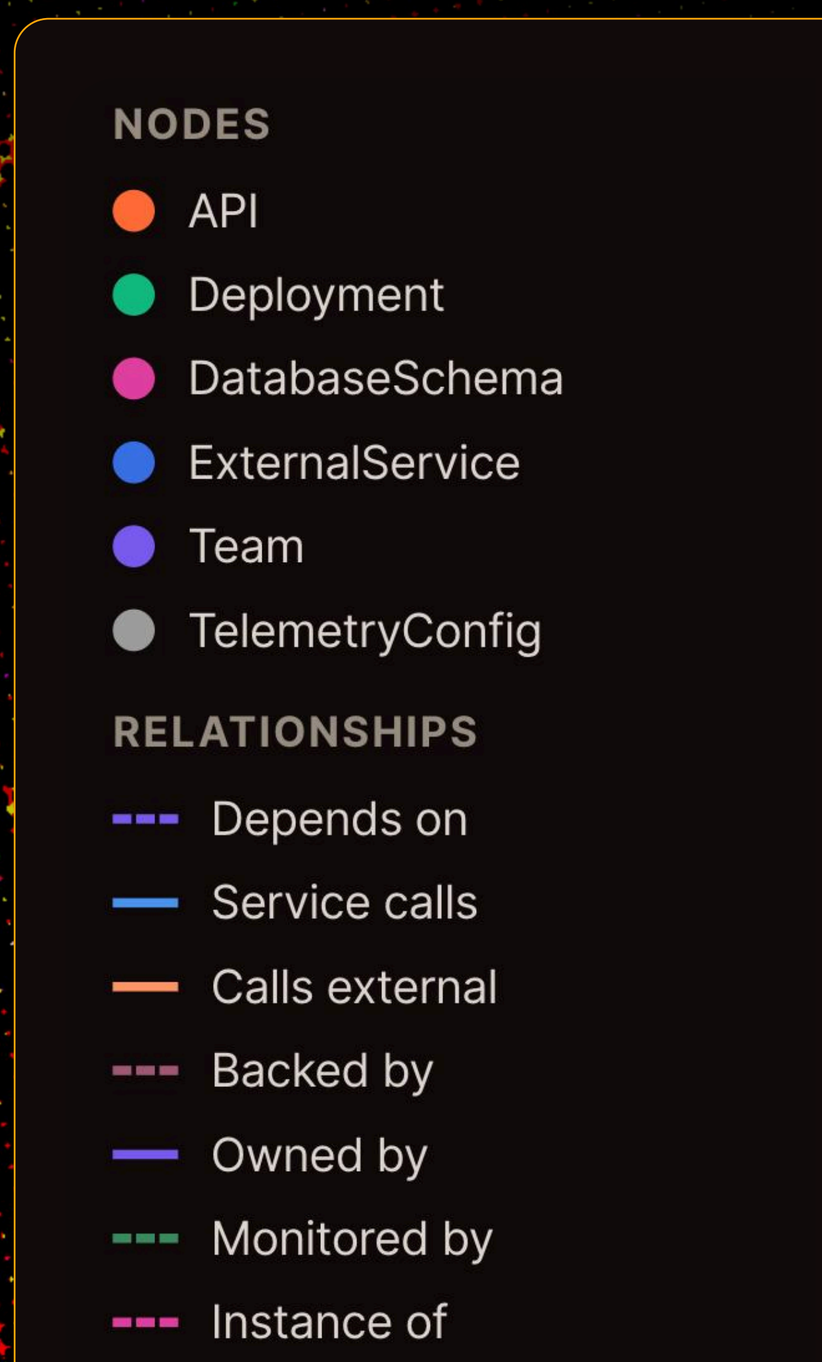
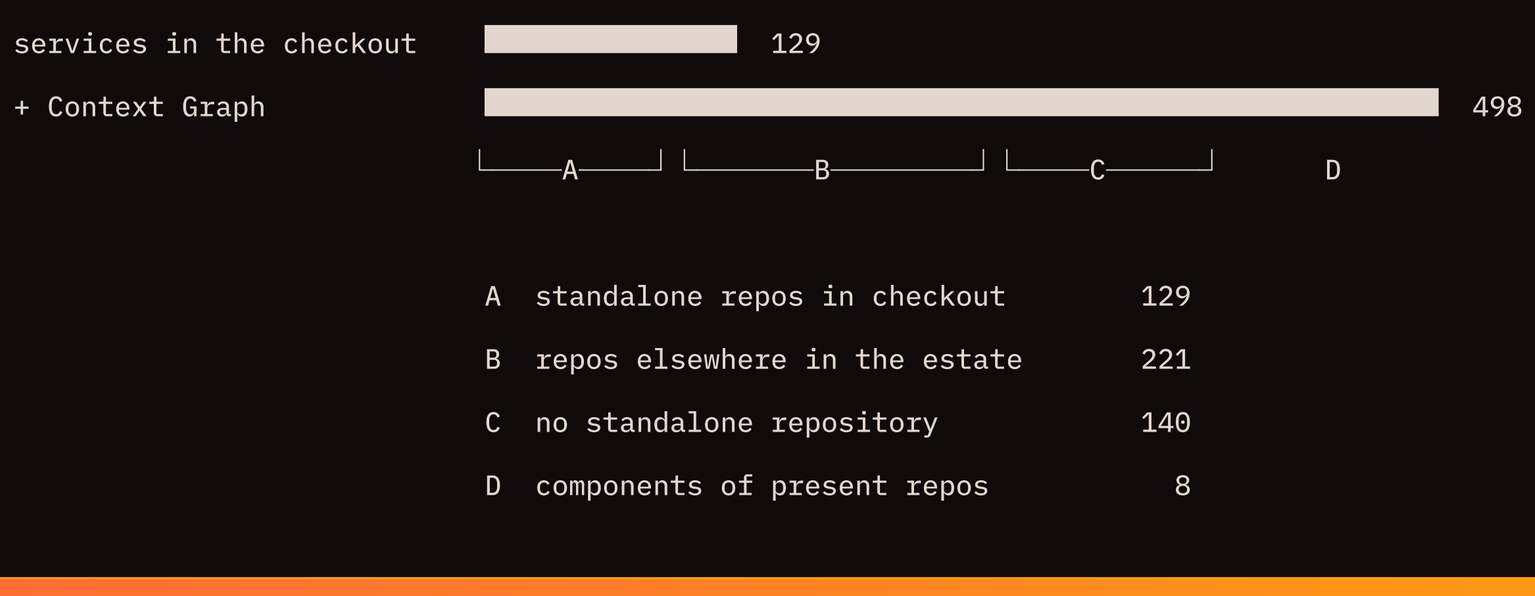
POSTMAN

Context Graph API Benchmarking Report





A	standalone repos in checkout	129
B	repos elsewhere in the estate	221
C	no standalone repository	140
D	components of present repos	8

POSTMAN

Table of Contents

Context Graph API Benchmarking Report

We wanted to understand what the Context Graph API actually adds when an agent is already searching a large codebase. So we tested frontier models with and without the Context Graph API across an estate with 468 repositories.

The simplest way to think about the difference is a library.

Imagine you walk into a library and ask the librarian to find everything about chess. There is no computer catalog, so the only way to answer is on foot. The librarian can walk through the building, search every shelf, and inspect individual books. A more experienced librarian might do that faster and notice more, but they are still searching the entire building from scratch.

Now give the librarian access to the library's computer catalog. Instead of starting with the first shelf and working aisle by aisle, they can search once and get a list of relevant books with their exact shelf locations. The catalog tells them where to begin, which books to pull, and which sections deserve a closer look.

It can also point them to a chapter inside a book with an unrelated title, a book stored at another location, or a resource that is not on a physical shelf at all, like an audiobook. The librarian can then open the available books and verify that they contain what you need.

In this analogy, the librarian is the model, the shelves are the checked-out repositories, and the catalog is the Context Graph API. The Context Graph API tells the model which services, APIs, endpoints, and relationships are likely to matter and where to focus its search. Instead of exploring hundreds of repositories one by one, the model can start with a specific set of candidates and verify those leads in the code wherever possible.

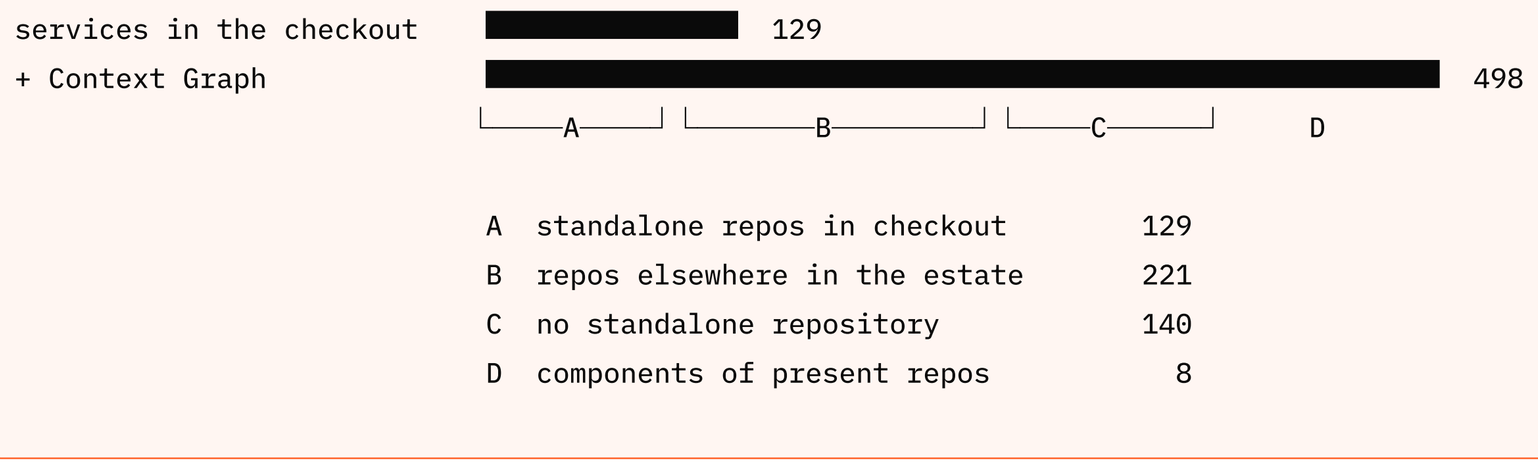
The catalog is also updated every night as books are added, removed, or moved. In the same way, the Context Graph refreshes nightly from your connected data, so it continues to reflect new services, changed endpoints, and updated relationships.

A better model is like a faster, more experienced librarian. It can search the shelves more efficiently, but it still cannot find something that is not on those shelves.

The Context Graph finds what file search cannot

To see what that system-wide catalog makes visible beyond the checked-out code, we asked the agent a straightforward question:

Which services expose HTTP endpoints, and what paths does each one serve?



A	standalone repos in checkout	129
B	repos elsewhere in the estate	221
C	no standalone repository	140
D	components of present repos	8

Of the 498 services the Context Graph returned, 129 have code in the checked-out repositories.

Think back to the library. File search could reach the 129 sitting on the shelves in front of the librarian. The Context Graph opened the catalog and pointed to 369 more entries.

Those additional services came from three places.

221 are repositories elsewhere in the wider ecosystem. The Context Graph is indexed across everything connected to it, which reaches further than the set of repositories this benchmark ran against. The code exists; it is simply in a different part of the ecosystem. These are books held at another branch of the library, and no amount of searching the local shelves will turn them up.

140 have no standalone repository. Some are external systems called at runtime. Others are services known through connected data but not mapped to a repository of their own. They appear in the catalog, but there is no standalone book on the shelf for the agent to open.

8 are components inside repositories already in the checked-out code. Their code is present, but they are more like chapters inside larger books. File search sees the repository and names it once, while the Context Graph identifies the component inside it as a separate service. Finding those components by searching would mean working out how the whole repository is assembled, and nothing in the code marks one as a service of its own.

These were not obscure edge cases. They included ordinary services in the middle of real request paths, such as gateways, accounting, ad serving, and booking. They are running, something is already calling them, and nothing in the checked-out code tells you they are there. If one of their endpoints changes, the first time you hear about it should not be when you're paged for an incident.

This is not a search-quality problem. It is an access problem. The agent was not searching badly; the files that would have named those services were not there to find. Better search will not close that gap. You cannot grep what was never in the checkout. The librarian searching for a book on foot will never find a book that is not on the shelves.

How the AI harness works

Each model answers eight questions covering all the parts of an API ecosystem: services, endpoints, schemas, event consumers, deployments, upstream dependencies, telemetry, runtime call relationships, and the callers of a changing API.

The prompt contains only the question. It does not tell the model where to look, how to search, or how many answers it should find. Those are exactly the things the Context Graph is supposed to help with. If we gave the baseline model the same clues for free, we would shrink the difference we were trying to measure. The harness even fails its build if a prompt hints at the search method or expected answer count.

The code decides what is correct

Every answer is graded in three ways. Two graders are deterministic code, and one is a language model.

The first compares the answer with a key built directly from the checked-out code. Every expected result in that key has been independently verified with grep.

The second checks every cited file path against the repository at a pinned commit. If the model invents a path or cites a file that does not exist in that version of the code, the grader catches it.

The third grader is a language model following a rubric. It carries the least weight because, unlike the other two graders, it cannot inspect the repository itself.

We also score only the answer the model commits to in a structured results block. The surrounding explanation does not count. Otherwise, a model could mention a repository, investigate it, rule it out, and still receive credit for naming it. That would reward whichever arm produces more prose rather than the one that produces the better answer.

Most importantly, the answer keys always come from the code, never from the Context Graph. Grading the Context Graph API against its own output would be marking its own homework. It would also make the results impossible to challenge or reproduce independently.

One variable changes

Both arms receive the same question, the same tools, and the same code at the same pinned commit. They run at the same temperature. The only difference is that one arm starts with querying the context graph, while the baseline starts without it. The baseline is not even told that the Context Graph exists.

Each question runs three times per arm. That matters because one model response is not a measurement. It is one sample from a distribution. Ask the same question twice and the results can differ dramatically. Language models are variable by nature: the same prompt, the same code, and the same settings can still produce a different search path and a different answer, so a single run tells you as much about luck as it does about capability.

The harness compares the two arms question by question and repeat by repeat. That controls for some questions being naturally harder than others and helps isolate the effect of the context graph.

We only call a result a win when both the significance test and confidence interval support it. If there are not enough paired observations, the harness reports insufficient data instead of forcing a conclusion.

The judge does not know which answer used the Context Graph API

Whenever a model is used as a grader, it never judges a run in which it is competing. We also prefer a judge from a different model family because models can favor answers that resemble their own.

All answers to the same question are sent to the judge together, anonymized and shuffled. The judge cannot tell which answer came from the baseline and which used Context Graph data. It has to score the answer rather than the condition.

If the judging call fails, that score is dropped. We do not replace it with a neutral value because that would make a broken grader look like a working one.

Model performance with and without the Context Graph API

We ran eight prompts through Claude Opus 4.7, Claude Sonnet 4.5 and GPT-6 Astra, each one asked twice: once with the model alone, and once with the model plus a single query to the Context Graph. Three runs of every combination, both arms, 144 cells in all.

After the context graph completed ingestion, it contained 6,712 nodes, including 4,828 endpoints, 1,424 APIs and services, 309 external systems, 139 database schemas and 11 telemetry configurations.

Each section below covers one prompt: an explanation of what it asks and why it matters, the question as each arm received it, and a scorecard per model.

Each card carries six rows. Two of them are the result and four are what it cost to get there. Correct services found is how many things in the answer key the arm actually named, not how long its list was, because a long list is easy and being right is not. Accuracy is that same answer scored against the key.

The Context Graph API did not improve its score by finding services outside the checkout. It improved because it helped the model find more of the services that were already there. For example, on runtime calls, Opus found 36.5% of the known services with the Context Graph API, compared with 11.1% on its own. It also found more of the known HTTP endpoints and API callers. Both runs had access to the same code, but the Context Graph API helped the model find more of what was inside it.

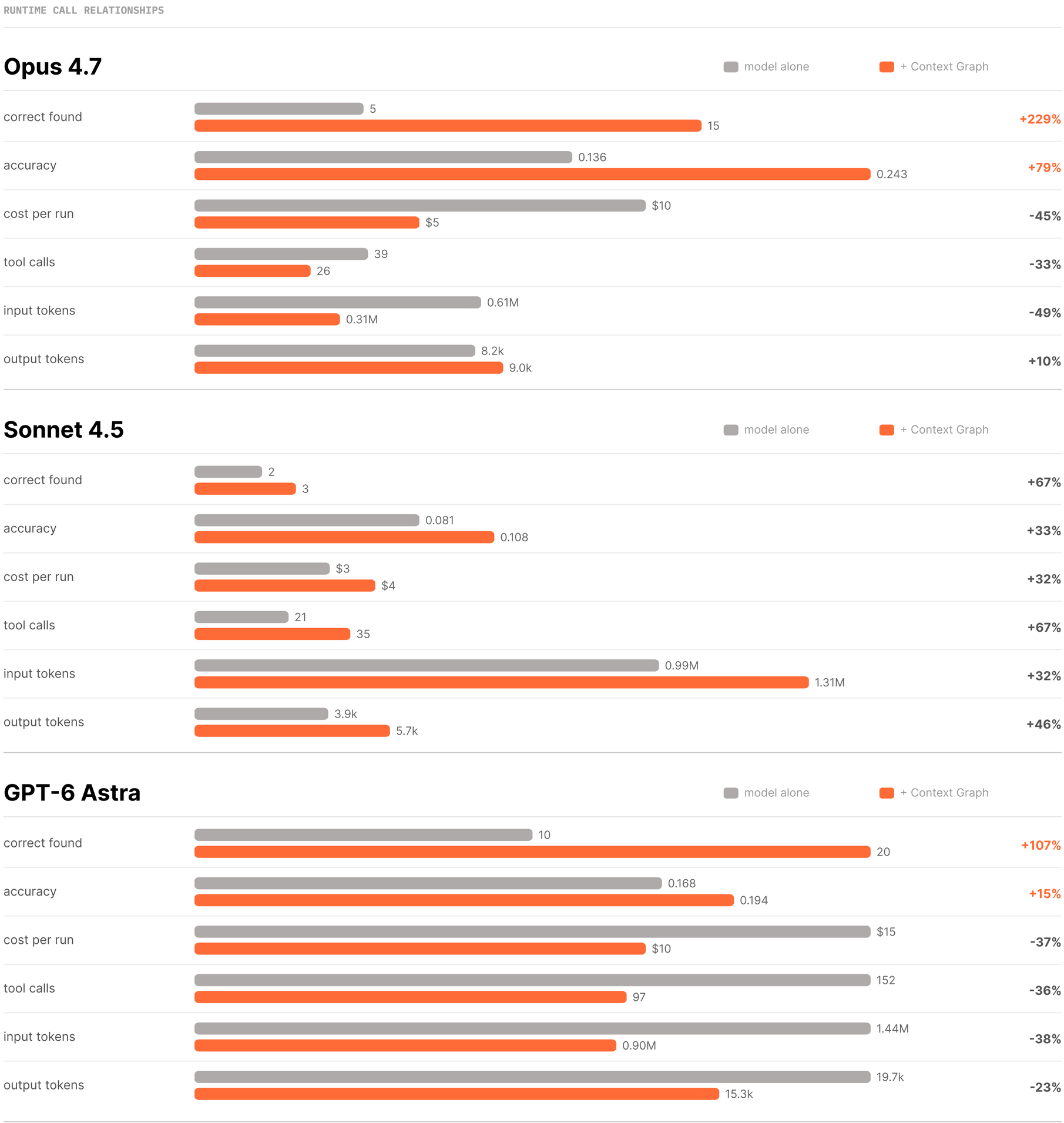
So the Context Graph API is helping in two separate ways. First, it helps the model find more relationships within the code it already has. That improvement appears in the accuracy score. Second, it can surface services and relationships that are not represented in the checkout at all. That value cannot appear in the score, so we call it out separately in the commentary.

We tested runtime call relationships, telemetry, deployed services, HTTP endpoints, message-bus consumers, API callers, schemas and upstream dependencies: eight questions an engineer has to answer before updating code. The first seven are scored against an answer key. The eighth is scored on coverage, for a reason its own section explains.

Runtime call relationships

Let's say you are responding to an incident where several services start slowing down at the same time. At first, they look like separate problems, but they all depend on the same service at runtime. That connection is not obvious in the code because runtime calls do not always appear as build dependencies. Before you can diagnose the actual problem, you have to trace requests across multiple services just to understand how they are connected. That missing context turns what could have been a quick diagnosis into a much longer incident.

What are the runtime dependencies between services?



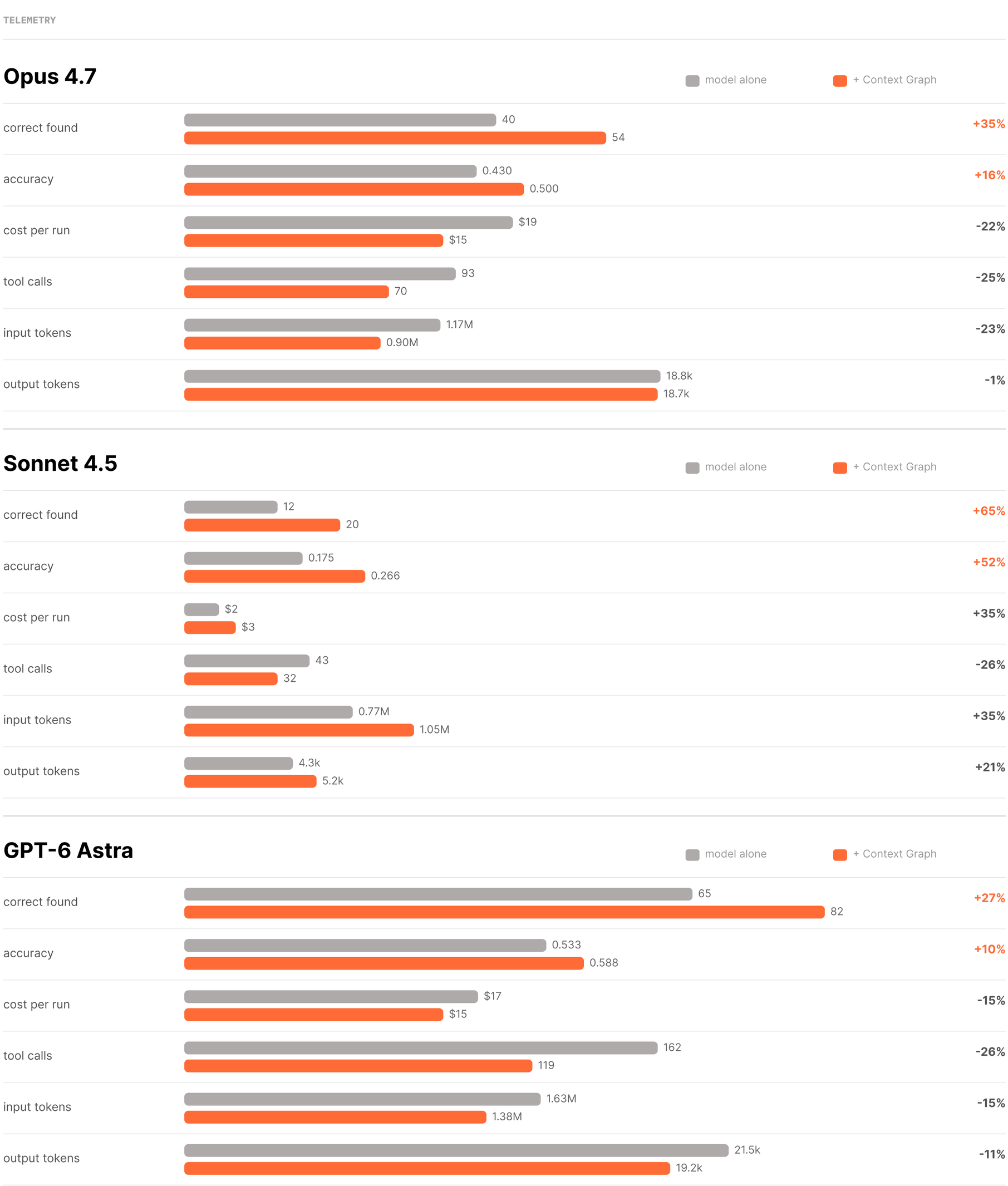
All three models saw significant improvement with the Context Graph.
The reason is that this relationship is not written down. When one service calls another at run time, neither repository records it: the target is a URL assembled from configuration at startup, so there is no string to grep for. The Context Graph API stores the call as an edge rather than as text, and where it returns enough of them the effect is large. In practice this is the question behind most bad incidents.

You change a timeout, a response shape, a retry policy, and something you have never heard of falls over, and you find out from another team's alert channel rather than from your own review.

Telemetry

Let's say an incident starts and several services could be responsible. Before you can debug it, you need to know which services collect the logs, metrics, and traces that can show you what happened. If a service has no instrumentation, you have no clear way to see what it was doing when the failure occurred. That leaves you debugging blind, working backward from symptoms instead of following the request to the source.

> **List each service and the logs, metrics, and traces it produces.**



All three models improve, and this prompt is the clearest case in the report of the query mattering as much as the index. Opus went from 40 correct out of 128 to 54 and accuracy 0.430 to 0.500, for $15 a run against $19. Astra went from 65 to 82. Sonnet improved most in relative terms, 12 correct to 20, accuracy 0.175 to 0.266.

Our first attempt at this question returned 11 rows, three times out of three, and we nearly published that as a finding about the index. It was a finding about our query. The graph holds exactly 11 TelemetryConfig nodes against 1,424 APIs and services, and asking which services "have monitoring or observability instrumentation" lands on the scarcest node type in the whole graph. Asked as a relationship instead, which services emit logs, metrics, traces or profiles and where they send them, the same endpoint returns over 200 rows. We re-ran the graph arm against that phrasing; the numbers above are the result, and the model-alone arm is untouched because it never sees the query.

What the better query bought is visible in the part the answer key cannot score. On the old phrasing the graph contributed no off-filesystem services at all. On the new one Opus named 24 instrumented services with no repository in the checkout and Astra named 21, nearly all of them traceable to their own query results rather than recalled from a public estate. Sonnet is the exception and went backwards, 0.383 on the old query to 0.266 on the new one: 219 rows helped it less than 11 did, which is worth knowing if you are pairing a small model with a broad query.

The practical version of this question is the one you ask at the start of an incident: which of these services can actually tell me what they were doing? A service with no instrumentation is a blind spot you find at the worst possible moment, and the ones with no code in your checkout are the blind spots you do not even know to look for.

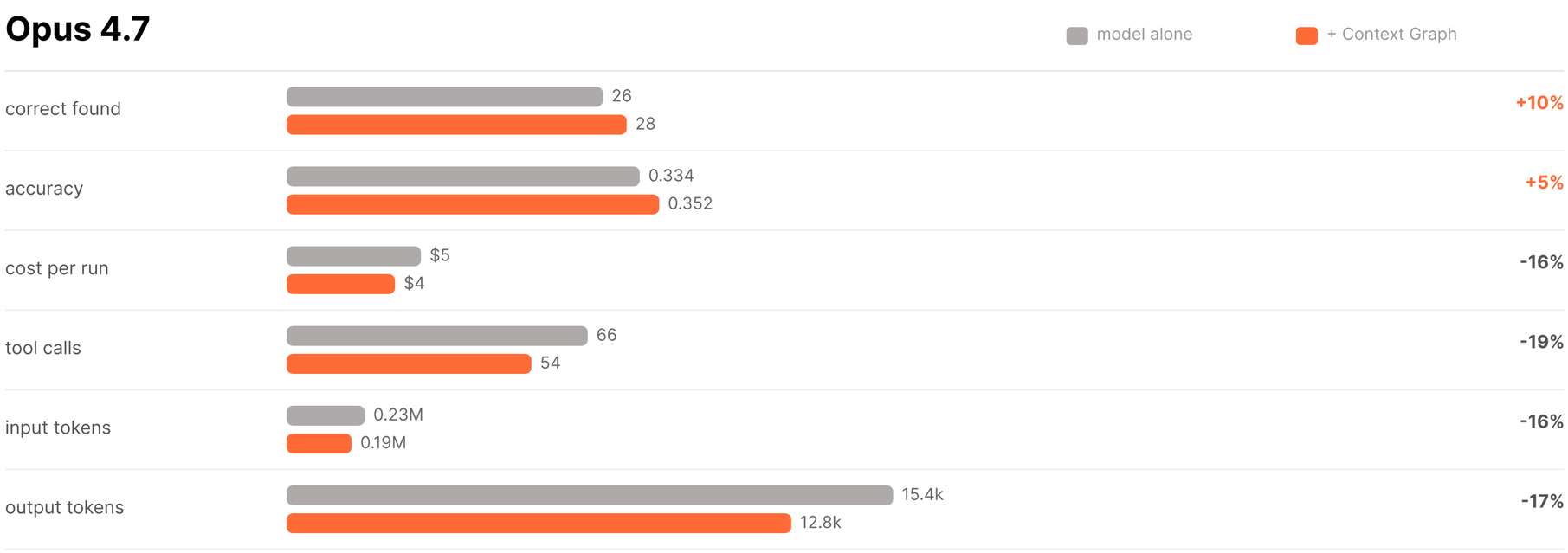
Deployed services

Let's say you're doing a cost audit of all the services running in production. You start with a list of repositories, but most of them are libraries, shared packages, and internal tools that do not run on their own. Before you can connect infrastructure spending to individual services, you need to figure out which repositories actually represent deployed services.

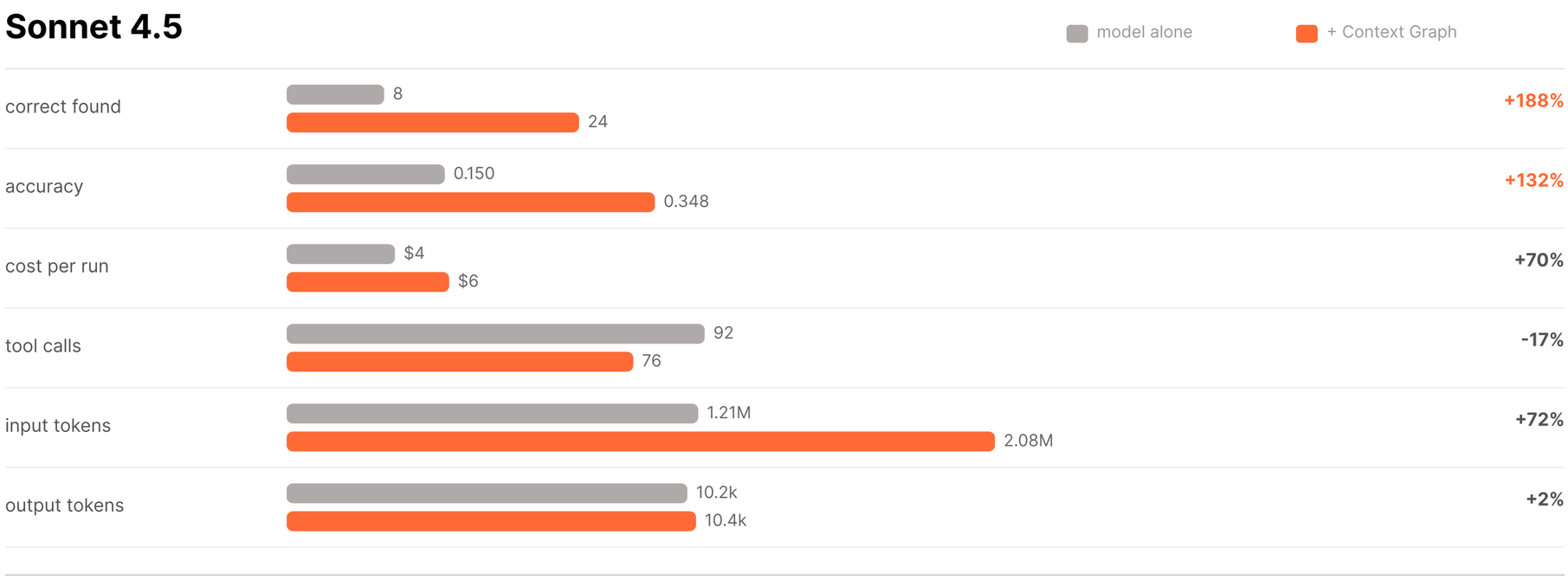
Which services do we actually deploy, and how is each one deployed?

DEPLOYED SERVICES

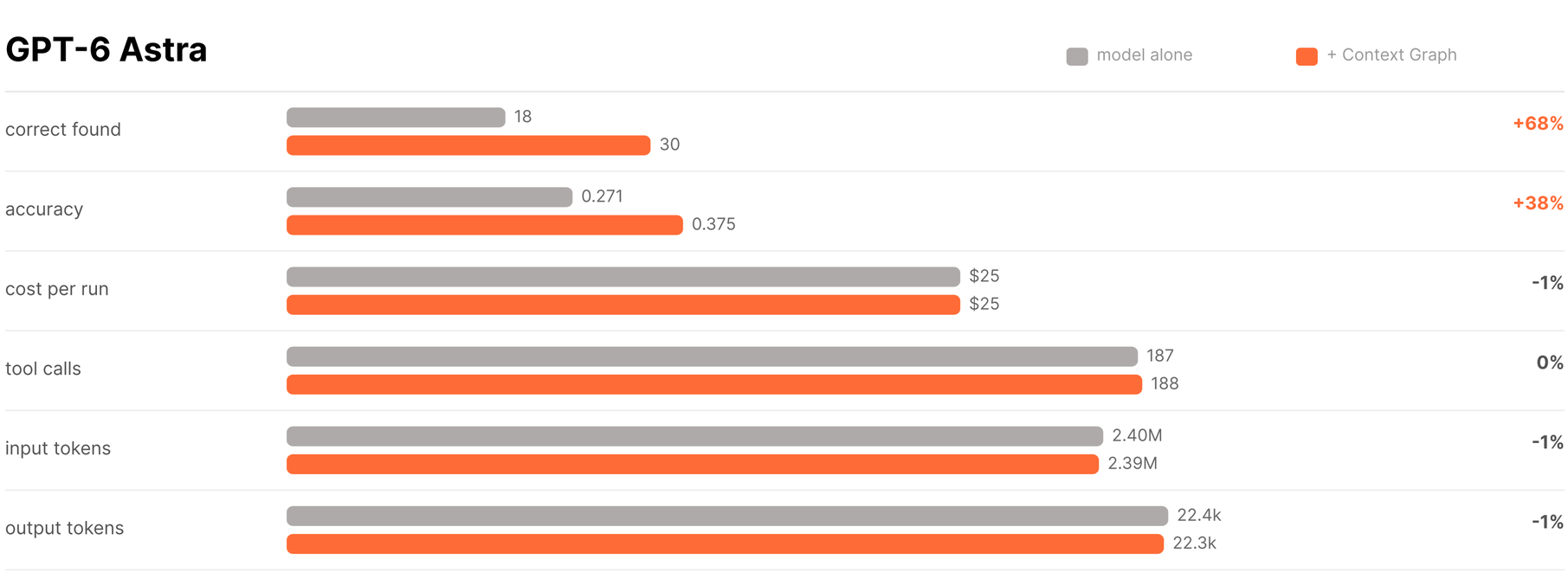
Opus 4.7



Sonnet 4.5



GPT-6 Astra



Every model did better with the graph.

The hard part is that every repository describes deployment differently. One might use a Kubernetes manifest, another a Helm chart, and another Terraform or Docker Compose. A model searching the code has to recognize all those formats and then work out whether a service is actually deployed or simply has the files needed to deploy it. The Context Graph stores deployment as a direct relationship.

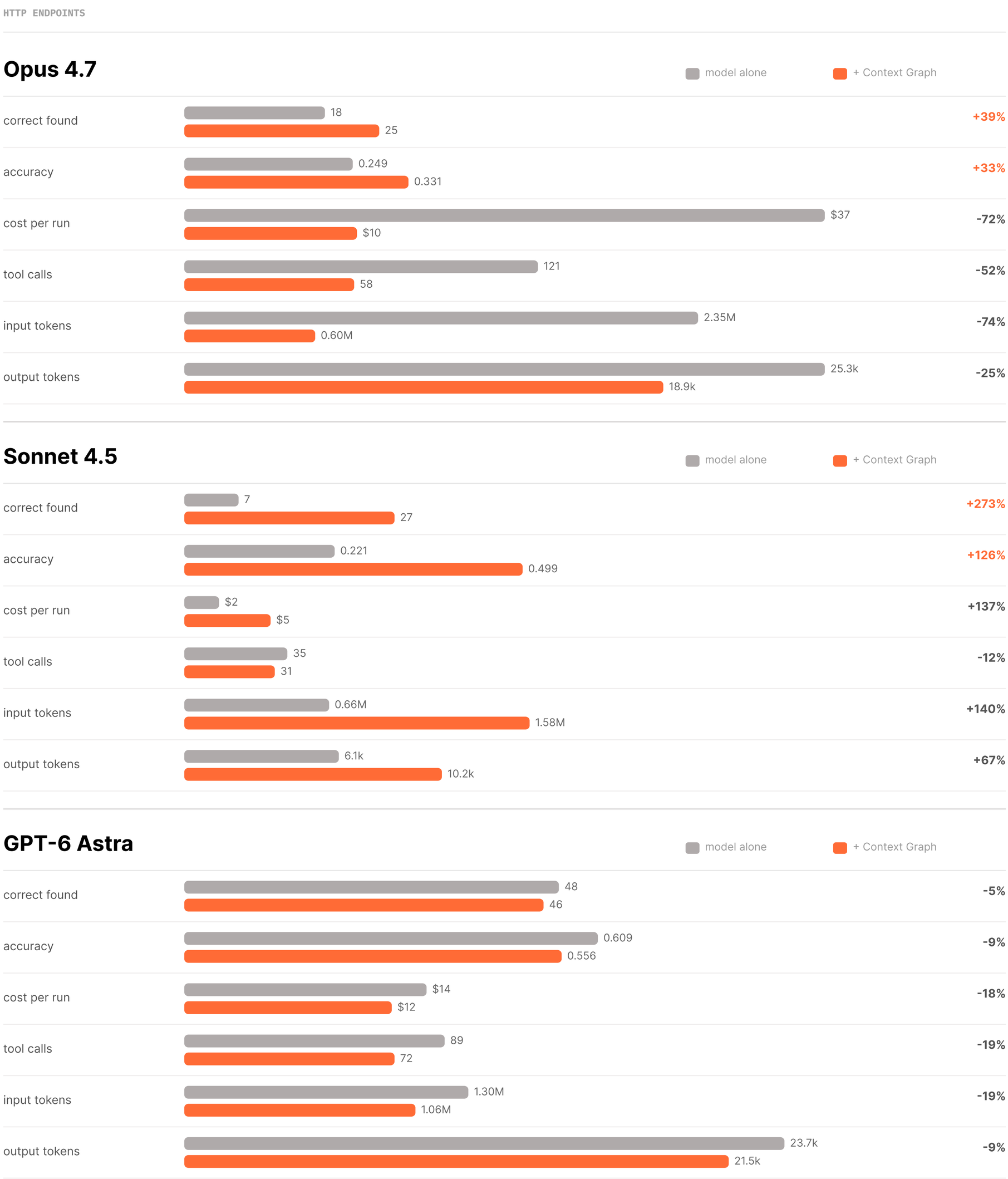
The graph also helped find services that code search could not. Opus named 10 deployed services with no repository in the checkout, and 5 came directly from the graph results. These were real services running in the environment, even though there was no local folder for the model to inspect.

This is the list you need when finance asks what your services cost to run. It is also the list you need before moving a cluster, changing a base image, or rotating a secret. A repository list includes libraries and internal tools that never run on their own, while potentially missing deployed services that can actually break.

HTTP endpoints

Let's say you get a task to require authentication for every API endpoint. Before you can safely roll that out, you need to know exactly which endpoints exist and who is calling them. If you miss a caller, you can break legitimate traffic the moment authentication is enabled. If you miss an endpoint, it may remain publicly accessible without anyone realizing it.

Which services serve HTTP endpoints, and what paths does each one serve?



Every model except Astra did better with the graph, and the improvement was especially large for Sonnet.

On its own, Sonnet found 7 of the 53 known services. With the graph, it found 27, and its accuracy more than doubled from 0.221 to 0.499. Opus improved from 18 correct services to 25 while cutting the cost from $37 per run to $10. It also used half as many tool calls and about a quarter of the input tokens.

The graph returned 456 results, but what it found matters more than the size of the list. Opus identified 48 services that had no repository in the checkout, and 47 came directly from its graph query. These were not services the model happened to remember. The graph retrieved them.

It also found 9 separately running components inside repositories the model had already searched, including an operator's webhook and a gateway's readiness probe. Each component exposed its own paths under its own name. A repository search usually finds the repository once and moves on, so it can easily miss components running inside it.

This matters when you are making a change like requiring authentication on every endpoint. If you miss a caller, you can break legitimate traffic when authentication is enabled. If you miss an endpoint, it may stay open even though everyone believes the rollout is complete. In this case, 48 services had no code in the checkout. You would not miss them because you searched badly. You would miss them because there was nothing there to search.

Astra was the exception because it already performed extremely well on its own, finding more than 90% of the known services. The graph gave it new information, including 28 services outside the checkout, but it also pulled its attention toward services that did not expose endpoints. As a result, it found two fewer services from the answer key and its precision dropped. The graph still helped Astra discover things it could not find on disk, but it had very little room to improve on the services that were already there.

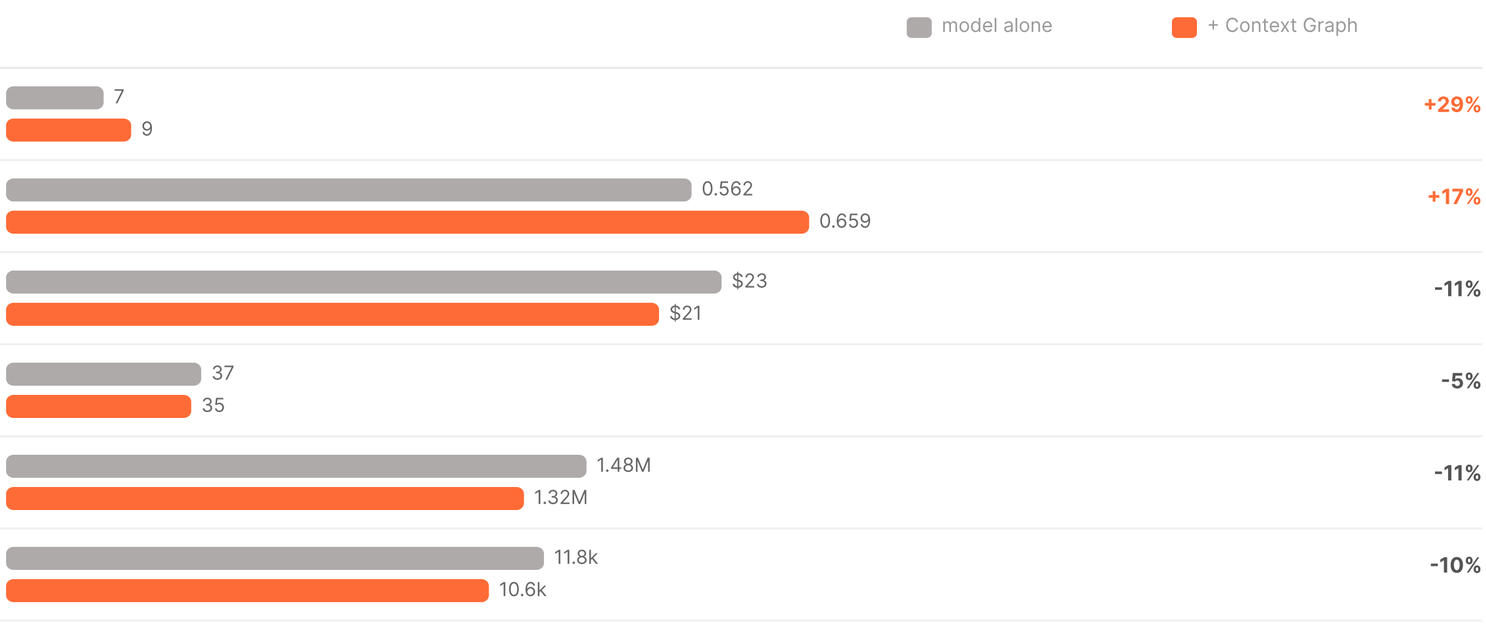
Message-bus consumers

You are preparing to migrate the message bus. You already know which services publish events, but there is no reliable list of everything consuming them. Each consumer may register its subscription only in its own startup code, so there is no central file you can check. To migrate safely, you have to search every service and piece together the full list of consumers yourself.

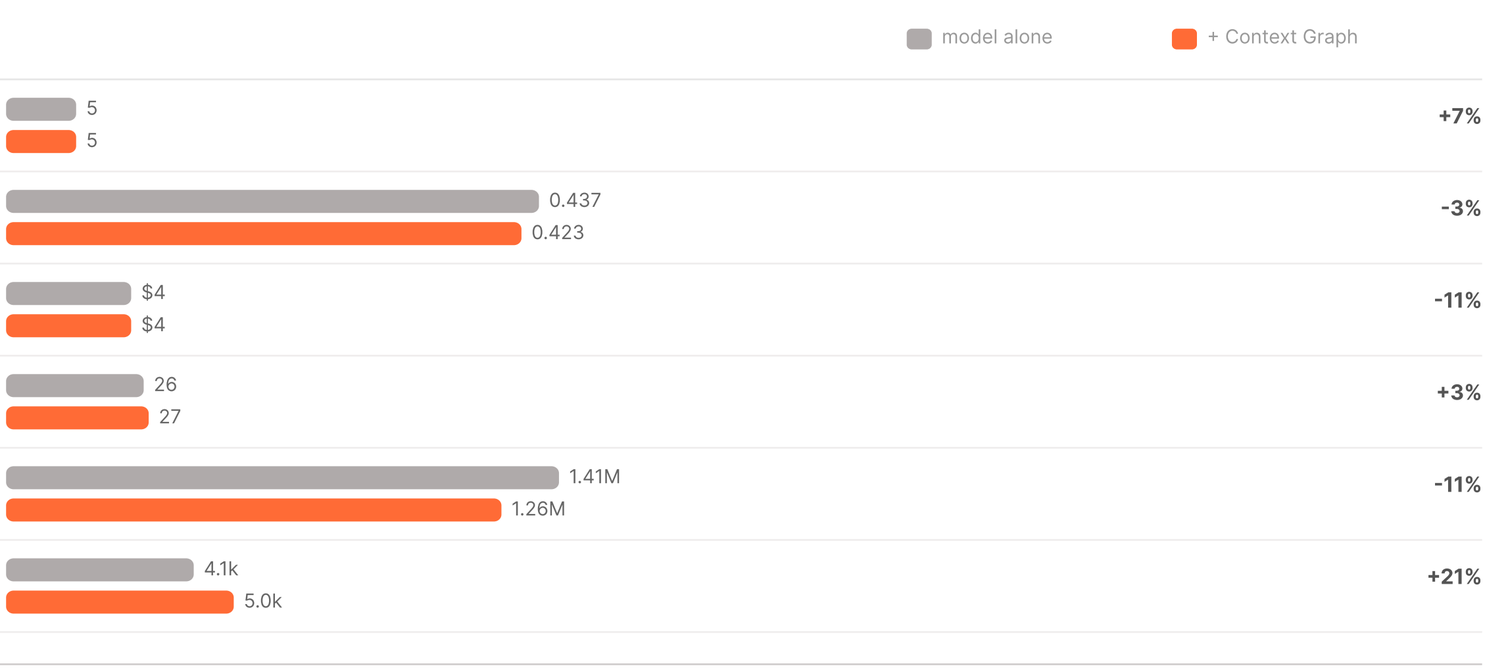
Which services consume from a message bus, and which topics do they read?

MESSAGE-BUS CONSUMERS

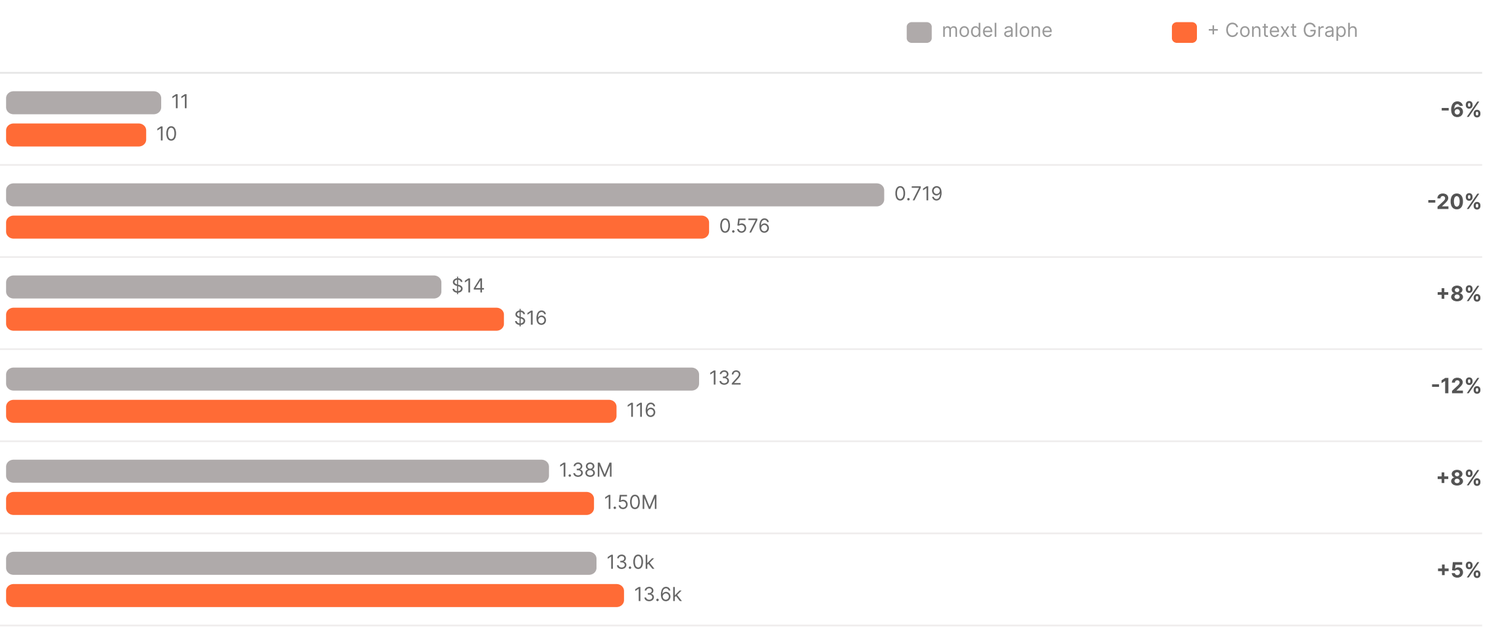
Opus 4.7



Sonnet 4.5



GPT-6 Astra



The results here split by model. Opus improved clearly, from 7 correct out of 16 to 9, with accuracy rising from 0.562 to 0.659. Sonnet was flat at 5 and slipped slightly on accuracy. Astra went the other way, from 11 correct to 10, and its accuracy fell from 0.719 to 0.576.

Finding a consumer by searching is hard because nothing declares it. A service subscribes to a topic in its own startup code, in whatever client library its language uses, and no shared file records that it happened. Building the answer key for this prompt took thirteen different library idioms across five languages.

The index does not model this the way you would expect, and that limits what the graph can do here. It holds no topic nodes at all, and no subscribe or publish relationships. What it does hold is links between services and 13 messaging systems treated as external services, so a service that reads from Kafka shows up as connected to Kafka rather than as subscribed to a topic. Asking broadly enough to reach that representation returned 186 results instead of 62, and it is why Opus improved. It is also why the gain stops there: the graph can tell you a service talks to Kafka, but not that it consumes rather than publishes, which is exactly the distinction this question asks for.

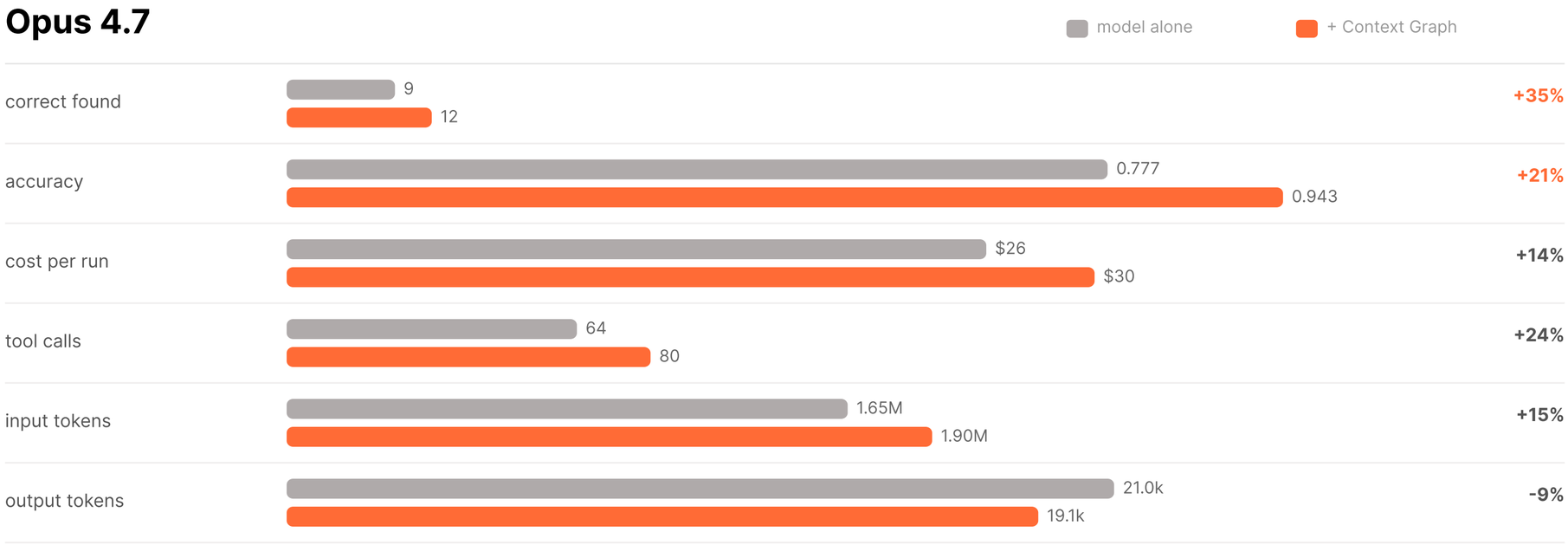
API callers

You are about to change an API and you need to know what else has to change with it.

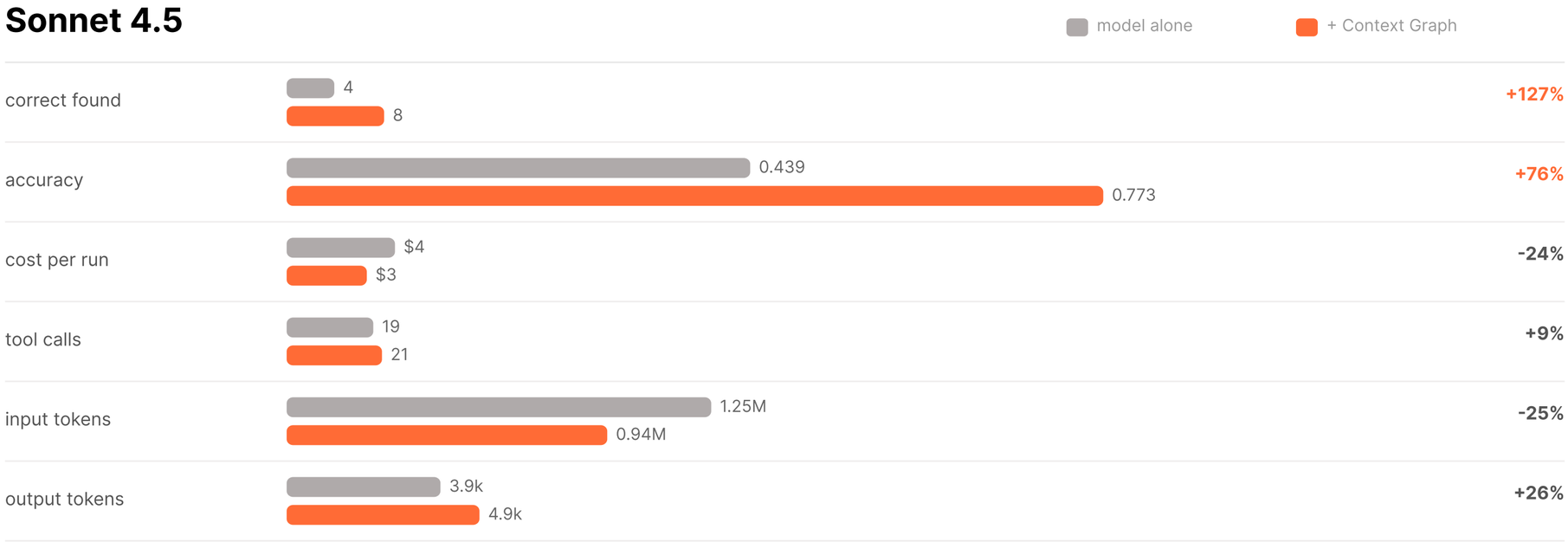
Which services call the platform's HTTP API?

API CALLERS

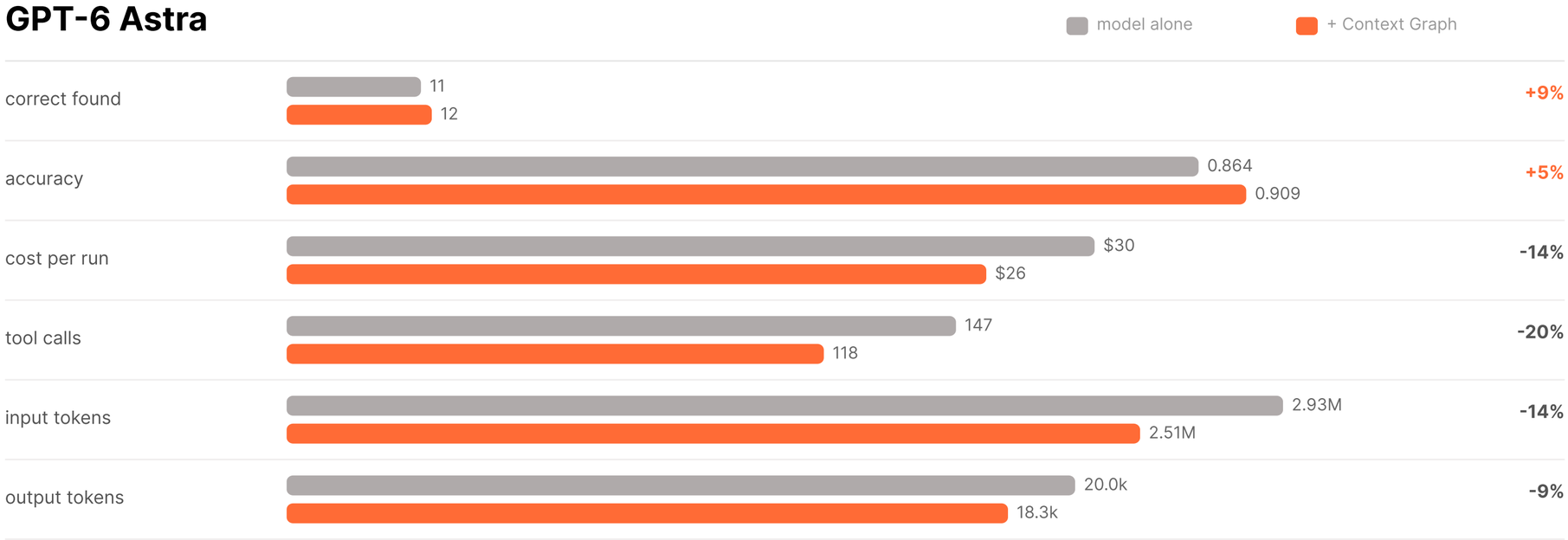
Opus 4.7



Sonnet 4.5



GPT-6 Astra



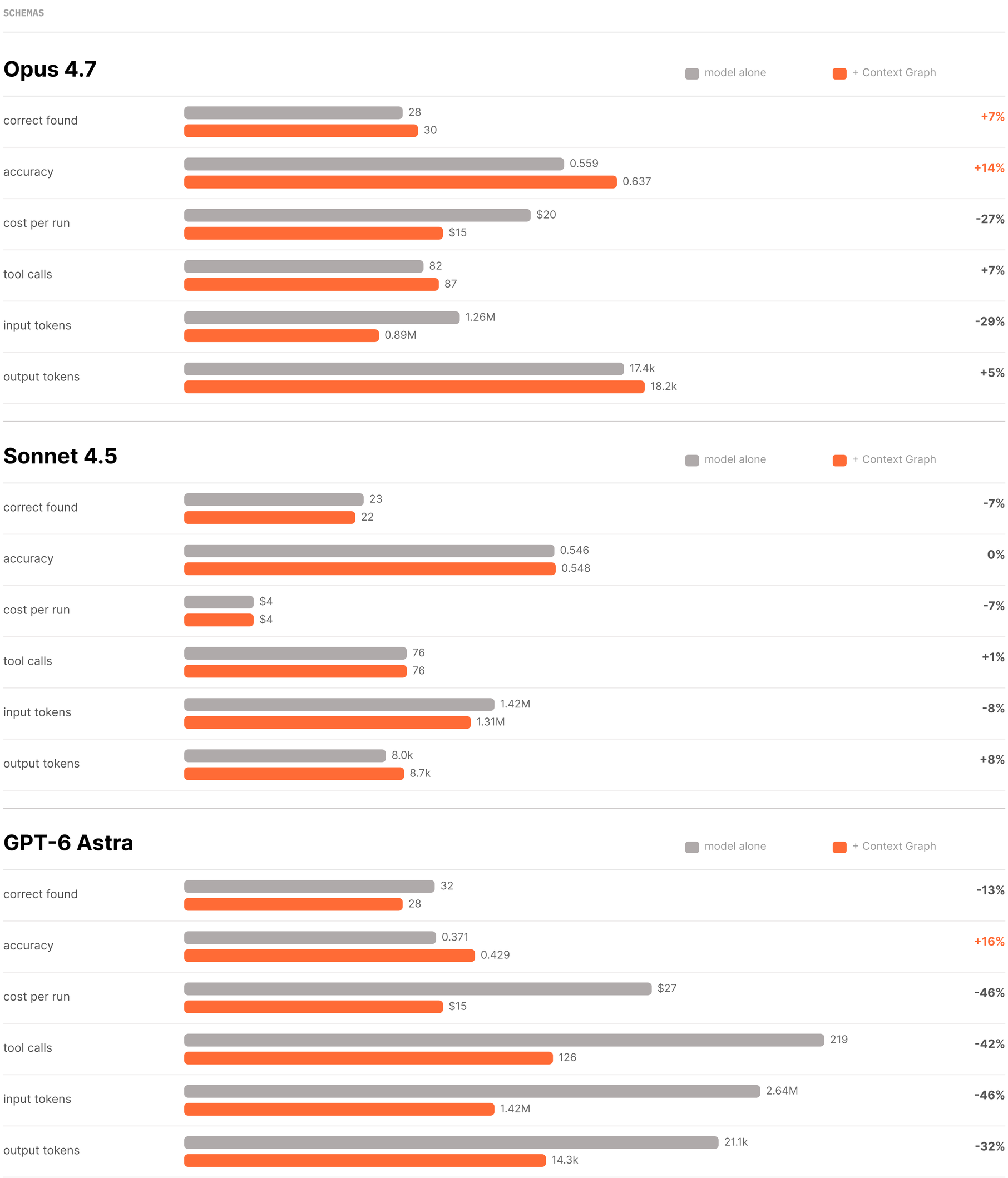
Every model improved with the Context Graph API to find the full list of 13 API callers.

Opus found 12 of the 13, compared with 9 on its own, and its accuracy rose from 0.777 to 0.943. Sonnet more than doubled its average from 3.7 callers to 8.3, while Astra improved from 10.7 to 11.7. This is exactly what matters when you are changing an API response and need to understand the blast radius before release. Without the graph, you might notify 54 teams and still miss 4 services that actually depend on the API.

Schemas

You are introducing contract testing, but first you need to understand what is already in place. Some services define formal API contracts, while others expose endpoints without documenting their expected requests and responses. Before you can decide what to test or where to start, you need a reliable list of the services that already have contracts you can test against.

Which services declare a machine-readable data or API contract, and what does each contract describe?



Code search can tell you that a schema file exists. The Context Graph can tell you which service owns it, what contract it represents, and whether several services inside the same repository each have their own. That is the difference between finding a file and understanding what it means.

Opus did not find many more correct services with the graph, but it returned far fewer wrong ones. Its precision rose from 0.530 to 0.642. Astra also found 11 services with no repository in the checkout, most of them directly from the graph results. There was no local file the model could have searched to find them.

That is what makes the graph so useful for contract testing. One repository might contain four services with four independently versioned API contracts. Repository search gives you one folder. The Context Graph gives you the four services you actually need to test. Without that detail, you may build one test suite for four different APIs and discover the gap only after a consumer breaks.

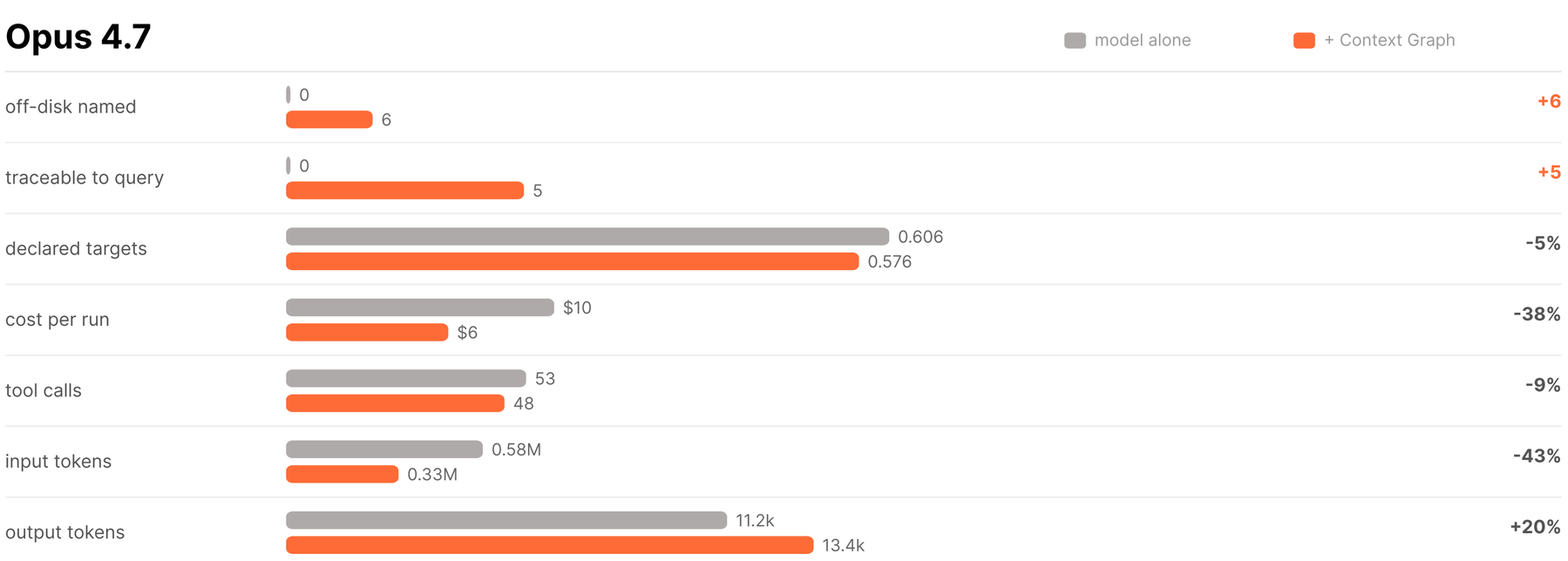
Upstream dependencies

You are preparing to change a shared service, database, queue, or external API. Before you touch it, you need to know which systems sit upstream of each service and keep it running. These runtime dependencies often do not appear as build dependencies, so searching imports and package files will not give you the full picture. To understand what each service relies on, you need to ask:

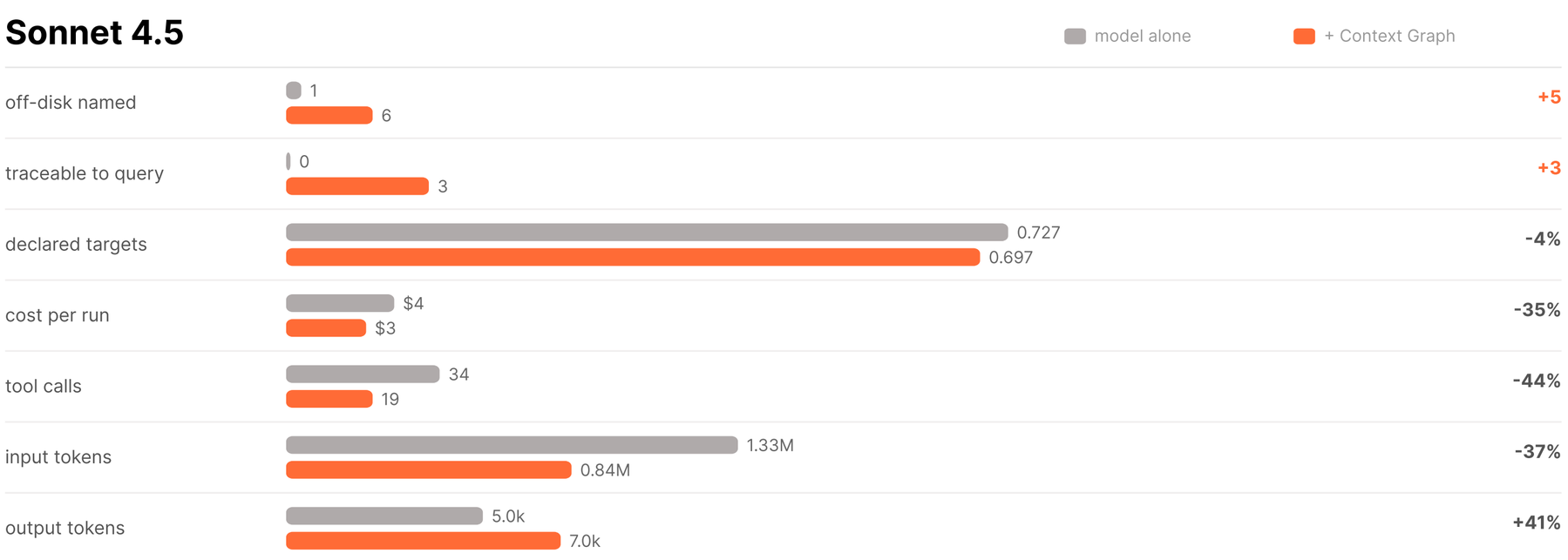
What does each service depend on at run time?

UPSTREAM DEPENDENCIES

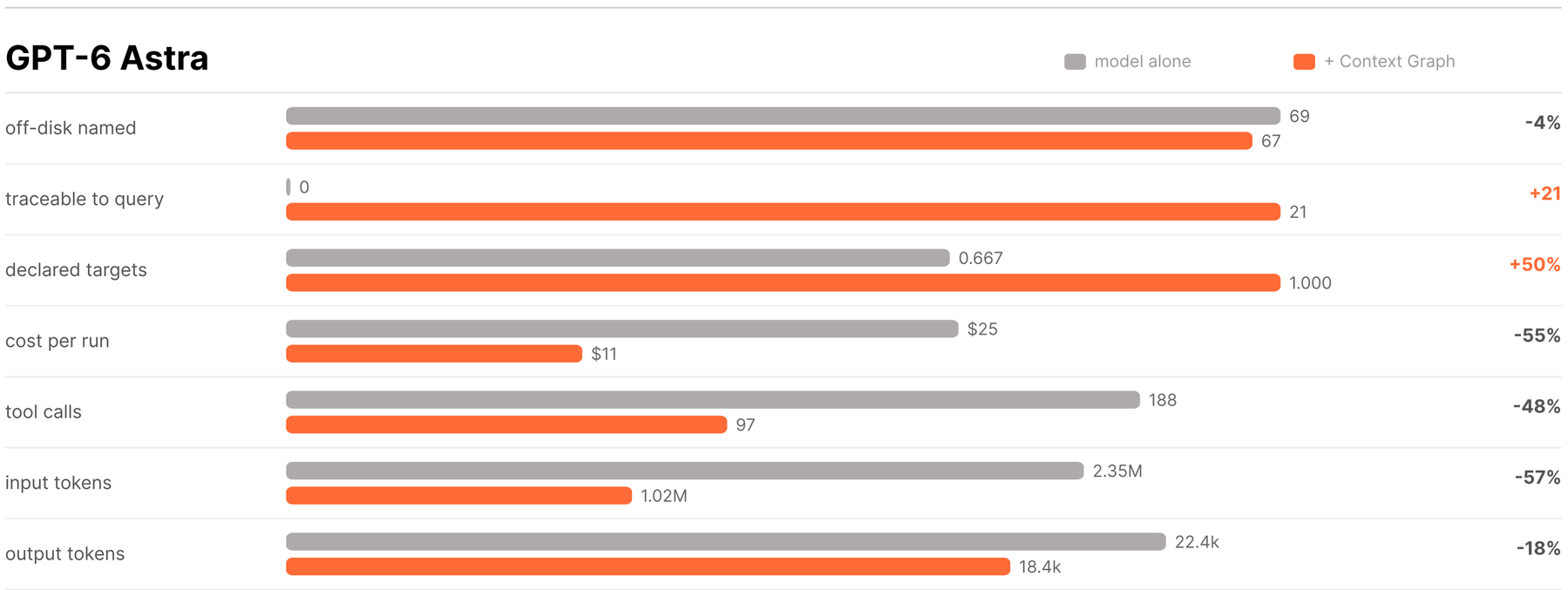
Opus 4.7



Sonnet 4.5



GPT-6 Astra



We scored this prompt differently because the code only contains 11 unique upstream dependencies, and one search can find all of them. If we graded the prompt on those 11 alone, every model would end up with roughly the same score and we would learn very little. That row is mainly there to catch an answer that is clearly wrong. Astra still stands out, though: with the graph, it found all 11 in every run, compared with about two-thirds on its own.

The more useful question is what the models found beyond the checkout. We track two things: how many off-disk dependencies the model named, and how many came directly from the graph query. That second number matters most. This is a public codebase, so a model might remember service names from its training data. A graph result gives us evidence that the dependency was actually retrieved.

Astra shows the difference clearly. On its own, it named 69 dependencies that were not in the checkout, but none could be verified. With the graph, it named 67, and 21 came directly from the query. The lists were almost the same size, but the graph-assisted list came with evidence. Opus and Sonnet went from finding nothing verifiable outside the checkout to finding 5 and 3 dependencies. All three models also became cheaper, with costs falling by 35% to 55%.

Some upstream dependencies can be found in the code. We found 245 declarations across Docker Compose files and Helm values, but they pointed to just 11 unique targets. The rest are configured at deployment time, often as URLs passed through environment variables. That means there may be no repository file connecting a service to the system it calls. When an unknown dependency goes down and takes several services with it, the problem is not that someone searched badly. The relationship was never in the code to begin with.

Where the Context Graph API earns its place

The Context Graph is most useful when the answer is a relationship, not something you can find by opening a file. Across three models and seven scored prompts, it improved accuracy in 18 of the 21 prompt-model pairs. The strongest results all followed the same pattern: the Context Graph API helped most when the evidence was spread across systems or was never written down in one place.

That showed up clearly in the runtime call question, where a service builds the destination URL at startup and no repository records the complete relationship. It also showed up in the HTTP endpoint question, where the Context Graph API found 48 services that did not have repositories in the checkout at all. Each service came directly from the Context Graph API's results, not from the model trying to remember a public project.

The same thing happened with API callers. The Context Graph API helped Opus find 12 of the 13 real callers, compared with 9 on its own, while returning 30 services instead of 54. On a question with a fixed answer, that tighter list matters. The model was not just finding more of the right services. It was guessing less. Message-bus consumers followed a similar pattern because the subscription might exist only in one service's startup code, with nothing elsewhere connecting that consumer back to the event.

For two of the three models, the Context Graph API also made these questions cheaper. Opus cost 27% less and Astra cost 20% less. When the Context Graph API can return the relationship directly, the model can skip much of the searching it would otherwise need to do.

The cases where the Context Graph API helped less were just as consistent. If the answer was already sitting in a file, the model was often better off reading that file without the context graph API.

So the takeaway is more specific than "the Context Graph API makes models better." It gives the model access to relationships that file search cannot reliably uncover. If the answer is already written in a repository, simply read the files. If the answer exists between repositories, services, and systems, that is where the Context Graph earns its place.