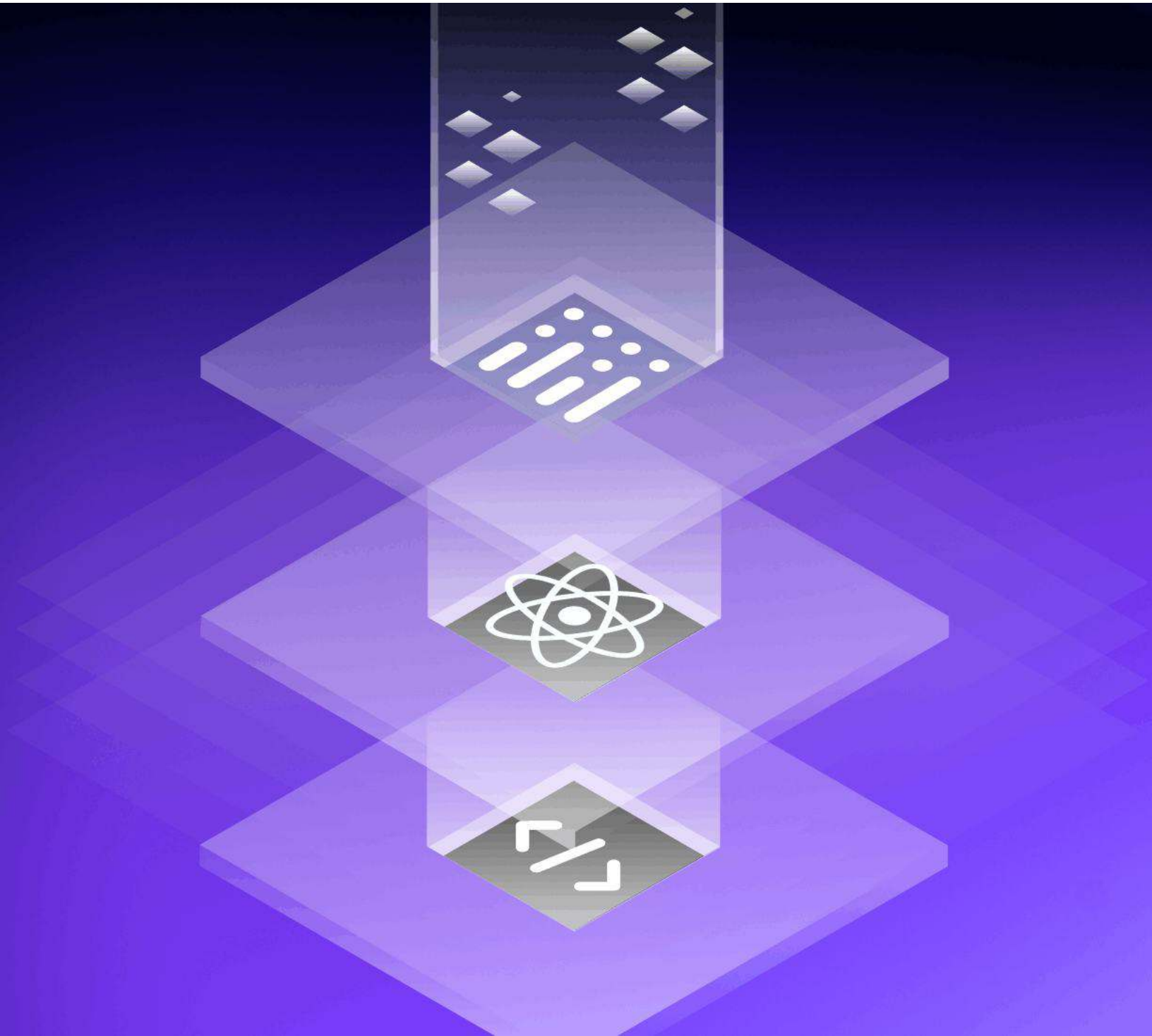
plotly

Dash vs. Full Stack in the AI Era

Matt Brown · Robert Claus

Combined from the three-part Plotly series: The Architecture Case, Hidden Risks, and Token Costs



TL;DR | Dash vs. Full Stack in the AI Era

When you ask a coding agent such as Claude or Codex to build a data app, naming the framework in the prompt matters.

Here is the short version of why that framework should be Dash across architecture, risk, and cost.

Architecture

- **One app, not two.** You asked for an app, but full stack gave you two: a front end and a back end with an API layer gluing them together. That's two things that have to be built, hosted, and kept in sync, or worse, a front end that never got a back end at all. Dash is one application, one language, one process.
- **No translation tax.** Your data already speaks Python. Dash callbacks call pandas, NumPy, and scikit-learn directly. Full stack has to flatten that same data into JSON first, a boundary that breaks on ordinary things like datetimes and NaN values if not handled appropriately.
- **Fewer decisions, fewer forks in the road.** Full stack doesn't have a default state-management pattern, so Claude has to pick one for every project introducing possible inconsistencies and complexity. Dash's callback model is one tried-and-true default that works every time.
- **The wiring is just as important as the charts.** Cross-filtering and linked components in Dash are just another callback. In full stack, Claude has to invent the state architecture connecting components from scratch, every time.
- **A smaller blast radius when something breaks.** One language, one runtime, one place for a bug to live, and a much shorter path to understanding why.

Risk

- **A smaller, checkable dependency tree.** A fresh Dash install resolves 27 packages vs. a full-stack app which resolves to hundreds of dependencies and sub-dependencies, in an ecosystem seeing growth in malicious and AI-hallucinated packages.
- **Easier hand off.** Someone else has to read the code eventually. A Dash app can be reviewed by anyone on a Python-fluent data team. A full-stack app asks whoever inherits it to either trust Claude's account of unfamiliar code or learn a new language.
- **Dash has a steward.** Plotly created the Python visualization stack and deployment platform: Dash (10M downloads/month), Plotly graphing libraries (68M/month), and on-prem/cloud hosting. No competing visualization stack is as vertically integrated or widely deployed. Full stack has no single company to call when something breaks.

Cost

- **It shows up in the token bill.** We ran the numbers and Dash apps cost roughly 33% less than the equivalent full-stack app costs.
- **The exception:** For a highly custom, consumer-facing product with a designer in the loop, full stack's flexibility is still the right call. This piece is about the data apps, internal tools, and reports that make up most of what people are actually vibe-coding.

For a walk through of the full argument, watch the on-demand webinar.

Table of Contents

Part 1: The Architecture Case
Why it still matters which framework you pick when AI writes all the code

Part 2: Hidden Risks
The risks you don't see on day one

Part 3: Token Cost
Doing the math on vibe-coded apps

Part 1: The Architecture Case

Why it still matters which framework you pick when AI writes all the code

YOLO-ing data apps

Type "build a dashboard that tracks churn by cohort" into Claude and you might get a Dash or a full-stack app back. That's the whole pitch of vibe-coding: describe the outcome, skip the syntax, let the model handle the rest. So why does it matter if you don't pick a framework in your prompt?

Because Claude isn't the one who has to live with what it builds. You are. And the choice between full stack and Dash isn't really a choice about which framework is more powerful. It's a choice about how much surface area you're handing an AI to get right, and how much of that surface area you can personally inspect when it doesn't.

You're actually asking for two apps

Ask Claude for a data app in full stack and you're asking for two applications: a front end that renders the UI, and a back end that talks to your data, with an API contract stitched between them. That's two languages, two runtimes, and a handshake between them that has to stay in sync every time either side changes.

Or, just as often, you get one application instead of two, and the wrong one. Claude builds the front end, wires it up to sample data or a hardcoded JSON file to get something on screen fast, and never gets around to the back end at all. It looks done. It renders, it's interactive, the charts show up. Then you go to point it at your actual data source and discover there's no API to call, no data layer underneath it, nothing to connect to, just a UI built around the shape of whatever placeholder data it started with. Now you're not adding a feature, you're building the whole back end Claude skipped, from scratch.

Ask for the same thing in Dash, and you get one application. Layout and logic live in the same codebase, deploy as the same process, and talk to your data directly instead of across a wire. There's no API contract to keep in sync because there's no second service on the other end of it, and no separate build step, host, or network boundary to set up and maintain.

This isn't purely about file count, but also how many moving parts Claude has to get right at once, and how many of those parts can quietly drift out of sync with each other after the fact. A front end and a back end built in the same session can still disagree about a field name or a response shape six prompts later, but a single application doesn't have a seam to drift across.

It also changes what Claude has to hold in its head on every turn. Less to generate and re-read on every follow-up prompt means fewer tokens spent over the life of a project, and less room for the model to lose track of an architecture that's split across two code bases instead of one.

When you want to open the hood yourself, there's only one application to understand – a much smaller job for anyone reviewing the work – especially when the person doing the reviewing is a data analyst or scientist.

Your data already speaks Python, don’t make it learn a second language

Data teams already live in Python. Pandas, NumPy, Polars, SciPy, scikit-learn, and whatever proprietary ETL pipeline your company has built up over the years are all native to it. It goes to the heart of where the actual analysis work already happens, in a notebook, in a script, in a scheduled job, long before anyone thinks about putting it in front of a wider audience. A Dash callback runs in that same process. It can call `pd.read_sql`, run a `groupby` and an aggregation, call `.predict()` on a scikit-learn model, and hand the result straight to a Plotly figure, in the same file, the same language, the same objects your notebook already produces. It’s your raw analysis – not an adaptation of your work for the web – wired up directly to a callback.

Full stack can't do any of that directly, because the browser doesn't run Python natively, and WebAssembly, although promising, hasn’t yet eliminated the need for a true server-side application in most cases. Whatever pandas or scikit-learn work happens has to happen somewhere else, get flattened into JSON, and cross a wire before a React component ever sees it, and this boundary can be brittle.

Consider this simple mistake: a plain `pandas` DataFrame with a datetime column fails to serialize. Did you spot the mistake?

Python's own `json.dumps` raises a `TypeError: Object of type Timestamp is not JSON serializable`. Of course, there's a workaround for this, but it's dependent on your chosen API (e.g. instead use Flask's `jsonify()` function).

How about this one?

```python
# api/endpoints.py

import json
from flask import Flask, Response
import pandas as pd

app = Flask(__name__)

def load_inventory_counts():
    df = pd.read_csv("inventory.csv")
    counts = (
        df.groupby("warehouse")["sku"]
        .count()
        .reset_index(name="item_count")
    )
    return counts

@app.route("/api/inventory-counts")
def inventory_counts():
    df = load_inventory_counts()
    payload = df.to_dict(orient="records")
    return Response(json.dumps(payload), mimetype="application/json")

if __name__ == "__main__":
    app.run(debug=True)
```

The default `numpy.int64` integer type that pandas hands you fails the same way: `TypeError: Object of type int64 is not JSON serializable`.

And what about this last one?

```python
# api/endpoints.py

import json
from flask import Flask, Response
import pandas as pd

app = Flask(__name__)

def load_inventory_counts():
    df = pd.read_csv("inventory.csv")
    summary = (
        df.groupby("warehouse")["sku"]
        .agg(avg_discount=("discount", "mean"))
        .reset_index()
    )
    return summary

@app.route("/api/inventory-counts")
def discount_summary():
    df = load_discount_ summary()
    payload = df.to_dict(orient="records")
    return Response(json.dumps(payload), mimetype="application/json")

if __name__ == "__main__":
    app.run(debug=True)
```

This one doesn't throw a Python error at all, nor is there any issue with the code itself, but the data set includes missing values that are printed as `NaN` values in the payload, which is not valid JSON. Try to parse that same payload in a browser and `JSON.parse` throws immediately: `Unexpected token 'N', "...NaN}" is not valid JSON`.

Again, can these issues be worked around? Absolutely, but it's something the AI and you will need to stay on top of as you develop. Claude will have to solve this on every project, because there's no default answer for how it should be done. The fixes an AI agent reaches for under time pressure tend to be the ones that make the error disappear rather than the ones that preserve the data's meaning: `fillna(0)` to stop the crash, `str()` on a timestamp to get past the TypeError. A data scientist glancing at that fix would catch it instantly. Zero is not the same thing as missing, and a stringified date in one format upstream can silently stop matching a differently formatted date downstream. Whether anyone catches it depends entirely on how easily the person reviewing it can spot the problem, which is much harder when you're working across the full stack.

Plotly's own libraries, `plotly.py`, `dash_table`, `dcc.Graph`, already know how to carry a DataFrame's dtypes, its `NaN`s, its timestamps, and its categorical columns into a rendered chart without losing or misrepresenting any of it. That translation still happens somewhere under the hood. The difference is that it happens inside code Plotly maintains and tests against exactly these edge cases, not inside code Claude is inventing for your project for the first time.

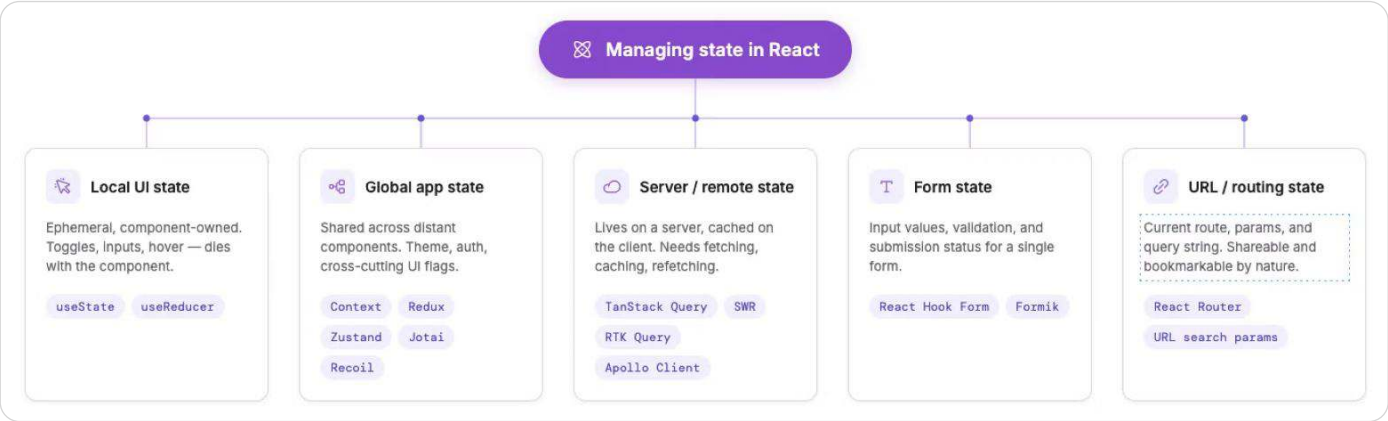
Here's the code that *just* works in Dash.

```python
import dash
from dash import html, dash_table
import pandas as pd

df = pd.read_csv("orders.csv", parse_dates=["order_date"])

summary = (
    df.groupby("region")
    .agg(
        order_count=("order_id", "count"),      # numpy.int64
        last_order=("order_date", "max"),       # pandas.Timestamp
        avg_discount=("discount", "mean"),      # NaN for regions with none
    )
    .reset_index()
)

app = dash.Dash(__name__)
app.layout = html.Div([
    dash_table.DataTable(data=summary.to_dict("records")),
])

if __name__ == "__main__":
    app.run(debug=True)
```

It also changes the scope of small edits. Decide you want a different aggregation, a new model feature, a rolling average instead of a daily sum, and in Dash that's a change to one Python function, the same kind of edit you'd make in a notebook. In a split React + back end app, the same idea touches two files in two languages: the back end logic, and then whatever the front end was expecting the response to look like, whether or not the underlying change had anything to do with the front end at all. That's slower for you to review and slower for Claude to get right, because part of every iteration goes to keeping a contract in sync instead of improving the analysis.

Lastly, it determines who can actually check the work afterward. A data scientist can read a Dash callback built around a `groupby` and know immediately whether the logic is right, wrong, or subtly off, because it's written in the language they think in. That same person usually has no way to tell whether a `to_dict(orient="records")` on a Flask endpoint actually matches what a `useEffect` hook expects on the other end, or whether a timestamp quietly picks up a timezone shift somewhere in the handoff. It's obvious but worth stating plainly: It's much easier to review and debug code you already understand.

Fewer forks in the road



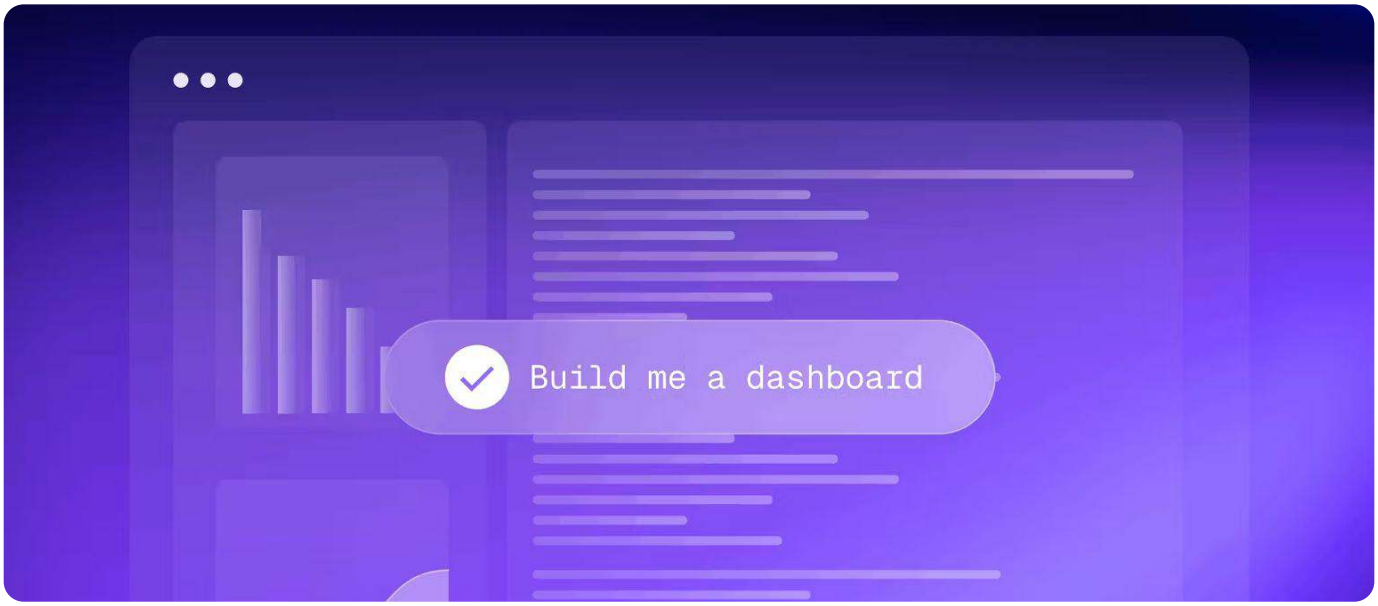
On the front end, React doesn't ship an opinion on how to manage state, route between views, or fetch data. That flexibility is the point, but it means every new project starts with a round of decisions: Redux, Zustand, Jotai, or Context for state; TanStack Query or something homegrown for server data; React Hook Form here, plain state there. Developer surveys going into 2026 describe this landscape in terms of "decision fatigue," and the industry's own response has been to split state into five separate categories, each with its own recommended library.

That's a lot of forks in the road for a human team to navigate deliberately. It's worse for an AI agent improvising a project structure from scratch, because there's no single "correct" pattern for it to reach for.

Dash doesn't force that decision on you as often. Though the choice technically still exists (Dash components are React components under the hood, and you can write your own if you need to) but Plotly and the community have already built and battle-tested the ones most apps need, so callbacks are the default way state moves without anyone having to reach for a custom component or a separate state library to get there. There's one tried-and-true pattern, so there's one thing for Claude to get right, and one thing for you to learn if you need to change it later. Fewer decisions during development keeps both the AI and the human on a maintainable path instead of a bespoke one.

It’s not just state, the wiring matters too

A "build me a dashboard" prompt rarely wants just one chart sitting on a page by itself. It wants three or four charts that talk to each other: click a bar in one, the table below filters to match; hover a point on the scatter plot, a detail panel updates; drag a selection across a time series, everything else on the page recomputes against just that window. It’s called cross-filtering, and it has less to do with any individual chart library and everything to do with where state lives and how it gets from one component to another.



In Dash, cross-filtering is the same pattern already used for everything else in the app. One chart's `clickData` or `selectedData` prop is an Input, another chart's `figure` prop is an Output, and the callback in between recomputes whatever the second chart needs to show. It's one more Input-Output pair, wired the same way as the dropdown that populates a table. Nothing new to learn, and nothing new for Claude to invent.

In React, that same interaction has no single answer, because React doesn't have an opinion on where shared state should live, the same fork-in-the-road problem from the last section, applied to interactivity instead of app-wide state. Claude has to decide whether to lift the selection into a parent component, hoist it into context, or push it into whatever state library the project already happened to pick. Then it has to write an event handler on the first chart that updates that shared value, a subscription on the second chart that reads it, and, if the update needs new data rather than just a re-render, a fresh call back to whatever back end is supplying it, the same fragile seam described earlier in this piece. Multiply that by four or five linked charts on one dashboard and it stops being a small decision. It's now a thorny state-architecture problem, solved per project, with no default pattern to fall back on the way Dash's callback model already provides.

Enterprise-grade components make this gap wider. Dash AG Grid ships as a Dash component, so filtering, sorting, pivoting, and cell edits all flow through the same callback model as everything else in the app. An edit fires an Output, and whatever needs to react to it does, automatically. Wire the plain AG Grid library into a React app instead, and the grid manages a lot of its own state imperatively, through its own API, `gridApi.setFilterModel`, `onCellValueChanged`, and so on, which then has to be manually bridged back into React's declarative state model any time something outside the grid needs to know what changed inside it. That bridge is code an AI agent writes once and rarely revisits, which is where the inconsistent UX and the filter that quietly stops filtering tend to stem from.

Performance follows the same shape. Downsampling a chart that's about to choke on a hundred thousand points is, in Dash, a pandas operation, the same resampling or aggregation you'd already reach for in a notebook. In React, keeping a page of linked, high-frequency components responsive usually means correctly reaching for `useMemo`, `useCallback`, and `React.memo` to stop components from re-rendering each other into a crawl, a set of tools whose entire job is managing the consequences of the state-sharing problem above. Get a dependency array wrong on any of them, one of the most common bug classes in React code, and a chart quietly stops updating, or the whole page re-renders incessantly.

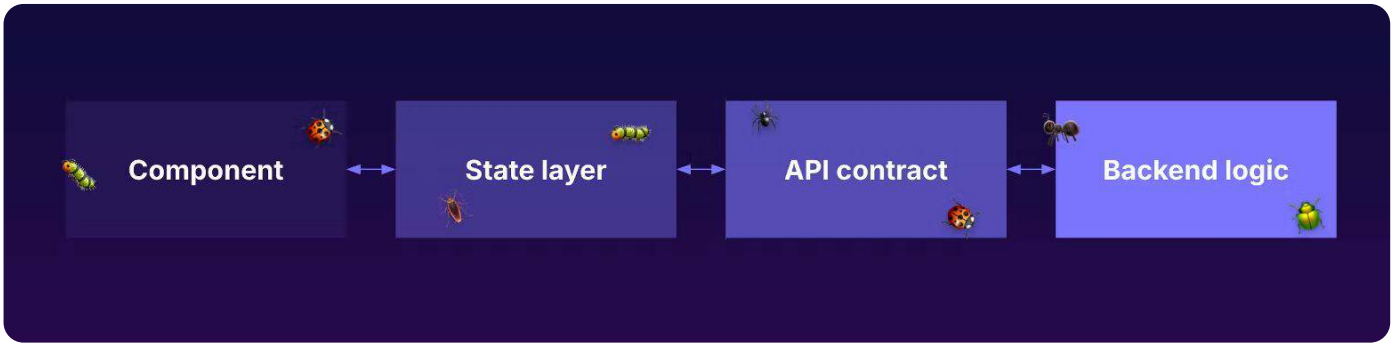
None of this is a knock on any individual charting library, but Dash's components already know how to talk to each other, and React's don't, not natively, which means every dashboard with more than one interactive element asks Claude to build the wiring between components from scratch, get it right the first time, and hope nobody needs to touch it again in six months.

Don’t chase bugs across the seams

Every AI-generated app eventually has a bug. The question is how far you have to dig to find it.

In a typical full-stack app, a bug could be in the component, the state layer, the API contract, or the back end logic, and it's often some interaction between two of those. Fixing it means understanding both halves and the handshake connecting them, whether you're debugging it yourself or asking Claude to.

In a single-process Dash app, there's just less of it. One language, one runtime, one place for the problem to live – a smaller blast radius for Claude to reason about when something goes wrong, and a much shorter path for you to walk when you need to understand why.



That's the case for why Dash is the easier, safer bet on day one, but most vibe-coded apps don't get judged on the day they ship. They get judged eight months later, when the person who built it has moved on, or when something buried in a thousand-package dependency tree gets flagged by a security scanner. The next part examines the risks you don’t see on day one.

Part 2: Hidden Risks

The risks you don’t see on day one

Most vibe-coded apps don't fail on day one. They pass every test that matters at launch: it renders, the charts update, the demo goes fine. The risk shows up later, on a timeline nobody was tracking when the app got built, as a dependency that turns out to be malicious, or a key person who leaves the company, or a laptop-grade prototype that just quietly became load-bearing.

A smaller attack surface

Here's a number worth sitting with. We installed both stacks ourselves.

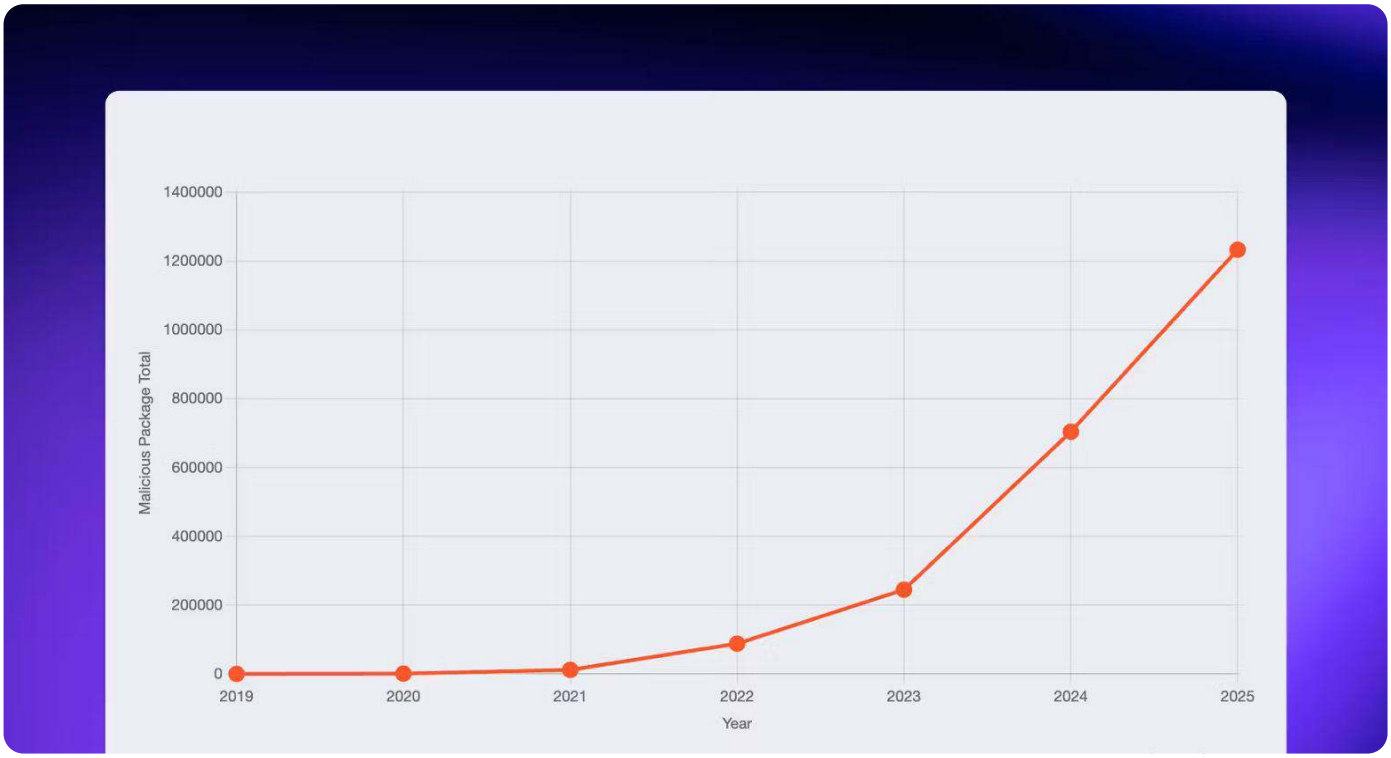
A fresh `pip install dash` resolves to 1 direct package and 12 total packages installed, Dash and everything underneath it included. Assembling a realistic React equivalent, a Vite app with React Router, TanStack Query, Recharts, Axios, Zustand, and Material UI, the kind of stack you'd actually need to chart and manage data, resolves 12 direct dependencies and **194 total packages**. And that's the front end alone. It doesn't include the separate back end service that stack still needs to talk to your data, which a Dash app already comes with. Every one of those packages is a file that runs on your machine or your user's browser, written by someone you've never met.

	Direct dependencies	Total installed
React (bare)	1	1
React (full stack)	12	194
Dash	1	27

Full Stack dependency list: react, react-dom, react-router-dom, @tanstack/react-query, recharts, axios, zustand, @mui/material, @emotion/react, @emotion/styled (deps) + vite, @vitejs/plugin-react (devDeps).

It's worth noting that Dash is built on React. The graphs, dropdowns, tables, and layout pieces you import in Python, are React components under the hood, and Plotly builds them with an npm toolchain: webpack, babel, React itself, and everything underneath those. The difference is where and when the dependency tree gets resolved. Plotly's engineers resolve it once, upstream, and ship the compiled output as a pinned, static JavaScript bundle inside the Python wheel. Nothing on your machine touches npm, and nothing re-resolves that tree when you or Claude run `pip install` six months from now. The 27 packages above are the dependencies you and Claude are on the hook for: the ones that get re-resolved on every install, the ones a new hallucinated name could slot into, the ones that show up fresh every time the project changes. It's not that Dash has no npm dependencies, but the dependencies are fewer, curated, and limit your exposure to a direct supply-chain risk, and doesn't compound every time your agent decides to include a new package.

That tree has gotten more dangerous over time. Sonatype's 2026 State of the Software Supply Chain report counted more than 454,600 new malicious open-source packages in 2025 alone, pushing its cumulative blocked total past 1.233 million, a 75% jump year over year. Over 99% of that malware showed up on npm specifically.



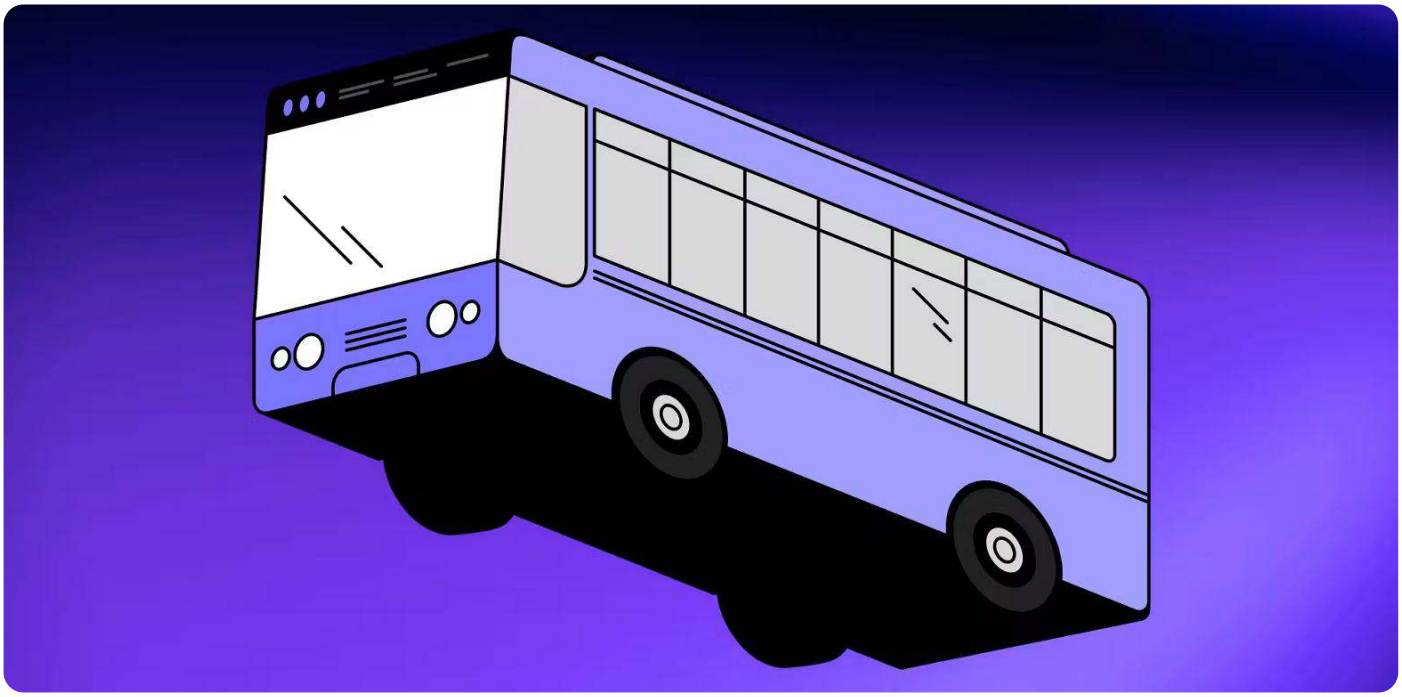
Some of the risk now targets AI agents directly. Researchers at USENIX Security 2025 generated 2.23 million code samples across 16 popular models and found that 19.7%, nearly one in five, recommended at least one package name that doesn't exist, with 43% of those invented names reappearing consistently enough for an attacker to register and wait. Security researcher Bar Lanyado proved this by publishing an empty package under a name he'd watched models invent, and it picked up 30,000 downloads in three months. In January 2026, researcher Charlie Eriksen found a hallucinated npm package called `react-codeshift`, a conflation of two real tools, spreading through 237 GitHub repositories via copy-pasted AI agent skill files, with agents themselves driving the daily download count. (Source: Aikido Security's slopsquatting research, building on the USENIX study.)

No ecosystem gets to sit this one out, and it's worth pointing out that Lanyado's proof of concept ran on PyPI, the same registry Dash depends on. If you know npm well, you might also object that its nested node_modules can hold several versions of the same package side by side, while Python's flatter environment can't, so version conflicts bite harder in Python when they do surface. That's fair. It's also part of why npm's tree grows the way it does: tolerating five different versions of the same dependency across a project avoids forcing a choice, it doesn't eliminate the conflict, and it's a contributor to why a realistic front end stack resolves to hundreds of packages instead of dozens. ERESOLVE errors, npm's shorthand for "these two packages want incompatible versions of the same thing," are common enough to have their own extensive troubleshooting literature. Fewer packages, regardless of ecosystem, means fewer places for these kinds of problems to arise, and Dash's 27 packages instead of React's 194 (plus whatever you're using as a back end) is a meaningfully smaller number.

The "bus factor" of a vibe-coded app

Most data apps aren't maintained by the person who built them, not for long. An analyst vibe-codes a churn dashboard on a Tuesday afternoon, it works, people start relying on it, and eight months later that analyst has moved teams or moved companies. Someone else on the data team inherits it. Whether it survives that handoff boils down to whether the person who inherits it can actually read it.

This well-studied problem in software engineering, called "the bus factor", is the minimum number of people who'd have to leave before a project stalls because the knowledge needed to maintain it is gone. A vibe-coded app starts with a bus factor of one by default, whoever wrote the prompts, and the question is how fast that number grows once other people on the team actually need to touch it. It's tempting to assume the answer is mostly about language: a Python analyst inheriting a full-stack app is stuck until they've put in months learning a stack that was never their job. That's not really true anymore. Claude narrows that gap on any codebase. The analyst can point Claude at the full-stack app and ask what a component does, ask it to add a filter, ask it to track down why a chart stopped updating, the same way they'd work through a Dash app. Whoever inherits a vibe-coded project isn't starting from a textbook any more, instead they get an interactive conversation with AI, which is a huge advantage.



What doesn't carry over is the ability to tell whether Claude got it right. Making an edit and knowing the edit is correct are different problems, and Claude is better at the first than the second. When Claude explains a pandas `groupby` to someone who already works in pandas, they can catch errors if the explanation is subtly off, because they have independent judgment to check it against. When Claude explains a `useEffect` and a piece of Redux state to someone who has never touched React, they're evaluating Claude's account of the code, not the code itself. If that account is confidently wrong, or the fix quietly breaks something adjacent, there's no independent backstop catching it before it ships to whoever's relying on the dashboard. Inheriting a codebase Claude can narrate to you is a thinner form of ownership than inheriting one you can actually think in yourself, even though Claude smooths over the friction of touching either one. The difference is borrowed confidence versus earned confidence. Confidence built on reading the code yourself travels with you: it holds up on a new bug next year, on an edge case nobody's explained yet, on a day Claude gets something wrong. Confidence built on being walked through unfamiliar code only holds up as long as the walkthrough is accurate and happens to cover the situation you're actually in.

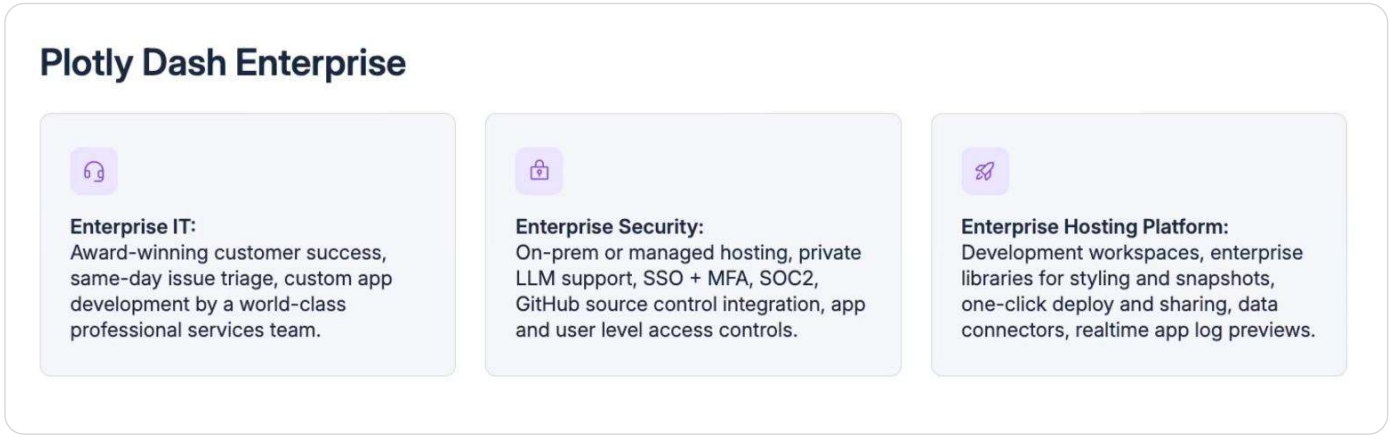
The second thing that doesn't go away is architecture, and it has nothing to do with who or what is doing the reading. A split front-end-and-back-end app still has more seams, more dependencies, and more places for a change to break something adjacent, whether the person making the change is a full-stack developer, a Python analyst working through Claude, or Claude itself navigating the codebase unsupervised. It doesn't shrink just because an agent is available to explain it. More seams means more places for an agent making a fast fix to introduce a mismatch as we covered in Part 1.

Undoubtedly, Claude lowers the cost of touching any codebase, but it doesn't teach the reviewer the finer details of a language they don't know by default, and it doesn't reduce the number of seams in an app. A Dash app keeps both of those advantages for Python users regardless of who, or what, ends up maintaining it. A full-stack app asks a Python developer who inherits it to either trust Claude's account of unfamiliar code or spend the time to stop needing to, and it hands them more architecture to hold that judgment over while they do it.

Who you gonna call?

Most vibe-coded apps start as a side project. Some of them turn into something a whole team depends on, and at that point "reach for Claude and hope" doesn't feel like such a great strategy.

This is where the two ecosystems diverge sharply. Full stack has no commercial steward. There's no single company you can call when you need SOC2 compliance, on-prem deployment behind your firewall, or a support contract with a human on the other end. You're either self-hosting on borrowed infrastructure or shopping around for a third party who didn't build the framework and doesn't control its roadmap.



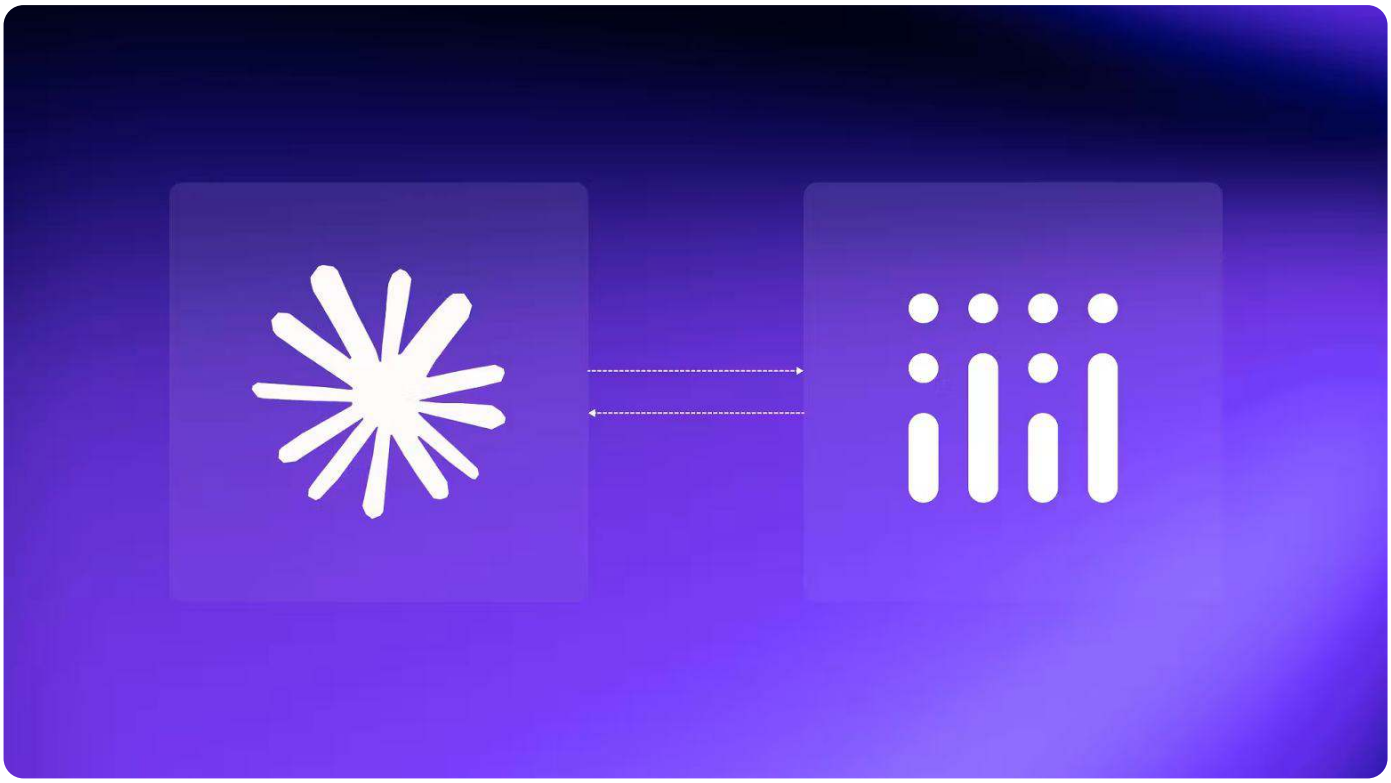
Dash has Plotly behind it, and Dash Enterprise turns "the app Claude built me" into something with a real deployment story: one-click publishing, private model support so your data never leaves your network, and an actual team to call when it matters. That's a tremendous value for anyone whose vibe-coded project outgrows a laptop.

Everything in this part has been about risk exposure: what's sitting in your dependency tree, who can actually maintain the app, whether anyone's picking up the phone when it breaks. The next part looks at the cost difference between AI development using Dash vs. full stack. It's a look at what building the same app in Dash versus a traditional full stack actually costs in tokens, and the counterpoint of when full stack is still the right call.

Part 3: Token Costs

Doing the math on vibe-coded apps

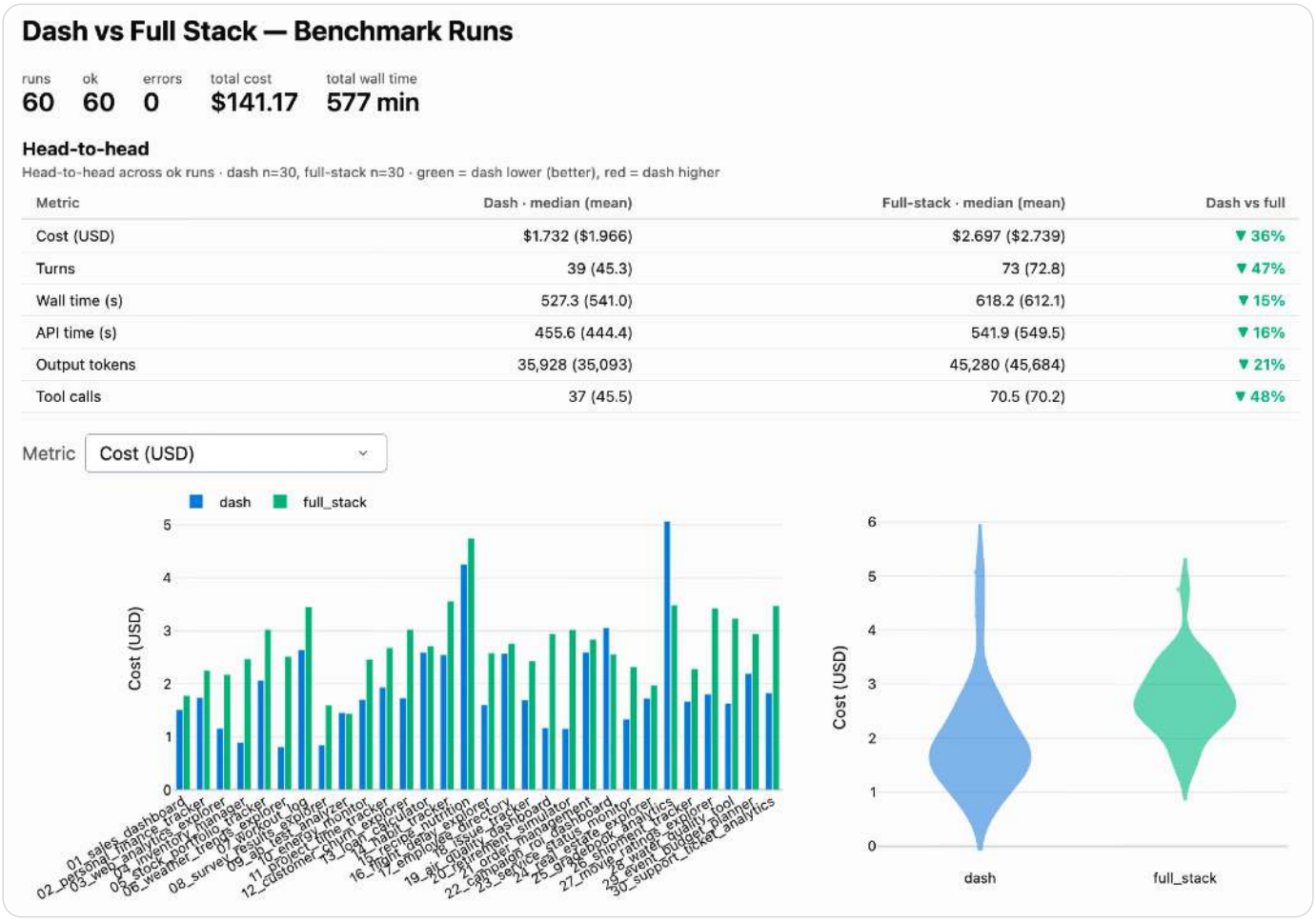
Everything in this series so far has been an architectural argument: fewer moving parts, a smaller dependency tree, an easier app to inherit. All of that should cost less to build and maintain with Claude. We wanted a number instead of just a theory.



Let's see the receipts

We ran a benchmark: the same set of data app prompts, built twice, once as a traditional full-stack app, a React front end plus a back end, and once in Dash, tracking actual token spend across the build.

Early results point to Dash apps costing roughly 33% less than what the equivalent full-stack app costs. In both cases, there's generally one big cost up front to generate the boilerplate. Feature additions in Dash tend to be cheaper given the more limited scope of the code that is needed to be written. Testing/probing costs once the app is operational tend to be the most expensive and higher for full-stack apps.



The sample is just a handful of app builds and the numbers could vary depending on app complexity, the model doing the building, and how disciplined the prompting is on either side. We plan to publish a separate article going in depth into the benchmark itself with the prompts, apps, and raw token counts, so anyone can take a closer look and check our math. Stay tuned!

When to reach for full stack

None of this makes full stack a bad choice for everything. If what you're building is a highly custom, consumer-facing product with novel interactions and a designer in the loop, full stack's flexibility is the reason to reach for it. Dash is a framework built specifically for data apps and internal tools, and it shows: the more your project looks like a general-purpose consumer product, the less that focus works in your favor.

But that's a narrower slice of what people are actually vibe-coding. Most of the "build me an app" prompts flying into Claude right now describe a dashboard, an internal tool, a report, something to explore or share data through. For that job, the smaller footprint is desirable – less for the AI to get wrong, less for you to review, and less standing between a good idea and something you can actually put in front of your team.

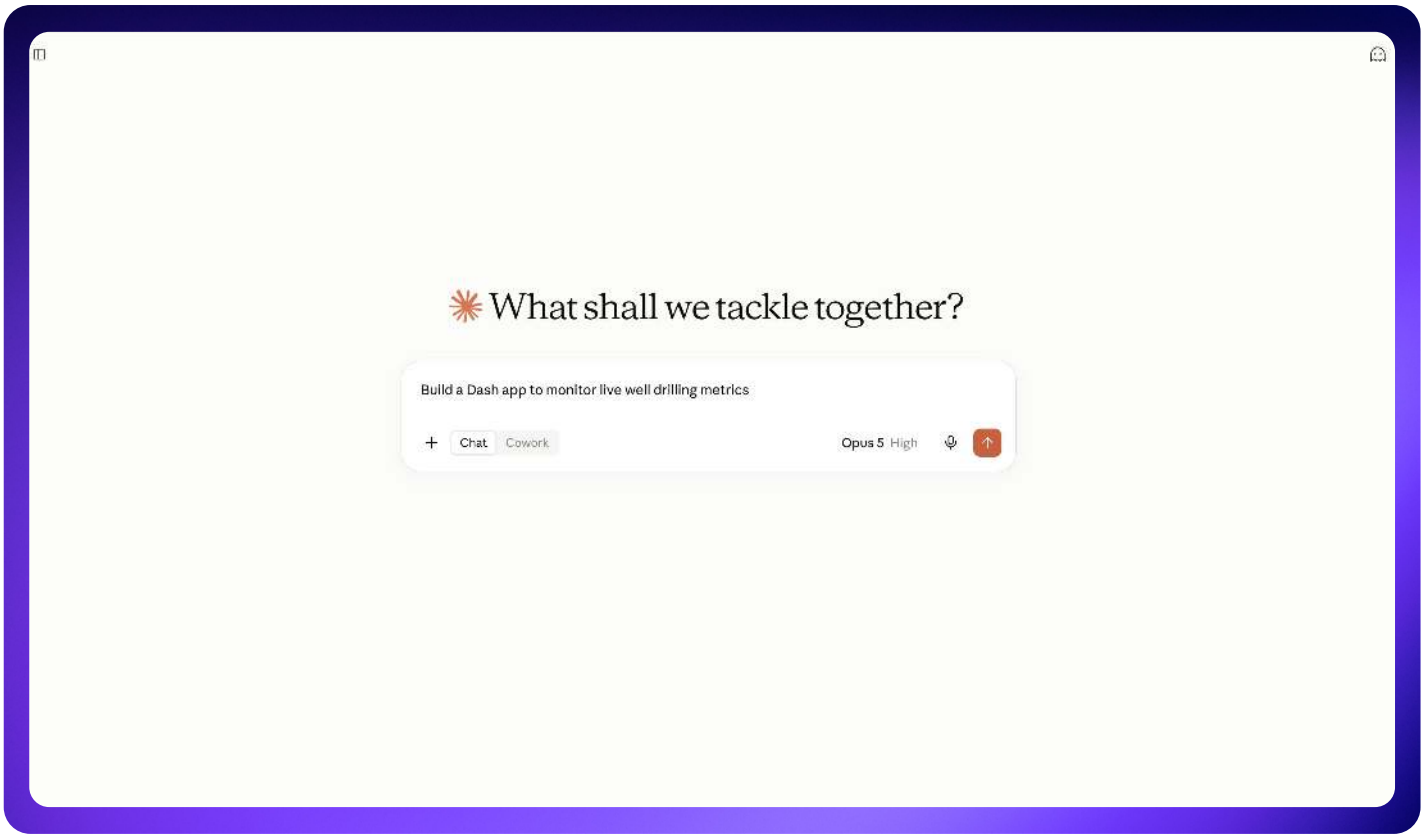
Keeping a front end and a back end in sync isn't a problem a lightweight model can shrug off. It takes something like Fable, a genuinely capable model, to hold both halves of the contract in its head at once and catch the drift before it becomes a bug, and that's not a cost you pay once. It compounds every time the project grows, every feature that touches both sides, every year the app stays in production and someone asks Claude to change something.

If the dashboard or internal tool you're building never needed two applications in the first place, none of that spend was necessary. A little care at the outset, naming Dash instead of leaving Claude to default to full stack, is the difference between paying for a bigger model to manage a problem you created and not having the problem at all. Over the life of a project, that's a material number of tokens, plus the harder-to-see cost of a team's time spent debugging a seam that never needed to exist.

Pick a framework, any framework, preferably Dash

So the next time you sit down to describe an app to Claude, consider naming the framework instead of leaving it up to chance. One line in the prompt is the difference between inheriting a single Python application you can read end-to-end, and inheriting two complex applications stitched together with a brittle API and backed by a dependency tree hundreds of packages deep.

Tell Claude to build it in Dash.



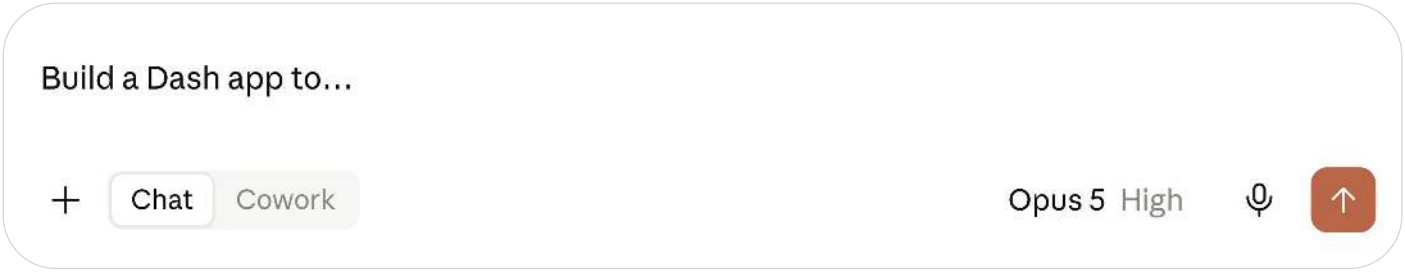
Start your prompt with ***Build a Dash app to...***

Let's review the full list of reasons to choose Dash over full stack for data apps.

1. **One app, not two.** You asked for an app, but full stack gave you two: a front end and a back end with an API layer gluing them together. That's two things that have to be built, hosted, and kept in sync, or worse, a front end that never got a back end at all. Dash is one application, one language, one process.
2. **No translation tax.** Your data already speaks Python. Dash callbacks call pandas, NumPy, and scikit-learn directly. Full stack has to flatten that same data into JSON first, a boundary that breaks on ordinary things like datetimes and NaN values.
3. **Fewer decisions, fewer forks in the road.** Full stack doesn't have a default state-management pattern, so Claude has to pick one for every project introducing possible inconsistencies and complexity. Dash's callback model is one well-worn default that works every time.
4. **The wiring is just as important as the charts.** Cross-filtering and linked components in Dash are just another callback. In full stack, Claude has to invent the state architecture connecting components from scratch, every time.
5. **A smaller blast radius when something breaks.** One language, one runtime, one place for a bug to live, and a much shorter path to understanding why.
6. **A smaller, checkable dependency tree.** A fresh Dash install resolves 27 packages vs. a full stack app which resolves to hundreds, in an ecosystem seeing growth in malicious and AI-hallucinated packages.
7. **Easier hand off.** Someone else has to read the code eventually. A Dash app can be reviewed by anyone on a Python-fluent data team. A full-stack app asks whoever inherits it to either trust Claude's account of unfamiliar code or learn a new language.
8. **Dash has a steward.** Plotly owns the Python visualization stack and deployment platform: Dash (10M downloads/month), Plotly graphing libraries (68M/month), and on-prem/cloud hosting. No competing visualization stack is as vertically integrated or widely deployed. Full stack has no single company to call when something breaks.
9. **It shows up in the token bill.** We ran the numbers and Dash apps cost roughly 33% less than the equivalent full-stack app costs.
10. **The exception:** for a highly custom, consumer-facing product with a designer in the loop, full stack's flexibility is still the right call. This piece is about the data apps, internal tools, and reports that make up most of what people are actually vibe-coding.

One app, one language, one codebase to maintain

Ask Claude to build it in Dash.



Where to go next

Watch the webinar: The full Dash vs. full stack argument, walked through end to end. plotly.com/webinars

Try Dash Enterprise: One-click deploy, on-prem or managed hosting, SSO and SOC2, and a team to call. plotly.com/dash-enterprise

Read the docs: Callbacks, components, and Dash AG Grid. Where you go when you want to check Claude's answer yourself. dash.plotly.com

About Plotly

This paper is brought to you by Plotly, the company behind Dash and Dash Enterprise. Dash is downloaded 10M times a month; the Plotly graphing libraries, 68M. With customers across the Fortune 500, Plotly is a category-defining leader in enabling data-driven decisions from advanced analytics, machine learning, and artificial intelligence.