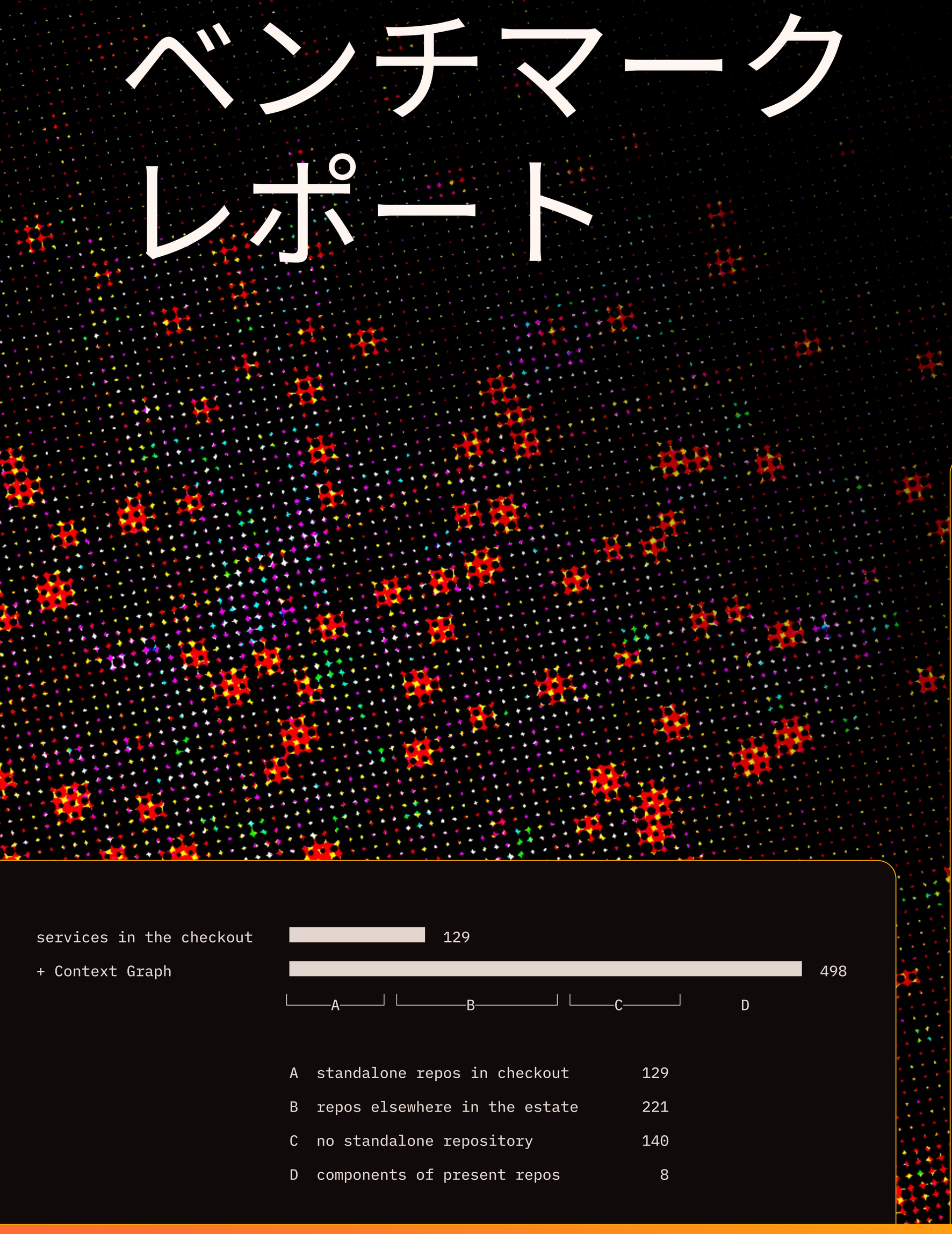
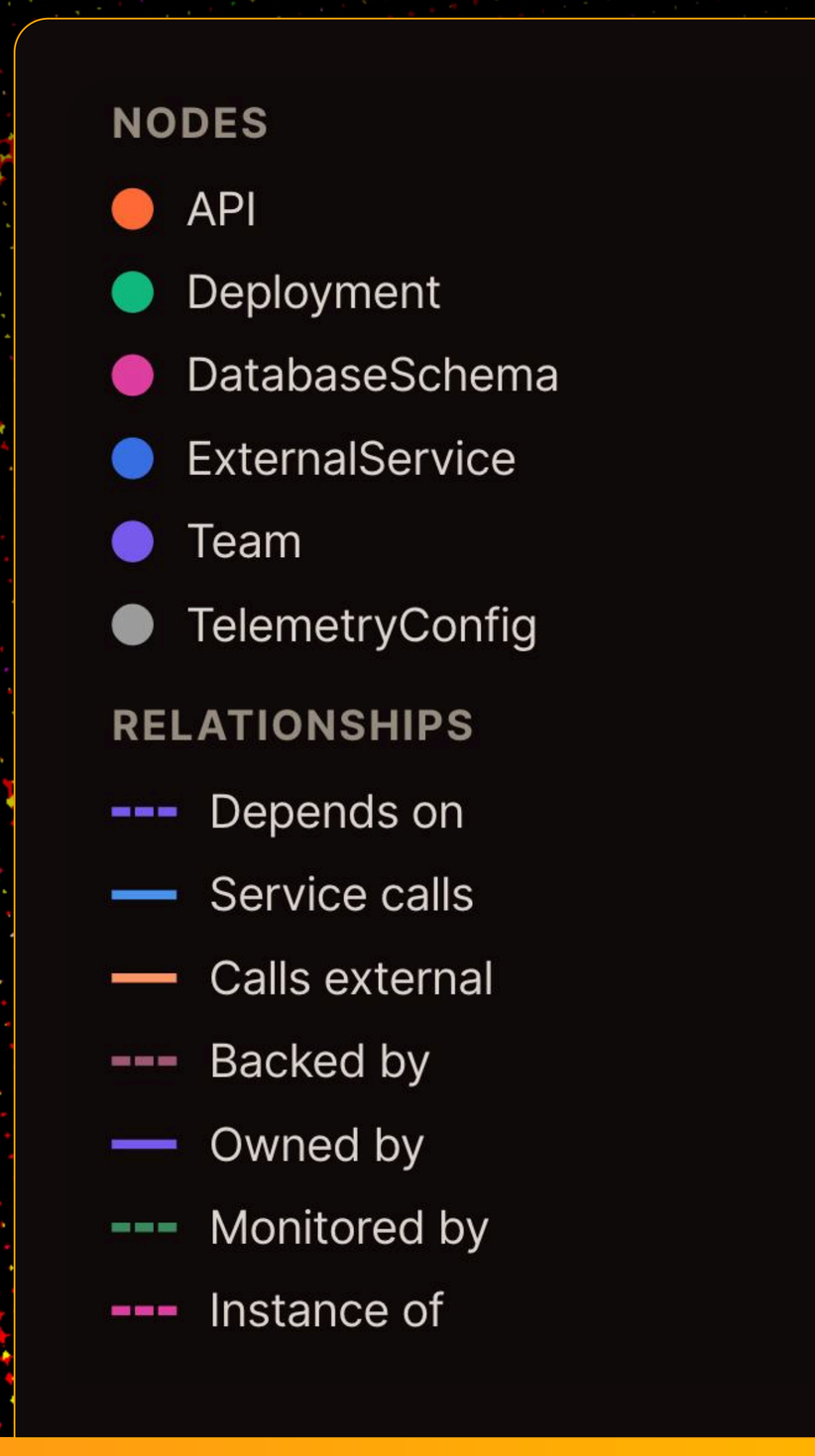
POSTMAN

Context Graph API ベンチマークレポート





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

目次

Context Graph API ベンチマークレポート 1

Context Graphはファイル検索では見つけられないものを発見する 2

AI評価ハーネス（検証機構）の仕組み 3

Context Graph APIの有無によるモデルのパフォーマンス比較 4

- ランタイム呼び出しの関係性 5
- テレメトリ（計測・監視情報） 6
- デプロイ済みサービス 8
- HTTP エンドポイント 10
- メッセージバスのコンシューマー（受信サービス） 12
- API呼出元（呼び出し側サービス） 14
- スキーマ 15
- アップストリーム（上流）依存関係 17

Context Graph APIが真価を発揮する領域 19

Context Graph API
ベンチマーク レポート

私たちは、AIエージェントがすでに大規模なコードベースを検索している状況において、Context Graph APIが実際にどのような価値を付加するのかを理解したいと考えました。そこで、468個のリポジトリが存在する環境で、Context Graph APIありとなしの両方の条件でフロンティアモデルのテストを実施しました。

その違いを理解する最もシンプルな方法は「図書館」に例えることです。

図書館に入り、司書にチェスに関するあらゆる資料を探してほしいと頼んだと想像してみてください。コンピュータの蔵書検索カタログがない場合、調べる唯一の方法は足を使うことです。司書は館内を歩き回り、すべての本棚を探し、一冊一冊の本を点検することになります。熟練した司書であれば、より早く探し出し、多くのことに気づくかもしれませんが、それでも建物全体をゼロから探していることに変わりはありません。

では、その司書に図書館のコンピュータカタログへのアクセス権を与えてみましょう。最初の本棚から始めて通路ごとに一列ずつ探していく代わりに、一度検索するだけで、関連する本の一覧とその正確な棚の位置を手に入れることができます。カタログは、どこから探し始めるべきか、本棚からどの本を抜き出すべきか、どのセクションを詳しく調べるべきかを教えてくれます。

また、無関係なタイトルの本の中にある特定の章や、別の場所に保管されている本、あるいはオーディオブックのように物理的な棚にないリソースを指し示してくれることもあります。司書は、手元にある利用可能な本を開き、要求に必要な内容が含まれているかを確認するだけで済みます。

この比喩において、**司書は「モデル」、本棚は「チェックアウトされたリポジトリ」**、そしてカタログが「Context Graph API」にあたります。Context Graph APIは、どのサービス、API、エンドポイント、関係性が重要になりそうか、そして検索の焦点をどこに絞るべきかをモデルに教えます。何百ものリポジトリを一つずつ探索する代わりに、モデルは特定の候補セットから探索を開始し、コード内で可能な限りそれらの手がかりを検証することができます。

カタログはまた、本が追加・削除・移動されるのに合わせて毎晩更新されます。同様に、Context Graphも接続されたデータから毎夜更新されるため、新しいサービス、変更されたエンドポイント、更新された関係性を反映し続けます。

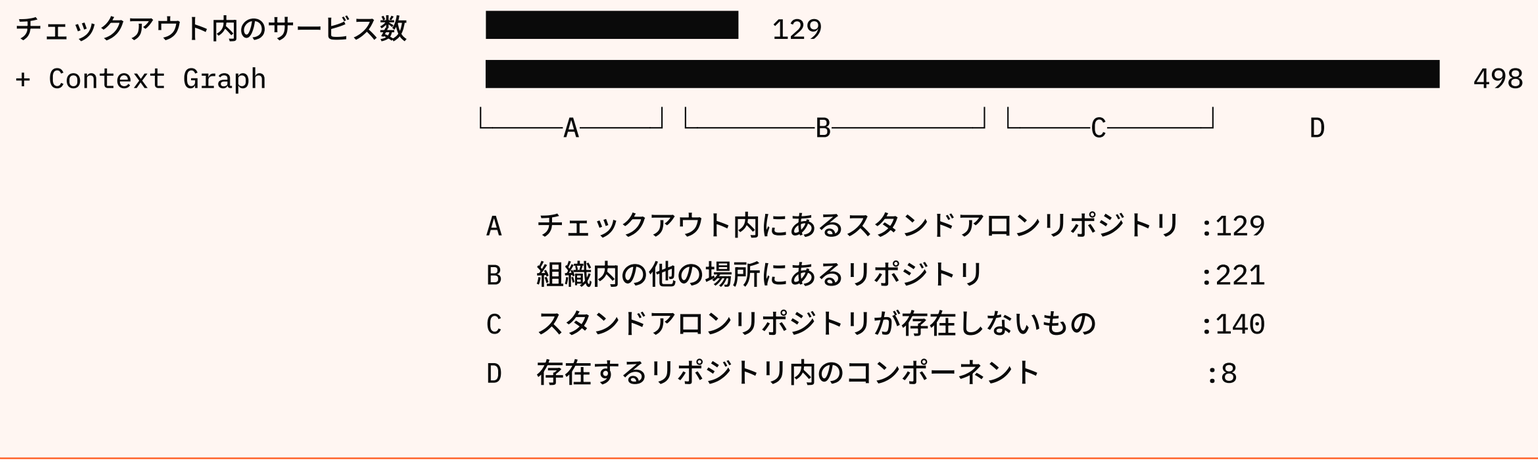


より優れたモデルとは、いわば「より作業が早く、経験豊富な司書」のようなものです。本棚をより効率的に検索することはできますが、それでも**その本棚の上に存在しないものを見つけることはできません**。

Context Graphはファイル検索では見つけられないものを発見する

システム全体を網羅するこのカタログが、チェックアウトされたコードを超えて何を可視化するのかを確認するため、私たちはエージェントにシンプルな質問を一つ投げかけました。

どのサービスがHTTPエンドポイントを公開しており、それぞれどのようなパスを提供していますか？



Context Graphが返した498個のサービスのうち、チェックアウトされたリポジトリ内にコードが存在していたのは129個でした。

再び図書館の例えを思い出してください。ファイル検索で到達できたのは、司書の目の前の本棚に置かれていた129冊だけでした。一方、Context Graphはカタログを開き、さらに369個のエントリ（項目）を指し示したのです。

これらの追加されたサービスは、主に3つの場所から見つかりました。

221個は、より広大なエコシステム内の他の場所に存在するリポジトリです。 Context Graphは接続されたすべてのデータにわたってインデックスを作成するため、このベンチマークが実行されたリポジトリ群よりも広い範囲に到達します。コード自体は存在していますが、単にエコシステムの別の場所にあるだけです。これらは図書館の別の分館に保管されている本のようなものであり、ローカルの本棚をいくら探しても見つかることはありません。

140個は、独立した単体のリポジトリを持たないものです。 一部は実行時に呼び出される外部システムであり、その他は接続データを通じて認識されているものの、自身のリポジトリにマッピングされていないサービスです。これらはカタログ上には表示されますが、エージェントが手に取って開くための独立した本が棚に存在するわけではありません。

8個は、すでにチェックアウトされたコード内にあるリポジトリ内部のコンポーネントです。 コード自体は存在していますが、これらは大きな本の中の「章」のような存在です。ファイル検索はリポジトリ全体を検知して1つのサービスとして名前を挙げますが、Context Graphはその内部にあるコンポーネントを独立した別々のサービスとして識別します。これらを検索によって見つけ出すには、リポジトリ全体がどのように構成されているかを解読する必要があり、コード内にはそれ自体が単独のサービスであることを示すマークは存在しません。

これらはマニアックなコーナーケース（例外的な事例）などではありません。ゲートウェイ、会計、広告配信、予約管理など、実際のリクエストパスの中核にある極めて一般的なサービスが含まれていました。これらは稼働しており、すでに何らかのシステムから呼び出されているにもかかわらず、チェックアウトされたコード内にはその存在を示す情報が一切含まれていないのです。もしこれらのエンドポイントのいずれかが変更された場合、障害対応の呼び出し（ページング）を受けて初めてそれを知る、という事態は避けるべきです。

これは検索クオリティの問題ではありません。「アクセス権」の問題です。エージェントの検索方法が悪かったのではなく、それらのサービスを特定するためのファイルがそもそも検索対象の中に存在しなかったのです。どれほど検索機能を向上させても、そのギャップを埋めることはできません。最初からチェックアウトされていないものを grep することは不可能です。徒歩で本を探す司書が、棚に置かれていない本を見つけることが絶対にできないのと同様です。

AIハーネスの仕組み

各モデルは、APIエコシステムのあらゆる領域（サービス、エンドポイント、スキーマ、イベントコンシューマー、デプロイメント、アップストリーム依存関係、テレメトリ、ランタイム呼び出し関係、および変更されるAPIの呼び出し元）を網羅する8つの質問に回答します。

プロンプトには質問文のみが含まれています。モデルに対して、どこを探すべきか、どう検索すべきか、あるいは答えをいくつ見つけるべきかといった指示は一切与えられません。これらはまさに、Context Graphが支援することを目的としている要素そのものだからです。ベースライン（比較対象）となるモデルに同様のヒントを無償で与えてしまうと、測定しようとしている差分が縮小してしまいます。そのため、プロンプトが検索方法や期待される回答数を少しでも匂わせた場合、ハーネスはビルドを失敗させる仕組みになっています。

正解の判定はコードが行う

すべての回答は3つの方法で採点されます。2つの採点器（グレーダー）は決定論的なコードであり、1つは言語モデルです。

1つ目の採点器は、チェックアウトされたコードから直接作成された正解集（解答キー）と回答を比較します。その正解集に含まれるすべての期待値は、grep によって個別に検証されています。

2つ目の採点器は、引用されたすべてのファイルパスを、固定されたコミット（pinned commit）時点のリポジトリと照合します。モデルがパスを捏造したり、そのバージョンのコードに存在しないファイルを引用したりした場合、この採点器がそれを検出します。

3つ目の採点器は、ルーブリック（評価基準）に従う言語モデルです。他の2つの採点器とは異なりリポジトリ自体を検証できないため、この採点器の評価重みは最も低く設定されています。

また、モデルが構造化された結果ブロック内に最終確定させた回答のみを採点対象とします。周辺の説明文はカウントされません。そうしないと、モデルがリポジトリに言及し、それを調査した上で除外した場合でも、名前を挙げたというだけで評価されてしまう可能性があるためです。それでは、より良い回答を出力したモデルではなく、より多くの文章を生成した側のグループを優遇することになってしまいます。

最も重要な点として、正解集は常にコードから作成され、Context Graphから作成されることはありません。Context Graph APIを自身の出力内容に基づいて採点することは、自分の宿題を自分で採点するようなものです。それでは結果を独立して検証したり再現したりすることも不可能になってしまいます。

変更する変数は1つだけ

両方のグループ（Context Graphあり／なし）が、同じ質問、同じツール、そして同じ固定コミットのコードを受け取ります。実行時の Temperature（ランダム性の設定）も同じです。唯一の違いは、一方のグループは Context Graph へのクエリ実行から開始し、ベースライン側はそれなしで開始する点です。ベースライン側には、Context Graph が存在することすら伝えられません。

各質問は、グループごとに3回ずつ実行されます。これは、1回のモデルの応答だけでは正しい測定にならないためです。それは単なる分布の1サンプルに過ぎません。同じ質問を2回投げかけても、結果が大きく異なることがあります。言語モデルは本質的に変動しやすいものであり、同じプロンプト、同じコード、同じ設定であっても、異なる検索経路や異なる回答を生み出す可能性があります。そのため、1回限りの実行結果は能力というよりも運の要素を大きく反映してしまうのです。

ハーネスは、質問ごと・反復ごとに2つのグループを比較します。これにより、一部の質問が本質的に他より難しいといった影響を抑え、Context Graph の純粋な効果を抽出することができます。

有意差検定（Significance test）と信頼区間（Confidence interval）の両方が結果を裏付けている場合にのみ、その結果を「勝利（Win）」とみなします。対となるデータが不足している場合、ハーネスは無理に結論を導き出さず、データ不足として報告します。

判定モデルは、どの回答が Context Graph API を使用したかを知らない

モデルが採点者として使用される場合、自身が参加している実行セッションを判定することはありません。また、モデルは自分自身の出力と似た回答を好む傾向があるため、別のモデルファミリーの判定者を使用することを優先しています。

同じ質問に対するすべての回答は、匿名化されシャッフルされた状態でまとめて判定者に送信されます。判定者は、どの回答がベースラインから得られたもので、どれが Context Graph のデータを使用したものかを区別できません。実験条件ではなく、回答そのものを評価せざるを得ない仕組みになっています。

判定呼び出しが失敗した場合、そのスコアは破棄されます。故障した採点器を正常に見せかけてしまうことを防ぐため、中立的な値に置き換える処理は行いません。

Context Graph APIの有無によるモデルのパフォーマンス比較

私たちは8つのプロンプトを Claude Opus 4.7、Claude Sonnet 4.5、および GPT-6 Astra で実行しました。各モデルにおいて「モデル単体」と「モデル ＋ Context Graphへの1回のクエリ」の2つの条件（アーム）を設定し、それぞれ2通りの条件でテストを行いました。すべての組み合わせにおいて各3回ずつ実行し、両条件合わせて計144個のセル（検証データ）を記録しました。

Context Graph のデータ取り込み（インジェクション）が完了した時点で、グラフ内には 4,828 個のエンドポイント、1,424 個の API およびサービス、309 個の外部システム、139 個のデータベーススキーマ、11 個のテレメトリ設定を含む、計 6,712 個のノードが保持されていました。

以下の各セクションでは、1つのプロンプトを取り上げています。その質問が意図することと重要性、各条件のモデルが受け取った実際の質問文、そしてモデルごとのスコアカード（評価表）を掲載しています。

各スコアカードには6つの項目が記載されています。そのうち2項目は「結果（成果）」を、4項目は「そこに到達するまでにかかったコスト」を示しています。「正解サービスの発見数（Correct services found）」は、モデルが出力したリストの長さではなく、解答キー（正解集）に含まれる項目を実際にいくつ特定できたかを表しています。リストを長くすることは容易ですが、正しく答えることは容易ではないためです。「精度（Accuracy）」は、その回答を解答キーと照合して採点したスコアです。

Context Graph API がスコアを向上させた理由は、単にチェックアウト外のサービスを発見したからだけではありません。すでに手元に存在するサービスをより多く見つける手助けをしたからです。例えば「ランタイム呼び出し」において、Opus は単体では判明しているサービスの 11.1% しか発見できませんでしたが、Context Graph API を併用することで 36.5% を発見しました。また、判明している HTTP エンドポイントや API 呼出元についてもより多く特定できました。双方の条件ともアクセスできたコードは全く同じですが、Context Graph API によってその内部に存在する要素をより多く発見できたのです。

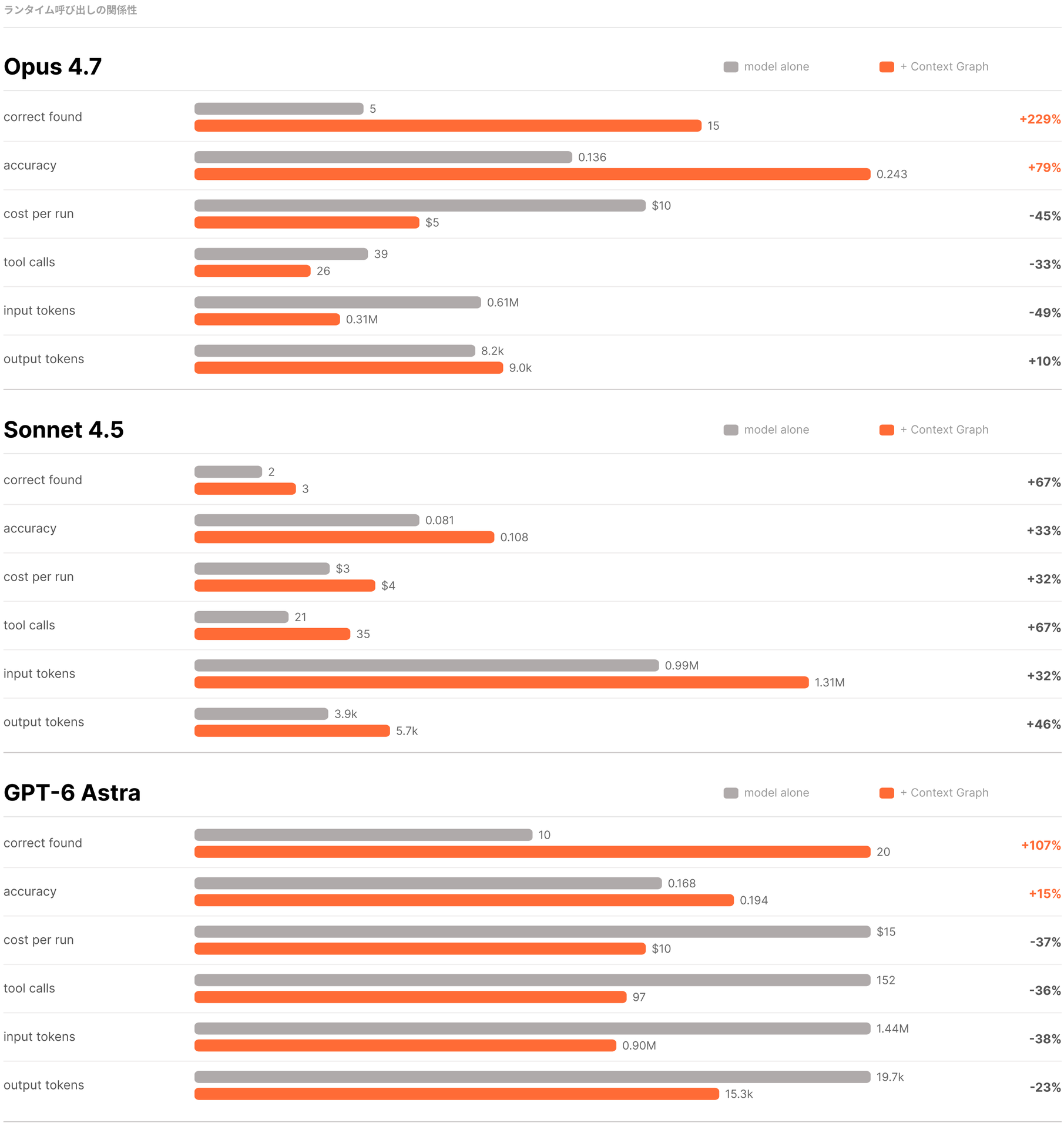
つまり、Context Graph API は2つの異なる方法で貢献しています。第一に、**既存のコード内にある関係性をより多く発見する手助けをすること**です。この改善は精度スコア（Accuracy）に表れます。第二に、**チェックアウトされたコード内にはまったく表現されていないサービスや関係性を表面化させること**です。後者の価値は従来のスコアには表れないため、解説の中で個別に取り上げて説明しています。

私たちは、エンジニアがコードを更新する前に答える必要のある8つの質問（ランタイム呼び出し関係、テレメトリ、デプロイ済みサービス、HTTPエンドポイント、メッセージバスのコンシューマー、API呼出元、スキーマ、アップストリーム依存関係）を検証しました。最初の7つは解答キーと照合して採点され、8つ目の質問は、そのセクションで説明する固有の理由により「カバー率（網羅性）」に基づいて採点されています。

ランタイム呼び出しの関係性

複数のサービスが同時に遅延し始める障害（インシデント）に対応しているとします。最初はそれぞれ別の問題のように見えますが、実はすべてのサービスが実行時に同じサービスに依存しています。ランタイムでの呼び出しは必ずしもビルド時の依存関係として現れないため、その繋がりはコード上からは一目でわかりません。本当の問題を診断する前に、それらがどのように接続されているかを理解するためだけに、複数のサービスにわたってリクエストを追跡しなければなりません。この失われたコンテキスト（背景情報）のせいで、本来なら迅速に診断できたはずの障害対応が長期化してしまうのです。

サービス間のランタイム依存関係はどのようになっていますか？



3つのモデルすべてで、Context Graph による大幅な改善が見られました。
その理由は、この関係性がコードとして書き残されていないためです。あるサービスが実行時に別のサービスを呼び出す際、どちらのリポジトリにもその記録は残りません。呼び出し先は起動時に設定情報から組み立てられるURLであるため、grep で検索できる文字列が存在しないのです。Context Graph API はこの呼び出しをテキストではなく「エッジ（接続関係）」として記録するため、十分な情報が返される場合には非常に大きな効果を発揮します。実務において、これは深刻なインシデントの多くで背景にある問ひと言えます。

タイムアウト、レスポンスの形式、リトライポリシーなどを変更した際、聞いたこともないシステムが突然ダウンし、自分のレビューではなく他チームのアラートチャンネルを見て初めてそれを知る、といった事態が発生するのです。

テレメトリ（計測・監視情報）

障害（インシデント）が発生し、複数のサービスが原因である可能性があるとします。デバッグを開始する前に、何が起きたかを示すログ、メトリクス、トレースをどのサービスが収集しているかを知る必要があります。サービスにインストゥルメンテーション（計測処理）が施されていない場合、障害発生時にそのサービスが何をしていたかを正確に把握する手段がありません。その結果、リクエストを追跡して根本原因を特定するのではなく、症状から逆算して当て推量でデバッグを行う羽目になります。

> **各サービスと、それが生成するログ、メトリクス、トレースを一覧で示してください。**

TELEMETRY

Opus 4.7

model alone　+ Context Graph

	model alone	+ Context Graph	
correct found	40	54	+35%
accuracy	0.430	0.500	+16%
cost per run	$19	$15	-22%
tool calls	93	70	-25%
input tokens	1.17M	0.90M	-23%
output tokens	18.8k	18.7k	-1%

Sonnet 4.5

model alone　+ Context Graph

	model alone	+ Context Graph	
correct found	12	20	+65%
accuracy	0.175	0.266	+52%
cost per run	$2	$3	+35%
tool calls	43	32	-26%
input tokens	0.77M	1.05M	+35%
output tokens	4.3k	5.2k	+21%

GPT-6 Astra

model alone　+ Context Graph

	model alone	+ Context Graph	
correct found	65	82	+27%
accuracy	0.533	0.588	+10%
cost per run	$17	$15	-15%
tool calls	162	119	-26%
input tokens	1.63M	1.38M	-15%
output tokens	21.5k	19.2k	-11%

3つのモデルすべてで改善が見られ、このプロンプトは本レポートの中で「インデックス（データ構造）と同じくらいクエリ（質問の形式）が重要である」ことを最も明確に示した例となりました。Opusは正解数が128個中40個から54個へ増え、精度は0.430から0.500に向上し、1実行あたりのコストも19ドルから15ドルに下がりました。Astraは正解数が65個から82個に増加しました。Sonnetは相対的な伸び率が最も大きく、正解数が12個から20個に増加し、精度は0.175から0.266に向上しました。

この質問に対する私たちの最初の試みでは、3回の実行すべてで11行しか結果が返されず、私たちはもう少しでそれを「インデックスに関する検証結果」として公開するところでした。しかし実際には、それは「私たちのクエリに関する問題」だったのです。グラフ内には1,424個のAPIやサービスに対してわずか11個の TelemetryConfig ノードしか存在せず、「どのサービスが監視・可視化のインストゥルメンテーションを持っているか」と尋ねると、グラフ全体で最も希少なノードタイプにヒットしてしまっていたのです。代わりに関係性として「どのサービスがログ、メトリクス、トレース、プロファイルを送信し、それらをどこへ送っているか」と尋ねたところ、同じエンドポイントから200行以上の結果が返ってきました。私たちはその言い回しでグラフ使用グループを再実行しました。上記の数値はその結果であり、クエリを直接見ることがないモデル単体グループは影響を受けていません。

より優れたクエリによって何が得られたかは、解答キーでは採点できない部分に表れています。古い言い回しでは、グラフはファイルシステム外のサービスを一切提供できませんでした。しかし新しいクエリでは、Opusはチェックアウトされたリポジトリが存在しない24個の計測対象サービスを特定し、Astraは21個を特定しました。そのほぼすべてが、公開情報からの記憶ではなく自身のクエリ結果まで追跡可能なものでした。Sonnetは例外で、古いクエリの0.383から新しいクエリの0.266へと精度が低下しました。219行の情報は11行の情報よりも役に立たなかったことになり、これは小規模なモデルと広範なクエリを組み合わせる際に知っておくべき貴重な知見です。

この質問の実用的な側面は、インシデントの初期段階で投げかける「これらのサービスのうち、実際に何を行っていたかを教えてくれるものはどれか？」という疑問そのものです。インストゥルメンテーションのないサービスは最悪のタイミングで発覚する死角であり、チェックアウト内にコードが存在しないサービスは、探すことすら思いつかない死角なのです。

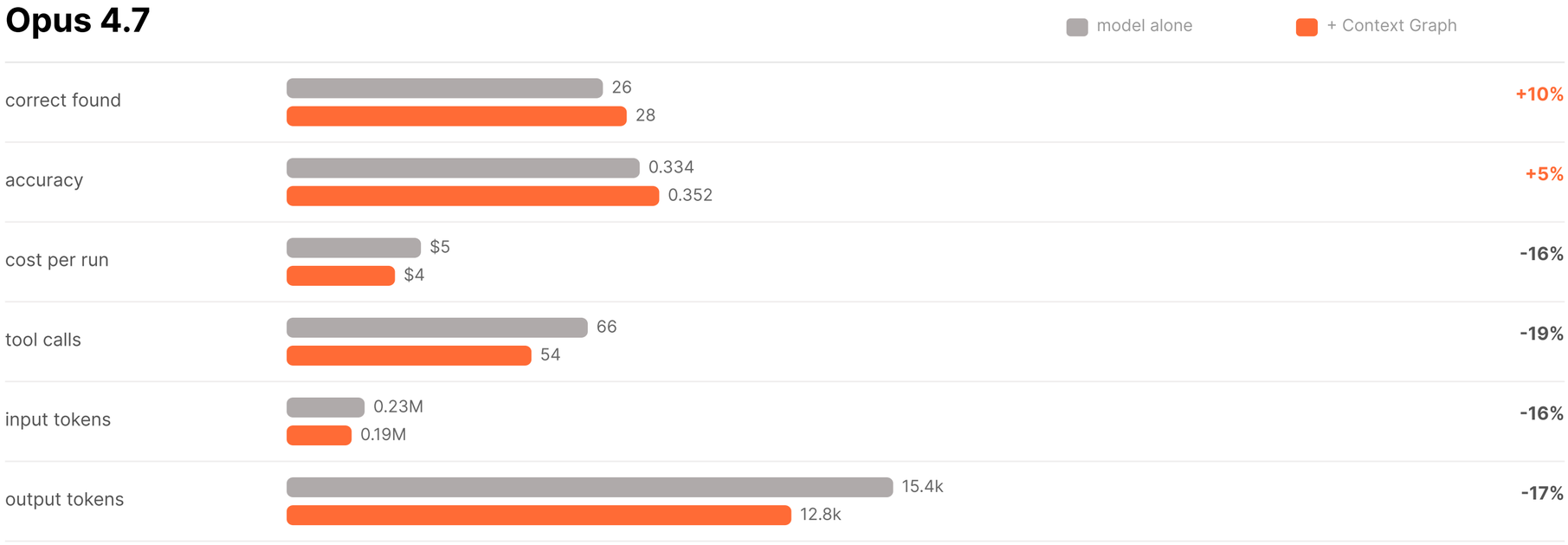
デプロイ済みサービス

本番環境で稼働しているすべてのサービスのコスト監査を行っているとします。まずはリポジトリの一覧から確認を始めますが、その大部分は単体では動作しないライブラリ、共有パッケージ、社内ツールなどです。インフラ支出を個々のサービスに結び付ける前に、どのリポジトリが実際にデプロイされているサービスを表しているのかを把握する必要があります。

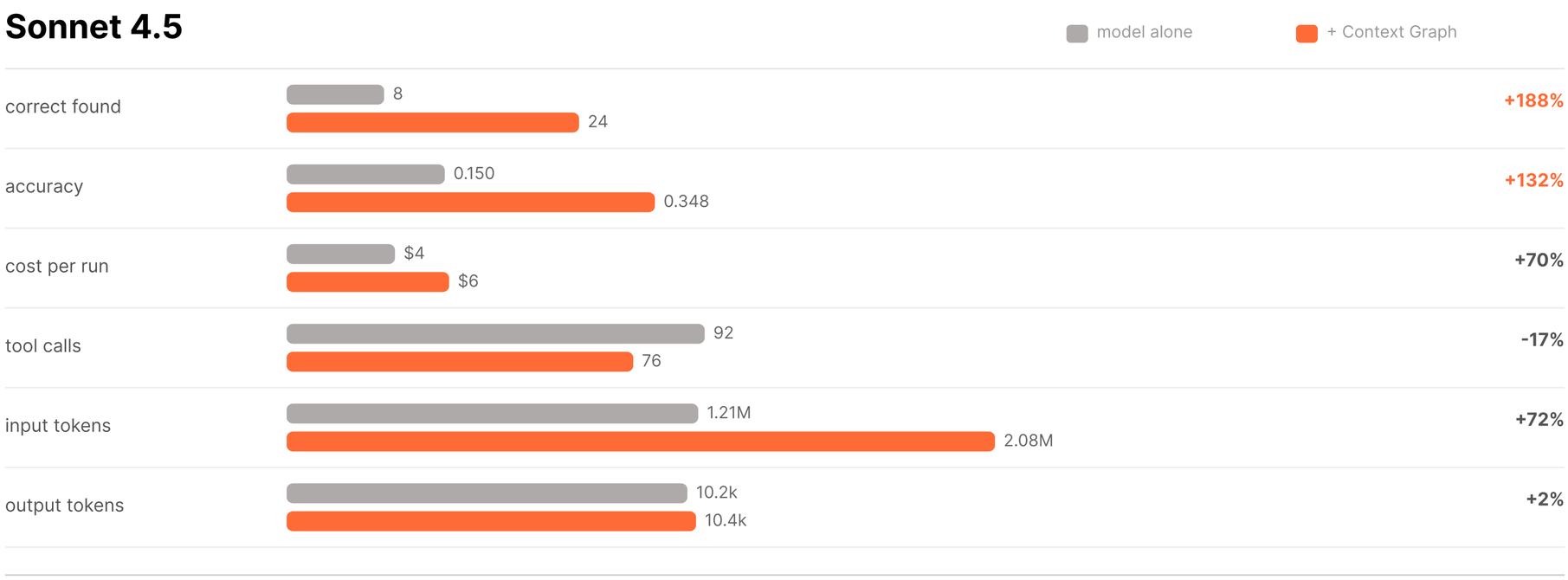
実際にデプロイされているサービスはどれで、それぞれどのようにデプロイされていますか？

DEPLOYED SERVICES

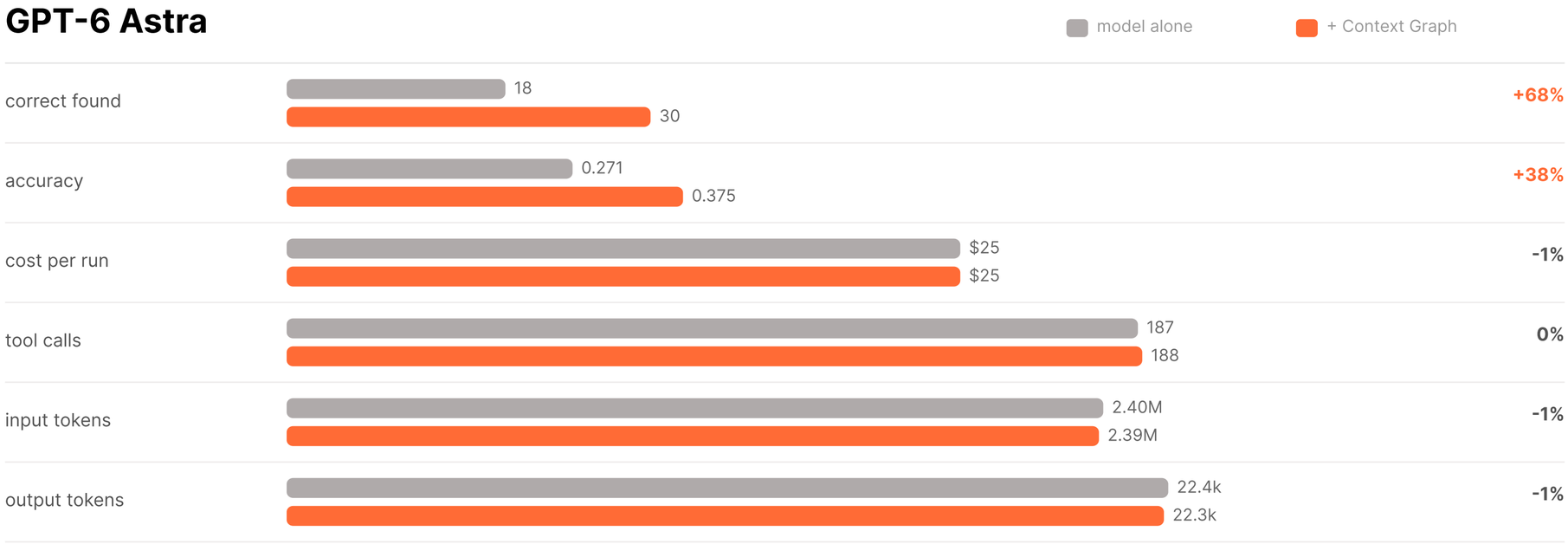
Opus 4.7



Sonnet 4.5



GPT-6 Astra



すべてのモデルにおいて、グラフを使用した方がパフォーマンスが向上した。

難しいのは、各リポジトリがデプロイ方法をそれぞれ異なる形で記述している点です。あるリポジトリは Kubernetes マニフェストを使用し、別のリポジトリは Helm チャート、さらに別のものは Terraform や Docker Compose を使用しているかもしれません。コードを検索するモデルは、これらすべてのフォーマットを認識した上で、サービスが「実際にデプロイされているのか」、それとも「単にデプロイに必要なファイルが存在するだけなのか」を判別する必要があります。一方、Context Graph はデプロイ情報を「直接的な関係性（リレーションシップ）」として保持しています。

また、グラフはコード検索では見つけられなかったサービスの発見にも寄与しました。Opus はチェックアウト内にリポジトリが存在しないデプロイ済みサービスを 10 個特定し、そのうち 5 個はグラフの検索結果から直接得られたものでした。これらは、モデルが調査できるローカルフォルダが存在しなかったにもかかわらず、環境内で実際に稼働していた本物のサービスでした。

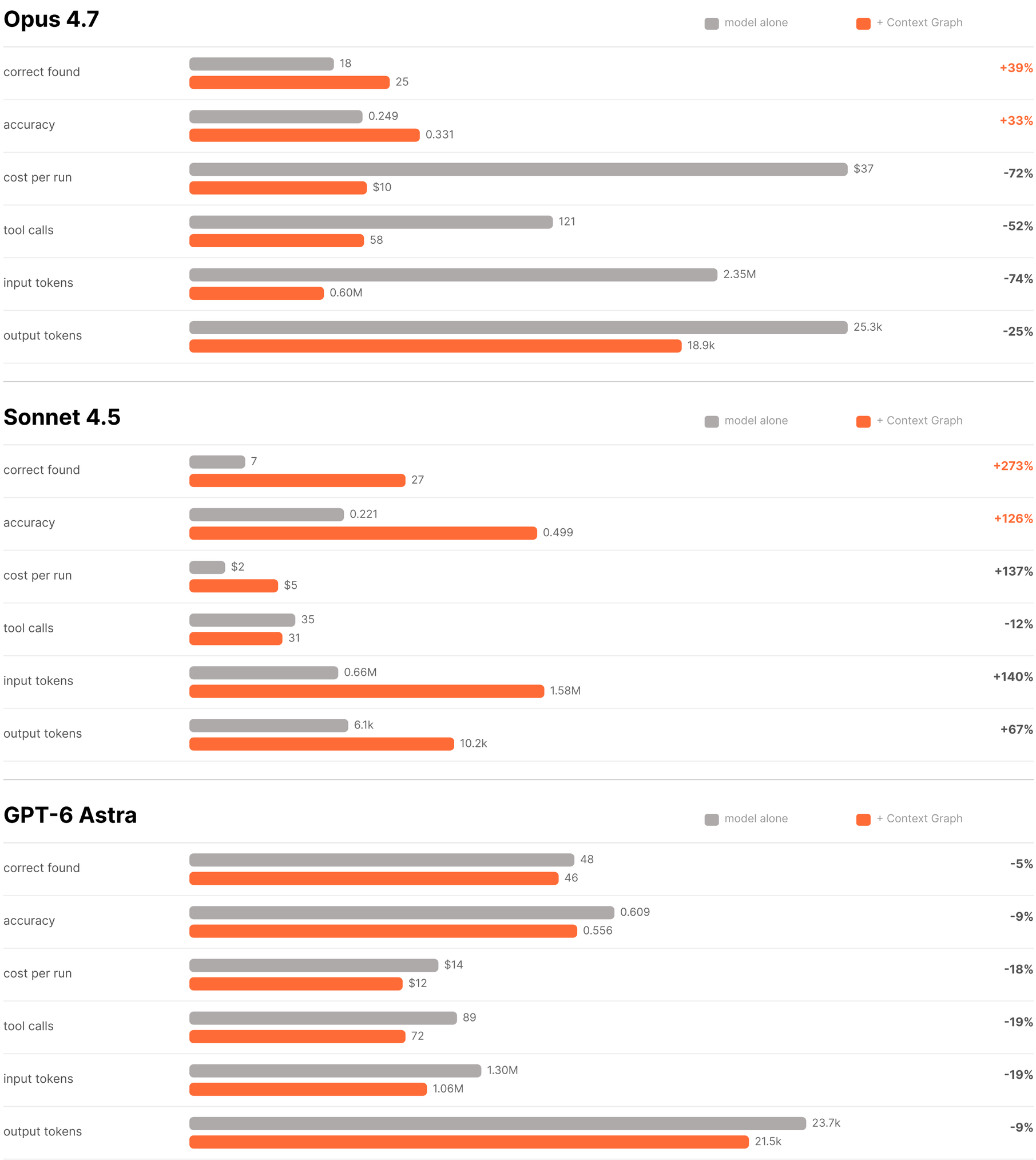
これこそが、財務部門から各サービスの運用コストを尋ねられた際に必要となるリストです。また、クラスタの移行、ベースイメージの変更、あるいはシークレットのローテーションを行う前に手元に置くべきリストでもあります。単なるリポジトリ一覧には、単体では動作しないライブラリや社内ツールが含まれる一方で、障害を発生させ得る実際のデプロイ済みサービスを見落としてしまう可能性があるのです。

HTTP エンドポイント

すべてのAPIエンドポイントに対して認証を必須化するタスクを割り当てられたとします。これを安全にロールアウトする前に、どのエンドポイントが存在し、誰がそれを呼び出しているのかを正確に把握する必要があります。呼び出し元を見落とすと、認証を有効にした瞬間に正当なトラフィックを遮断してしまう可能性があります。一方でエンドポイントを見落とすと、誰にも気づかれないままパブリックにアクセス可能な状態が残ってしまうかもしれません。

どのサービスがHTTPエンドポイントを提供しており、それぞれどのようなパスを提供していますか？

HTTP ENDPOINTS



Astraを除くすべてのモデルにおいて、グラフを使用した方がパフォーマンスが向上し、特にSonnetの改善幅は顕著でした。

Sonnet単体では、判明している53個のサービスのうち7個しか発見できませんでした。しかしグラフを使用することで27個を発見し、その精度は0.221から0.499へと2倍以上に跳ね上がりました。Opusは正解サービス数を18個から25個に伸ばすと同時に、1実行あたりのコストを37ドルから10ドルへと削減しました。また、ツール呼び出し回数は半分になり、入力トークン数もおよそ4分の1に減少しました。

グラフは456個の結果を返しましたが、リストの大きさ以上に重要なのは「そこで何が見つかったか」です。Opusはチェックアウト内にリポジトリが存在しないサービスを48個特定しましたが、そのうち47個はグラフクエリから直接得られたものでした。これらはモデルが偶然覚えていたサービスなどではありません。グラフが検索・取得（リトリーブ）してきたものなのです。

さらにグラフは、オペレーターの Webhook やゲートウェイの Readiness Probe（準備状態チェック）など、モデルがすでに探索済みだったリポジトリ内部で独立して稼働している 9 個のコンポーネントも発見しました。各コンポーネントは、独自の名称のもとで固有のパスを公開していました。通常のリポジトリ検索では、一度リポジトリを見つけるとそこで探索を終えてしまうため、内部で稼働しているコンポーネントを見落としがちになります。

これは「すべてのエンドポイントに認証を必須化する」といった変更を行う際に非常に重要となります。呼び出し元を見落とせば認証を有効化した際に正常なトラフィックを破損させてしまいますし、エンドポイントを見落とせば全員がロールアウト完了と信じ込んでいる裏でアクセス可能なまま放置されてしまいます。今回のケースでは、48個のサービスについてチェックアウト内にコードが存在しませんでした。それらを見落としてしまうのは検索方法が悪かったからではなく、**検索すべき対象がそこに存在しなかったから**なのです。

Astraが例外となったのは、単体でも判明しているサービスの90%以上を発見するなど、すでに極めて高いパフォーマンスを発揮していたためです。グラフはチェックアウト外の28個のサービスを含む新しい情報をもたらしたものの、同時にエンドポイントを公開していないサービスへとモデルの注意を引いてしまいました。その結果、正解集から発見できたサービス数が2個減少し、適合率（Precision）が低下しました。それでもグラフは、Astraがディスク上では見つけられなかった要素の発見に役立ちましたが、すでにそこにあったサービスに関しては改善の余地がほとんど残されていなかったと言えます。

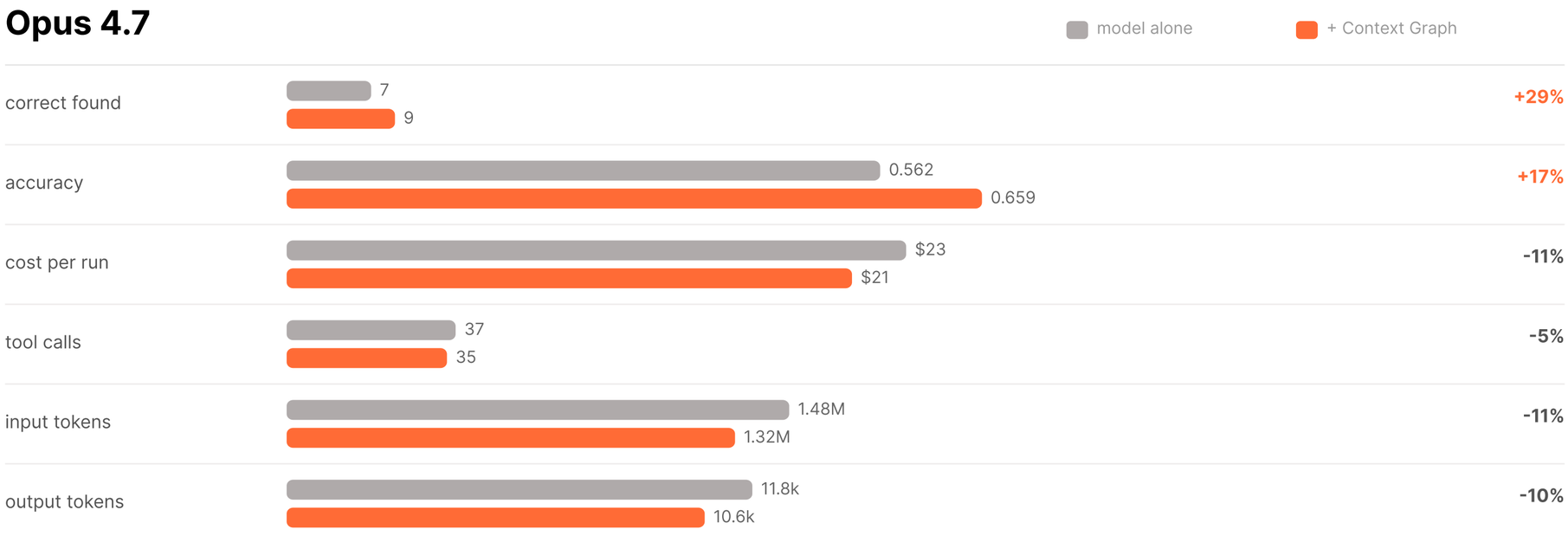
メッセージバスのコンシューマー（受信サービス）

メッセージバスの移行を準備しているとします。どのサービスがイベントを発行（Publish）しているかはすでに把握していますが、それを受信・消費（Consume）しているすべてのサービスを網羅した信頼できるリストが存在しません。各コンシューマーは自身の起動コード内でのみサブスクリプション（購読設定）を登録している可能性があるため、確認できる中央の共通ファイルが存在しないのです。安全に移行を行うには、すべてのサービスを検索し、コンシューマーの完全なリストをご自身でつなぎ合わせて作成する必要があります。

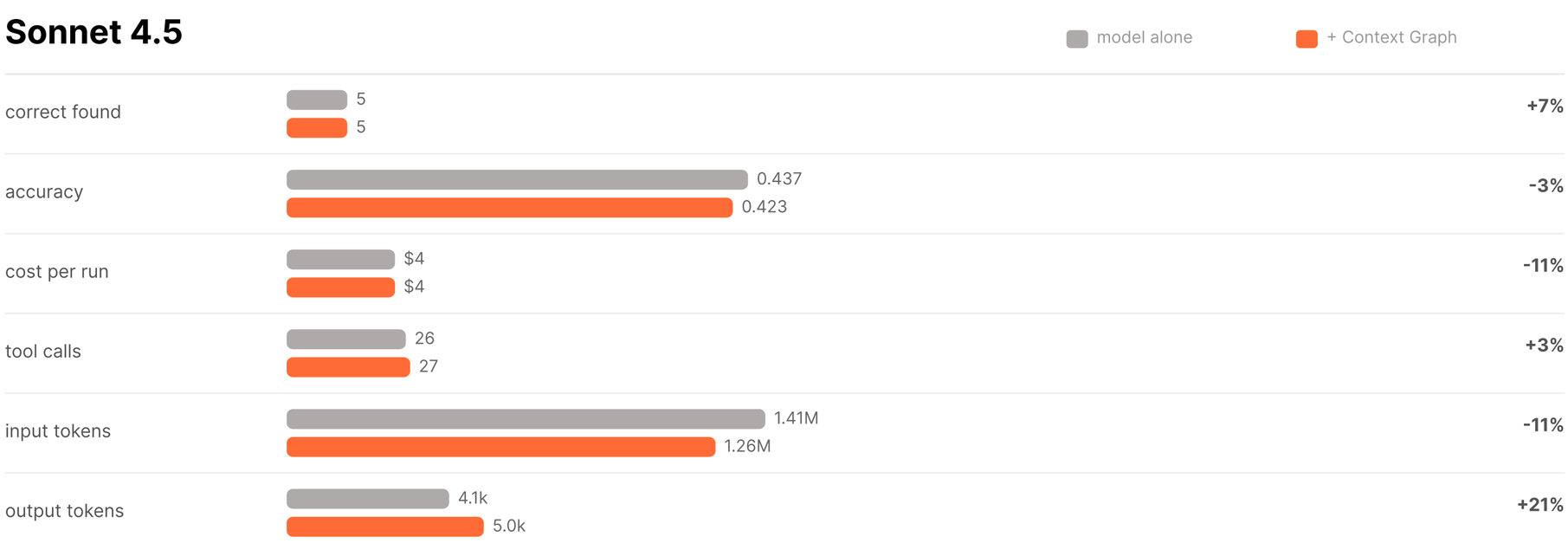
メッセージバスから受信・消費を行っているサービスはどれで、それぞれどのトピックを読み取っていますか？

MESSAGE-BUS CONSUMERS

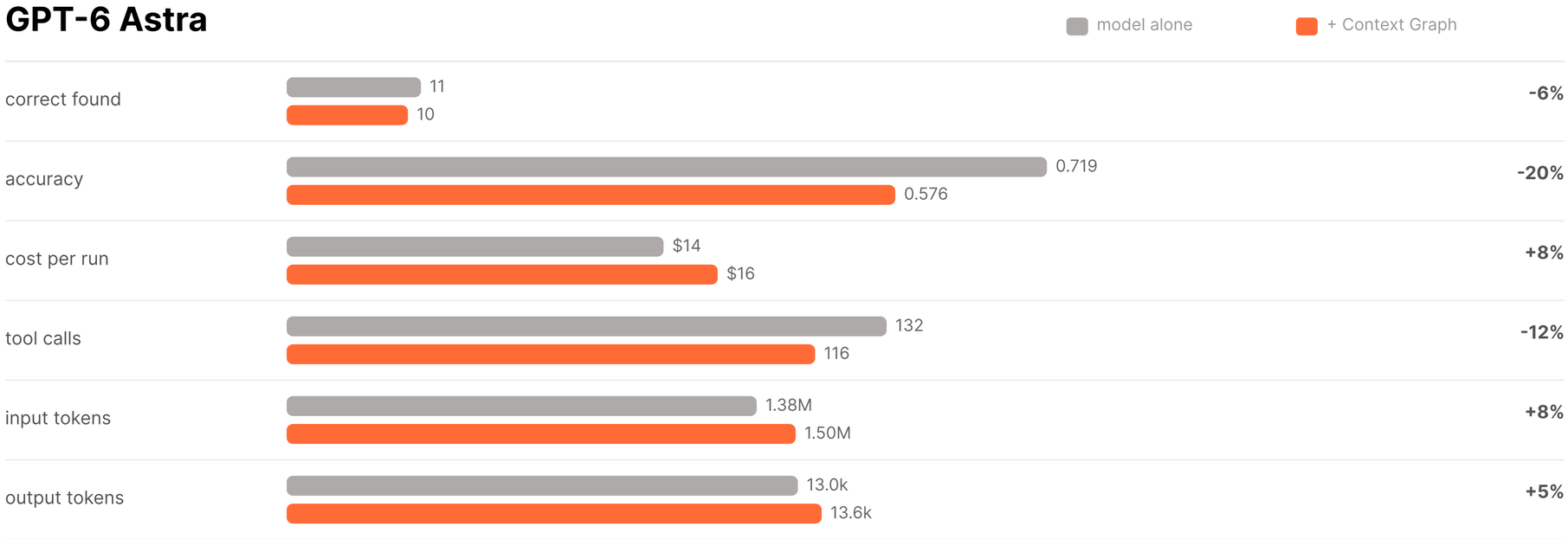
Opus 4.7



Sonnet 4.5



GPT-6 Astra



ここでの結果はモデルによって分かれました。Opusは明確に改善し、16個中7個正解だった状態から9個へと増加し、精度も0.562から0.659に上昇しました。Sonnetは5個のままで変動がなく、精度はわずかに低下しました。Astraは逆の動きを見せ、正解数が11個から10個に減少し、精度も0.719から0.576へと低下しました。

検索によってコンシューマーを見つけるのが難しいのは、それを明示的に宣言するものが存在しないためです。サービスは自身の起動コード内で、その言語が使用しているクライアントライブラリを介してトピックを購読（サブスクライブ）しており、その事実を記録した共通ファイルは存在しません。このプロンプトの正解集（解答キー）を作成するだけでも、5つの言語にわたる13種類の異なるライブラリのイディオムを解読する必要がありました。

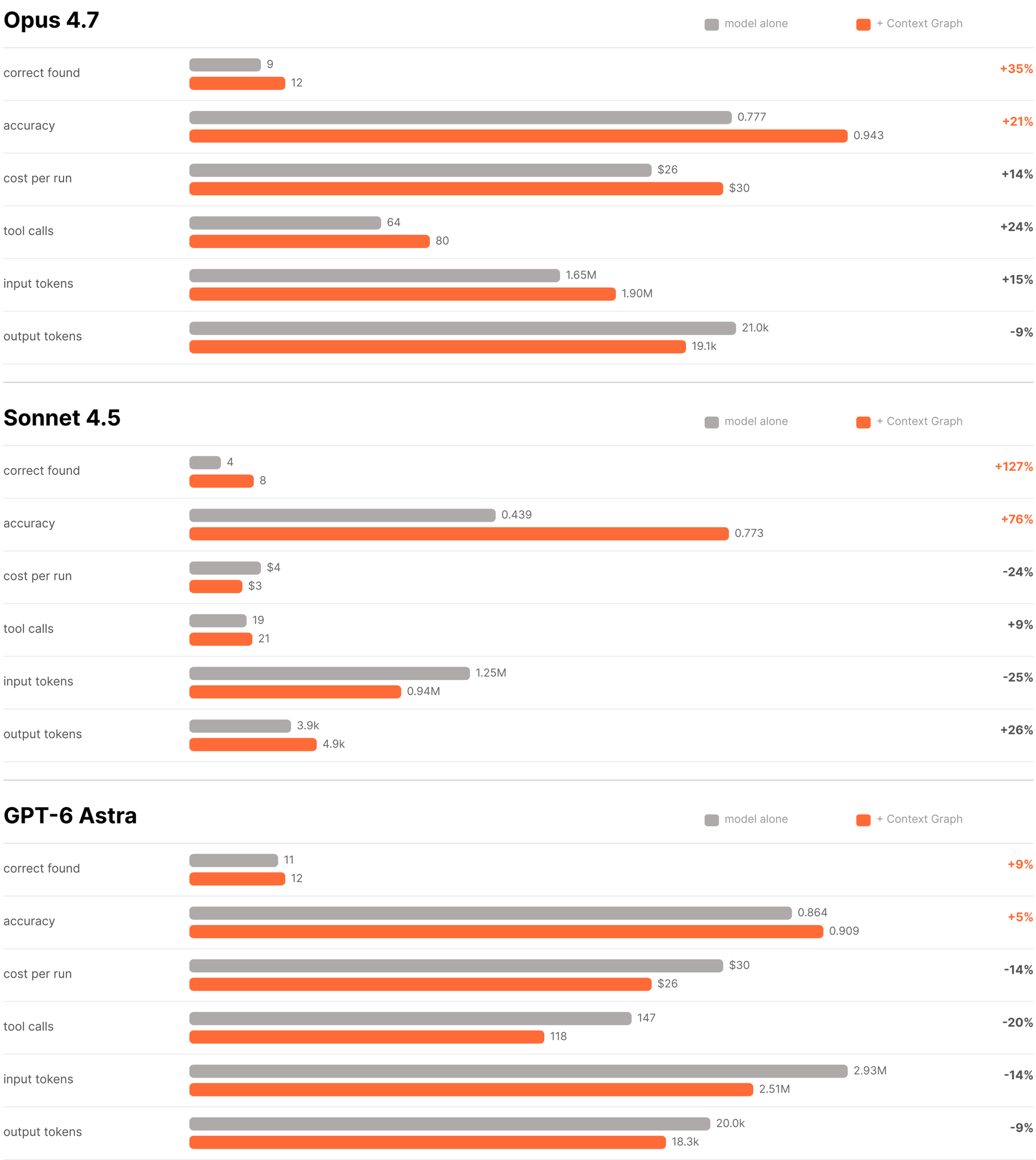
インデックス側もこれを予期されるような形ではモデル化しておらず、それがここでのグラフの限界となっています。インデックス内にはトピックノード（トピックに関するデータ）が一切存在せず、購読（Subscribe）や発行（Publish）といった関係性も保持されていません。保持されているのは、サービスと「外部サービスとして扱われる13のメッセージングシステム」とのリンクだけです。そのため、Kafkaから読み取りを行うサービスは「トピックを購読している」ではなく「Kafkaに接続している」と表示されます。その表現に届くよう十分に広範な質問を投げかけた結果、62個ではなく186個の結果が返され、これがOpusのスコア向上につながりました。同時に、向上がそこでストップした理由もここにあります。グラフはサービスがKafkaと通信していることは教えられても、「発行ではなく受信を行っている」ことまでは教えられず、それこそがまさにこの質問で求められていた区別だったからです。

API呼出元（呼び出し側サービス）

あるAPIを変更しようとしており、それに伴って他にどのサービスを変更する必要があるかを知る必要があります。

プラットフォームのHTTP APIを呼び出しているサービスはどれですか？

API CALLERS



Context Graph APIを使用することで、全13個のAPI呼び出し元リストを発見する精度がすべてのモデルで向上しました。

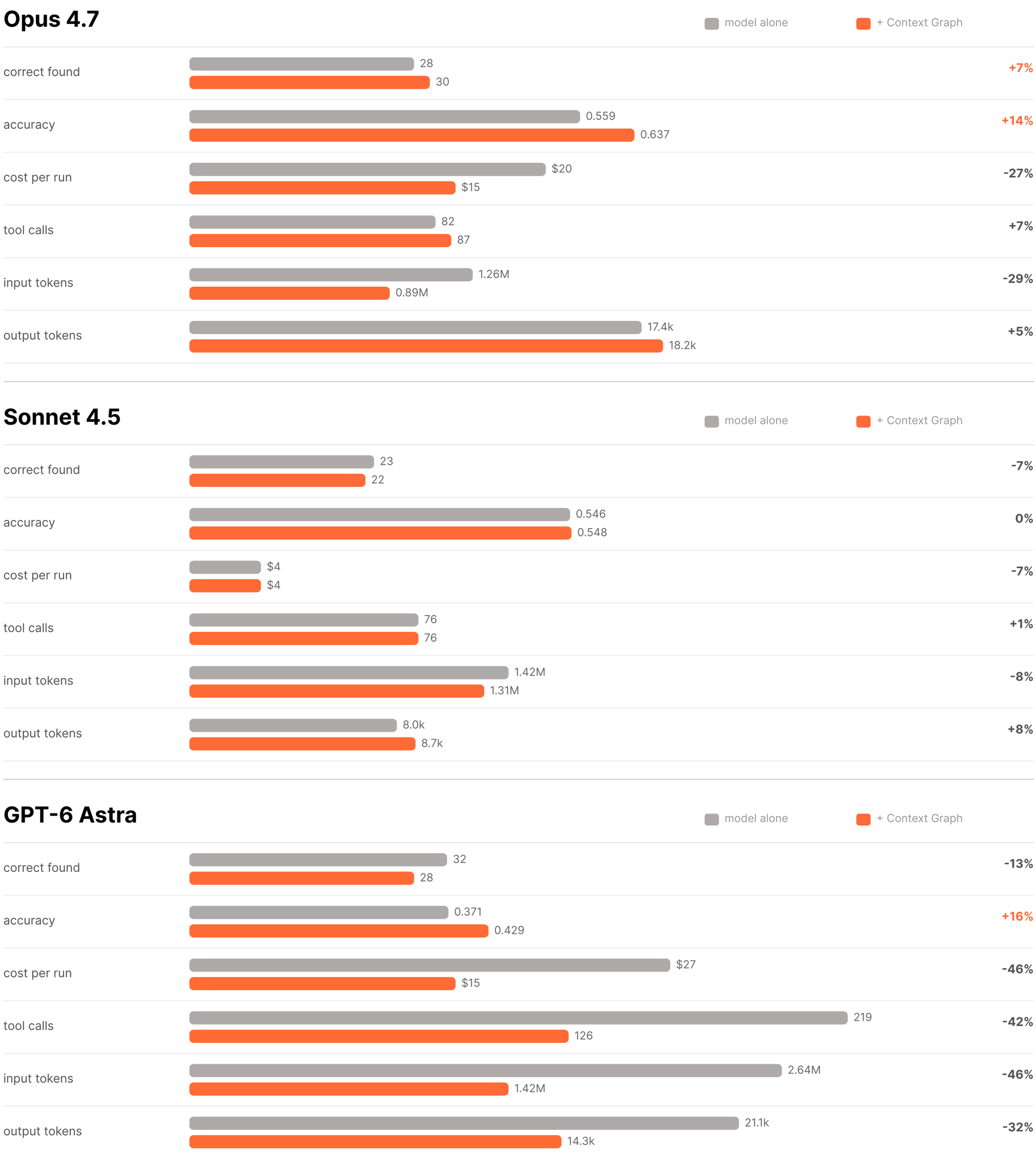
Opusは単体時の9個から13個中12個を発見し、精度は0.777から0.943に上昇しました。Sonnetは平均して3.7個の呼び出し元から8.3個へと2倍以上に増加し、Astraも10.7個から11.7個へと改善しました。これこそが、APIのレスポンスを変更する際にリリース前の影響範囲（ブラストライナース）を把握する上で極めて重要な要素です。グラフがなければ、54ものチームに通知を送ったにもかかわらず、そのAPIに実際に依存している4つのサービスを見落としてしまう可能性があるのです。

スキーマ

契約テスト（コントラクトテスト）の導入を進めようとしていますが、まずはすでに何が存在しているかを把握する必要があります。正式なAPI契約を定義しているサービスもあれば、期待されるリクエストやレスポンスを文書化せずにエンドポイントを公開しているサービスもあります。何をテストすべきか、あるいはどこから始めるべきかを決定する前に、テストの基準となる契約をすでに持っているサービスの信頼できるリストが必要です。

機械判読可能なデータ契約またはAPI契約を宣言しているサービスはどれですか？また、各契約は何を記述していますか？

SCHEMAS



コード検索は「スキーマファイルが存在すること」を伝えることができます。一方、Context Graph は「どのサービスがそのファイルを持ち、どのような契約（コントラクト）を表しているか、そして同じリポジトリ内の複数のサービスがそれぞれ独自のスキーマを持っているか」まで伝えることができます。これこそが「単にファイルを見つけること」と「そのファイルの意味を理解すること」の違いです。

Opus はグラフを使用しても正解となるサービス数をそれほど多く増やしたわけではありませんが、間違ったサービスの出力数を大幅に減らしました。その結果、適合率（Precision）は 0.530 から 0.642 に上昇しました。また Astra は、チェックアウト内にリポジトリが存在しないサービスを 11 個発見し、そのほとんどはグラフの検索結果から直接得られたものでした。モデルがそれらを見つけるために検索できるローカルファイルは一切存在しなかったのです。

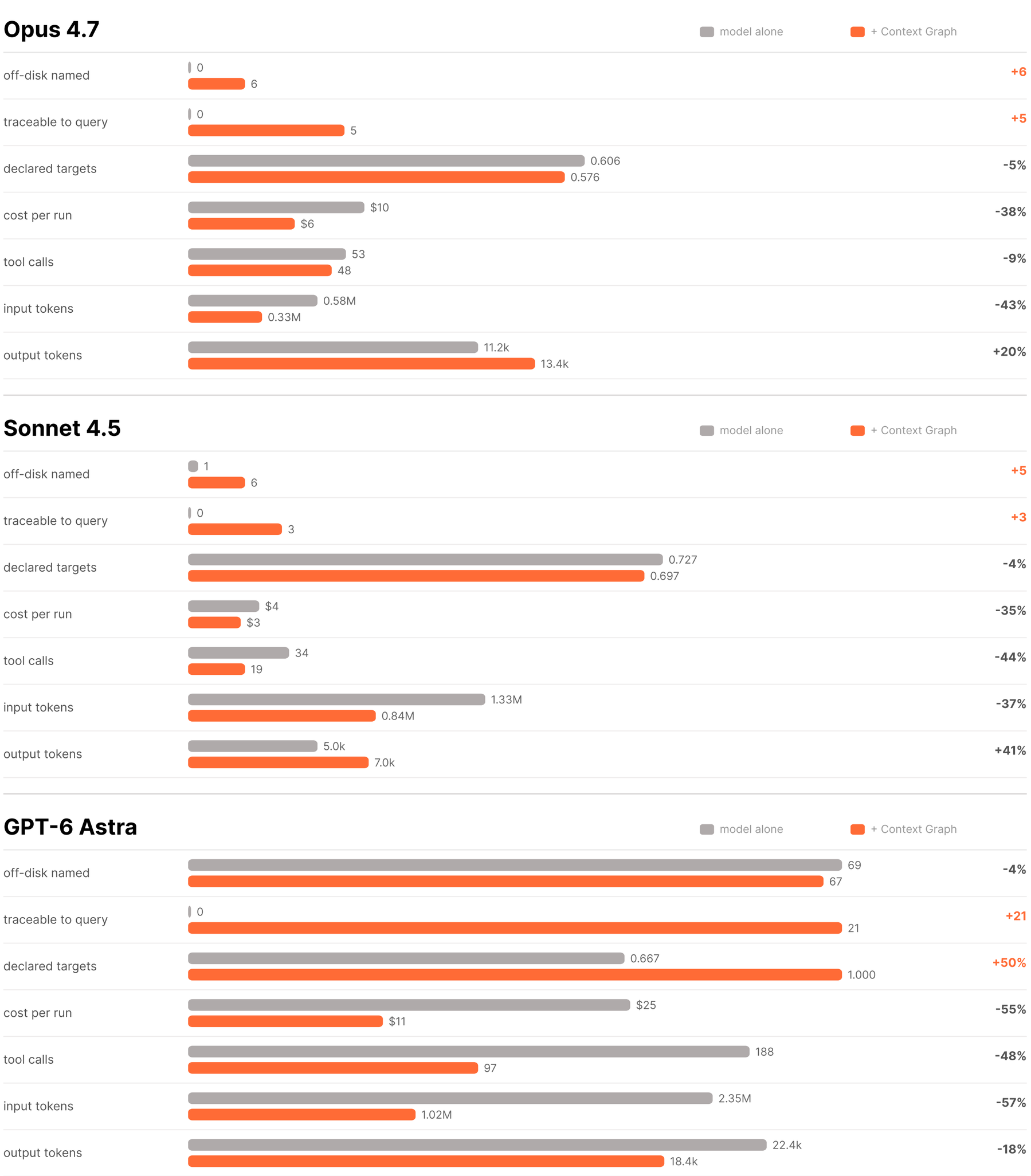
これこそが、契約テストにおいてグラフがこれほど有用である理由です。1つのリポジトリ内に、独立してバージョニングされた API 契約を持つ 4 つのサービスが含まれている場合があります。リポジトリ検索では 1 つのフォルダしか得られませんが、Context Graph は実際にテストする必要がある 4 つのサービスを提供します。その詳細情報がなければ、4 つの異なる API に対して 1 つのテストスイートを作成してしまい、利用側（コンシューマー）で障害が発生して初めてそのギャップに気づくことになりかねません。

アップストリーム（上流）依存関係

共有サービス、データベース、キュー、または外部APIを変更する準備をしているとします。対象に手を加える前に、どのシステムが各サービスの上流に位置し、その稼働を支えているのかを知る必要があります。これらのランタイム（実行時）依存関係はビルド時の依存関係として現れないことが多いため、インポート文やパッケージファイルを検索するだけでは全体像を把握できません。各サービスが何に依存しているのかを理解するには、次のように質問する必要があります。

各サービスは実行時に何に依存していますか？

UPSTREAM DEPENDENCIES



コード内には固有のアップストリーム依存関係が11個しか存在せず、1回の検索でそれらすべてを発見できてしまうため、このプロンプトについては異なる採点方法を採用しました。もしこの11個だけで採点した場合、すべてのモデルがほぼ同じスコアになり、得られる知見はごくわずかになってしまいます。そのため、該当の採点項目は主に「明らかに間違っている回答を検出するため」に存在しています。それでも Astra は際立っていました。グラフを使用することで、単体では約3分の2だった検出率が、毎回の実行で11個すべてを発見するようになりました。

より有益な問いは、モデルが「チェックアウト（ローカル環境）を超えて何を発見できたか」です。私たちは「モデルが挙げたディスク上に存在しない依存関係の数」と「そのうちグラフクエリから直接得られた数」の2点を追跡しました。特に重要となるのは2つ目の数値です。これは公開コードベースであるため、モデルが学習データからサービス名を記憶している可能性があります。グラフの検索結果があることで、その依存関係が実際に情報として取得（リトリーブ）されたという証拠が得られます。

Astra はその違いを明確に示しています。単体ではチェックアウト内にない依存関係を69個挙げましたが、検証可能なものは1つもありませんでした。一方、グラフを使用した場合は67個を挙げ、そのうち21個はクエリから直接得られたものでした。リストの規模はほぼ同じですが、グラフによって補助されたリストには「証拠」が伴っていたのです。

Opus と Sonnet も、チェックアウト外で検証可能なものを全く発見できなかった状態から、それぞれ5個と3個の依存関係を発見できるようになりました。さらに3つのモデルすべてで費用が抑えられ、コストは35%から55%削減されました。

アップストリーム依存関係の一部はコード内からも発見できます。Docker Compose ファイルや Helm の設定値（values）全体で245箇所の宣言が見つかりましたが、それらが指し示していたのはわずか11個の固有の対象だけでした。残りはデプロイ時に設定され、多くは環境変数経由で渡される URL として存在しています。つまり、サービスと呼び出し先システムを紐付けるファイルがリポジトリ内に存在しない可能性があるということです。未知の依存関係がダウンし、それに引きずられて複数のサービスが不通になった場合、問題は「誰かの検索方法が悪かったこと」ではありません。その関係性は最初からコード内に存在しなかったのです。

Context Graph APIが真価を発揮する領域

Context Graphが最も威力を発揮するのは、答えが「ファイルを開けば見つかるもの」ではなく「システム間の関係性」である場合です。3つのモデルと採点対象となった7つのプロンプト（全21ペアのプロンプト×モデル検証）において、Context Graphは18ペアで精度向上をもたらしました。最も大きな成果を上げたケースには共通するパターンがありました。それは、根拠となる情報が複数のシステムに分散している場合や、一箇所に明記されたことがない場合に、Context Graph APIが最大の効果を発揮したという点です。

この特徴は「ランタイム呼び出し」に関する質問で明確に表れました。ここではサービスが起動時に送信先URLを構築するため、完全な関係性を記録したリポジトリが存在しません。また「HTTPエンドポイント」の質問でも同様で、Context Graph APIはチェックアウト内にはまったくリポジトリが存在しない48個のサービスを発見しました。これらのサービスはいずれもモデルが公開プロジェクトを思い出そうとした結果ではなく、Context Graph APIの検索結果から直接得られたものです。

「API呼出元」に関する質問でも同じ現象が見られました。Context Graph APIにより、Opusは単体での9個から増加して実際の呼び出し元13個中12個を発見し、提案するサービスリストの数も54個から30個へと絞り込みました。確定した答えが存在する質問においては、この洗練された（無駄のない）リストが大きな意味を持ちます。モデルは正解のサービスをより多く見つけただけでなく、「不確実な推測をする頻度」を減らしたのです。「メッセージバスのコンシューマー」も同様の傾向を示しました。購読情報（サブスクリプション）が特定のサービスの起動コード内にのみ存在し、そのコンシューマーをイベントに紐付ける情報が他のどこにも存在しない可能性があるためです。

検証した3つのモデルのうち2つにおいて、Context Graph APIはこれらの質問に対する実行コストも削減しました。Opusは27%、Astraは20%のコスト削減を達成しています。Context Graph APIが関係性を直接返すことができる場合、モデルは本来必要としていた膨大な検索プロセスをスキップできるからです。

一方で、Context Graph APIの貢献度が低かったケースも同様に一貫していました。答えがすでに特定のファイル内に記述されている場合、モデルにとってはContext Graph APIを使わずにそのファイルを直接読み込む方が効果的であることが多かったのです。

したがって、得られた結論は単に「Context Graph APIを使えばモデルの性能が上がる」というような大雑把なものではなく、より具体的なものです。それは「Context Graph APIは、ファイル検索では確実に解き明かせない『関係性』へのアクセスをモデルに与える」ということです。答えがすでにリポジトリ内のコードとして書かれているのであれば、素直にファイルを読み込めば済みます。しかし、答えがリポジトリ、サービス、そして各種システム「同士の間（行間）」に存在するならば、それこそがContext Graphが真価を発揮する領域なのです。