🧠 What was your role on the Radeon Display Insights Portal project at AMD?

I joined AMD in mid-2023 as a front-end developer on a small team building an internal dashboard for the Display QA group. At the time, the tools they were using were pretty outdated — lots of manual log reading and inconsistent ways of validating new display features. So we set out to build a centralized web portal that could show real-time GPU data in a way that was easier to understand and act on.

I focused mostly on the front-end, building out the React and TypeScript interface. We had a lot of custom visualizations — things like showing when HDR kicked in during a test run or tracking display faults over time on a heatmap. I also handled live telemetry updates using WebSockets and structured caching with React Query, which helped the app stay fast even with large streams of GPU data.

On top of the code, I worked closely with QA and firmware engineers — syncing often during sprints, testing new features, and making sure what we built actually helped their day-to-day work.

🧭 What was the main goal of the Display Portal in AMD?

So before we built the new portal, AMD's display QA team was using a really outdated tool. It had this super basic, pixelated UI — barely showed anything beyond raw numbers. If something went wrong during a test, they had to scroll through huge log files just to figure out what happened.

That worked okay years ago, but with newer GPUs, there's just way more going on. You've got things like dynamic refresh rates, screen flickering, color shifting... and they needed a better way to see all of that in real time.

That's where the Display Insights Portal came in. We built a modern web dashboard that let engineers actually watch test data as it streamed in. So if something flickered or the screen colors didn't switch correctly, they could catch it right away — without having to dig through logs.

It also let them look at past runs, compare issues across different devices, and export test results if they needed to dig deeper. Basically, it gave them way more visibility and saved a lot of time during debugging.

🧑‍🤝‍🧑 What was the team structure like in AMD?

We had a tight six-person team: two frontend devs (including me), two backend, one QA engineer, and a product manager. Because we were small, we all worked pretty closely — frontend and backend paired up on API design, QA helped us understand real test scenarios, and the PM kept us aligned with what the engineers actually needed. We worked in sprints, had regular standups, and iterated quickly based on feedback from the QA lab.

What component UI libraries did you use in AMD?

Since we were building the new portal to replace an older internal tool, we did inherit a bit of legacy code — mostly on the backend. But on the frontend, we had to build a lot from scratch because the old UI just didn't support the kind of visualizations we needed.

We used Tailwind for most of the layout and styling — it was great for moving quickly and keeping things consistent. For components, we pulled in Radix primitives through ShadCN to handle accessibility stuff like dropdowns and tabs, and we reused a few Chakra UI elements where it made sense. There were also some old MUI components from the legacy app that we kept around, but only in a few low-impact areas.

For charts and visualizations, we mostly built custom components using Chart.js. The built-in chart libraries didn't give us enough control — especially when we needed to highlight things like flickers, color transitions, or drift events frame-by-frame.

So yeah, it ended up being a layered setup. We reused what we could, but most of the frontend was tailored for how QA actually debugged these new GPU features.

🧱 How did you structure the React app in AMD?

We used a feature-based folder structure to keep things organized as the app grew. For the core visualizations — like the HDR timeline and the fault heatmap — we built custom components from scratch since those were really specific to AMD's hardware testing.

That said, for things like forms, tables, and general layout, we reused components from AMD's internal design system to keep everything consistent. It was a hybrid setup that let us go custom where it mattered and reuse where it didn't.

🎨 What styling libraries did you use in AMD?

We used a mix. Tailwind CSS was our go-to for quick layout and spacing. For components, we pulled from Radix (ShadCN) and Chakra UI where it made sense — mostly for accessibility and keyboard support. There were also some legacy parts that still used MUI.

For charts, we mostly used Chart.js, but heavily customized it. The interactivity and design needed to match how QA folks worked — not just pretty graphs but tools that actually helped them see problems fast.

🧭 How did you handle routing and filters in the portal in AMD?

We used React Router v6 to handle navigation. The dashboard had a lot of deep views — think filtering by GPU model, firmware version, test type — so we stored all that in the URL. That way, someone could share a link or reload the page without losing context.

To keep things fast, we debounced filter inputs and synced them with the URL, so the state didn't get out of sync. We also used parent-child routes to pass context down cleanly, which helped make the code easier to reason about and test.

📡 What kind of telemetry data were you working with in AMD?

We were pulling in real-time GPU data from AMD's internal test framework — mostly for the display team. So things like refresh rates, frame timing, screen flickers, and signal dropouts. If something went wrong visually during a test, this data would help explain why.

The data came in as JSON — each event tied to a specific device and timestamp. Some were live streaming metrics, others were like snapshots — for example, a firmware version or a summary of recent faults. So we had to structure all of it in the frontend in a way that made time-based filtering and charting feel smooth.

🌐 Why did you use WebSockets instead of polling in AMD?

Since we were dealing with live test runs, polling wasn't really fast enough. The QA team wanted to see things happen second by second — like when the screen blinked or when frame rates dropped suddenly.

So we went with WebSockets. It gave us a constant connection to the backend, and anytime new data came in, the server could just push it straight to the frontend. No delays, no need to ask for updates — the stream just kept flowing.

We wired that stream into our state layer — React Query — so the UI updated instantly without re-rendering everything. It was just a smoother and more efficient way to keep the data flowing without bogging down the network or the browser.

🚧 Did you run into issues with WebSocket backpressure in AMD?

Yeah, especially when multiple users were testing at once or running high-frequency telemetry. Sometimes we'd get flooded with updates — way more than the browser could handle in real time. That caused some noticeable lag and even occasional dropped frames in the UI.

To fix it, we built a small smoothing layer on the frontend. It filtered out tiny fluctuations that didn’t matter and prioritized stuff like error flags or threshold breaches. So even if 500 updates came in, we’d only show the important ones that second.

That kept the UI snappy and helped prevent the WebSocket connection from choking during heavy runs.

🚀 How did you keep the dashboard fast with so much telemetry in AMD?

Yeah, early on we were seeing test runs dump over 100,000 data points — and the page just froze.

So we broke the problem down. First, we trimmed the data itself. The backend used to send huge logs with everything; we got it down to just the fields the UI actually used — timestamp, value, maybe a label.

Second, we added list virtualization using `react-window`. That way, even if the dataset was huge, we only rendered what was visible on screen — like 30 or 40 items max — instead of trying to draw thousands of DOM nodes.

And third, we leaned on caching and data filtering. We stored everything in React Query’s cache, but only pulled out what was needed for the active view. So if someone filtered by a device or test, the UI wouldn’t bother looking at unrelated data.

Once we had those pieces in place, the dashboard went from freezing for 10–15 seconds to loading almost instantly, even with giant payloads.

🛠️ [AMD - Error - Resilience] How did you handle front-end errors and keep the app from crashing?

Yeah, we ran into two big types of front-end issues. First were data mismatches — the telemetry format sometimes changed when the GPU team added new features. So we'd get unexpected fields or missing ones, and that could break some of our components. Second, because the data was coming in live, we had timing issues — sometimes a component would try to render before the data it needed had even arrived, which led to undefined errors or layout glitches.

To keep the app stable, we wrapped our most fragile UI pieces — like the graphs and charts — in React error boundaries. So if something broke, it wouldn't crash the whole dashboard. It would just show a small fallback message for that panel, and users could reload it without refreshing the whole page.

We also logged every error: the message, the stack trace, and even a sample of the data that caused it. We didn't use a tool like Sentry since it was all internal, but we had a lightweight custom logger that gave us everything we needed to debug. That combo — error boundaries and logging — made sure we didn't leave QA stuck with a blank screen, and we could track down weird bugs fast.

[AMD - CrossTeam] How did you collaborate with non-frontend teams on this project?

When I joined, I honestly didn't know much about display firmware. So I made it a point to sit in on the firmware and QA team's triage calls. They'd talk through things like flicker bugs or refresh rate drift, and I'd just listen — and ask questions whenever something didn't make sense.

For example, I remember being confused about how they labeled firmware versions. It didn't match the filters I had built in the dashboard. So I asked them to walk me through how they track builds in the lab, and based on that, I changed the filter input to accept their exact version format.

That way, if a firmware engineer saw an issue in the lab, they could paste that version right into our dashboard and jump to the test — no translation needed. That kind of back-and-forth really shaped the tool. By the time we were doing demos, the firmware and QA folks were already using it as part of their daily workflow.

🧪 [AMD - Collaboration - Feedback] How did you get feedback from QA and incorporate it?

We baked feedback into the workflow from the start. Every two weeks, we had a sprint review where the QA team would try out the latest version of the dashboard and walk us through what made sense, what didn't, and what they wished they could do better.

But honestly, the most useful feedback came informally. I sat near a couple of the QA leads, and they'd just swing by or ping me whenever something felt off or if they had an idea. Like one time, someone said, "Hey, I'm trying to compare two test runs, but it's really clunky." So I added a toggle that made side-by-side comparison easier.

It wasn't just bug reports — it was small quality-of-life stuff they noticed while doing real work. That loop of hearing them out and making quick changes helped us build trust and catch a lot of edge cases early.

🔌 [AMD - KeyTech - LiveData] How did you manage live data subscriptions safely on the frontend?

We used WebSockets for streaming, so managing connections was a big part of keeping things smooth. I set up a hook that would open the WebSocket connection when the component mounted, and clean it up when it unmounted. That helped avoid zombie connections sitting around in the background.

But the bigger issue was how fast the data came in. If we updated React state on every single message, the app would've choked. So I used a little buffer with `useRef`, and instead of setting state every time, I'd batch the updates and push them in maybe 2–3 times per second. It still felt real-time to the user, but it was way more efficient for rendering.

That combo kept things clean — no memory leaks, and no overloaded React updates.

🧠 [AMD - KeyTech - StateMgmt] Why React Query instead of Redux for telemetry state?

Redux is great when you need to manage app-wide state manually, but in our case, most of the data was remote and constantly changing. That's exactly what React Query is built for.

With React Query, we didn't need to write our own fetching logic or reducers. It handled caching, background updates, and stale data out of the box. We gave each feed its own cache key — tied to things like test ID or device ID — so we didn't mix up streams.

We also tuned how long different types of data stayed fresh. For fast-changing stuff, we kept the stale time short so it always refreshed. For historical runs, we could cache longer.

It kept the data fresh, reduced the amount of boilerplate code, and made the dashboard feel a lot more responsive without adding complexity.

📊 [AMD - KeyTech - FreeSyncHDR] How did understanding FreeSync and HDR impact your UI decisions?

I spent time digging into how FreeSync and HDR actually worked, since a lot of the issues QA was catching were tied to those.

With FreeSync, I learned that the problem wasn't just the refresh rate — it was how much it drifted over time. If it drifted too far and failed to correct, that meant the GPU wasn't syncing properly. So in the dashboard, I didn't just show the raw values — I built a visual that highlighted those drift patterns and flagged when a re-sync happened. That way, engineers could spot issues without needing to interpret the numbers manually.

HDR was similar. When HDR handshakes failed, the screen might flicker or flash for a split second. I built a timeline view that showed HDR metadata frame by frame, grouped in little buckets, so QA could pinpoint exactly when that switch happened — and see if there was a gap or spike.

So instead of dumping logs, the UI helped tell the story. It made it way easier for the team to figure out when and where something went wrong.