SHABELAB

ホワイトペーパー

できないことを定める

― AIエージェントの安全性を、
承認ではなく権限の設計で得る ―

2026年8月

発行

株式会社 喋ラボ

代表取締役　大橋 功

目次

- エグゼクティブサマリー
- はじめに ― 承認ボタンを押したあなたは、何にYESと言ったのか
- 第1章 世界はどこまで来たか
- 第2章 それでも残る3つの穴
- 第3章 経路を断つ
- 第4章 操作の二分と、何にYESと言うのか
- 第5章 実装と、二つの実例
- 第6章 摩擦と、その扱い
- 第7章 結びにかえて ― 4つの問い
- 参考文献

エグゼクティブサマリー

本ホワイトペーパーの主張

AIエージェントのセキュリティをめぐる議論は、いま一つの結論に収束している。権限を最小限にし、入力を分離し、出口を監視し、サンドボックスに閉じ込め、重要な操作には人間の承認を挟む。多層防御である。

本稿は、その最後の一枚である承認ゲートを検討する。承認ゲートは、権限を持ったまま許可を求める仕組みだ。エージェントは実行できる。ただ、その前に聞いてくるだけである。聞く手順が何らかの理由で飛べば、実行はそのまま続く。

世界の到達点と、そこに残る穴

この3年で、モデルを信用しない設計は大きく前進した。汚染を計画から遠ざけるDual LLM、データの出所を追跡するCaMeL、考える系から実行能力そのものを取り上げるParallax。日本語でまとまった紹介がほとんどないため、本稿の第1章はその整理に費やす。

しかし、いずれも最後の砦として人間の承認を置き、提案から実行へ至る機械の経路は残している。経路が残る限り、システムの安全性は、その途中に置いた判定器の精度と等価になる。判定器はソフトウェアであり、ソフトウェアは確率的にしか正しくない。

本稿が示す構成

我々の構成は、不可逆な操作についてこの経路そのものを断つ。エージェントが行うのはURLを生成して提示することまでであり、実行器を起動できるのは利用者のクリックだけである。そして実行は、エージェントではなく利用者自身の認証情報で行われる。経路もなく、鍵もない。

操作は可逆性によって二分する。検索と追加は可逆群、削除と更新は不可逆群とし、どちらに置くかはテナントごとの設定とする。分類は設定だが、遮断は構造である。設定は境界の位置を動かすが、境界の存在そのものは動かさない。

現時点で言えないこと

この構成はマネーフォワード連携とタスク管理において本番稼働している。ただし運用期間は短く、情報システム部門による正式な審査を経た事例はなく、敵対的な試験も行っていない。本稿が述べる性質は設計から導かれる主張であって、測定された結果ではない。

また、本稿が示す構成に原理的に新しい要素はほとんどない。我々が加えるのは、境界をどこに置くかという選択と、本番環境で動かした報告の2つだけである。

読者への問い

最後に、読者に対して問いを残しておきたい。

あなたの会社が使っているAIエージェントは、承認の手順を飛ばしたとき、それでもその操作を実行できるか。できるのであれば、そこに置かれているのは境界ではなく検問である。

はじめに — 承認ボタンを押したあなたは、何にYESと言ったのか

画面に確認が出る。顧客マスタを更新します、よろしいですか。あなたは押す。

そのとき、何が何に変わるのかを、あなたは見ていない。見せられていない。同意したのは操作の名前であって、その結果ではない。

この画面は、いま多くのAIエージェント製品で、最後の安全装置として置かれている。日本語で読めるAIエージェントのセキュリティ解説は、ほぼ例外なく同じところに着地する。権限を最小限にし、入力を分離し、出口を監視し、サンドボックスに閉じ込め、そして重要な操作には人間の承認を挟む。多層防御である。

異論はない。どれも必要である。

本稿が問うのは、その最後の一枚が、我々が思っているほど厚いのかということである。

承認ゲートは、権限を持ったまま許可を求める仕組みだ。エージェントは実行できる。ただ、その前に聞いてくる。聞く手順が何らかの理由で飛べば、実行はそのまま続く。検問は置かれているが、道は通っている。

では、道そのものを断てるか。それが本稿の問いである。

先に断っておきたいことがある。**本稿が示す構成に、原理的に新しい要素はほとんどない。**　ケイパビリティに基づく安全設計も、可逆性による操作の分類も、考える系と実行する系の分離も、すでに提案され、実装され、評価されている。1章はその整理に費やす。日本語でまとまった紹介がほとんどないため、そこだけでも読む価値があるはずだ。

我々が付け加えるのは2つだけである。ひとつは、境界をどこに置くかという選択。もうひとつは、それを本番環境で動かした報告である。

2章で既存の設計に残る穴を3つ挙げ、3章で我々の構成を述べ、4章でその境界の置き方を説明する。5章は実装の報告、6章は未解決の課題である。解けていない問題のほうが、解けた問題より多い。

第1章 世界はどこまで来たか

図1　エージェント防御設計の系譜（2023〜2026）

1-1 命令とデータが区別できない

プロンプトインジェクションが厄介なのは、対策を怠ってきたからではない。LLMの動作原理そのものに根がある。

利用者からの信頼できる指示と、メールやWebページから来る信頼できないテキストが、同じトークン列に連結されてモデルに入る。モデルの側には、どこまでが命令でどこからがデータなのかを構造的に区別する手段がない。Simon　Willisonがこれを「プロンプトインジェクション」と名づけたのは、SQLインジェクションとまったく同じ形をした問題だからだった。

ただしSQLと違い、こちらにはプレースホルダに相当するものが存在しない。ある種類のテキストに書かれた指示には従い、別の種類のテキストに書かれた指示には従わない。その区別を確実に行わせる方法は、いまだ見つかっていない。

だから防御の主戦場は、モデルの中ではなくモデルの外に移った。ここから先の系譜は、すべて「モデルを賢くする」のではなく「モデルの周囲をどう設計するか」の話である。

1-2 Dual LLM（2023年）

最初の分岐点は2023年4月、Willisonが提案したDual LLMパターンだった。

LLMを2つに分ける。ひとつは特権LLM（P-LLM）で、ツールを呼ぶ権限を持ち、利用者のプロンプトだけを見る。もうひとつは隔離LLM（Q-LLM）で、信頼できないテキストを読むが、ツールを呼ぶ権限を一切持たない。

肝は、Q-LLMが扱った内容がP-LLMの入力に入らないことにあった。Q-LLMは中身のかわりに参照だけを返す。P-LLMは「`$email-summary-1`を利用者に表示せよ」と書けるが、その参照が指す実際のトークンには触れない。汚染された文字列が計画立案に届かない構造である。

権限と汚染を分離するという発想は、この時点で出揃っている。

1-3 それでも空いていた穴

2年後、Google DeepMindの論文がこの設計の欠陥を指摘した。

例として挙げられたのは「前回の打ち合わせでBobが依頼した資料を送っておいて。Bobのアドレスと資料名は議事録ファイルにある」という指示である。P-LLMは計画を立て、アドレスの抽出をQ-LLMに委ねる。ここまでは設計通りだ。

しかしQ-LLM自身は、依然として悪意ある指示に晒されている。攻撃者が議事録に細工をしておけば、抽出されるアドレスを攻撃者のものに差し替えられる。P-LLMは汚染されていない。計画も正しい。それでも資料は攻撃者の手に渡る。

計画（制御フロー）を守っても、データフローが守られていなければ流出は起きる。

1-4 CaMeL（2025年）

DeepMindの回答がCaMeLである[1]。

利用者の指示を、P-LLMがPythonのサブセットにあたる限定された言語のコードへ変換する。そのコードを独自のインタプリタで実行し、どの変数がどの変数から導かれたかを追跡する。各変数にはケイパビリティと呼ばれるタグが付き、データの出所と読み取り権限が記録される。

先の例であれば、メール本文は信頼できない出所なので、そこから導かれたアドレスも信頼できないと判定される。「送信先が信頼済みのときのみ送信を許す」というポリシーを置けば、差し替えられたアドレスへの送信は止まる。

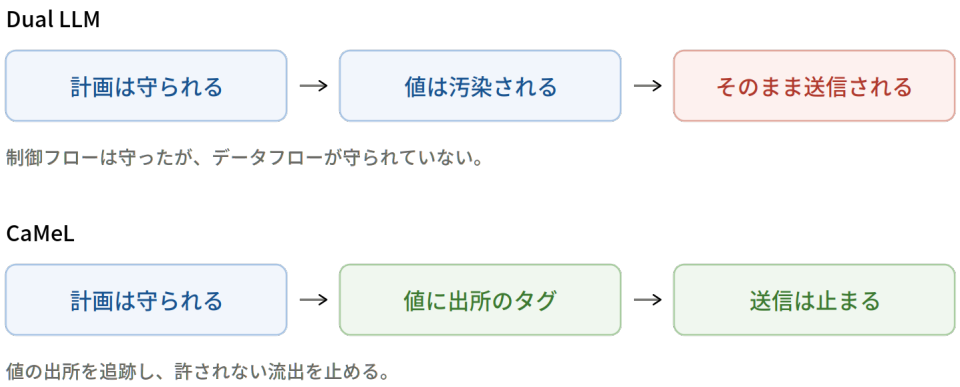

図2　Dual LLMとCaMeLの違い

この設計の見識は、AIを重ねなかったことにある。インジェクションを検知するモデルをもう一段積む解法は多いが、それは確率的な防御にすぎない。Willisonの言い方を借りれば、アプリケーションセキュリティにおいて99%は落第点である。攻撃者の仕事は、通る1%を見つけることだからだ。CaMeLはかわりに、ケイパビリティとデータフロー解析という、枯れた技術に寄った。

AgentDojoベンチマークでは、証明可能な安全性を伴って67%のタスクを解いている。

1-5 CaMeL自身が認める限界

ここは日本語でほとんど紹介されていない部分なので、強調しておきたい。

論文の8-3節は「プロンプトインジェクションは解決したのか」と自問し、していない、と答えている。挙げられている限界は2つある。

ひとつは、利用者がセキュリティポリシーを自分で記述し、保守しなければならないこと。もうひとつは、利用者への負荷である。とりわけ、確認を求められ続けた人間が何にでも「はい」と答える状態に陥る問題を、論文はuser fatigueとして名指ししている。

この2点は、本稿の6章でそのまま我々に返ってくる。覚えておいていただきたい。

1-6 Parallax（2026年）

さらに踏み込んだのが、2026年4月に公開されたParallaxである[2]。副題は「考えるAIエージェントは、決して行動してはならない」。本稿が扱う問題に最も近い先行研究であり、以降の章でも繰り返し参照する。

4つの原則からなる。

第1に、Cognitive-Executive Separation。 考える系は構造的に実行できず、実行する系は構造的に考えられない。推論プロセスはサンドボックスの中で動き、ファイルシステムへのアクセスも、ネットワークへのアクセスも、シェルを実行する能力も持たない。エンジンプロセスとのgRPC通信だけが唯一の経路である。OpenSSHの特権分離をエージェントに持ち込んだ設計だと、著者自身が位置づけている。

第2に、Adversarial Validation with Graduated Determinism。 提案された操作を、独立した検証層が4段階で評価する。Tier 0は決定論的な、つまり同じ入力に必ず同じ結果を返すポリシー規則、Tier 1は分類器、Tier 2は別のLLMによる評価、Tier 3が人間である。決定性の高い順に並べ、下の段で決着しなければ上へ上げる。どの段でもエラーが起きれば拒否する。安全側に倒れる設計だ。

第3に、Information Flow Control。 読み取ったデータに機密ラベルを付け、以降の操作へ伝播させる。認証情報を読み、それをURLに書き、送信する。個々には無害な操作の連鎖を、これで検出する。

第4に、Reversible Execution。 破壊的な操作の前に、巻き戻せるだけの状態を保存しておく。ただし但し書きがあり、外部システムに及ぶ操作、たとえば第三者APIの呼び出しや送信済みのメールは巻き戻せない場合がある。その場合は事前の防御の比重が上がり、人間の承認を要求すべきだとされる。

そして、ここから導かれる原則としてValidator Immutabilityが置かれる。推論システムは検証層に対して、読むことも書くことも変更することも干渉することもできない。検証層の設定はコンパイル済みのコードに埋め込まれ、変更可能なファイルには置かれない。検証者を誰が検証するのかという古典的な問いへの、アーキテクチャによる回答である。

評価も踏み込んでいる。推論システムを完全に迂回してツール呼び出しを直接注入するAssume-Compromise Evaluationという手法で、9カテゴリ280件の攻撃を試験した。既定の設定で98.9%を遮断

し、誤検知はゼロ。最大強度の設定では280件すべてを遮断するが、そのとき正当な操作の36%が巻き添えで止まる。

この280件は、著者自身が構成した攻撃セットである。CaMeLが用いたAgentDojoのような、第三者が公開しているベンチマークではない。また本稿執筆時点で、この論文は査読を経ていない。

なお、人間による承認（Tier　3）が最終的に発火したのは280件中1件、0.4%である。この数字は、本稿の3章で意味を持つ。

1-7 机上ではなく現場で

ここまでは設計の話である。2026年に入って、状況は具体的になった。

OWASPが公開している agentic AI のセキュリティ報告は、前年版が起こりうる脅威のカタログだったのに対し、最新版はCVE、ベンダー勧告、侵害報告のカタログに変わった。脅威の想定ではなく、起きたことの記録である。

なかでも本稿の主題に直結する事例をひとつ挙げておきたい。コーディングエージェントCursorに対して開示されたCVE-2026-22708である[3]。

仕組みは二段構えだった。まず、`export` や `typeset` といったシェルの組み込みコマンドが、許可リストに載っていなくても、承認を求められることなく実行できた。検査そのものを素通りしていたのである。攻撃者はこれを使って環境変数を汚染する。すると、`git branch` や `python3 script.py` のような、利用者が正規に承認したコマンドが、任意コード実行の経路に変わる。

許可リストに載っているということが、攻撃の条件になった。　利用者が自分で安全だと判断して登録した操作ほど、汚染された環境の下では確実に走る。

発見者はあわせて、サニタイズに基づく防御は、コードが入力として想定される文脈では原理的に成立しないと指摘している。

安全性を高めるはずの機構が、攻撃の踏み台になる。この反転は本稿の6章で我々自身にも返ってくるので、頭の片隅に置いていただきたい。

そして直近の総括として、適応的な攻撃は公開されているほぼすべての防御を回避する、という評価が出ている。ここまで見てきた研究群も例外ではない。

1-8 到達点

整理すると、こうなる。

汚染を計画から遠ざける（Dual　LLM）。データの出所を追跡して流出経路を塞ぐ（CaMeL）。考える系そのものから実行能力を取り上げ、検証層を触れなくする（Parallax）。

いずれもモデルを信用しないという前提を共有している。この10年で最も筋の良い進歩が、この3年に集まっている。

しかし3つとも、最後の砦として人間の承認を置いている。そしてCursorの事例が示したのは、その砦が条件次第で攻撃側に回るということだった。

次章では、砦に空いている穴を3つに絞って見ていく。

第2章 それでも残る3つの穴

前章で見た設計は、いずれもモデルを信用しないという前提に立っている。その前提は正しい。問題は、信用しないと決めたあとで、何を信用することにしたかである。

2-1 承認という行為の空洞

第一の穴は、承認ゲートそのものにある。

承認ゲートが機能するのは、押す人間が何を承認したのか理解している場合に限られる。ところが実務では、エージェントは「顧客マスタを更新します」としか言わない。何が何に変わるのかは示されない。押した人間は、操作の名前に同意しただけで、結果に同意したわけではない。

そして名前しか見えない承認は、繰り返されるほど形骸化する。CaMeLの論文がuser fatigueと呼び、Parallaxが承認疲労攻撃として対策を組み込んだのは、この問題である。50回目のほとんど同じ確認画面を、人間はもう読まない。

だが空洞の本質は、心理の問題ではない。承認ゲートは、権限を持ったまま許可を求める仕組みだということだ。エージェントは実行できる。ただ、その前に聞いてくるだけである。聞く手順が何らかの理由で飛べば、実行はそのまま続く。ゲートは経路の上に置かれた検問であって、経路を断つものではない。

2-2 計画を立てられない相手

第二の穴は、CaMeLの適用範囲にある。

CaMeLは、利用者の指示を先にコードへ変換する。手順が事前に確定していることが前提になっている。ところが実務のエージェントは、途中で得た情報を見て次に呼ぶ道具を決める。取引先の回答を読んでから照会先を変え、ファイルの中身を見てから処理を分ける。動的にツール呼び出しを決めるエージェントとCaMeLの相性が悪いことは、後続の研究でも指摘されている。

加えて、CaMeL自身が認めるとおり、セキュリティポリシーは利用者が書き、保守しなければならない。AWSのIAMを完全に理解している人間がどれだけいるかを考えれば、これは軽い前提ではない。

2-3 経路が残る限り、安全性は確率になる

第三の穴が、本稿の中心である。

Parallaxはエージェントから実行能力そのものを取り上げた。ここまでは徹底している。しかし取り上げたその先で、実行器を起動するのは依然として機械である。エージェントが提案し、Shieldが検証し、通れば実行器が動く。人間は4段目に控えてはいるが、280件の試験で最終的に発火したのは1件、0.4%にすぎない。

つまりこの設計において、システムの安全性はShieldの判定精度と等価である。

そしてShieldは、著者自身が構成した280件の攻撃のうち277件を止めた。98.9%である。誤検知はゼロだった。

しかし1章で引いたWillisonの基準を、ここに当てはめてみたい。アプリケーションセキュリティにおいて99%は落第点だった。攻撃者の仕事は通る1%を見つけることだから、というのがその理由である。この基準は、インジェクションを検知するモデルにも、操作を検証する層にも、等しく当てはまるはずだ。98.9%は、その基準では落第点である。

Parallax自身、限界を正直に列挙している。エンジンのバイナリが一箇所でも破られれば全体が崩れること。`python script.py`というコマンドは評価するが、スクリプトの中身は見ないこと。最大強度の設定なら280件すべてを止められるが、そのとき正当な操作の36%も巻き添えで止まること。

これらは実装の粗ではない。提案から実行へ機械の経路が通っている限り、その途中に置いた判定器の精度が、そのままシステムの安全性になる。その構造から出てくる帰結である。

そしてCursorの事例が示したのは、判定器が攻撃側に反転しうるということだった。許可リストは、載っている操作を素早く通すために作られる。攻撃者から見れば、それは素早く通る操作の一覧表でもある。

2-4 問いの置き直し

3つの穴には、共通の形がある。

エージェントを信用しない設計は、この3年で十分に成熟した。だが信用しないと決めたあと、これらの設計は判定器を信用することにした。判定器はソフトウェアであり、ソフトウェアは確率的にしか正しくない。

ならば、問いはこうなる。

判定器の精度に依存しない設計はありうるか。言い換えれば、提案から実行へ至る機械の経路そのものを、断つことはできるか。

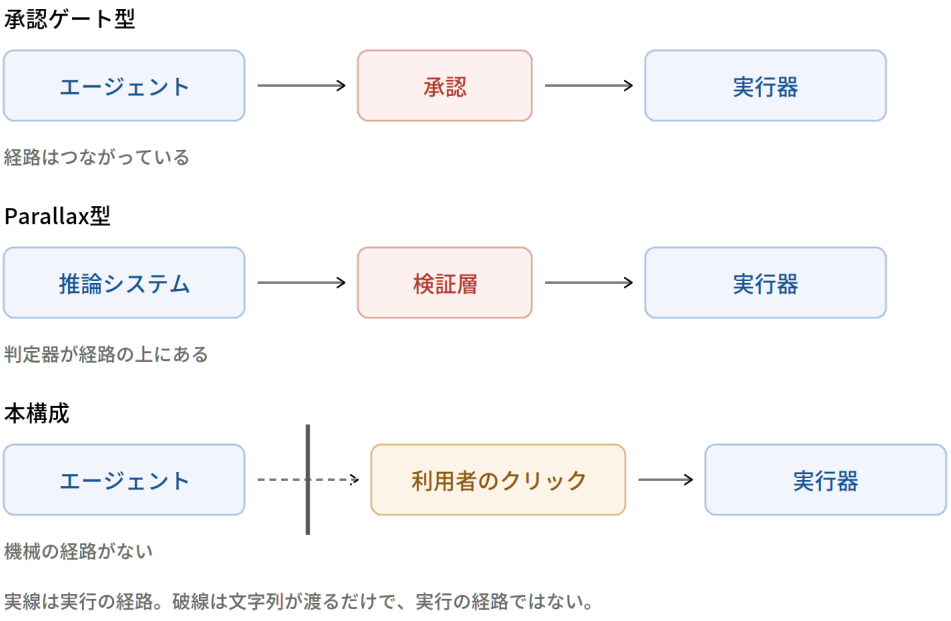

図3　三つの設計における実行経路

第3章 経路を断つ

3-1 盗ませない設計と、起こさせない設計

前章の問いに答える前に、言葉を分けておきたい。

これまでの研究が達成してきたのは、盗ませないことだった。認証情報を読み出せない。機密データを外へ出せない。CaMeLのケイパビリティも、ParallaxのInformation Flow Controlも、データが行ってはならない場所へ行くのを止める仕組みだった。

本稿が扱うのは、起こさせないことである。データを盗ませないのではなく、操作そのものを起こさせない。エージェントが顧客マスタの更新を実行しようとしたとき、それを止めるのではなく、そもそも実行という行為に手が届かない状態をつくる。

言葉遊びに見えるかもしれないが、実装はまったく別のものになる。止めるには判定が要る。手が届かないようにするには、判定は要らない。

3-2 経路の断ち方

本構成では、操作を2つに分けている。検索と追加を可逆群、削除と更新を不可逆群と呼ぶ。境界をここに置いた理由は次章で述べる。

不可逆群について、処理は次の順序で進む。

1. エージェントが、操作を表すURLを生成する
2. そのURLを、対話の中で利用者に提示する
3. 利用者が、自分のブラウザでそれをクリックする
4. 変更前と変更後を並べた確認画面が表示される
5. 利用者がYESを選ぶと、事前に登録された決定論的なコードが実行される

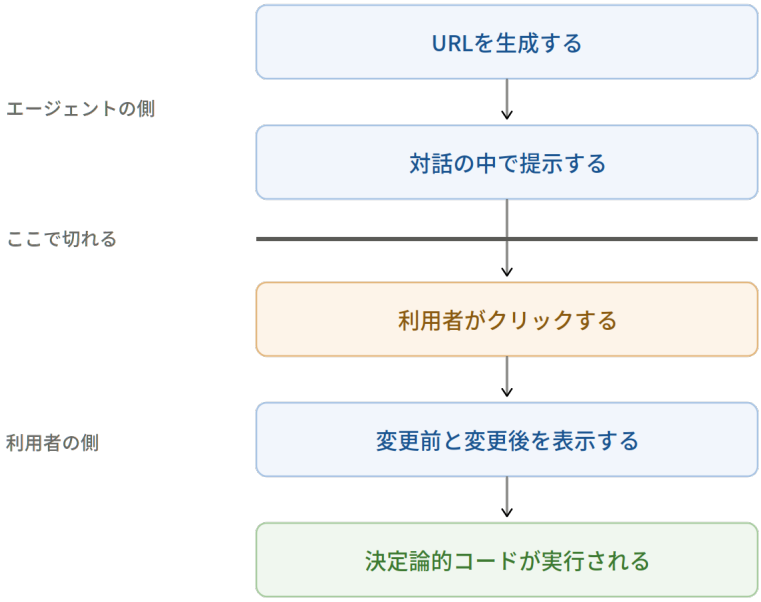

図4　不可逆群の操作における実行フロー

エージェントが行うのは1と2だけである。3以降に、エージェントが介入できる箇所はない。

肝心なのは、1でエージェントが作っているのが文字列だという点だ。URLを生成することと、URLを訪問することは別の行為である。エージェントは前者しかできない。

当然の疑問として、エージェント自身がそのURLを叩けばよいのではないか、という指摘がありうる。これは二重に成立しない。

経路がない。　エージェントの実行環境から実行器へ至る通信経路は、出口の許可リストで塞がれている。そしてこの許可リストは、エージェントの実行ユーザーからは書き換えられない。書き換えられるのは、エージェントとは別の、sudoを持つ利用者だけである。

鍵がない。　不可逆群の操作は、利用者自身の認証情報によって実行される。エージェントはその認証情報を保持していない。仮に経路が何らかの理由で通ったとしても、手元に使える鍵がない。

この2つは互いに独立している。片方が破られても、もう片方が残る。

結果として、実行器を起動できるのは利用者の操作だけになる。判定器の精度が問題にならないのは、判定が行われないからではなく、判定を通り抜けたところで届く先がないからである。

3-3 なぜ任意のエージェントで成立するのか

この構成には、副次的ではあるが実務上は大きい性質がある。

エージェント側に要求されるのは、URLを含むテキストを出力する能力だけである。特定のフレームワークも、専用のインタプリタも、制限された言語も要らない。

2章で見たとおり、CaMeLは手順が事前に確定していることを前提としており、動的にツールを決めるエージェントとは相性が悪かった。本構成にはその制約がない。エージェントが途中で方針を変えようが、何段の推論を重ねようが、出てくるのは文字列だからである。

言い換えれば、この設計はエージェントの賢さに対して中立である。モデルが賢くなっても、壊れても、境界の位置は動かない。

ただし前提がひとつある。生成したURLを利用者に届けるチャネルが要る。そしてチャネルには固有の制約がある。たとえばLINEを窓口にした場合、こちらから送信できるのはターンの終了時に一度だけで、処理の途中に確認を差し込むことができない。操作の提示をどの時点に寄せるかを、チャネルごとに設計し直す必要がある。

エージェント側の要件が軽いことは、要件が消えたことを意味しない。軽くなった分は、チャネル側の制約に置き換わる。

3-4 何が達成されていないか

誇張を避けるために、明確にしておきたいことが3つある。

第一に、これはエージェントを安全にする仕組みではない。不可逆群の操作が届かなくなるだけであって、可逆群、すなわち検索と追加は、これまでどおりエージェントが起動する。可逆群に対しては、1章で見た防御群が引き続き必要である。本構成はそれらの代替ではなく、直交する層である。

第二に、有限の操作カタログに登録されたものしか扱えない。任意のシェルコマンドを安全に実行する方法を、本構成は提供しない。汎用性を捨てることで境界を得ている。

第三に、利用者を騙してクリックさせる攻撃は防げない。エージェントが説得力のある文章で誤った操作を勧め、利用者が納得して押してしまう経路は残る。Parallaxが利用者へのソーシャルエンジニアリングを明示的に対象外としているのと同じ理由で、我々もここは扱えない。ただし次章で述べる構造化差分は、この攻撃の成立条件をいくらか狭める。

そして最大の未達成は、利用者に手を動かさせることそのものである。これは6章で扱う。

第4章 操作の二分と、何にYESと言うのか

4-1 なぜ可逆性で切るのか

前章で名前だけ挙げた境界を、ここで説明する。可逆群と不可逆群を分けているのは、名のとおり可逆性である。

素直な分け方は読み取りと書き込みだろう。状態が変わるかどうかで切る。だがこの軸は、新しいレコードを1件足す操作と、全レコードを消す操作を同じ側に入れてしまう。実務の感覚と合わない。

合わない理由は、問うべきことが状態の変化ではないからだ。問うべきは、戻れるかどうかである。

そして戻れるかどうかは、情報の増減で決まる。追加は情報を増やす。増えた分を取り除けば元に戻る。更新は、書き換えられた前の値を消す。削除は、レコードそのものを消す。**失われた情報は、外部に控えがない限り復元できない。** 可逆性という軸は、この非対称を素直に写したものである。

ただし但し書きが要る。追加が可逆なのは、その追加が系の内側で完結する場合に限られる。メールの送信は追加の一種だが、送ったものは取り消せない。系の外に出る操作は、追加であっても不可逆群として扱う。Parallaxが外部システムへの操作を巻き戻せないと明記したのと同じ線引きである。

4-2 なぜ段階ではなく、できるかできないかなのか

既存のフレームワークの多くは、操作を3段階や4段階に分ける。読み取り専用、可逆、外部影響、高リスク、といった具合だ。粒度が細かいほうが、運用は柔軟になるように見える。

我々は、できるかできないかの二つしか置かなかった。理由は運用の都合ではない。

6章で述べるとおり、本構成には、繰り返し承認された操作を不可逆群から可逆群へ自動的に降格させる仕組みがある。利用者の手間を減らすための機構だが、承認の蓄積で権限が広がる以上、攻撃者がそれを狙わない理由はない。

ここで境界の置き方が効いてくる。降格先が可逆群しかなく、可逆群が定義上すべて可逆であるなら、**攻撃者が承認を積み重ねて得られる最大の戦果は、可逆な操作にとどまる。** 分類の基準が承認履歴ではなく可逆性である以上、削除や更新が承認の回数によって可逆群に上がることは起こりえない。

段階を細かくすると、この保証が消える。4段階のうち3段目から2段目への降格は、依然として危険な場所への降格でありうるからだ。

粒度の粗さは、この設計では欠点ではない。降格の安全性が、分類の粗さによって保証されている。

4-3 何にYESと言うのか

冒頭で立てた問いに戻る。承認ボタンを押した人間は、何を承認したのか。

本構成の確認画面は、操作の名前を出さない。変更前の状態と変更後の状態を並べて出す。顧客マスタを更新しますではなく、この項目がこの値からこの値に変わります、と出す。

設計上、ここには外せない条件がひとつある。**差分を生成するのは、実際にその操作を行う決定論的なコードでなければならない。**　エージェントに差分を書かせては意味がない。表示された差分と実行される内容が食い違いうるからだ。見たものが実行されるという保証は、差分と実行が同じコードから出ていることによってのみ得られる。

これは3-4で残した課題、すなわち利用者を騙してクリックさせる攻撃の成立条件を、いくらか狭める。エージェントがどれほど巧みに操作を勧めても、画面に出るのは実際の変更内容だからである。

4-4 検証を実行時から登録時へ

不可逆群の実行器は、事前に登録された有限のカタログからのみ選ばれる。任意のシェルコマンドを受け付ける口はない。

この制約には、見落とされやすい効果がある。実行器は、渡された操作が安全かどうかを判定しなくてよい。安全だと確認済みの操作しか、そもそも実行器として存在しないからだ。

言い換えれば、**検証は実行時ではなく登録時に済んでいる。**　実行時に判定が要らないということは、実行時に判定を誤る余地もないということである。

2章で見たParallaxの限界のひとつは、`python  script.py`というコマンドは評価できてもスクリプトの中身は見えない、という点だった。任意のコードを実行できる口を残す限り、この穴は原理的に塞がらない。汎用性を捨てた設計は、この穴を持たない。

4-5 境界は設定であり、遮断は構造である

ここまで、可逆群と不可逆群の分け方を固定のものとして述べてきた。実際には違う。

どの操作をどちらの群に置くかは、テナントごとに決まる。　同じ操作でも、組織によって扱いは変わる。下書きの削除が日常の作業である会社もあれば、記録の保持が義務づけられている会社もある。可逆性の線引きは、技術的な性質だけでなく、その組織が何を失っては困るかによって決まる。

重要なのは、変えられるものと変えられないものの区別である。

変えられるのは分類である。どの操作をエージェントに任せ、どの操作をURLとして人に返すか。これはテナントごとの設定として、人が決める。

変えられないのは遮断のほうだ。不可逆群に置かれた操作について、提案から実行へ機械が到達する経路が存在しないことは、設定ではなく構造から来ている。どのテナントも、設定を変えることでこの経路を復活させることはできない。エージェント自身にできないのは言うまでもない。

Parallaxが検証層の完全性について、ポリシーの遵守ではなくアーキテクチャ上の分離によって保証されなければならないと述べたのと、同じ性質のことを言っている。**設定は境界の位置を動かすが、境界の存在そのものは動かさない。**

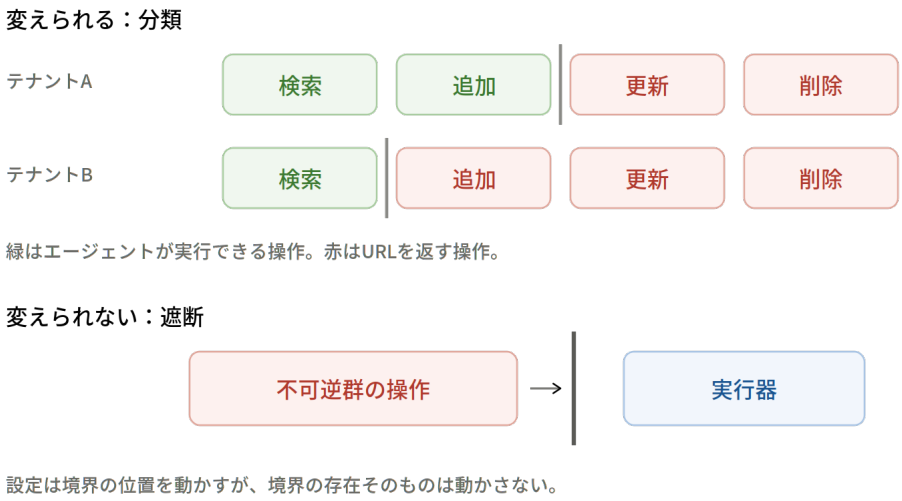

図5　設定が動かすものと、構造が動かさないもの

この区別は、次章以降で二度効いてくる。5章で見るとおり、テナントごとに実行環境を分離しているのは、設定が他のテナントに漏れないためである。そして6章で触れる自動降格は、この設定を運用の中で書き換えていく機構にあたる。

4-6 既存手法との対照

ここまでの整理を表にまとめる。

	承認ゲート	Dual LLM	CaMeL	Parallax	本構成
遮断の位置	実行の直前	計画と汚染の間	データフロー	推論と実行の間	提案と実行の間
実行器を起動するのは	機械	機械	機械	機械	利用者
動的なツール選択	可	可	不可	可	可
可逆性の扱い	なし	なし	なし	事後の巻き戻し	事前の二分
安全性が依存するもの	人の注意力	分離の完全性	利用者が書くポリシー	判定器の精度	経路の不在
実行対象	任意	任意	限定された言語	任意シェル	有限カタログ

とくに最後から二段目に注目してほしい。承認ゲートが依存しているのは判定器ではなく、押す人間の注意力である。2-1で見た空洞は、そこから来ている。

一段ずつ見ていくと、本構成が何かを増やして優位に立っているわけではないことが分かる。実行対象を有限のカタログに絞り、汎用性を手放した。その代わりに、実行器を起動する主体が機械から利用者へ移り、判定器への依存が不可逆群について消えた。

取引である。次章では、その取引が本番環境で何を意味したかを見る。

第5章 実装と、二つの実例

5-1 何が動いているか

我々はこの構成を、エージェントが外部システムを操作するすべての箇所に適用し、本番環境で稼働させている。マルチテナントのSaaSであり、テナントごとに実行環境を分離している。

以下では、マネーフォワード連携とタスク管理という2つの実例を挙げる。

エージェントから見た操作の可否は次のとおりである。参照は可能。追加も可能。更新と削除はできない。できないというのは、権限を持っているが拒否されるという意味ではなく、呼び出す手段が存在しないという意味である。

外部への通信は、出口の許可リストによってVM単位で制限されている。この許可リストは、エージェントを動かしている実行ユーザーからは変更できない。変更できるのは、別に用意したsudo権限を持つ利用者だけである。監査ログは各VMから中央のログ基盤へ直接送られ、エージェントの側からは読むことも消すこともできない。

5-2 担当者を変更するとき

不可逆群にあたる操作の例として、担当者の変更を挙げる。

エージェントは、この操作を実行できない。かわりにURLを返す。利用者がそのURLを開くと、エージェントの存在しない環境で動作する決定論的な仕組みがUIを表示する。画面には変更の内容が出る。利用者が確認して実行すると、**利用者自身のトークンで**処理が走る。

3章で述べた二重の遮断が、ここで具体的な形をとっている。エージェントは実行器への経路を持たず、不可逆群を実行するための認証情報も持たない。権限の所在が、はじめから人間の側にあるということである。

5-3 消せなかったタスク

タスク管理の機能でも、同じ構成を採っている。そこで次のことが起きた。

この機能では、エージェントに削除の権限を与えていなかった。ある場面で、追加したタスクを完了ではなく取り下げる必要が生じた。エージェントは、削除するためのツールが存在しないので削除できない、と報告して止まった。

これは我々の設計漏れである。必要な機能を登録し忘れていた。利用者には余分な手間をかけた。

だが同時に、この一件は仕組みが意図どおりに働いていることの証明でもあった。エージェントは迂回を試みなかった。別の操作を組み合わせて実質的な削除を行おうともしなかった。存在しない権限を使ったふりもしなかった。できないと言って止まった。

ないものは、できない。　当たり前に聞こえるが、承認ゲート型の設計ではこの当たり前が成立しない。権限があって許可を求める仕組みの下では、許可の手順を回避する経路が理論上つねに残るからだ。権限そのものがなければ、回避すべき手順もない。

もっとも、これは代償の実例でもある。有限のカタログで運用する以上、登録漏れは機能不全として現れる。判定の誤りではなく、単に何もできないという形で現れる。安全側の失敗ではあるが、失敗であることに変わりはない。

5-4 現時点で言えないこと

本章の記述には限界がある。明示しておきたい。

第一に、稼働からの期間が短い。長期運用によって設計を見直した経験を、我々はまだ持っていない。

第二に、情報システム部門による正式な審査を経た事例がない。これはAIエージェントという仕組み自体が新しく、審査する側の理解も、審査項目そのものも、まだ固まっていないためである。この状況は数年で変わるだろうが、現時点では審査を通ったという実績を主張できない。

第三に、敵対的な試験を行っていない。Parallaxが実施したような、推論システムを迂回して攻撃を直接注入する評価を、本構成に対して我々はまだ行っていない。3章で述べた性質は設計から導かれる主張であって、測定された結果ではない。

したがって本章は、有効性の証明ではない。本番で動いているという報告である。

第6章 摩擦と、その扱い

6-1 摩擦

本構成の最大の課題は、攻撃者ではなく利用者との関係にある。

人は、丸投げして結果だけを受け取りたいと思う。AIエージェントに期待されているのは「やっておいて」であって、「これでよろしいですか」ではない。ところが本構成は、不可逆群の操作が発生するたびに、利用者に手を動かすことを要求する。URLを開き、内容を読み、実行を選ぶ。

さらに5-3で述べたとおり、カタログに登録されていない操作については、できないと言われて止まる。利用者から見れば、賢いはずのものが目の前で立ち往生している。

我々はこの摩擦を、まだ解けていない。

6-2 これは我々だけの問題ではない

もっとも、この摩擦は本構成に固有のものではない。

1-5で見たとおり、CaMeLは論文の中で、利用者がセキュリティポリシーを書き保守しなければならないこと、そして確認を求められ続けた人間が何にでも同意するようになることを、未解決の限界として挙げている。Parallaxも、人間による承認の段に対して、承認疲労を突く攻撃への対策として回数制限を組み込んでいる。

安全側に倒した設計は、ほぼ例外なく、負担を人間へ移す。そして人間は摩耗する。**摩耗した承認は、承認ではない。** 押されたボタンの数は増えるが、そこで守られているものは何もない。

つまりこの問題を解かない限り、どの設計も、最後は同じところで破綻する。

6-3 素直な解と、その危険

素直な解は、繰り返し承認された操作を自動的に降格させることだろう。同じ操作を利用者が20回承認したなら、21回目からは確認を求めない。手間は減る。

ところがParallaxは、この方向を有望としつつ、明確な危険を指摘している。特定の利用者が一貫して承認する操作を学習して承認の対象を絞り込んでいく適応的なポリシーは、**攻撃者が無害に見える承認を積み重ねることで、緩い方向へ訓練できてしまう**、というものである。

そしてこれは机上の懸念ではない。1-7で触れたCursorのCVE-2026-22708において、攻撃を容易にしたのは許可リストそのものだった。安全のために作られた仕組みが、攻撃者にとっては素早く通る操作の一覧表として機能した。

安全機構が攻撃の踏み台になる。承認の履歴に基づいて権限を広げる仕組みは、この反転を構造的に抱えている。

6-4 分類の粗さが保証になる

先に現状を述べておく。**本構成に自動降格の仕組みは、まだない。** 4-5で述べたとおり、どの操作をエージェントに任せ、どの操作を人に返すかは、いまはテナントごとに人が設定している。承認の履歴によって設定が動くことはない。

仮にこの機構を入れるとすれば、承認の結果を記録し、繰り返された操作を可逆群へ移すものになるだろう。手間は減るが、Parallaxが指摘した危険と無縁ではない。

それでもこれが検討の対象になりうるのは、4-2で述べた境界の置き方があるからだ。

降格先は可逆群しかない。そして可逆群は、定義上すべて可逆な操作である。したがって、**攻撃者が承認を積み重ねて得られる最大の戦果は、可逆な操作にとどまる。** 削除や更新が、承認された回数によって可逆群へ上がることは起こらない。

より正確に言えば、こうなる。**承認の回数は、降格の必要条件ではあるが、十分条件ではない。十分条件は可逆性であり、可逆性は承認では変えられない。**

Cursorの事例との違いは、ここにある。許可リストは、この操作は通してよいという個別の記載の集合である。記載が増えれば、通る操作が増える。対して可逆性による二分は、記載ではなく性質による分類である。承認をいくら積み重ねても、削除が可逆になることはない。境界そのものが動かない。

4章で粒度の粗さを選んだのは、このためである。段階を細かくすれば、3段目から2段目への降格が依然として危険な場所への降格でありうる。二つしかないことが、降格先の安全性を保証する。

ここまでは、境界の置き方から導かれる話である。自動降格を実際に入れるかどうかは別の判断であり、我々はまだ決めていない。ただ、この境界の下では検討の対象になりうる、というところまでは言える。

6-5 それでも残るもの

正直に書いておきたいことが4つある。

第一に、可逆性の判定そのものは人間が行う。カタログへ操作を登録するとき、そして4-5で述べたとおりテナントごとに分類を決めるとき、これは可逆かどうかを判断するのは人である。その判断が誤り、実際には取り返しのつかない操作が可逆群に置かれれば、保証は根本から崩れる。構造は経路を断つが、判断の誤りは防がない。

第二に、要求を出す鍵と実行できる鍵を分ける仕組みや、実行の許可そのものを改竄できない形で受け渡す仕組みは、設計として検討しているだけで、実装していない。本稿の主張はこれらを前提としていない。

第三に、3-4で述べたソーシャルエンジニアリング、すなわち利用者を説得して押させる経路は、依然として開いている。構造化差分はその成立条件を狭めるが、閉じはしない。

そして第四に、6-1の摩擦は解けていない。仮に自動降格を入れたとしても、摩擦は減るだけでゼロにはならない。不可逆群の操作について、この設計は「やっておいて」という期待に構造的に応えられない。応えられるようにすれば、それは不可逆群でなくなる。

この取引を市場が受け入れるかどうかを、我々はまだ知らない。

第7章 結びにかえて ― 4つの問い

本稿が述べてきたのは、優れた設計ではなく、ひとつの取引である。

汎用性を手放した。任意のシェルコマンドは実行できない。有限のカタログに登録された操作しか扱えない。利用者には手間をかける。不可逆群の操作のたびに、URLを開き、内容を読み、実行を選んでもらう。

その代わりに得たものが、判定器への依存の消滅である。不可逆群について、我々のシステムの安全性は、分類器の精度にも、検証層の正しさにも依存していない。実行器を起動できるのが利用者だけだからである。

この取引が誰にとっても正しいとは思わない。だが、判定器の精度を上げ続ける競争とは別の方向があることは、示せたと思う。

最後に、読者が自分のシステムを点検するための問いを4つ挙げて終わりたい。

第一の問い。承認の手順を飛ばしたとき、エージェントはその操作を実行できるか。

できるなら、それは検問であって境界ではない。手順を飛ばす経路が存在しないことを、設計として説明できるかどうかが分かれ目になる。

第二の問い。承認画面は、何が何に変わるのかを見せているか。

操作の名前しか出ていないなら、押した人間は結果に同意していない。そして差分を生成しているのが誰かも問うてほしい。エージェントが書いた差分は、実行される内容と食い違いうる。

第三の問い。出口の許可リストを、エージェント自身が書き換えられるか。

書き換えられるなら、それは制限ではなく設定である。同じことが、監査ログにも言える。

第四の問い。承認を重ねることで、エージェントにできることが増える仕組みはあるか。あるなら、増える先はどこまでか。

手間を減らすための機構は、たいてい攻撃者にとっての目標にもなる。増える先が可逆な操作に限られているかどうかを、構造として説明できるかを確かめてほしい。

4つとも、我々が自分のシステムに対して繰り返し問うてきたものである。もっとも4つ目については、我々自身がまだ答えを持っていない。その機構を持っていないからである。

本稿は結論ではなく、途中経過である。

参考文献

※ Dual LLM パターンの原文（Willison, 2023）および CaMeL 原論文のURLは、公開前に確認して補うこと。

1. CaMeL については、Simon Willison による解説が読みやすい。 https://simonwillison.net/2025/Apr/11/camel/↩

2. Parallax の原典。査読前のプレプリントである。 https://arxiv.org/abs/2604.12986↩

3. CVE-2026-22708 の技術的詳細は、発見者であるPillar Securityの解説による。 https://www.pillar.security/blog/the-agent-security-paradox-when-trusted-commands-in-cursor-become-attack-vectors↩