Visualize the Invisible: Exposing Causal Layers in GPU Performance Analysis through Milestone Abstractions

Yuanhuan Deng, Katherine E. Isaacs, Yifan Sun

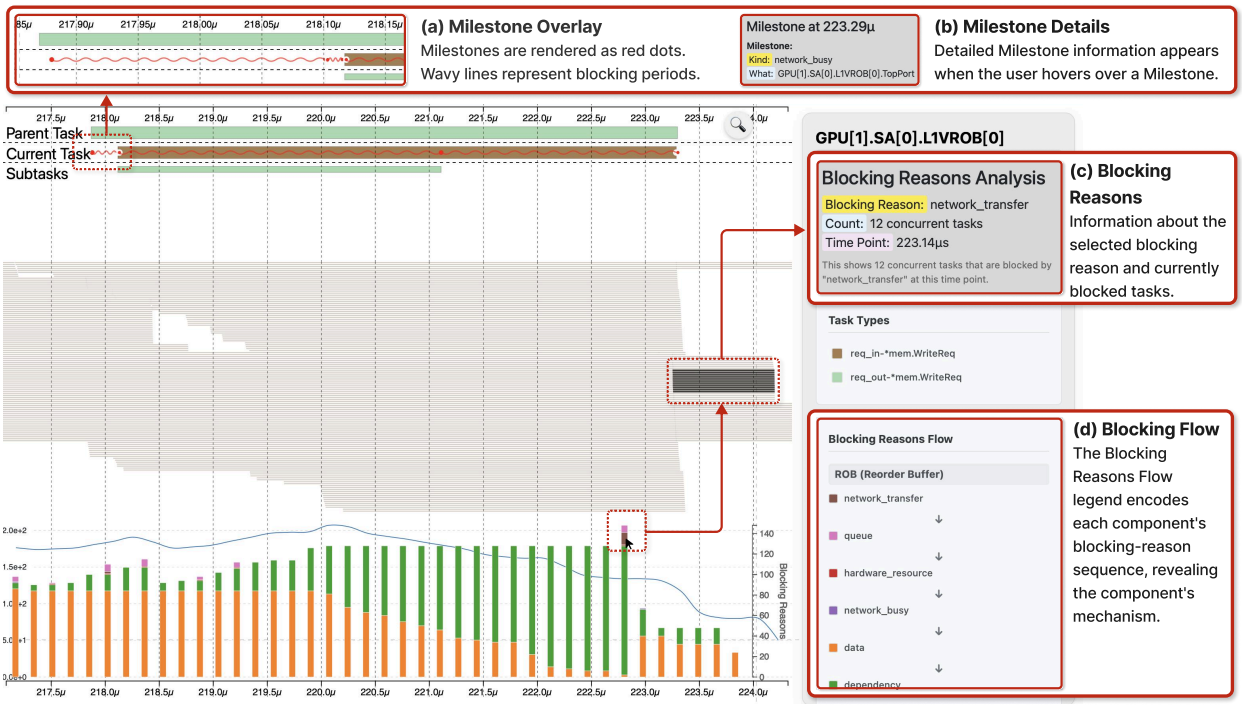

Figure 1: Blocking Reason Analysis for `GPU[1].SA[0].L1VROB[0]`, showing (a) the Milestone overlay, (b) Milestone Details, (c) the Blocking Reasons Analysis panel, and (d) the Blocking Reasons Flow.

Abstract—Visual analytics has the potential to provide critical support in computer architecture design by helping architects interpret execution behavior, but current tools are limited to exposing what happened during execution rather than why, relying on user expertise rather than visual evidence to identify true causes of performance issues. Expertise can help analysts formulate hypotheses, but analysts cannot validate those hypotheses through visualization when causal factors are not visible. We design a visual analytics system that makes causal factors, such as task blocking and underlying mechanisms, visible. At the core of our approach is a conceptual abstraction, the Milestone, a data primitive that simultaneously marks the resolution of prior blocking conditions and assigns reasons to waiting states. Milestones convert execution traces from records of activity into records of causation. We further design three complementary visualizations that leverage Milestones, enabling us to show summaries of blocking patterns, analyze how patterns evolve over time, and navigate causal relationships across parent-subtask hierarchies. We evaluate our approach through a study with GPU architecture researchers, showing that the abstraction and resulting visualizations enable participants to validate hypotheses about an architecture's performance directly from visual evidence. Finally, we discuss how the Milestone abstraction can be adapted to support causal analysis in other expertise-intensive domains.

Index Terms—GPU performance analysis, visual analytics, causal reasoning, execution tracing, milestone abstraction

1 Introduction

As GPUs have become essential infrastructure for modern AI, GPU performance analysis has grown into a critical discipline. Both hardware architects designing next-generation GPU microarchitectures and software developers optimizing applications depend on performance analysis to identify bottlenecks, diagnose inefficiencies, and guide their design decisions, leading to faster and more efficient computing.

Interactive visualization has proven effective in supporting GPU performance analysis by helping architects and developers navigate complex execution data [3, 12, 19]. Prior work has demonstrated that visualization can make instruction flow directly observable [35], support reasoning about resource contention in heterogeneous deployments [43], and provide macro-level views of irregular task execution in parallel systems [13]. Specifically, Daisen [31] advances this direction by supporting task-execution tracing and component-level performance inspection. Daisen is designed as a companion to MG-

- *Yuanhuan Deng is with Brandeis University. E-mail: yuanhuandeng@brandeis.edu.*
- *Katherine E. Isaacs is with the University of Utah. E-mail: kisaacs@sci.utah.edu.*
- *Yifan Sun is with William & Mary. E-mail: ysun25@wm.edu.*

Manuscript received xx xxx. 202x; accepted xx xxx. 202x. Date of Publication xx xxx. 202x; date of current version xx xxx. 202x. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org. Digital Object Identifier: xx.xxxx/TVCG.202x.xxxxxxx

PUSim and explores only the traces MGPUSim produces; MGPUSim executes unmodified GPU programs in simulation, so the traces reflect real workloads rather than synthetic playground loads. Daisen enables architects to examine task timelines, explore execution patterns, and trace dependencies across parent-subtask relationships.

Computer architects use performance analysis tools to understand execution behavior, learn the mechanisms behind performance anomalies, and identify the root causes of bottlenecks. Existing tools, including Daisen, primarily visualize what is easily observable (e.g., instruction pipelines, cache hit statistics, and execution counters) rather than exposing the intermediate causal layers that analysts must currently reconstruct mentally. Even for experienced analysts, this poses a core challenge: reasoning from what can be seen to what cannot, from observable metrics to the hidden causes beneath the surface. Our target users are GPU architecture researchers and hardware architects familiar with GPU execution concepts.

Our requirement study with four GPU architecture researchers (see §4) confirmed this problem in the context of Daisen: participants could observe timing and task behavior, but lacked the evidence needed to confirm or reject their performance hypotheses. All four participants reported that their conclusions relied on prior research experience or familiarity with the simulator, rather than on evidence obtained through the visualization itself. This disconnect between expert reasoning and available visual information is a recurring challenge in expertise-intensive visualization domains [37] where tacit knowledge substitutes for tool support, posing barriers to novices and limiting the reliability of expert conclusions that cannot be visually verified [20].

To address the gap between what analysts can observe and what they need to reason about, we first recognize that the problem extends beyond visualization design: the causal states they need to reason about are transient and absent from standard trace data. We propose a data primitive, *Milestone*, that records transient execution conditions as persistent, queryable records. Focusing on the Milestone data primitive, we then build three complementary visualizations on this shared abstraction. We validate our approach through a user study with GPU architecture researchers, demonstrating that the resulting system reduces reliance on prior expertise for validating performance hypotheses.

Analysts using the redesigned system were able to reason directly from visual evidence, independently forming and verifying performance analyses without fully relying on prior expertise.

This paper makes the following contributions: (1) Milestones, a data primitive that captures both state transitions and blocking reasons during task execution, addressing the often overlooked challenge of data design for causal reasoning, and (2) three complementary visualizations built on this shared abstraction that connect observable task behavior to its underlying causes. We further introduce a characterization of invisibility in expertise-intensive visualization, spanning invisible data, reasoning, causation, and barriers, and use it to organize and interpret our evaluation, before generalizing to other domains that require causal analysis in expertise-intensive settings.

2 Background

We provide the necessary domain background in performance analysis and GPU execution, as well as an overview of Daisen, the visual analytics system we extend to demonstrate our approach.

2.1 Computer Architecture Performance Analysis

Computer architects rely on performance analysis to understand how hardware behaves under specific workloads. In practice, architects work with two primary approaches: (1) profiling on real hardware, which uses built-in instrumentation to capture high-level aggregate metrics such as cache hit rates, instruction throughput, and memory bandwidth utilization; and (2) simulation-based analysis, where architects use software models of hardware designs to execute workloads, generating detailed execution traces. This paper focuses on the second approach, in which the simulator determines what data to record and visualization tools help analysts interpret the resulting data.

The ultimate goal of performance analysis is not simply to observe what happened during execution, but to identify the root causes of performance problems so that architects can propose targeted fixes. An analyst who sees that a workload runs slowly needs to determine which specific component, resource conflict, or dependency chain is responsible before any design improvement can be attempted. Both performance counters and execution traces primarily capture *what* occurred rather than *why*, leaving root-cause identification largely dependent on the analyst's expertise and reasoning.

2.2 GPU Execution Model

GPUs run programs that may contain millions of threads. CPUs offload parallel computation to GPUs by launching functions called kernels. When a kernel launches, groups of threads are dispatched to available GPU cores. The massive scale of concurrent execution, with threads competing for shared hardware resources and depending on each other across components, presents a challenge to visualization systems, making it difficult to reason about performance behavior.

Much of the complexity in GPU performance lies in the memory system. Memory requests from executing threads pass through a multi-level cache hierarchy, from a core-dedicated L1 cache to a shared L2 cache, and then to off-chip DRAM. Along the way, address translation is handled by Translation Lookaside Buffers (TLBs), and pending memory requests are tracked in structures such as Miss Status Holding Registers (MSHRs) and Reorder Buffers (ROBs). Any of these components can be limited by hardware resources and can become a source of blocking when resources are contended or dependencies are unresolved, yet these transient blocking conditions are not captured by standard execution traces.

Critically, these hardware components interact through dependencies that are not always apparent. A memory request from one warp may stall waiting for a TLB translation, which in turn depends on a cache access, which may be blocked by buffer pressure from other concurrent requests. These transient blocking conditions are central to understanding performance but are not directly visible in standard execution traces and visualization tools.

2.3 Daisen

Daisen [31] is a web-based visualization tool designed for detailed GPU execution analysis, built on traces generated by the MGPUSim simulator [30]. At the core of Daisen's data model is the Task abstraction: each unit of work processed by a hardware component (e.g., a memory request, an instruction dispatch) is represented as a task with a start time, end time, and links to its parent and subtasks in other components. Throughout this paper, *task* refers to this hardware unit of work and not to a user's analytical task. Components correspond to hardware units in the simulated GPU, such as compute units, caches, and memory controllers.

Daisen's entry point is a dashboard that lists all components and their task counts, enabling analysts to identify the most active parts of the architecture and select components for further inspection. From the dashboard, analysts can navigate to two primary views. The Component View displays all tasks within a selected component as a horizontal Gantt chart, with task bars on a shared timeline, allowing analysts to see when tasks execute, how long they take, and how they overlap with concurrent tasks. The Task View focuses on a single task, showing its attributes and providing navigation links to its parent and subtasks in other components.

Daisen was intended to support reasoning about execution dependencies through two design strategies. (1) It exposes parent-subtask relationships: when a task is selected, analysts can navigate to its parent or subtasks, allowing them to see which subtask is blocking the parent's progress. (2) It stacks the Task View on top of the Component View, so that analysts can observe a task's execution alongside the overall activity of the component, with the expectation that analysts can infer why a task starts late or finishes late based on the component's workload. In practice, however, these strategies provide only indirect support. Parent-subtask navigation is limited to one level at a time, making it easy to lose context during multi-level exploration. The stacked layout shows temporal correlation between a task and its components' activity, but does not make the causal connection explicit: analysts must still

mentally reconstruct why a task stalled based on what they know about the hardware.

3 Related Work

We discuss related work in performance analysis visualization, causal reasoning through visualization, and knowledge-assisted visualization.

3.1 Performance Analysis Visualization

Visualization tools for execution pipelines and parallel architectures share a common goal: making the runtime behavior of complex systems legible to developers. Prior work separates along three directions.

(1) Visual encoding. Several systems take trace or telemetry data as given and advance the visual representation itself, contributing new encodings and interactive views for parallel-system behavior, network traffic, process monitoring, and large-scale communication [6, 8, 10, 12, 21]. Trace flow visualization extends the same direction through color-encoded pipeline views that make instruction flow and stalls directly observable [35], interactive access to raw execution traces [25], reasoning about resource-contention-driven loss in heterogeneous OpenCL deployments [43], and top-down diagnosis of GPU thread-block memory conflicts and coalescing inefficiencies [26].

(2) Data collection. A complementary effort targets the data itself. Several groups have argued that performance counters alone cannot capture dynamic complexity at scale, advocating for animated, system-level summaries of aggregate program behavior [3, 15], macro-level views that accommodate the temporal irregularity of heterogeneous clusters [13], and the co-design of data collection alongside visualization design [36]. Instrumentation-focused work pursues the same goal by capturing execution signals not previously available for analysis [4, 22–24, 34].

(3) Data abstraction. Between collection and rendering, data abstractions restructure already-recorded traces into forms amenable to reasoning, such as logical-time orderings, call-stack-tree representations, and task-parallel trace structures [16, 28, 40].

Our work builds on (2) and (3) and combines them. We instrument the simulator to emit Milestones (a data-collection contribution) and define the Milestone as a data primitive that augments the trace structure (a data-abstraction contribution). What distinguishes our approach is not that we collect or abstract data, but what we collect and abstract. We go one step further, from capturing what execution leaves behind to capturing the invisible causal state that execution never records.

3.2 Causal Reasoning in Visualization

Visual analytics of causal reasoning generally fall into two categories, depending on where the analytical challenge lies. The first focuses on causal graph discovery, where the goal is to recover or verify causal structure from observed variables. Path diagram layouts have been proposed to expose directional causal flow with interactive model scoring for link verification [32, 33]. The second focuses on causal patterns in event sequences, where the events themselves are already recorded, and the challenge is identifying temporal causal relationships among them [17]. Outflow aggregates event sequences into pathway visualizations with outcome-colored edges for comparing alternative progressions [38], while recent work applies Granger causality via Hawkes processes with user-feedback mechanisms for iterative refinement [42].

Both lines of work assume that the data needed for causal reasoning is already available, whether as observed variables or recorded events. In GPU performance analysis, this assumption does not hold. The intermediate causal states are not captured by traditional task tracing. Our work addresses this prior step: defining a data abstraction that makes these invisible states observable, then designing visualizations that support causal tracing along the exposed chain.

3.3 Knowledge-Assisted Visualization

In visualization domains characterized by deep domain knowledge, prior work has consistently demonstrated that existing tools impose substantial prerequisite knowledge on end users [11, 14, 18, 20, 41], creating significant comprehension barriers for novice analysts. A particularly persistent challenge lies in how expert practitioners routinely perform implicit "mental corrections" during analysis, drawing upon internalized background knowledge to interpret and adjust visual information in ways that are neither surfaced nor captured by the visualization system itself. These subjective assumptions, while central to expert reasoning, remain structurally unrecorded in existing visual analytics tools [20].

The Knowledge-Assisted Visual Analytics (KAVA) theoretical framework [11] has directly engaged with the question of how visualization systems might bridge the expert knowledge gap, proposing that expert knowledge can be inferred and extracted from observable user analytic behaviors at runtime. Building upon this foundation, subsequent research has operationalized the KAVA model by constructing unified knowledge repositories [2, 9] that consolidate expert knowledge and establish guided interactions with end users by annotations, modeling, and recommendation [1, 5, 7, 29, 39].

While prior work on knowledge-assisted visual analytics has thus addressed how systems can leverage formalized knowledge during run-time analysis [11, 27], little work has investigated how to encode expert reasoning into visualization design itself. This reveals a unique opportunity: developing design-time methodologies that transform invisible cognitive procedures into explicit visual affordances.

4 Requirement Study

We conducted a preliminary user study with four GPU architecture researchers, including three doctoral students and one researcher holding a PhD, with GPU research experience ranging from 1–3 to 5+ years (see more participant background in supplementary materials). Our findings revealed difficulties in validating hypotheses due to information about dependencies being unavailable visually.

4.1 Requirement Study Method

We describe the study procedure and analysis. This research was approved by the Institutional Review Board (IRB) at our institution (IRB-2024-25). At the start of each session, we obtained verbal informed consent from all participants, both for participation and for audio recording; the consent script is included in the supplemental materials.

Procedure. We interviewed participants remotely via Zoom. Each interview session lasted approximately 60 minutes and was recorded and transcribed. The study was structured in two sections:

Section 1 consisted of general questions to understand participants' research domains and their usage patterns of GPU simulators, and to identify the data metrics they prioritize during the performance analysis.

Section 2 used a scenario-based approach. We prepared four representative scenarios in Daisen [31], covering diverse analytical challenges, and provided participants with remote access during the interview. Participants were asked to freely explore each scenario while sharing their screens. For each scenario, we asked participants to: (1) observe and identify distinctive patterns within each scenario, (2) conduct performance analysis based on the visualization, and (3) explain their analytical reasoning and conclusions.

We specifically asked participants whether they required additional information to validate their conclusions or whether modifications to the visualization would enhance their analytical capabilities. When participants could not articulate specific suggestions, we presented prepared mockups (early design ideas as static prototypes drawn in Figma) to ground their feedback. The detailed study guide can be found in supplementary files.

Analysis. We analyzed the transcripts using an open-coding approach. The lead researcher first read all the transcripts and then coded them, identifying recurring themes related to analytical barriers, missing information, and desired tool capabilities. Codes were then discussed within the whole research team and consolidated through iterative comparison until agreement was reached.

4.2 Requirement Study Findings

Our interviews revealed that while all participants found Daisen helpful for performance analysis, they consistently encountered barriers when

attempting to validate their analytical conclusions:

Participants could not validate their analyses through visual evidence alone. All four participants conducted analyses based on their prior knowledge, but were unable to verify their conclusions through the available visualization. P1 arrived at his conclusions based on close familiarity with Daisen's internals, while P2, P3, and P4 stated that their reasoning relied entirely on research and professional experience. P3 described Daisen as "very useful" and P2 appreciated that its tracing design clearly delineates dependencies across tasks, yet neither could point to visual evidence that confirmed their performance hypotheses.

Participants requested explicit visualization of dependencies between architecture components. Three out of four participants recommended incorporating a component dependency map. P1 suggested that the system should show more information about the components or the parameters of the components, while P3 emphasized that displaying the dependency between components would provide better support for visually tracing parent tasks and subtasks. P4 reinforced this notion by proposing that dependencies be drawn directly in the component view to show cross-component relationships.

Participants requested granular hardware state metrics. All participants stressed the need to expose dynamic resource states that were not available in the visualization. The requested metrics fell into three categories: (1) buffer and queue pressure (P2 noted that awareness of buffer status and ROB state could explain performance gaps; P4 requested information on queued requests and buffer size), (2) cache behavior (P1 indicated that knowing maximum cache capacity would help; P3 recommended displaying cache hit/miss rates), and (3) port status (P3 recommended revealing port origins to trace data flow across components). P4 suggested that marking state transitions on task bars would aid validation, noting that such markers are "really necessary."

Participants observed low efficiency in navigation and information retrieval. P2 and P3 reported that locating specific information required excessive clicking and was time-consuming. P2 described the process as "clicking on components and starting to dig deeper," which hindered quick access to necessary data. P3 noted that displaying summary statistics (e.g., the number of concurrent tasks) inline would reduce the steps needed to validate an analysis.

4.3 Design Goals

The findings above reveal a gap not only in how execution data is visualized, but in what data is available for visualization in the first place. Our work faces a prior design question: the causal states that analysts need to reason about are transient and absent from standard trace data. An improved data primitive design is a prerequisite for, rather than a byproduct of, the visualization design process. Therefore, we derive the following design goals from our findings:

DG1: Expose Component Dependencies. The visualization should expose how components depend on one another for the tasks under analysis. Without such contextual information, analysts struggle to correlate performance anomalies with upstream or downstream task interactions.

DG2: Expose Invisible Resource Contention. The visualization should surface dynamic, transient information (e.g., buffer pressure, port saturation, blocking reasons) that impacts performance but remains absent from current trace data and tools. Analysts should be able to map tasks and their resource states both temporally and spatially.

DG3: Reduce Navigation and Cognitive Overhead. The design should shorten the time required to locate key information, streamline navigation between related data elements, and reduce the number of interactions needed to reach critical evidence, while avoiding visual clutter that distracts from analytical focus.

5 Data Abstraction

We introduce *Milestones*: timestamped records that mark the moments when a blocking condition is resolved. Each Milestone simultaneously encodes two pieces of information: (1) the event that just occurred (e.g., a data transfer finished, a hardware resource became available) and (2) the *blocking reason*—what prevented the task from progressing until that moment. This dual encoding converts execution traces from records of activity into records of causation.

Table 1: Milestone Data Structure

Field	Format	Description
ID	String	Unique identifier for the Milestone (`id`)
TaskID	String	The ID of the associated task (`task_id`)
Time	Float/Time	The timestamp of the Milestone (`time`)
Kind	Enum	Classification of the blocking reason (see Table 2)
What	String	Description of the blocking reason

Table 2: Milestone (Blocking Reason) Taxonomy

Value	Description
hardware_resource	Waiting for a hardware structure (e.g., buffer slot, MSHR entry, ROB slot) to become available
network_transfer	Waiting for data to be transmitted across the on-chip network
network_busy	Waiting for a network port or link to become free due to congestion
queue	Waiting to reach the head of a processing queue behind other tasks
data	Waiting for required data (e.g., a cache line or memory response) to arrive
dependency	Waiting for a prerequisite task to complete before proceeding
translation	Waiting for a virtual-to-physical address translation (e.g., TLB lookup) to complete

Milestone Structure. A *Milestone* entry comprises several properties, as shown in Table 1. Each Milestone is assigned a unique identifier and a timestamp recording the precise moment the blocking condition was resolved. The Kind field classifies the blocking reason according to our taxonomy (Table 2). The What field provides a descriptive label for the specific event (e.g., "head" for reaching the front of a queue, or the name of the buffer that became available).

Each Milestone is associated with exactly one task through its TaskID field, and a task may have zero or more Milestones. The Milestones of a task partition its execution duration into a sequence of phases: the interval between two consecutive Milestones represents a period during which the task was blocked by the reason recorded in the later Milestone. A task with no Milestones has only its start and end times recorded, as in traditional tracing. A task with multiple Milestones provides a fine-grained timeline of what the task was waiting for at each point during its execution. Since each task already carries location information identifying which component processes it, Milestones inherit this context through the task association and do not need to redundantly record location. This one-to-many relationship between tasks and Milestones allows the Milestone abstraction to augment existing trace data incrementally, preserving backward compatibility while adding causal depth where instrumentation is available. To make the contrast explicit, a traditional Daisen task records only its start time, end time, parent, and subtasks. A Milestone augments the same task with a sequence of timestamped, typed blocking records (Table 1) that partition the execution into labeled blocking phases, while the original task fields remain unchanged.

Milestone Taxonomy. We define blocking-reason categories (see Table 2) based on our requirement study findings and the GPU execution model described previously. The categories correspond to the distinct resource types and interaction mechanisms that can cause a task to stall: hardware resources (e.g., buffer capacity), network conditions (transfer latency, congestion), queue occupancy, data availability, inter-task dependencies, and address translation. The Milestone taxonomy is extensible, allowing additional simulators or architectures to add domain-specific extensions while preserving the core structure.

Milestone Collection. Milestones are emitted by the simulator at runtime as we make the implicit data explicit. We instrument each component's control flow by inserting a function call at the points

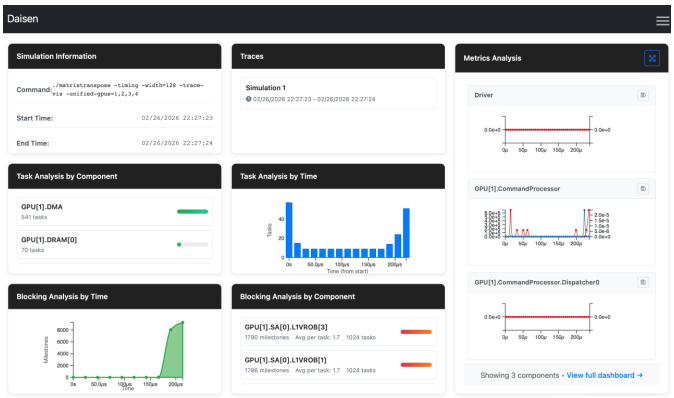

Figure 2: The new Daisen dashboard provides multiple entry points into performance analysis, allowing analysts to begin from task distributions, blocking reason summaries, or component-level metrics, depending on their analytical focus. The Blocking Analysis widgets (bottom row) surface system-level Milestone data, showing when and where blocking concentrates across the simulation.

where a task makes forward progress, indicating the resolution of a blocking state. Each Milestone has a Kind and a What field that records the previous blocking reason. We provide an extensible taxonomy (see Table 2) of the Kind field based on domain experts' experience. Other fields include the timestamp and the owning task ID.

Linearization Simplification. In practice, a task may be waiting on multiple resources simultaneously, and the true blocking structure can be a partial order rather than a sequence. While using more sophisticated data structures (e.g., a graph) can better represent blocking structure, implementing them can be prohibitive for two reasons: (1) Recording graph-based Milestones and dependencies significantly complicates the simulator due to the requirement of recording more context. (2) Visualizing a graph or other complex structure in the already cluttered Daisen interface can be challenging. We find that a linear blocking reason is sufficient for users to analyze blocking reasons with the help of existing Daisen data and visualizations.

6 Visual Design

Our proposed design extends the Daisen visualization tool with three new visualizations, each built on the Milestone abstraction and addressing a specific analytical need identified in our requirement study.

6.1 System-Level Blocking Overview

The redesigned Daisen dashboard provides analysts with a system-wide view of blocking activity, helping them spot which components and tasks are worth a closer look. The dashboard introduces two new widgets built on the Milestone abstraction (see Figure 2): (1) The Blocking Analysis by Component widget visualizes Milestone counts per component using horizontal bars, with total Milestone count, average Milestones per task, and total task count overlaid on each bar. The average-per-task metric normalizes comparison across components with different task volumes, allowing analysts to quickly identify which components experience the most blocking relative to their workload. (2) The Blocking Analysis by Time widget uses an area chart to show how blocking intensity varies over the course of execution, revealing whether blocking is concentrated in specific phases or distributed throughout. Clicking on a component in either widget navigates to the Component-Level Blocking Analysis for that component.

The dashboard retains several widgets from the original Daisen interface: Simulation Information, Traces, Task Analysis by Component and by Time, and a Metrics Analysis panel, which we leave unmodified.

Concrete example. For the `matrixtranspose` benchmark, the Task Analysis by Component widget shows `GPU[1].DMA` handling the most tasks (541), yet the Blocking Analysis by Component widget tells a different story. `GPU[1].SA[0].L1VROB[3]` accumulates 1,790 Milestones at 1.7 per task across 1024 tasks, the highest blocking load, which marks it rather than the busiest component as the primary candidate. The Blocking Analysis by Time widget shows blocking concentrates in the later execution phase (rising sharply toward 200 μs) rather than spreading evenly. Clicking the ROB bar navigates directly to its Component-Level Blocking Analysis. This supports DG3 by providing a fast entry point that reduces the navigation needed to locate where blocking concentrates.

6.2 Component-Level Blocking Analysis

Daisen's Component View displays all tasks running within a selected hardware component on a shared timeline, intended to show how tasks and hardware components interact. While analysts can observe task concurrency and duration, they cannot see why tasks stall or what resources they are waiting for. To address this, we design the *Component-Level Blocking Analysis* (see Figure 1), which overlays blocking reason information directly onto the component's temporal context, enabling analysts to see not only when tasks execute but what prevents them from making progress.

The blocking reason information is encoded as a stacked bar chart positioned below the task timeline. At each time point, the visualization identifies all tasks that have not yet reached their next Milestone and computes the blocking reason for each. Each vertical bar represents a time point, with the bar's total height encoding the number of blocked tasks at that moment. Within each bar, colored segments are stacked vertically, where each segment's height represents the count of tasks blocked by a specific reason (see Table 2). In Figure 1, the component's tasks appear as horizontal bars in the middle region. Tasks that do not match the current selection are shown in faint grey so the highlighted tasks stand out. The x-axis is simulation time in microseconds with the blocked-task count on the right y-axis. The overlaid line is the concurrent task load.

The stacked blocking reason bar chart is tightly coupled with the task timeline above it. Clicking on a bar segment, which represents a specific blocking reason at a specific time point, highlights all tasks currently blocked by that reason in the task view, and the Blocking Reasons Analysis panel (Figure 1(c)) reports the selected reason, the number of concurrent blocked tasks, and the time point. This allows analysts to move from an aggregate observation (e.g., "dependency blocking is growing") to the specific tasks impacted, and to further investigate their attributes, parent/subtasks, or Milestones to understand the root cause.

The Blocking Reasons Flow widget (Figure 1(d)) displays the component-specific sequential order of blocking reason types. For each component type, blocking reasons are listed in the order they typically occur as a task progresses through that component's execution pipeline. For example, in the ROB component, the ordering is: `network_transfer`, `queue`, `hardware_resource`, `network_busy`, `data`, `dependency`. A task entering the ROB first waits for network transfer, then queues for processing, then waits for hardware resources, and so on. This ordering is derived from the control flow written in the simulator code. The panel serves as a reference for analysts to understand the component's mechanisms. By reading the flow from top to bottom, analysts can see the expected progression of blocking stages without needing to consult the simulator source code directly.

Concrete example. We walk through a concrete example (see Figure 3) to understand how blocking reason analysis can be performed. The component is a reorder buffer (ROB; component name: `GPU[1].SA[0].L1VROB[0]`), which ensures that received memory requests complete in order. In the beginning, the orange segments (blocking reason: `data`) dominate the execution, indicating that the component is primarily blocked because the connecting memory module has not yet provided the requested data.

The pink segments (`queue`) appear briefly throughout the chart. Their transient presence shows that requests sometimes wait in the queue for some time. They are not dominating because their episodes are short-lived, suggesting that the ROB has sufficient entries overall.

The green segments (blocking reason: `dependency`) reveal a more interesting pattern. In an ROB, dependency blocking occurs when a task

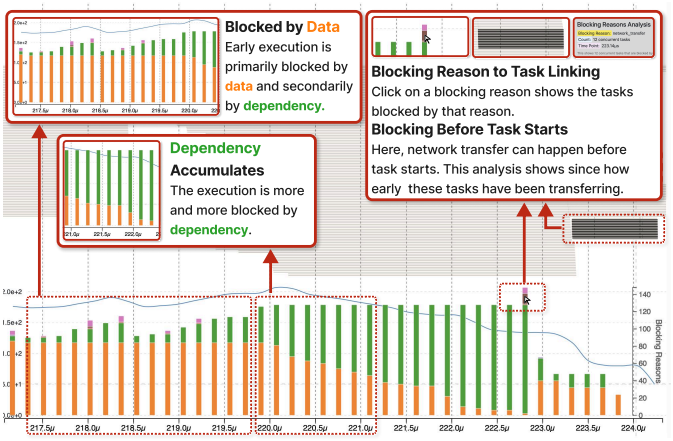

Figure 3: An example of how to perform a blocking reason analysis.

cannot finish because an earlier task has not yet completed, enforcing reorder constraints. The green segments first appear around 220.0μs and grow progressively taller: initially, execution proceeds smoothly, but then one or more tasks begin to block the rest of the pipeline, causing an increasing number of tasks to stall for the same reason. By approximately 222.0μs, the situation resolves. A brief pink (blocking reason: `queue`) and brown (blocking reason: `network_transfer`) follow, indicating that new requests have arrived at the component.

To investigate further, the analyst can click on a brown `network transfer` segment at time 222.8μs. As shown in Figure 1, the system reports that 12 tasks are blocked by this reason at that moment. Interestingly, these tasks have later start times. The later start times show how long these tasks wait for network transfer before execution begins. Overall, the blocking reason analysis enhances Daisen's ability to understand the invisible factors that impacted execution, addressing DG2 by surfacing the otherwise-invisible blocking reasons behind each stall.

6.3 Task-Level Blocking Analysis

The Component-Level Blocking Analysis reveals blocking patterns across many tasks within a component, but performance diagnosis often requires a deep understanding of a single task: what sequence of blocking states it encountered, how long each phase lasted, and how its subtasks across other components contributed to its overall duration. To support this level of investigation, we design a task-level blocking analysis method that focuses on a single task and exposes its internal Milestone sequence, along with its full subtask hierarchy.

The first element of the Task-Level Blocking Analysis is the Milestone overlay on task bars (see the top part of Figure 1(a)). Each Milestone is rendered as a red dot positioned at its temporal location on the current task bar, and wavy lines connect consecutive dots to indicate the blocking periods between them. The dots and lines transform the bar from a uniform duration indicator into a segmented record of execution phases. When multiple Milestones coincide at the same time point, the dot is rendered larger with a contrasting border to signal temporal density and prompt the analyst to zoom in for individual inspection. Clicking on a Milestone displays its attributes (kind, what, and timestamp) in the right-side information panel (Figure 1(b)). Together, the overlay and interaction allow analysts to read the sequence of blocking states a task encountered without leaving the Component View.

The Milestone overlay on the current task bar provides a quick summary of blocking states, but the limited screen space of a single task bar constrains how much detail can be shown. To dig deeper into a task's execution, a magnifier button on the top-right corner of the Task View opens the Dissect View for the currently selected task. The Dissect View (see Figure 4) provides the task's full Milestone sequence, its parent task for upstream context, and its complete subtask hierarchy across unlimited depth levels, all on a shared time axis.

In GPU execution, a task in one component often spawns subtasks in other components, which in turn spawn further subtasks of their own. Understanding why a task stalls frequently requires tracing this chain across multiple components and hierarchical levels. The previous Daisen interface restricted navigation to one parent-subtask level at a time, forcing analysts to click in and out repeatedly and mentally reconstruct the full picture. The Dissect View addresses this by displaying the complete subtask hierarchy of the selected task, with subtasks rendered using depth-based coloring and horizontal indentation to encode nesting level. Analysts can expand subtasks on demand across unlimited levels, keeping the causal branch visible in a single view.

Below the subtask hierarchy, a Milestone achievement bar visualizes the selected task's full Milestone sequence as a segmented progress bar. Each colored block spans the interval between two consecutive Milestones. Zooming in separates overlapping Milestone markers and reveals labels identifying each blocking reason along with a completion flag at the final Milestone.

All elements of the Dissect View, including the parent task, current task, subtask hierarchy, and Milestone achievement bar, share the same time axis as the original Task View. Panning and zooming are always synchronized across all elements, ensuring that analysts maintain their temporal orientation throughout the analysis. The shared axis enables vertical scanning, where at any time position, analysts can simultaneously observe which subtasks are active, what Milestones are being achieved, and what the parent task is doing, making cross-component causal relationships readable through visual alignment rather than mental reconstruction.

Concrete example. Toggling a selected `L1VROB[3]` task (`req_in`, `*mem.ReadReq`) into the Dissect View (see Figure 4), the Milestone achievement bar reads its execution as consecutive phases. Clicking a Milestone reveals its attributes. For example, Milestone 1 (`queue`, *head*) is followed by Milestone 2 (`hardware_resource`, the ROB *Buffer*), showing the task first waited in the queue and then for a buffer slot. As each subtask is labeled with its owning component, the cross-component subtask hierarchy makes visible how a task depends on work carried out in other components, addressing DG1. It also addresses DG3 by consolidating multi-level navigation into a single view.

7 User Study

We validate the utility of our Milestone abstraction and subsequent visualizations through a user study with six GPU architecture researchers: five doctoral students and one researcher holding a PhD, with GPU research experience ranging from 1–3 to 5+ years. All had GPU-simulation experience; four were familiar with Daisen, and two had limited prior exposure. Three participants had participated in the requirement study, while three were new. Detailed participant backgrounds are provided in the supplementary materials. This study was approved under the same IRB protocol as the requirements study.

7.1 User Study Method

Sessions were conducted remotely via Zoom, recorded with participant consent, and lasted about 60 minutes each. Each session followed a semi-structured format. We began with background questions: for returning participants, whether they had continued working with performance analysis since the first study, and whether they recalled specific pain points; for new participants, their research context, simulator usage, and typical diagnostic workflow. We then provided a brief overview of the four new features (the redesigned dashboard, blocking reason analysis, Milestone markers on task bars, and the Dissect View). Next, participants were given a simulation dataset and asked to identify performance bottlenecks with supporting visual evidence using the new features, while sharing their screens. The session closed by asking participants to reflect on whether the new features helped them validate their performance analyses compared to the previous version of Daisen.

We transcribed all sessions and analyzed them using the same open-coding approach as in the requirement study. Two researchers independently coded the transcripts, identifying recurring themes related to analytical capability, validation strategies, and remaining barriers. Codes were discussed and consolidated through iterative comparison.

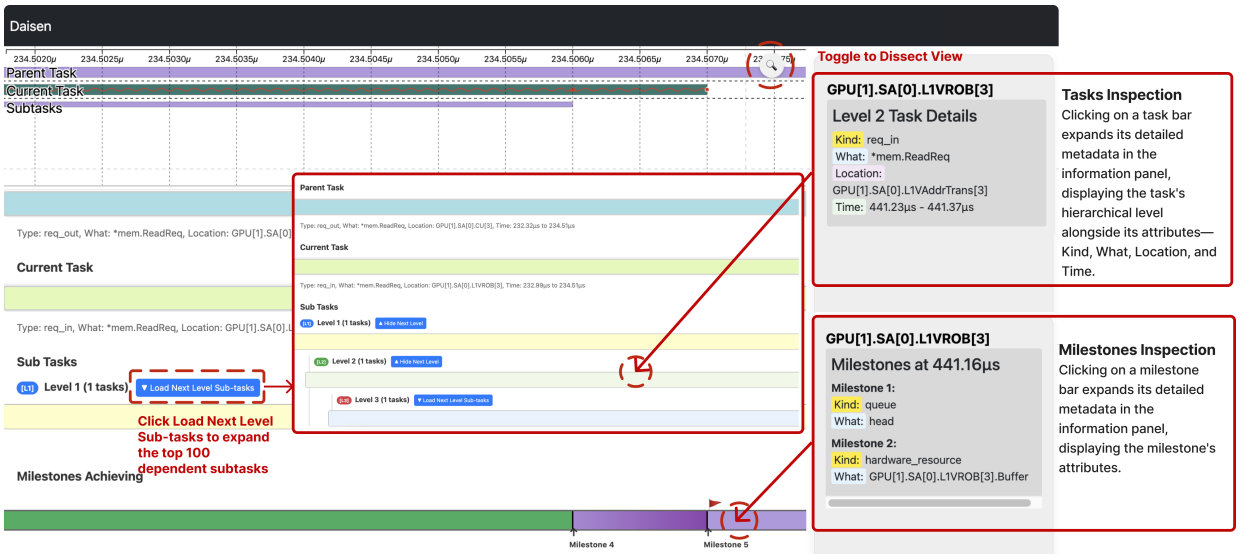

Figure 4: The Dissect View for hierarchical causal tracing. This view consolidates parent, current, and subtask timelines on a shared time axis, replacing the previous one-level-at-a-time navigation. Subtasks can be expanded on demand across unlimited depth levels. The Milestone achievement bar (bottom) segments the selected task's execution into consecutive blocking phases, each color-coded by reason.

7.2 User Study Findings

All participants engaged with the new features along the intended workflow and demonstrated independent analytical reasoning without intervention throughout the sessions. We organize our findings around four layers of invisibility that the system exposes (see Figure 5), each capturing a different reason why causal information goes missing. *Invisible data* is what the original Daisen system did not record. *Invisible reasoning* is the inference the analyst makes mentally, but the tool never shows. *Invisible causation* is a causal chain spanning levels the analyst cannot simultaneously view. *Invisible barriers* are expertise gaps that keep users from interpreting what is shown. Each finding below maps to one layer; we revisit the framework in Section 8.

Figure 5: A conceptual characterization of layers of invisibility in expertise-intensive visualization.

7.2.1 Participants identified bottlenecks from the visualization.

In the requirement study, all participants confirmed they could not validate their analysis through Daisen alone, relying entirely on prior expertise instead. In this study, both P1 and P2 demonstrated reasoning directly from visual evidence. P2 described a clear inference strategy: *"Whichever blocking reason has a larger proportion in a branch, I conclude it is a bottleneck,"* and independently identified a component-level bottleneck from the dashboard by explaining, "The ROB has average per task 1.7 and 1024 tasks, so it should be the bottleneck for this benchmark." P3 used the blocking reason distribution to confirm a prior hypothesis: *"Our intuition is that it is because of the queue, and this gave us the reason. It is just what our intuition was correct."* P3 went further, articulating how the visualization could redirect research priorities: *"In the past, it's like the researcher, they found that, okay, we think that TLB is a bottleneck. Then we analyzed, and we found the evidence to prove our idea is correct. If we have a tool like Daisen, it can instead tell us what the performance bottleneck is, and then give them the research direction."* This shift, from researchers seeking evidence to confirm pre-existing hypotheses to the visualization proactively surfacing bottlenecks, illustrates a qualitative change in the analytical workflow.

P1 assessed the blocking reason flow as the most actionable part of the new design: *"I can indicate what the relationship is between different blocking reasons, and I can further double-check them from the parts below."* P1 demonstrated this by panning the task view temporally and observing that two blocking reasons (`hardware_resource` and `network_busy`) disappeared after a peak in concurrent tasks, independently concluding that these blocking conditions were concentrated in a specific execution phase. P5, navigating the blocking reason analysis in an address translator component, identified two dominant blocking categories and stated: *"There are two bottlenecks. The first one is translation, and the second is data."*

Across these participants, we observe that analysts in our study could form specific, component-level performance judgments directly from the blocking reason visualization, without relying on external analysis or prior familiarity with the workload. Our study confirms the *invisible-data* layer was addressed: the evidence analysts previously lacked is now present and directly readable.

7.2.2 Milestones made execution dynamics explicit.

First, participants were able to identify which blocking reason dominated execution within a component. P3 reasoned from the blocking reason stacked bar, and said *"It can give us some intuition on which one dominates our performance bottleneck inside this ROB."* P3 also recognized that the temporal visualization captured a phenomenon discussed in prior architecture research: *"In some prior MICRO paper, they talk about how the performance bottleneck changed along with the execution, so it's not like if it is always an ALU bottleneck. I think this figure can show the same intuition that they gave."* This observation validates that the blocking reason analysis captures execution dynamics that were previously accessible only through manual data extraction. P3 confirmed this directly: *"In the past, I usually draw the data and then just manually do it."*

Second, participants were able to reconstruct the temporal sequence of blocking states within a single task. P3 read the Milestone sequence on a task bar and derived the execution narrative without guidance, *"It means that in the beginning, when we issue to this component, we waste*

our time on the queue, and when we commit this one from this component, most of the time we spend is on dependency." P1 independently connected a Milestone marker to a subtask transition: *"I see a Milestone named translation. So I assume at this time point, a translation request has been finished, because the translation keyword disappeared at this time point. Furthermore, I see a Milestone point here [that] indicates there should be something about translation happened."* P1's reasoning chain, connecting a Milestone's label to the disappearance of a subtask type, demonstrates that Milestones bridge the gap between task-level observations and causal explanations.

Third, participants were able to connect blocking evidence to architectural causes. P3 recognized the architectural implication of a translation Milestone immediately: *"Oh, translation. That makes sense. It is address translation, so it will wait for TLB."* P6 described a scenario where blocking analysis would be essential: *"When they consider the multiple GPU system, they will have communication between these multiple GPU processes... I can know what happened exactly, and what's the reason why it stopped sending the new message. So maybe I think it can give me an explanation."* P2 confirmed that these capabilities together addressed the core validation problem from the first study: *"I think it has been solved, largely solved this problem already."* The qualifier "largely" is worth noting: P2 recognized that while the Milestone abstraction substantially closes the gap between observation and causal reasoning, some analytical steps still require domain expertise that visualization alone cannot replace. Overall, being able to connect evidence with architectural behavior corresponds to the *invisible reasoning* layer: inferences once performed silently became observations readable from the visualization.

7.2.3 Participants were able to trace causal relationships across hierarchical levels without losing context.

In the previous interface, multi-level navigation was a persistent pain point. P1 described his experience with the earlier interface: *"After a period of time, when I find my clues and I want to go back, I will fully lose my way. I have no idea how to go back to that certain task view again, because each parent may have multiple children, and if we just go to the top layer, it is hard to find the way back."* The Dissect View resolved this problem. In this study, P1 stated, *"I love this feature, to be frank,"* and noted that analysis had become reproducible: *"Under this feature, I think most of the analysis is reproducible. We made some analysis, and I can double-check them again and again, and see exactly the same feature again."* The word "reproducible" is important to note as it indicates that the Dissect View transforms performance analysis from a one-shot exploration into a verifiable, repeatable process. P2 confirmed: *"This is much better than before. Navigating subtasks is much more convenient. I can also see the task hierarchy clearly and identify where problems might occur."* P3 noted that the hierarchical view eliminated manual reconstruction: *"In the past, I will click in, click out, and draw a figure by myself to find it. And this one can have us automatically do it."* P6 echoed this: *"If you don't have this function, you have to click this subtask again and again to get the next one."*

Across all participants who commented on the Dissect View, the consistent theme was that consolidating hierarchical navigation into a single, temporally aligned view removed a structural barrier to multi-level causal tracing that had previously forced analysts to maintain context mentally or through external tools. Combining multi-faceted data into one view maps to the *invisible-causation* layer: causal chains that previously had to be reconstructed mentally are now visible in a single aligned view.

7.2.4 The new features reduced but did not eliminate reliance on domain expertise.

P2 described a general analysis workflow that the visualization now directly supports: *"First I will look at which component takes more time, then go into it, and then look more carefully at which blocker takes more time, and which subtask takes longer, and figure out which GPU component needs to be optimized."* P2's top-down strategy, from component overview to blocking reason to subtask inspection, is precisely the analytical flow encoded in the system's multi-level design.

P1, who has spent over 100 hours using Daisen, confirmed the features were effective for experienced users. Meanwhile, they noted that users without a solid background in GPU architecture may still struggle to interpret the terminology, suggesting that documentation or in-tool guidance (such as hoverable explanations) would be beneficial. P4, who had less GPU architecture experience, corroborated this: *"It feels a bit like there's no guidance. Newcomers would still struggle."* P2 made a related observation about the Milestone abstraction: *"Looking at this chart, I can tell which blocking reason has a larger proportion, so I conclude it is a bottleneck. But I don't need to understand the Milestone concept itself to reach that conclusion."* This finding suggests that the blocking reason visualization is intuitive in its visual encoding, even when the underlying data abstraction is unfamiliar, though users would benefit from explicit guidance connecting the two. P6 similarly noted: *"If the blocking analysis is necessary for the researcher, then I can know what happened exactly. I think it can give me an explanation."* P6's framing of the visualization as providing "an explanation" rather than merely displaying data captures a key outcome of the redesign: the system shifts from showing what happened to articulating why, reducing the inferential burden that previously fell entirely on the analyst's expertise.

From blocking evidence to optimization. The blocking evidence also points analysts toward the lever to adjust. When `translation` dominates on an address translator, the indicated lever is translation or TLB capacity. The tool localizes the bottleneck and the responsible resource, while selecting the precise fix still draws on architectural expertise, as P4 observed that they could tell a task was blocked on data yet not how to optimize it. This result maps to the *invisible-barriers* layer. The evidential barrier was largely removed, leaving a terminological barrier discussed in §8.

8 Discussion

We discuss how analyzing what remained invisible at both specific and abstract levels informed our design.

8.1 Towards a Characterization of Invisibility in Expertise-Intensive Visualization

Visualization research [5, 11, 37, 41] in expertise-intensive domains has long recognized that analysts bring substantial background knowledge to their interpretation of visual displays. For example, Lin et al. [20] have examined how expert practitioners perform implicit "mental corrections" during analysis, drawing on internalized knowledge that remains structurally unrecorded in the visualization system itself, and personal knowledge routinely supplements and sometimes contradicts what the visualization shows. The KAVA framework [11] has proposed that expert knowledge can be inferred from observable analytic behaviors at runtime. These contributions share a common concern: something important is invisible, and that invisibility impedes analysis. Our work suggests that invisibility in expertise-intensive visualization is not a single phenomenon but operates at distinct layers, each with different causes and requiring different design interventions. The four layers we propose may help designers in other expertise-intensive domains diagnose where their tools fall short.

Invisible data. The most fundamental form of invisibility occurs when the states analysts need to reason about are never recorded. In GPU performance analysis, transient blocking conditions resolve during execution without producing any trace record. No visual encoding can surface what does not exist in the data. This layer is invisible, not because of a design choice to hide information, but because the instrumentation was never built to capture it. The appropriate intervention is data primitive design: defining new abstractions, such as our Milestone, that convert transient states into persistent, queryable records. Our requirement study confirmed this diagnosis: all four participants could form hypotheses but could not validate them, not because the visualization was inadequate, but because the underlying data lacked the causal information they needed.

Invisible reasoning. Even when data is present, the inferential steps that connect observations to conclusions may remain internal to the analyst. An experienced GPU architect who sees a timing gap

on a task bar may immediately recognize it as cache contention, but this reasoning is neither visible to others nor captured by the tool. Prior work has addressed this layer through knowledge repositories, annotation systems, and recommendation mechanisms that attempt to externalize and share expert reasoning. Our work does not directly address this layer, but the Milestone abstraction partially reduces its scope: by making blocking reasons explicit in the data, it converts some inferences that were previously invisible (performed entirely in the analyst's head) into observations that are visible (readable from the visualization). Recall that P3's comment that *"our intuition is that it is because of the queue, and this gave us the reason"* illustrates this conversion: what was previously an unsupported intuition became a verifiable observation.

Invisible causation. In systems where behavior emerges from chains of interacting components, the causal relationships connecting a root cause to an observable symptom may span multiple levels of a hierarchy. An analyst examining a slow task in one component may need to trace through parent tasks, subtasks, and cross-component dependencies to identify the underlying cause. Even when each individual level is visible, the causal alignment across levels may not be. In the previous Daisen interface, analysts could inspect any single level but lost context when navigating between levels, making it impossible to hold the full causal chain in view simultaneously. The Dissect View addresses this layer by encoding the parent-subtask-Milestone structure in a single, temporally aligned layout that supports vertical scanning across hierarchical levels. P1's observation that this made analysis "reproducible" reflects the shift from navigation that consumes causal context to navigation that preserves it.

Invisible barriers. The expertise gap itself can be invisible to tool designers. The conventional framing treats domain knowledge as a monolithic prerequisite: either users have enough expertise, or they do not. Our two-phase study revealed a finer structure. In the requirement study, the barrier was evidential: the data lacked the information analysts needed, and no amount of expertise in using the tool could compensate. In the evaluation study, the evidential barrier was substantially resolved, but a terminological barrier remained: users could read the visual encodings but struggled with domain-specific labels like "dependency" or "network_transfer." P2 could identify bottlenecks from blocking proportions without understanding the Milestone concept itself. Collapsing these two barriers into a single "expertise gap" obscures the fact that they require fundamentally different interventions.

These four layers are not exhaustive, and we do not claim they constitute a complete taxonomy. Other forms of invisibility likely exist in other domains: invisible assumptions in model-based analysis, invisible biases in data collection, or invisible constraints imposed by tool architecture. Rather, we offer this characterization as a diagnostic lens. When a user study reveals that analysts "cannot validate their hypotheses," the layers suggest distinct questions, such as "Is the relevant data captured at all? Are the inferential steps externalized? Is causal alignment preserved across levels? Is the barrier evidential, terminological, or something else?" Each question points toward a different class of design intervention. In our case, recognizing that the primary barrier was invisible data, rather than invisible reasoning or inadequate visual encoding, redirected our design effort from representation to instrumentation, and the Milestone abstraction was the result.

8.2 From Invisible Evidence to Unfamiliar Terminology

A key outcome of our study is that the nature of the expertise barrier shifted. In the requirement study, the barrier was evidential. Recall that our participants had the knowledge to form hypotheses but lacked visual evidence to verify them. P1 arrived at conclusions *"based on close familiarity with Daisen's internals,"* P2 through *"research and professional experience."* The visualization could not help because the relevant information was absent. In the evaluation study, the evidential barrier was substantially lowered. Participants formed and verified performance judgments directly from the visualization. P2 identified bottlenecks from blocking distributions. P3 confirmed prior intuitions from the stacked bar chart. P1 traced Milestone markers to subtask transitions. The information was now present, and analysts used it.

The remaining barrier, however, was terminological. P1 noted that *"users without a solid background in GPU architecture may still struggle to interpret the terminology,"* and P4 observed that newcomers would find the system difficult without guidance. P2 made a particularly revealing observation: the blocking reason visualization was interpretable without understanding the Milestone concept itself. P2 could read proportions from the stacked bar chart and conclude which reason dominated, even while finding the Milestone abstraction *"not very intuitive"* as a concept.

As characterized in our discussion of invisible barriers (Section 8.1), the expertise gap is not monolithic: the evidential barrier that dominated in the first phase and the terminological barrier that emerged in the second require fundamentally different interventions. This decomposition carries a practical implication for visualization researchers working in expertise-intensive domains.

When user studies reveal that analysts "cannot validate their hypotheses," the natural response is to improve the visualization: adding views, refining encodings, or supporting new interactions. Our experience suggests that designers should first diagnose which type of barrier is operating. If experts can form hypotheses but cannot find confirming or disconfirming evidence, the problem is upstream of the visualization, in the data itself, and no visual redesign will resolve it. The appropriate intervention is to define new data primitives that capture the missing states. If, on the other hand, the evidence is present but users struggle to interpret domain-specific labels or concepts, the problem is downstream, and the appropriate intervention is pedagogical: glossaries, guided onboarding, contextual tooltips, or progressive disclosure that introduces terminology incrementally. We propose that future design studies in expertise-intensive domains explicitly diagnose which type of barrier accounts for observed analytical failures, as each demands a distinct design response.

8.3 Limitations

Four limitations bound our findings. (1) Our design and evaluation use MGPUSim/Daisen. Validation across other GPU simulators, CPU architectures, and distributed systems is needed. (2) Our six-participant study included three returning participants, limiting generalizability and potentially introducing priming. Returning participants enabled direct before-and-after comparison, while three new participants partially mitigated this risk. (3) The expert-derived Milestone taxonomy may omit architecture-specific blocking behaviors. Broader workloads are needed to test its coverage. (4) We evaluated traces from hundreds of MB to a few GB. On-demand expansion and per-component aggregation limit load, but substantially larger traces remain unevaluated.

9 Conclusion

We presented a visual analytics system that exposes the causal layers underlying GPU behavior. Central to the system is the Milestone data abstraction, a primitive that records the moment each blocking condition resolves, encoding both what became available and what had been blocking progress. Three visualizations build on the Milestone abstraction: (1) Blocking Summaries locate where and when blocking concentrates, (2) Blocking-Reason Analysis reveals dominant blocking patterns and their temporal evolution, and (3) the Dissect View enables hierarchical causal tracing without loss of context. Our two-phase study with GPU architecture researchers showed a qualitative shift in analytical capability. Participants who previously relied on prior expertise could now form and verify performance judgments directly from visual evidence, leaving a barrier that is terminological rather than evidential. Our central methodological claim is that in expertise-intensive domains, the visualization design problem can begin upstream of visual encoding, at the level of what data to capture. When the causal states analysts reason about are transient and absent from standard traces, defining a data primitive to record them is a core part of visualization design, and the Milestone abstraction generalizes to other domains where analysts reason backward along causal chains that current instrumentation leaves invisible.

Supplemental Materials

We provide the following supplemental documents to support the reproducibility of our two studies. We also archive all materials on OSF at https://doi.org/10.17605/OSF.IO/U85H3 under a CC BY 4.0 license.

Supplement S1 (Study Guide). S1 contains the full interview protocols for both the requirement study (Section 4) and the evaluation study (Section 7), including the consent scripts, the question lists, the four scenarios with their Daisen views, and the four low-fidelity mockups we showed as a fallback elicitation aid during the requirement study, placed alongside the scenario to which each applies.

Supplement S2 (Demonstration Video). S2 is a screen-recorded walkthrough of our system, demonstrating the redesigned Daisen dashboard, the Blocking-Reason Analysis, the Milestone markers on task bars, and the Dissect View on an example simulation trace.

Supplement S3 (Coding Results and Participant Backgrounds). S3 contains the open-coding results for both studies, organized as themes, findings, participants, and supporting quotes, together with per-participant background for both studies, covering prior Daisen experience, years of GPU research experience, and degree.

Acknowledgments

We thank the anonymous reviewers for their constructive feedback. We also thank all study participants for their time and insights. This project was supported by the United States National Science Foundation (NSF) under award OAC-2441804 and the United States Department of Energy (DOE) under award DE-SC0024635.

References

[1] I. Aedo, T. Onorati, C. Tucci, P. Díaz, A. Montero, and J. Castro. Bridging the gap between knowledge and human expertise: Integrating explicit and tacit knowledge in maintenance operations. In *Proc. 1st Int. Workshop on Augmented Artificial Intelligence (A^2ID 2025), co-located with IS-EUD 2025*, vol. 3978 of *CEUR Workshop Proc.* CEUR-WS.org, Aachen, 2025. 3

[2] G. Alicioglu and B. Sun. A survey of visual analytics for explainable artificial intelligence methods. *Comput. Graph.*, 102(C):502–520, 2022. doi: 10.1016/j.cag.2021.09.002 3

[3] A. Ariel, W. W. L. Fung, A. E. Turner, and T. M. Aamodt. Visualizing complex dynamics in many-core accelerator architectures. In *Proc. ISPASS*, pp. 164–174. IEEE, White Plains, 2010. doi: 10.1109/ISPASS.2010.5452029 1, 3

[4] I. Beschastnikh, P. Liu, A. Xing, P. Wang, Y. Brun, and M. D. Ernst. Visualizing distributed system executions. *ACM Trans. Softw. Eng. Methodol.*, 29(2), art. no. 9, 38 pp., 2020. doi: 10.1145/3375633 3

[5] A. J. Böck, S. Größbacher, J. Vrablicz, C. Stoiber, A. Rind, J. Suschnigg et al. KAVAI: A systematic review of the building blocks for knowledge-assisted visual analytics in industrial manufacturing. *Applied Sciences*, 15(18), art. no. 10172, 2025. doi: 10.3390/app151810172 3, 8

[6] R. Bosch, C. Stolte, G. Stoll, M. Rosenblum, and P. Hanrahan. Performance analysis and visualization of parallel systems using SimOS and Rivet: a case study. In *Proc. HPCA*, pp. 360–371. IEEE, Toulouse, 2000. doi: 10.1109/HPCA.2000.824365 3

[7] D. Ceneda, T. Gschwandtner, T. May, S. Miksch, H.-J. Schulz, M. Streit et al. Characterizing guidance in visual analytics. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):111–120, 2017. doi: 10.1109/TVCG.2016.2598468 3

[8] S. Cheng, P. De, S. H.-C. Jiang, and K. Mueller. TorusVisND: Unraveling high-dimensional torus networks for network traffic visualizations. In *Proc. VPA*, pp. 9–16. IEEE, New Orleans, 2014. doi: 10.1109/VPA.2014.7 3

[9] S. I. Chuprina, K. V. Ryabinin, D. V. Koznov, and K. A. Matkin. Ontology-driven visual analytics software development. *Program. Comput. Softw.*, 48(3):208–214, 2022. doi: 10.1134/S0361768822030033 3

[10] J. Cottam, B. Martin, L. Dalessandro, and A. Lumsdaine. Pixel-oriented techniques for visualizing next-generation HPC systems. In *Proc. VISSOFT*, pp. 160–164. IEEE, Bremen, 2015. doi: 10.1109/VISSOFT.2015.7332429 3

[11] P. Federico, M. Wagner, A. Rind, A. Amor-Amorós, S. Miksch, and W. Aigner. The role of explicit knowledge: A conceptual model of knowledge-assisted visual analytics. In *Proc. VAST*, pp. 92–103. IEEE, Phoenix, 2017. doi: 10.1109/VAST.2017.8585498 3, 8

[12] T. Fujiwara, P. Malakar, K. Reda, V. Vishwanath, M. E. Papka, and K.-L. Ma. A visual analytics system for optimizing communications in massively parallel applications. In *Proc. VAST*, pp. 59–70. IEEE, Phoenix, 2017. doi: 10.1109/VAST.2017.8585646 1, 3

[13] V. Garcia Pinto, L. Schnorr, L. Stanisic, A. Legrand, S. Thibault, and V. Danjean. A visual performance analysis framework for task-based parallel applications running on hybrid clusters. *Concurrency and Computation: Practice and Experience*, 30, 2018. doi: 10.1002/cpe.4472 1, 3

[14] P. Gyarmati, D. Moritz, T. Möller, and L. Koesten. Structured visualization design knowledge for grounding generative reasoning and situated feedback, 2025. Preprint. doi: 10.48550/arXiv.2512.20306 3

[15] M. T. Heath and J. A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8(5):29–39, 1991. doi: 10.1109/52.84214 3

[16] K. E. Isaacs, P.-T. Bremer, I. Jusufi, T. Gamblin, A. Bhatele, M. Schulz et al. Combing the communication hairball: Visualizing parallel execution traces using logical time. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2349–2358, 2014. doi: 10.1109/TVCG.2014.2346456 3

[17] Z. Jin, S. Guo, N. Chen, D. Weiskopf, D. Gotz, and N. Cao. Visual causality analysis of event sequence data. *IEEE Transactions on Visualization and Computer Graphics*, 27(2):1343–1352, 2021. doi: 10.1109/TVCG.2020.3030465 3

[18] B. Karer, H. Hagen, and D. J. Lehmann. Insight beyond numbers: The impact of qualitative factors on visual data analysis. *IEEE Transactions on Visualization and Computer Graphics*, 27(2):1011–1021, 2021. doi: 10.1109/TVCG.2020.3030376 3

[19] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler et al. The Vampir performance analysis tool-set. In *Tools for High Performance Computing*, pp. 139–155. Springer, Berlin, 2008. doi: 10.1007/978-3-540-68564-7_9 1

[20] H. Lin, D. Akbaba, M. Meyer, and A. Lex. Data hunches: Incorporating personal knowledge into visualizations. *IEEE Transactions on Visualization and Computer Graphics*, 29(1):504–514, 2023. doi: 10.1109/TVCG.2022.3209451 2, 3, 8

[21] M. Meyer, F. Beck, and S. Lohmann. Visual monitoring of process runs: An application study for stored procedures. In *Proc. PacificVis*, pp. 160–167. IEEE, Taipei, 2016. doi: 10.1109/PACIFICVIS.2016.7465264 3

[22] C. Muelder, C. Sigovan, K.-L. Ma, J. Cope, S. Lang, K. Iskra et al. Visual analysis of I/O system behavior for high-end computing. In *Proc. LSAP*, LSAP '11, pp. 19–26. ACM, New York, 2011. doi: 10.1145/1996029.1996036 3

[23] D. K. Osmari, H. T. Vo, C. T. Silva, J. L. D. Comba, and L. Lins. Visualization and analysis of parallel dataflow execution with smart traces. In *Proc. SIBGRAPI*, pp. 165–172. IEEE, Rio de Janeiro, 2014. doi: 10.1109/SIBGRAPI.2014.2 3

[24] S. P. Reiss and S. Karumuri. Visualizing threads, transactions and tasks. In *Proc. PASTE*, PASTE '10, pp. 9–16. ACM, New York, 2010. doi: 10.1145/1806672.1806675 3

[25] J. Roberts and C. Zilles. TraceVis: An execution trace visualization tool. In *Proc. ISPASS*, pp. 232–241. IEEE, Austin, 2006. doi: 10.1109/ISPASS.2006.1620804 3

[26] P. Rosen. A visual approach to investigating shared and global memory behavior of CUDA kernels. In *Proc. EuroVis*, EuroVis '13, pp. 161–170. Eurographics Assoc., Leipzig, Germany, 2013. doi: 10.1111/cgf.12103 3

[27] D. Sacha, A. Stoffel, F. Stoffel, B. C. Kwon, G. Ellis, and D. A. Keim. Knowledge generation model for visual analytics. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):1604–1613, 2014. doi: 10.1109/TVCG.2014.2346481 3

[28] S. A. Sakin, A. Bigelow, R. Tohid, C. Scully-Allison, C. Scheidegger, S. R. Brandt et al. Traveler: Navigating task parallel traces for performance analysis. *IEEE Transactions on Visualization and Computer Graphics*, 29(1):788–797, 2023. doi: 10.1109/TVCG.2022.3209375 3

[29] Y. B. Shrinivasan and J. J. van Wijk. Supporting the analytical reasoning process in information visualization. In *Proc. CHI*, CHI '08, pp. 1237–1246. ACM, Florence, Italy, 2008. doi: 10.1145/1357054.1357247 3

[30] Y. Sun, T. Baruah, S. A. Mojumder, S. Dong, X. Gong, S. Treadway et al. MGPUSim: enabling multi-GPU performance modeling and optimization. In *Proc. ISCA*, pp. 197–209. ACM, Phoenix, 2019. doi: 10.1145/3307650.3322230 2

[31] Y. Sun, Y. Zhang, A. Mosallaei, M. D. Shah, C. Dunne, and D. Kaeli. Daisen: A framework for visualizing detailed GPU execution. *Computer*

Graphics Forum, 40(3):239–250, 2021. doi: 10.1111/cgf.14303 1, 2, 3

[32] J. Wang and K. Mueller. The visual causality analyst: An interactive interface for causal reasoning. *IEEE Transactions on Visualization and Computer Graphics*, 22(1):230–239, 2016. doi: 10.1109/TVCG.2015.2467931 3

[33] J. Wang and K. Mueller. Visual causality analysis made practical. In *Proc. VAST*, pp. 151–161. IEEE, Phoenix, 2017. doi: 10.1109/VAST.2017.8585647 3

[34] S. Wang, H. Yan, K. E. Isaacs, and Y. Sun. Visual exploratory analysis for designing large-scale Network-on-Chip architectures: A domain expert-led design study. *IEEE Transactions on Visualization and Computer Graphics*, 30(4):1970–1983, 2024. doi: 10.1109/TVCG.2023.3337173 3

[35] C. T. Weaver, K. C. Barr, E. D. Marsman, D. Ernst, and T. M. Austin. Performance analysis using pipeline visualization. In *Proc. ISPASS*, pp. 18–21. IEEE, Tucson, 2001. doi: 10.1109/ISPASS.2001.990670 1, 3

[36] K. Williams, A. Bigelow, and K. E. Isaacs. Visualizing a moving target: A design study on task parallel programs in the presence of evolving data and concerns. *IEEE Transactions on Visualization and Computer Graphics*, 26(1):1118–1128, 2020. doi: 10.1109/TVCG.2019.2934285 3

[37] Y. L. Wong, K. Madhavan, and N. Elmqvist. Towards characterizing domain experts as a user group. In *Proc. BELIV*, pp. 1–10. IEEE, Berlin, 2018. doi: 10.1109/BELIV.2018.8634026 2, 8

[38] K. Wongsuphasawat and D. Gotz. Exploring flow, factors, and outcomes of temporal event sequences with the Outflow visualization. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2659–2668, 2012. doi: 10.1109/TVCG.2012.225 3

[39] K. Wongsuphasawat, D. Moritz, A. Anand, J. Mackinlay, B. Howe, and J. Heer. Voyager: Exploratory analysis via faceted browsing of visualization recommendations. *IEEE Transactions on Visualization and Computer Graphics*, 22(1):649–658, 2016. doi: 10.1109/TVCG.2015.2467191 3

[40] C. Xie, W. Xu, and K. Mueller. A visual analytics framework for the detection of anomalous call stack trees in high performance computing applications. *IEEE Transactions on Visualization and Computer Graphics*, 25(1):215–224, 2019. doi: 10.1109/TVCG.2018.2865026 3

[41] C. Xiong, L. Van Weelden, and S. Franconeri. The curse of knowledge in visual data communication. *IEEE Transactions on Visualization and Computer Graphics*, 26(10):3051–3062, 2020. doi: 10.1109/TVCG.2019.2917689 3, 8

[42] S. Zhu, G. Sun, Y. Shen, Z. Zhu, W. Xia, B. Chang et al. VAC^2: Visual analysis of combined causality in event sequences. *IEEE Transactions on Visualization and Computer Graphics*, 31(9):6223–6240, 2025. doi: 10.1109/TVCG.2024.3496789 3

[43] A. K. Ziabari, R. Ubal, D. Schaa, and D. Kaeli. Visualization of OpenCL application execution on CPU-GPU systems. In *Proc. WCAE*, WCAE '15, art. no. 3, 8 pp. ACM, Portland, 2015. doi: 10.1145/2795122.2795125 1, 3