout (see Figure 1) is similar to that of a typical chatbot that has the user input at the bottom and the conversation history on top. DaisenBot specifically emphasizes multimodal input, including file, image, current Daisen screenshot, trace data (collected from GPU simulation), and simulator source code. The multimodal input, sent to the generative API together with the user's chat messages, provides the Daisen context, allowing the generative AI to provide more targeted answers.

3.2 Multimodal Input

For every text message that is sent to the ChatBot API, DaisenBot allows five types of contextual, multimodal data, including: file uploads (designed for data, text and code files), image uploads, attaching the current Daisen interface screenshot, the raw data of what is currently being visualized in Daisen, and the simulator's source code from GitHub.

Next, we describe each input type in detail, including text input.

Text Input. Users can enter any textual information in the input box, as shown in Figure 3(a). Similar to a terminal, the interface supports quick navigation through input history using the up and down arrow keys. This is the primary channel for interaction between the user and DaisenBot. DaisenBot generates answers in response to the questions entered here.

File Upload. As shown in Figure 3(b), users can upload any relevant text, data, or code files from their local machine. Supported formats include, but are not limited to, .txt, .json, .csv, .sqlite3, .py, .c, and .cpp. Once uploaded, files are displayed in a list with an icon representing their type, the file name, and file size, along with an option to quickly remove them. Files are converted to plain text before being sent to the backend for DaisenBot to process.

Image Upload. As shown in Figure 3(c), users can upload image files in formats such as .png, .jpeg, .jpg, .svg, and .gif. Similar to file uploads, the files are displayed with type icons, names, sizes, and quick removal options. Images are encoded in Base64 format before being sent to the backend for processing.

Attach Screenshot. When users wish to ask a question about the current Daisen view, DaisenBot provides the "Attach Screenshot" feature, which can attach the current Daisen interface screenshot using the *html2canvas* [25] package. As shown in Figure 3(d), clicking this button automatically captures and attaches a screenshot of the active Daisen window. This feature supports all Daisen pages, including the Overview Panel, Component View, and Task View. Once captured, the screenshot is processed in the same way as an uploaded image.

Attach Trace. The Daisen Visualization Tool operates on collected traces that contain both event information (e.g., name, location) and timing details (e.g., start and end times) for simulator components. For trace-specific questions, DaisenBot provides the "Attach Trace" feature, which can attach the currently opened simulation trace. Since traces from large simulations can result in very large files and extracted plain text may be prohibitively long, attaching the entire trace can be inefficient and costly in terms of tokens. To address this, DaisenBot enables users to attach only partial traces. As shown in Figure 3(e), the trace attachment interface allows users to attach selected, all, or currently displayed component traces. A pair of sliders are also provided to filter events within a selected time interval.

Attach Source Code. Since the execution logic of events in a trace is closely tied to the simulator's source code, DaisenBot allows users to attach the relevant source code directly from GitHub REST API

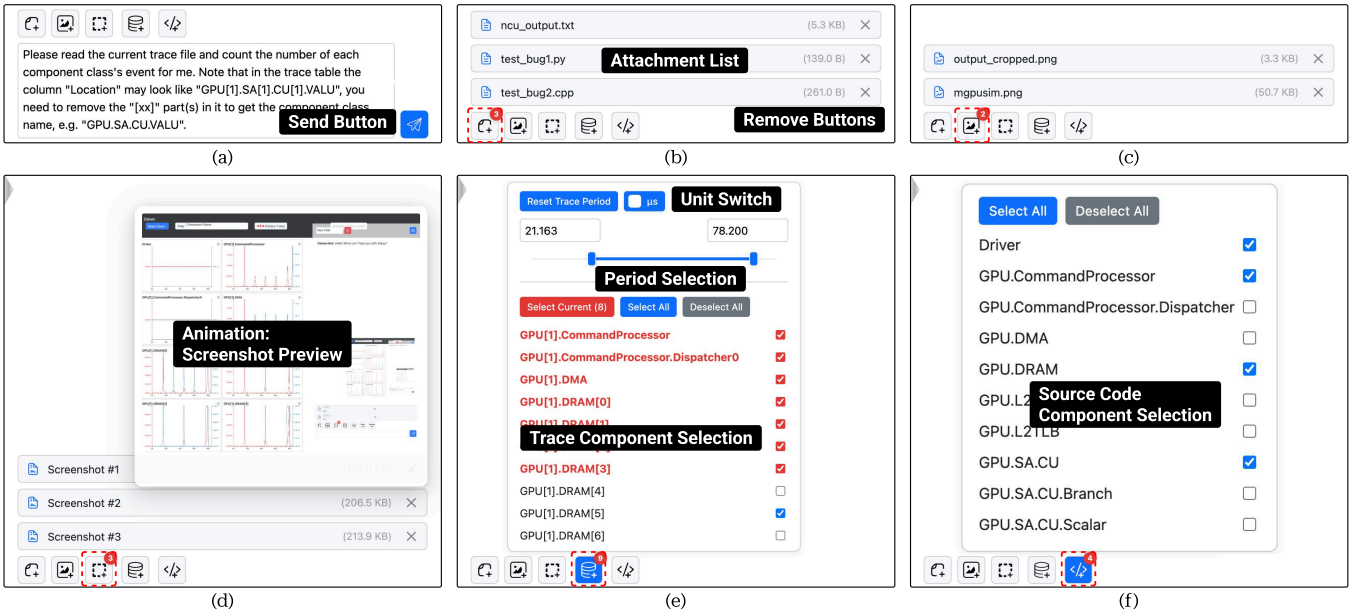

Figure 3: DaisenBot Multimodal Input. (a) Text Input. Users enter their primary question in plain text. (b) File Upload. Users can upload any relevant text, data, or code files from their local machine. (c) Image Upload. Users can upload image files from their local machine. (d) Attach Screenshot. Users can attach the current Daisen interface screenshot. (e) Attach Trace. Users can attach the currently opened simulation trace in Daisen. (f) Attach Source Code. Users can attach the simulator's source code from GitHub REST API.

[6]. As shown in Figure 3(f), to address token cost concerns similar to those for traces, users can attach only selected components' source code. These files are processed by the backend in the same manner as code uploaded through the File Upload feature.

3.3 Multi-Modal AI Assistant

Response Generation. DaisenBot integrates both plain text and multimodal inputs from the chat panel to generate responses. The workflow proceeds as follows:

(1) The frontend encodes image inputs into Base64, extracts plain text content from uploaded files, and sends these inputs—together with user-selected trace and source code options, plain-text question as well as the current URL information (base address and parameters)—to the backend.
(2) The backend applies the user's trace options to filter the full trace file, retaining only the specified trace events. The filtered events are then encoded into a CSV-formatted plain text representation.
(3) The backend retrieves the relevant simulator source code from the GitHub REST API, including file names and repository paths. Users may specify which simulator components are relevant to their query, as described in §3.2.
(4) The backend loads a predefined prompt from a text file that contains essential task environment information. This includes an introduction to Daisen, Akita, and MGPUSim; descriptions of visible components on Daisen pages; expected user input formats and outputs; and other necessary details.
(5) All collected inputs are concatenated into a plain text prompt, after which the backend sends an HTTP POST request to the OpenAI API. If the model name or API key has not yet been configured, users are prompted to provide them upon first use. The chat history is also included to preserve conversational context.
(6) Once a response is received, the backend forwards it to the frontend. The DaisenBot frontend then decodes the output and displays it in the chat panel. It additionally supports mathematical typesetting and rich text formatting (e.g., bold or italic fonts), rendered using the KaTeX package.

Visualization Suggestion in DaisenBot's Response.

In a visualization tool, the chatbot is not intended to replace the role of a human. Instead, the chatbot should facilitate. Accordingly, DaisenBot not only uses text to explain what users see, but also actively directs them to relevant visualizations. Its responses may include links to specific Daisen views (see Figure 4(a)) or to simplified charts (see Figure 4(b)).

When DaisenBot determines that navigating to a particular Daisen view can help the user diagnose a simulation issue or gain deeper insights into results, it provides one or more shortcut URLs. This functionality leverages a unique feature of Daisen: all views are URL-encoded and can be linked directly.

For example, when DaisenBot identifies the host-to-GPU memory transfer as the main bottleneck—"the host-to-GPU memory transfer is longer than the kernel execution time" (see Figure 4(a))—it suggests that the user examine the execution timeline to compare memory copy time against kernel execution time (see Figure 4(c)).

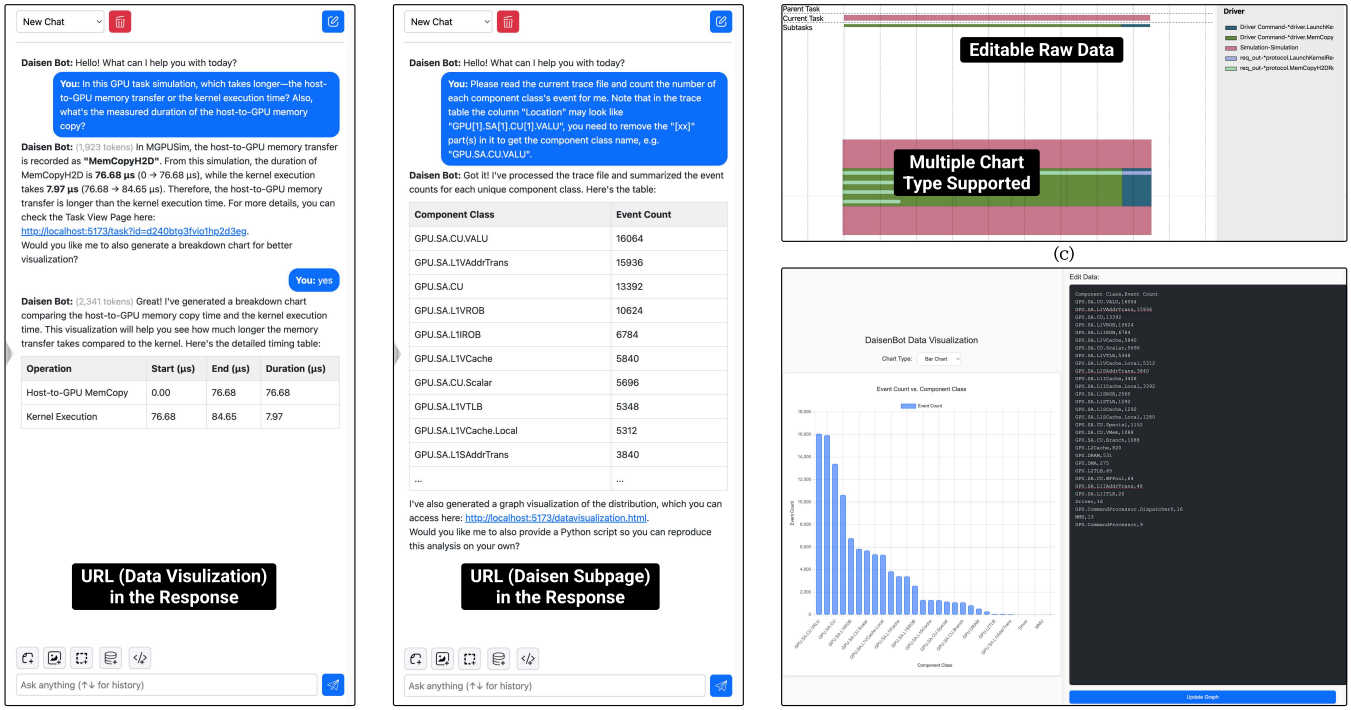

Figure 4: DaisenBot Response Examples with URLs. (a) Response with a Daisen subpage URL. (b) Response with a Data Visualization URL. (c) Navigate to a new browser tab to the Daisen subpage (Task View). (d) Navigate to a new browser tab to the Data Visualization page.

Occasionally, Daisen may lack the most effective visualization for deeper analysis, as it does not provide standard charts such as line or bar plots. To address this gap, DaisenBot can generate a shortcut URL to a **Data Visualization Page**, where it produces simple charts (e.g., bar or pie). An example is shown in Figure 4(d): a bar chart visualization page that enables users to quickly explore data patterns and obtain an overview before engaging in more detailed analysis.

4 Evaluation

We evaluate DaisenBot's capabilities in three aspects, from the most basic to the most advanced: clarifying users' questions about the Daisen visualization tool (§4.1); providing organized simulation data, relevant subpages locating and simulation settings (§4.2); and supporting users in understanding and analyzing simulations (§4.3).

We evaluate DaisenBot with a trace collected from MGPUSim [21]. We run the Finite Response Filter (FIR) benchmark from the HeteroMark benchmark suite [22] with 4096 input data samples and a tap count 16. FIR is selected since it is a small, commonly used GPU algorithm. Using other benchmarks should yield a similar evaluation result. The GPU simulated is the AMD R9 Nano GPU [1, 10], default to MGPUSim.

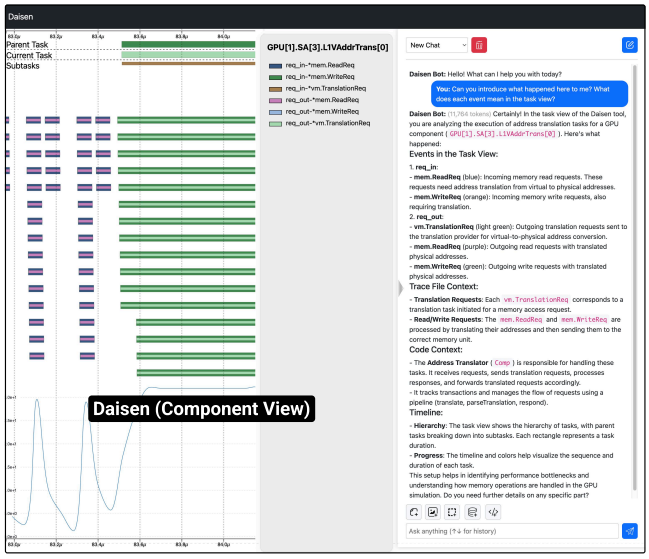

Figure 5: Example of a user inquiry about the meaning of each colored event in the component view of L1VAddrTrans.

4.1 Explaining Daisen

We first evaluate whether DaisenBot can clarify users' questions about Daisen. In this case, we assume that the user seeks to understand the meaning of each colored event in the component view

of `L1VAddrTrans`. The corresponding conversation is shown in Figure 5.

DaisenBot leverages multimodal information when generating explanations. For instance, in the response, the "Event in the Task View" section demonstrates that DaisenBot can correctly explain most task types using pre-trained knowledge (e.g., virtual addresses must be translated to physical addresses). Moreover, DaisenBot introduces the term "translation provider", which appears in the simulator's implementation, indicating that it can integrate source code information into its explanations. The statement, "The `mem.ReadReq` and `mem.WriteReq` are processed by translating their addresses and then sending them to the correct memory unit", is a good example of a concise summary of this process.

However, the response remains unsatisfactory in two respects. First, DaisenBot does not synthesize the information into a single coherent explanation; instead, the answer is fragmented across multiple contexts (Task View, Trace File Context, and Code Context). Second, its visual reasoning is limited, as it incorrectly associates colors with task types, despite the presence of clear legends in the Daisen interface.

4.2 Summarizing Data

Next, we evaluate whether DaisenBot can effectively summarize data within the Daisen tool. In one example, the user asks whether host-to-GPU memory transfer or kernel execution time dominates the overall simulation time (see Figure 4(a)). DaisenBot not only provides a direct answer but also generates a link to the relevant subpage (*Task View*, as shown in Figure 4(c)) for further inspection. This demonstrates its ability to both interpret the user's question and retrieve the corresponding data from attached files (e.g., Daisen trace files) to support its response.

In another example, the user requests a count of events for each component class in the trace file (see Figure 4(b)). DaisenBot responds with a static table in the chat panel and additionally generates a link to a new Data Visualization page (see Figure 4(d)), where the user can interactively edit and visualize the data. This highlights DaisenBot's capability to organize and summarize complex data from attached files in a way that directly supports user queries.

4.3 Facilitating Understanding

Finally, we assess DaisenBot's performance on more challenging analysis tasks.

As illustrated in Figure 6, we consider a scenario where the user asks DaisenBot to identify the potential bottleneck of a specific simulation—an inherently difficult task even for experienced GPU architecture developers. DaisenBot explores four possible bottlenecks and concludes that memory bandwidth and latency are the most significant contributors, with cache and TLB contention further aggravating the problem. Although its analysis leaves room for refinement, DaisenBot demonstrates the ability to integrate multiple data sources, producing an explanation that is not generic but grounded in concrete evidence, including event names drawn from both the source code and attached traces.

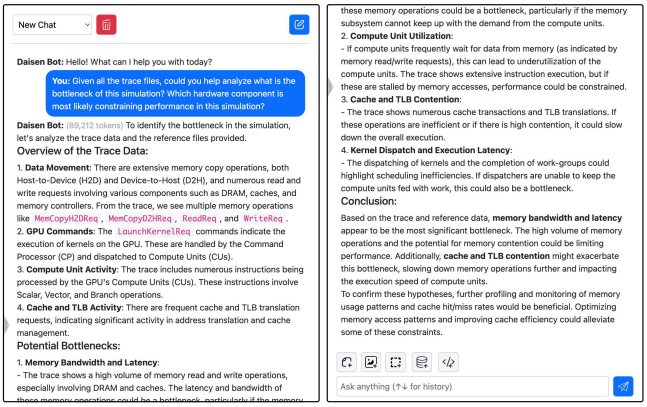

Figure 6: Example of a user inquiry where DaisenBot analyzes and identifies potential bottlenecks in a specific simulation.

5 Discussion

In this section, we reflect on the findings from our evaluation of DaisenBot and examine its limitations, along with potential directions for future improvements in DaisenBot.

Lost-in-the-Middle Issue. The lost-in-the-middle phenomenon is a well-documented challenge in the field of large language models (LLMs), referring to the model's diminished ability to effectively utilize relevant information located in the middle of a long input context. Studies have shown that performance can degrade significantly when relevant information is positioned centrally, even in models designed to handle extensive contexts [13].

When evaluating DaisenBot, we observe similar limitations. When users upload lengthy or complex inputs—such as large simulation traces containing tens of thousands of events—DaisenBot sometimes overlook key details buried in the middle of the file. This results in incomplete, generic, or inaccurate responses, even though the relevant data is present.

To mitigate these challenges, a potential solution is to adopt an embedding-based retrieval approach, inspired by recent work integrating LangChain [12], OpenAI [17], and vector databases such as Pinecone [18]. Instead of passing the entire trace file to the model, the trace can be segmented into smaller, semantically meaningful chunks, each represented by embeddings and indexed in a vector store. When a user submits a query, only the most relevant segments are retrieved and provided as context to the model. This approach not only reduces context overload but also ensures that information located in the "middle" of large inputs remains accessible and prioritized during reasoning.

Future work will focus on integrating such retrieval-augmented generation (RAG) techniques into DaisenBot to better handle large-scale traces (e.g., 100k+ rows) and improve the accuracy of responses for complex simulation analysis tasks.

Visual Reasoning Failure. Visual reasoning failure is a recognized limitation in multimodal LLMs, where the model struggles to accurately interpret and reason about visual inputs, such as images or diagrams. Recent research has highlighted significant performance gaps in visual reasoning tasks, even among advanced multimodal models [28].

In our assessment of DaisenBot, we observe cases where the model misinterprete visual information—for instance, miscounting elements, overlooking subtle visual cues, or drawing incorrect inferences from diagrams. To address these issues, improvements in the integration of visual processing pipelines are needed. Specifically, the potential solutions include: (1) avoiding reliance on *html2canvas* [25] for screenshots and instead enabling the backend to generate high-resolution images, and (2) supplementing screenshots with the corresponding HTML DOM tree structure to provide richer contextual information for reasoning.

In the future, advancing DaisenBot's visual reasoning ability will require a combination of higher-quality visual data representations and tighter coupling between visual and structural inputs.

6 Conclusion

In this paper, we presented DaisenBot, an interactive AI assistant designed to support users of the GPU simulator visualization tool, Daisen. By leveraging pre-trained large language models, DaisenBot can generate accurate, context-specific answers from multimodal inputs, including text, images, simulation traces, and source code, without requiring paired training data. Our evaluation demonstrates that DaisenBot effectively clarifies user questions, organizes simulation data, guides users to relevant subpages and settings, and assists in analyzing simulation results. We also discussed limitations such as the lost-in-the-middle issue and visual reasoning failures, which highlight opportunities for future improvements. Overall, DaisenBot provides a practical solution for reducing cognitive load and enhancing the usability of GPU simulator visualization tools.

Acknowledgement

We thank the anonymous reviewers of CAMS2025 for their constructive feedback. This work is supported by the US National Science Foundation (NSF) under OAC-2441804.

References

[1] AMD. 2015. AMD Radeon(TM) R9 Nano, World's Smallest and Most Power-Efficient Enthusiast Graphics Card, Brings 4K Gaming to the Living Room. Press release, August 27, 2015.

[2] Aaron Ariel, Wilson WL Fung, Andrew E Turner, and Tor M Aamodt. 2010. Visualizing complex dynamics in many-core accelerator architectures. In *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*. IEEE, 164–174.

[3] Henk W De Regt. 2014. Visualization as a tool for understanding. *Perspectives on science* 22, 3 (2014), 377–396.

[4] Weihong Du, Jia Liu, Zujie Wen, Dingnan Jin, Hongru Liang, and Wenqiang Lei. 2024. CARE: A Clue-guided Assistant for CSRs to Read User Manuals. *arXiv preprint arXiv:2408.03633* (2024).

[5] Anders Eklund, Paul Dufort, Daniel Forsberg, and Stephen M LaConte. 2013. Medical image processing on the GPU–Past, present and future. *Medical image analysis* 17, 8 (2013), 1073–1094.

[6] GitHub, Inc. 2025. GitHub REST API Documentation. https://docs.github.com/rest Accessed: 2025-08-15.

[7] Albert Gu and Tri Dao. 2023. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752* (2023).

[8] Everton Hermann, Bruno Raffin, François Faure, Thierry Gautier, and Jérémie Allard. 2010. Multi-GPU and multi-CPU parallelization for interactive physics simulations. In *European Conference on Parallel Processing*. Springer, 235–246.

[9] Yu Huang, Yan Li, Zhaofeng Zhang, and Ryan Wen Liu. 2020. GPU-accelerated compression and visualization of large-scale vessel trajectories in maritime IoT industries. *IEEE Internet of Things Journal* 7, 11 (2020), 10794–10812.

[10] Dave James. Sept. The AMD Radeon R9 Nano Review: The Power of Size. *AnandTech* (Sept).

[11] Maria Knobelsdorf, Essi Isohanni, and Josh Tenenberg. 2012. The reasons might be different: Why students and teachers do not use visualization tools. In *Proceedings of the 12th koli calling international conference on computing education research*. 1–10.

[12] LangChain Contributor Community. 2025. LangChain. https://en.wikipedia.org/wiki/LangChain. Accessed: 2025-08-21.

[13] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2023. Lost in the middle: How language models use long contexts. *arXiv preprint arXiv:2307.03172* (2023).

[14] Songsong Liu, Shu Wang, and Kun Sun. 2024. Having Difficulty Understanding Manuals? Automatically Converting User Manuals into Instructional Videos. *Proceedings of the ACM on Human-Computer Interaction* 8, EICS (2024), 1–19.

[15] David G Novick and Karen Ward. 2006. What users say they want in documentation. In *Proceedings of the 24th annual ACM international conference on Design of communication*. 84–91.

[16] David G Novick and Karen Ward. 2006. Why don't people read the manual?. In *Proceedings of the 24th annual ACM international conference on Design of communication*. 11–18.

[17] OpenAI. 2025. OpenAI API Models Documentation. https://platform.openai.com/docs/models.

[18] Inc. Pinecone Systems. 2025. Pinecone Vector Database. https://www.pinecone.io/. Accessed: 2025-08-21.

[19] Donghao Ren, Bongshin Lee, and Tobias Höllerer. 2017. Stardust: Accessible and transparent gpu support for information visualization rendering. In *Computer Graphics Forum*, Vol. 36. Wiley Online Library, 179–188.

[20] Tal Ridnik, Hussam Lawen, Asaf Noy, Emanuel Ben Baruch, Gilad Sharir, and Itamar Friedman. 2021. Tresnet: High performance gpu-dedicated architecture. In *proceedings of the IEEE/CVF winter conference on applications of computer vision*. 1400–1409.

[21] Yifan Sun, Trinayan Baruah, Saiful A Mojumder, Shi Dong, Xiang Gong, Shane Treadway, Yuhui Bao, Spencer Hance, Carter McCardwell, Vincent Zhao, et al. 2019. Mgpusim: Enabling multi-gpu performance modeling and optimization. In *Proceedings of the 46th International Symposium on Computer Architecture*. 197–209.

[22] Yifan Sun, Xiang Gong, Amir Kavyan Ziabari, Leiming Yu, Xiangyu Li, Saoni Mukherjee, Carter McCardwell, Alejandro Villegas, and David Kaeli. 2016. Hetero-mark, a benchmark suite for CPU-GPU collaborative computing. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 1–10.

[23] Yifan Sun, Yixuan Zhang, Ali Mosallaei, Michael D Shah, Cody Dunne, and David Kaeli. 2021. Daisen: A framework for visualizing detailed gpu execution. In *Computer Graphics Forum*, Vol. 40. Wiley Online Library, 239–250.

[24] Stone Tao, Fanbo Xiang, Arth Shukla, Yuzhe Qin, Xander Hinrichsen, Xiaodi Yuan, Chen Bao, Xinsong Lin, Yulin Liu, Tse-kai Chan, et al. 2024. Maniskill3: Gpu parallelized robotics simulation and rendering for generalizable embodied ai. *arXiv preprint arXiv:2410.00425* (2024).

[25] Niklas von Hertzen. 2011. html2canvas. https://github.com/niklasvh/html2canvas. JavaScript library for rendering HTML elements into canvas. Accessed: 2025-08-21.

[26] Shengqiong Wu, Hao Fei, Leigang Qu, Wei Ji, and Tat-Seng Chua. 2024. Next-gpt: Any-to-any multimodal llm. In *Forty-first International Conference on Machine Learning*.

[27] Mengwei Xu, Wangsong Yin, Dongqi Cai, Rongjie Yi, Daliang Xu, Qipeng Wang, Bingyang Wu, Yihao Zhao, Chen Yang, Shihe Wang, et al. 2024. A survey of resource-efficient llm and multimodal foundation models. *arXiv preprint arXiv:2401.08092* (2024).

[28] Weiye Xu, Jiahao Wang, Weiyun Wang, Zhe Chen, Wengang Zhou, Aijun Yang, Lewei Lu, Houqiang Li, Xiaohua Wang, Xizhou Zhu, et al. 2025. Visulogic: A benchmark for evaluating visual reasoning in multi-modal large language models. *arXiv preprint arXiv:2504.15279* (2025).

[29] Qingchen Zhang, Changchuan Bai, Zhuo Liu, Laurence T Yang, Hang Yu, Jingyuan Zhao, and Hong Yuan. 2020. A GPU-based residual network for medical image classification in smart medicine. *Information Sciences* 536 (2020), 91–100.

[30] Amir Kavyan Ziabari, Rafael Ubal, Dana Schaa, and David Kaeli. 2015. Visualization of OpenCL application execution on CPU-GPU systems. In *Proceedings of the Workshop on Computer Architecture Education*. 1–8.