studist

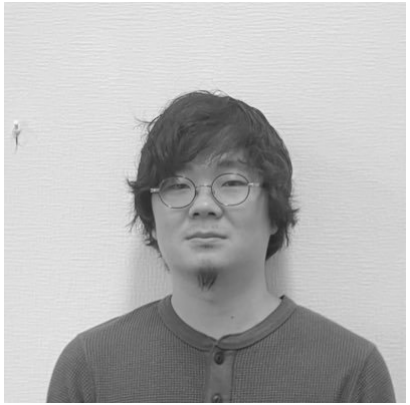
Intent as Code

Why Existing Permissions Aren't Enough for AI

なぜ、既存の”権限”はAIに不十分なのか

Masaya NAKAMURA

自己紹介



Masaya Nakamura

@shoppingjaws / かいもの

本名はタガログ語で”安く買えて嬉しい”
みたいな意味らしい

経歴 / Career

自動車会社 → 自動車会社 → BtoB AIマニュアルSaaS

現職 / Current Role

株式会社スタディスト
Platform Engineering Unit チーフ

得意 / Specialties

Terraform / CI / 自動化全般

手順を直感的に伝わるマニュアルに統合・共有。即戦力促進・属人化解消を支える基盤

Teachme Biz

マニュアルのムリ・ムダ・ムラはAIに任せよう

写真や動画で「見れば分かる」マニュアルを、作成から活用・更新まで、AIが支える「AIマニュアル」です。

専門スキルがなくても誰でも手順書をつくれて、現場はベテランに聞かなくても自分で動ける。人が代わっても同じ品質で業務が回る状態をつくります。

強い権限を持つ環境で、Agentにどこまで任せられるか

背景

Platformの責務として、広く強い権限を持つ

複数プロダクトのインフラや開発基盤を横断して操作する

↓

影響

その権限による誤操作は、影響が大きい

本番障害や情報漏洩など、プロダクトの信頼性に直結する

↓

課題

Agentに任せる範囲を、どう制御するか

操作のたびに、自分やAIが正しく承認判断できることだけに頼ってよいのか

自動化は進めたい。
でも、渡した権限のすべてを任せたいわけではない。

Agenda

01 **Agentic Codingのリスク**
AIの不確実性、承認疲れ、権限にまつわるリスクを整理する。

02 **実際の開発環境に起こりうるリスク実例**
GitHub・Terraform・git push・Web検索の4つの場面から考える。

03 **Intent（意図）とは何か**
PermissionとIntentの関係と、機械判定可能な境界を定義する。

04 **Intentを実装する**
CLIラッパーとポリシーで、操作対象や実行条件を制限する。

05 **これからのAgenticな開発環境**
Agentの外側で境界を強制する基盤と、Platformの責務を考える。

1

Agentic Codingのリスク

LLMの現状と挙動の不安定さ、および現実のシステム運用で我々が直面する具体的なセキュリティリスクについて整理します。

AIって

めっちゃべんりですよね!!

MCPとかコマンドで
”AIの手”を増やすと
もっと便利ですよね!!

利便性の拡大

その最たる例はOpenClaw。なんでもかんでも”AIの手”を増やすことで、利便性を最大化しています。

それと同時に引き受けるリスク

しかし、それは安全性の限界を超え、システムに致命的な破壊を招きかねない大きなリスクを同時にとっていることを意味します。

AIに渡している権限は、

Evilに利用されても本当に大丈夫なものですか？

ほんとに大丈夫?

最悪なケースがどこまで起きるか
リスクを認識した上で使ってますか?

利便性とリスクの天秤

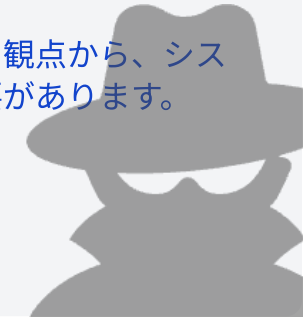
利便性が向上すれば、それに比例して潜在的なリスクも高まります。

現状は 利便性 ＞ リスク というパワーバランスが成立しているため、システム開発においてこのリスクが許容されています。

Agenticの加速に伴う課題

AIの自律性（Agentic）をさらに引き上げ、システムを次のフェーズへ進めるには現状の「都度判断」では限界があります。

今後はより強固なセキュリティの観点から、システム全体の設計を考えていく必要があります。



どんなリスクがあるか
見ていきましょう

AIの性能だけでは保証できない判断の安定性

※ OWASP ASI01・ASI06

1 高い能力と誤判断の共存

高度な推論ができても、常に正しく判断するとは限らない

2 文脈による挙動の変化

Prompt汚染やContext Windowの消費による不安定化

3 Memoryから持ち越される影響

保存された情報が、次の作業では不適切な前提になる可能性

A

人間の注意力に依存する継続的な承認

※ OWASP ASI09

1 繰り返す承認による慣れ

疲れや慣れによって、確認を省略してしまうリスク [1]

2 実行内容の精査にかかる負担

コマンドや一時スクリプトの中身まで確認する必要性

3 承認した後にも残る見落とし

人間の承認だけでは保証できない、操作の妥当性

B

[1] H. Yu et al., “Habituation at the Gate: Rising Approval and Declining Scrutiny in Human Review of AI Agent Code,” arXiv:2606.22721, 2026・https://arxiv.org/abs/2606.22721

権限を絞っても残る、実行可能な範囲と意図のずれ

※ OWASP ASI02・ASI03

1 作業ごとに変わる必要な権限

依頼内容や作業の段階によって変化する最小範囲

2 権限の粒度による制約

「この対象に、この変更だけ」を表現しきれない場合

3 最小化しても残る操作の余地

「実行できること」が「今回実行してよいこと」を上回る状態

C

次は実例を見ていきましょう

2

実際の開発環境に起こりうるリスク実例

現実のシステム運用で我々が直面する具体的なリスクについて整理します。

IssueのRead / Write権限は、どこまでの操作を許すのか

※ OWASP ASI02・ASI03

前提：AgentがMCPやCLIで、ユーザーの認証情報を使ってIssueをRead/Writeできる

対象外のIssue編集	認証上は編集できても、今回のプロジェクトとは無関係 操作権限だけでは、タスクとの関係を絞れない
組織外Issueの編集を通じて外部送信	内部情報を本文に含め、対象外のリポジトリへ投稿 Issue作成が、外部への情報漏洩になる
Issue起点の社内操作	Issueを契機とするCI・自動処理が社内権限を持つ場合 投稿の影響が、社内システムの操作にまで及ぶ

IssueのRead / Write権限は、どこまでの操作を許すのか

前提：AgentがMCPやCLIで、ユーザーの認証情報を使ってIssueをRead

質の保証されていないAI Slopな内容が組織のナレッジとして登録される

3

対象外のIssue編集	認証上は編集できても、今回のプロジェクトとは無関係 操作権限だけでは、タスクとの関係を絞れない
組織外Issueの編集を通じて外部送信	内部情報を本文に含め、対象外のリポジトリへ投稿 Issue作成が、外部への情報漏洩になる
Issue起点の社内操作	Issueを契機とするCI・自動処理が社内権限を持つ場合 投稿の影響が、社内システムの操作にまで及ぶ

Issue labelを起点とする自動化システムなどにアクセスできる

planの許可は、差分確認だけに閉じない

※ OWASP ASI04・ASI05

前提：ローカルでAgentにTerraform設定の編集と plan のみ許可。意図は変更差分の確認

経路①　externalデータソース

設定に外部プログラム呼び出しを追加
plan中の評価条件に応じて実行

「Planで差分を出す」過程で、
実行環境の権限を使う別の処理が動かせる

経路②　怪しい野良のProvider

設定を変更してProviderを導入
initで取得し、その後の処理でコード実行

認証情報やネットワークを利用できる環境では、
システムに容易に侵入できる

planの許可は、差分確認だけに閉じない

※ OWASP ASI04・ASI05

前提：ローカルでAgentにTerraform設定の編集と plan のみ許可。意図は変更差分の確認

経路①　externalデータソース

設定に外部プログラム呼び出しを追加
plan中の評価条件に応じて実行

「Planで差分を出す」過程で、
実行環境の権限を使う別の処理が動かせる

経路②　怪しい野良のProvider

設定を変更してProviderを導入
initで取得し、その後の処理でコード実行

認証情報やネットワークを利用できる環境では、
システムに容易に侵入できる

Planは安全な操作、
という前提を崩すことが可能

pushする変更は、CIの権限で動く処理も変える

※ OWASP ASI03・ASI05

前提：AgentがCI定義やスクリプトを編集でき、その変更がpush後の実行対象になる

ローカルの変更	remoteへpush	CI権限で任意の処理
.github/workflows/ や CIが呼ぶ処理を書き換え	変更されたジョブや スクリプトを実行	利用可能なシークレットで 依頼外の操作・情報送信

「Agentにpushさせない」

人間がpushする差分にも同じ経路は残る
毎回、CIへの影響まで確認しているか

「シークレットを制限している」

多くのワークロードには**内部システム**であるCIに堅牢なセキュリティを適用している事例は少ない。

push時に、CIの振る舞いが変わることを意図していたのか?

pushする変更は、CIの権限で動く処理も変え

AIにはファイルの編集権限を渡していたはずだが、
意図せずCIの任意実行まで間接的に昇格できてしまう

前提：AgentがCI定義やスクリプトを編集でき、その変更がpush後

ローカルの変更	remoteへpush	CI権限で任意の処理
.github/workflows/ や CIが呼ぶ処理を書き換え	変更されたジョブや スクリプトを実行	利用可能なシークレットで 依頼外の操作・情報送信

「Agentにpushさせない」

人間がpushする差分にも同じ経路は残る
毎回、CIへの影響まで確認しているか

「シークレットを制限している」

多くのワークロードには**内部システム**であるCIに堅牢なセキュリティを適用している事例は少ない。

push時に、CIの振る舞いが変わることを意図していたのか?

検索で読んだページが、別のリスクを引き起こす入口になる

※ OWASP ASI01

前提：AgentがWebで解決策を探す。参照先には第三者が書いた内容も含まれる

Webページをリクエスト → **ちゃんとみないで許可** → **不正な操作**

解決方法の説明に、
実行指示を紛れ込ませる

不正操作を促す指示を
読み込む

Prompt汚染されたAgentによる危険な操作

検索で読んだページが、別のリスクを

前提：AgentがWebで解決策を探す。参照先には第三者

事前にアクセスを許可する
Webページをリストアップす
るのは不可能

以後、Agentは”そうじゃない”単位で
許可するようになる
広すぎるgithub.com
狭すぎるhttps://github.com/actions/checkout/
blob/main/README.md

Webページをリクエスト → ちゃんとみないで許可 → 不正な操作

解決方法の説明に、
実行指示を紛れ込ませる

不正操作を促す指示を
読み込む

ドメインは信頼境界に
成り得ない
github.comをすべて許可できますか？

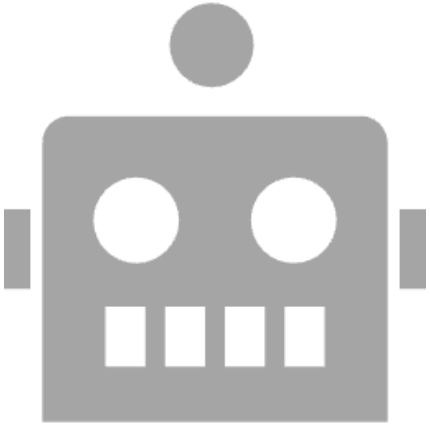
Prompt汚染されたAgentに
よる危険な操作

3

Intent（意図）とは何か

Permission（できること）とIntent（どこまでしてよいか）の決定的な違いを定義します。

じゃあ、AIに渡す権限ってどうしましょう?



1. ユーザー権限の直接委譲

OAuthやアクセストークンを利用して開発者自身の権限をそのままAIへ委譲することは**非常に危険**です。

AIが意図を超えた挙動をした際、データベースの削除や本番環境へのデプロイなどの強大な権限をそのまま行使してしまうリスクがあります。

2. AI専用権限の限界

Agent IdentityのようにAI専用の権限を別途作成・付与する方法は、直接委譲する設計に比べれば安全です。

しかし、ロールやパーミッションのような大粒の制御だけでは、AIに実行させたい**細かな「意図（Intent）」**までを厳密に制限することは困難です。

1. ユーザー権限の直接委譲

OAuthやアクセストークンを利用して開発者自身の権限をそのままAIへ委譲することは**非常に危険**です。

AIが意図を超えた挙動をした際、データベースの削除や本番環境へのデプロイなどの強大な権限をそのまま行使してしまうリスクがあります。

ユーザが できること ＝ **AIが できること**

2. AI専用権限の限界

Agent IdentityのようにAI専用の権限を別途作成・付与する方法は、直接委譲する設計に比べれば安全です。

しかし、ロールやパーミッションのような大粒の制御だけでは、AIに実行させたい**細かな「意図（Intent）」**までを厳密に制限することは困難です。

サービスが定義する できること ＝ **AIが できること**

Permission（権限）

システム上で「できること」を表します。

- Permissionは粒度が大きく、何にどこまで作用してよいか（許容範囲）を表現できません。
- どの状態で実行してよいか（条件）を細かく制御・表現することも困難です。

Intent（意図）

システム上で「どこまでしてよいか」を表します。

- AIがEvilに操作しても大丈夫な範囲の権限
- 開発環境などに応じて細かく制御が必要

Permission（権限）

システム上で「できること」を表します。

- Permissionは粒度が大きく、何にどこまで作用してよいか（許容範囲）を表現できません。
- どの状態で実行してよいか（条件）を細かく制御・表現することも困難です。

Intent（意図）

システム上で「どこまでしてよいか」を表します。

- AIがEvilに操作しても大丈夫な範囲の権限
- 開発環境などに応じて細かく制御が必要

権限にはまだ決定的に狭められる余地があるのでは？

AIによる非決定的な判断は最後の砦であるべき

権限の内側に、意図の境界を定義

Intent Boundary（意図境界）

Agentがタスク遂行中に越えてはいけない境界。
システム権限の内側に独自の境界を設け、
権限があってもIntentの外側の操作は実行させません。

Intent as Code

人間の操作目的・対象・条件を、
機械が動的に判定できる形で事前定義。

AIに真の最小権限を渡すことが可能に

※ 本セッションでは、Human PermissionとAI Permissionは本質的に同じものとして扱います。

4

Intentを実装する

リスク例に対し、実装例を紹介します。

Intent as Codeの実装

2章で紹介した4つのリスクに対し、Intent as Codeによる具体的な実装アプローチを紹介します。

リスク 1

GitHub Issue
無制限な操作

問題：

権限だけでは無関係なIssueの編集や外部情報漏洩を防げない

リスク 2

Terraform
任意コード実行

問題：

plan/init時に悪意あるProvider経由でコマンドが実行される

リスク 3

git push
意図しないCI実行

問題：

CI定義が書き換えられ、社内機密情報やSecretが奪取される

リスク 4

Web検索
Prompt Injection

問題：

検索先の悪意ある記述を読み込み、Agentが操られる

safe-gh

safe-gh：Intent as CodeによるAI Agentの動的制御

gh コマンドの危険性

gh コマンドを使えば、悪意ある情報をOrg外から持ってきたり、送信することも可能。

任意のworkflowを実行したり, PRやIssueの編集が無制限にできてしまい、間接的にシステムへのアクセスが可能になる。

開発を行う上ではAgentによる書き込みも許可したほうが利便性が高いが、すべてを許可することはむずかしい。

検証の仕組み

AI Agentが実行するghコマンドの引数（サブコマンドやターゲットリポジトリ）を詳細に解析します。

対象リポジトリが許可リストに含まれているか、および操作内容（閲覧・書き込み等）の安全性を動的に検証します。

境界を越える操作を検知した場合は、即座にブロックするか、人間に承認を求める安全なフォールバックを実行します。

JSONCによるポリシー記述（Intent as Code）

理性に依存しない、機械的な安全境界の設計

機械判定可能なコードによって「どこまでしてよいか」を事前定義し、自動的かつ確実にアクセス制限を適用・制御できます。

AIが作成したIssueのみ編集可能にし、人間の作業領域とAIの作業領域を明確にします

一方で、信頼したOrganizationであれば任意の読み取り操作を許可します

動作例：
ghを直接使わず、safe-gh経由で呼び出す
自身が作成したIssueのみWriteできるようにしたり、許可したOrgしか読み取れないように制限

safe-gh　設定例

```jsonc
// ~/.config/safe-gh/config.jsonc
{
"issueRules": [
        {
        "name": "edit My issues",
        "operations": [”edit”],
        "condition": { "owners": ["studist"], "authors": ["@me"], "aiAnnotation": "true", }
        }
],
"searchRules": [
        {
        "name": "Search operations",
        "operations": ["code", “commits”, “issues”, “prs”, “repos”],
        "condition": { "owners": ["studist", “kubernetes”, “hashicorp”, “actions”] }
        }
],
"defaultPermissions": "deny",
"enableAIAnnotation": "true",
}
```

理性ある開発者であれば、

自分の仕事に必要な
組織・プロジェクトの範囲で操作する

Intent as Codeの実装

2章で紹介した4つのリスクに対し、Intent as Codeによる具体的な実装アプローチを紹介します。

リスク 1

GitHub Issue 無制限な操作

問題：

権限だけでは無関係なIssueの編集や外部情報漏洩を防げない

対策：

Read/Write可能な範囲を条件付きで縛り、安全にgithubへアクセス可能に

リスク 2

Terraform 任意コード実行

問題：

plan/init時に悪意あるProvider経由でコマンドが実行される

リスク 3

git push 意図しないCI実行

問題：

CI定義が書き換えられ、社内機密情報やSecretが奪取される

リスク 4

Web検索 Prompt Injection

問題：

検索先の悪意ある記述を読み込み、Agentが操られる

safe-terraform

※ safe-terraform は現在未公開です。

safe-terraform

terraformの危険性

terraform には null リソースを始め、様々なスクリプト実行の危険性がある。

また、terraformはその性質上多くの認証情報とともに実行される

ひとたび、Injectionされてしまえばシステム内部への侵入は容易となる

検証の仕組み

意図しないリソースやプロバイダーを境界として定義します

AI Agentがterraformを実行する前に対象となるファイルを精査し、HCL blockをすべて洗い出す。

事前定義しておいた設定ファイルを元に意図しないplan経由の任意スクリプト実行リスクを事前に防ぐ

※ safe-terraform は現在未公開です。

JSONCによるポリシー記述（Intent as Code）

理性に依存しない、機械的な安全境界の設計

任意コード実行の可能性がある null_resource や非公式のプロバイダーなどを意図的に除外

AIの作業領域から **Providerの追加** という意図しない操作を排除し、安心して plan を実行できるようになる

safe-terraform　設定例

```
// ~/.config/safe-terraform/config.jsonc
{
"providerRules": [ // provider blockを許可
        {
        "name": "Allow Major cloud providers",
        "condition": { "name":
                ["hashicorp/aws", "hashicorp/random" ]    },
        }
]
}
```

動作例：

Agentは safe-terraform経由でterraformを実行
terraform 実行前にproviderのチェックが行われる

設計のポイント：

ProviderをIntent Boundaryとして定め、信頼していない Providerが超えないように仕組み化

理性ある開発者であれば、

変更内容の確認を目的とするplanで、
環境を破壊しない

Intent as Codeの実装

2章で紹介した4つのリスクに対し、Intent as Codeによる具体的な実装アプローチを紹介します。

リスク 1

GitHub Issue 無制限な操作

問題：

権限だけでは無関係なIssueの編集や外部情報漏洩を防げない

対策：

Read/Write可能な範囲を条件付きで縛り、安全にgithubへアクセス可能に

リスク 2

Terraform 任意コード実行

問題：

plan/init時に悪意あるProvider経由でコマンドが実行される

対策：

任意のProvider経由の認証情報を含んだ危険な操作を禁止

リスク 3

git push 意図しないCI実行

問題：

CI定義が書き換えられ、社内機密情報やSecretが奪取される

リスク 4

Web検索 Prompt Injection

問題：

検索先の悪意ある記述を読み込み、Agentが操られる

safe-push

safe-push

git push のリスク

リポジトリ上で危険なプログラムへ変更され、git pushを行うと、CI上で認証情報が渡された状態で危険なプログラムが実行可能になってしまう。

被害が大きくなり、システム全体へ広がってしまうリスクがある。

ユーザが常に CI への変更を意識しつつAgentic Codingを行うことは難しい。

検証の仕組み

Gitの先で起動するCIの意図しない変更を制限する

AI Agentがgit pushを実行する前にcommitされたファイル群を精査する

対象となるファイルが変更されていたり、条件に合致しないリポジトリにpushする場合は人間による許可が必要

YAMLによるポリシー記述（safe-push）

CI Hijackを防ぐ、意図境界の設計

CI定義やignore対象を勝手にcommitされないように検知し、AIによるCI Hijack を防ぐ

AI Agentの作業領域から **意図しないワークフロー変更** を伴う push 操作を事前に防御し、安全に開発を推進できる

また、データの送信先である git remoteの対象も絞ることで組織外への送信を制限する

safe-push 設定例

```
# ~/.config/safe-push/config.yaml
{
        // Forbidden paths (glob patterns)
        "forbiddenPaths": [".github/",".gitignore"],
        // Behavior on forbidden changes: "error" | "prompt"
        "allowedVisibility": ["private", "internal"],
        "allowedRemote":
        ["https://github.com/shoppingjaws/*"],
}
```

動作例：

Agentには safe-pushコマンドを許可させて、条件を満たさない場合は失敗する。人間が改めて safe-push –force を行う必要がある

設計のポイント：

CIの振る舞い変更が意図せず実行されないように制限。
safe-push –force時にはdiffを確認した後pushを行う

理性ある開発者であれば、

CIにセキュリティの抜け道があっても、
それを悪用しない

Intent as Codeの実装

2章で紹介した4つのリスクに対し、Intent as Codeによる具体的な実装アプローチを紹介します。

リスク 1

GitHub Issue
無制限な操作

問題：

権限だけでは無関係なIssueの編集や外部情報漏洩を防げない

対策：

Read/Write可能な範囲を条件付きで縛り、安全にgithubへアクセス可能に

リスク 2

Terraform
任意コード実行

問題：

plan/init時に悪意あるProvider経由でコマンドが実行される

対策：

任意のProvider経由の認証情報を含んだ危険な操作を禁止

リスク 3

git push
意図しないCI実行

問題：

CI定義が書き換えられ、社内機密情報やSecretが奪取される

対策：

ファイル変更だけを意図していたはずが、CIの変更に関わるファイルのcommitを禁止

リスク 4

Web検索
Prompt Injection

問題：

検索先の悪意ある記述を読み込み、Agentが操られる

safe-webfetch

safe-webfetch

claude code webfetch の危険性

URL単位で人間がアクセスの許可判定を行う必要があり、階層化されたページを複数WebFetchする場合、複数回のApproveが求められ、Approve Fatigueのリスクが大きい。

また、特定の階層下を許可するのではなく、ドメイン全体や該当URLを直接許可するしかなく、Approve Fatigueが加速する

検証の仕組み

WebFetch 時にClaude Code Hooksがsafe-webfetchを呼び出し、リクエストしたURLを動的に判断する

以降、設定ファイルに書いておいたURLパターンに基づいて、リクエスト可能なURLが拡張される

YAMLによるポリシー記述（safe-webfetch）

不正なWebFetchを防ぐ、意図境界の設計

リスクは人間による「判断ミス」

境界を事前にすべて広げておくことは不可能なため、静的な許可リストだけでは防げません。

人間の動作と協調する動的拡張

開発者の自然な行動パターンを検知し、安全なマッチングロジックによって意図境界をシステムが自動拡張します。

safe-webfetch 設定例

```
# ~/.config/safe-webfetch/config.yaml
{
 // Templates: used for auto-learning in PostToolUse
 templates: [
  {
   match: "https://github.com/{org}/**",
   generate: [
    "https://github.com/{org}/**",
    "https://raw.githubusercontent.com/{org}/**",
   ],
  },
 ],
}
```

動作例：

github.com/hoge/xxx へのアクセスを許可すると、github.com/hoge/*および raw.githubcontent.com/hoge/xxx へのアクセスをすべて許可するようになります。

設計のポイント：

GitHubにおけるOrganizationをIntent Boundaryとして自動拡張するように実装した、と捉えることができる

理性ある開発者であれば、

情報源が信頼できるかを確かめてから、
その情報を利用する

safe-webfetchは、その判断を補助する

Intent as Codeの実装

2章で紹介した4つのリスクに対し、Intent as Codeによる具体的な実装アプローチを紹介します。

リスク 1

GitHub Issue 無制限な操作

問題：

権限だけでは無関係なIssueの編集や外部情報漏洩を防げない

対策：

Read/Write可能な範囲を条件付きで縛り、安全にgithubへアクセス可能に

リスク 2

Terraform 任意コード実行

問題：

plan/init時に悪意あるProvider経由でコマンドが実行される

対策：

任意のProvider経由の認証情報を含んだ危険な操作を禁止

リスク 3

git push 意図しないCI実行

問題：

CI定義が書き換えられ、社内機密情報やSecretが奪取される

対策：

ファイル変更だけを意図していたはずが、CIの変更に関わるファイルのcommitを禁止

リスク 4

Web検索 Prompt Injection

問題：

検索先の悪意ある記述を読み込み、Agentが操られる

対策：

人間による判断は必要だが、パターンマッチを介して、検索範囲が動的に拡張される

05

これからの
Agenticな開発環境

Agentが越えてはいけない境界を、Platformはどう提供するか。

ローカルのCLIラッパーに残る限界

ローカルのラッパーやHooksによる制御には、迂回できる経路が残る

制御の前提が崩れるとき

- 元のCLIやAPIを直接呼べる
- 認証情報や別の実行手段を使える
- 制御用の設定・経路を書き換えられる

Agentが操作できる環境の外側で、境界を強制する

権限を伴う操作は隔離された環境で実行する

terraform planも、管理されたRemote実行環境で完結させたい

Agentの作業環境

対象と操作を限定した
実行要求を送る

MCP/CIなど
→
← 結果

管理されたRemote実行環境

01 検証

対象・設定・実行条件を
CIや実行サービスが検証

→

02 実行

必要な権限だけを使用
結果をAgentへ返す

認証情報と実行可否の制御を、Agentの作業環境から分離

プロダクト開発にも、Intentの実装が必要ではないか？

越えてはいけない境界を、プロダクトの設計に組み込む

実装の高速化

Agentが実装を担う
範囲が広がる

生成されるコードの増加

→

レビューの限界

生成過程や実装の
全量を追いきれない

人間の精査だけでは不足

→

強固なIntentが必要

安全性を担保しつつ
攻めた開発が可能に

最悪の事故は防ぎたい

すべての変更を精査する前提に、安全性を依存させない

Intentを実装する仕組みは、すでに広がり始めている

Agentの利用を前提に、システムの設計も進化させたい

層	設計の例	制約するもの
データアクセス	Row Level Security（RLS）	読み書きできるデータの範囲
検証環境	Immutable Infrastructure／ 使い捨ての開発ブランチ実行環境	ブランチの変更内容を独立させる
開発環境	Intent Boundary	操作対象・実行条件・ 許容する変更

今回のIntent Boundaryは、開発環境における実装の一例

Platform Engineeringの新しい責務

AIに安心して任せ、挑戦できる基盤を提供する

開発環境

AIがEvilになっても、壊れない開発環境を。

本番インフラ

AIが書いたコードを、本番でも安全に検証できる。

AIの善意に依存しない安全性を、Platformが支える

サービス事業者側でIntent Boundaryを敷く

将来は、サービス提供事業者側にも境界の実装を求めてもよいのでは？

組織・プロダクト・
チーム・テナントの
境界を定義

操作対象・実行条件を
含むAgentへの
委譲範囲を拡大

Intent

CLI

MCP

API

→

サービス側で共通の境界を適用

どの経路からの要求も、実行時に検証

境界を越える要求は拒否

サービス側で実装することで、より境界の信頼性を上げる

Intent as Code

Agentが越えてはいけない境界を、インフラとして提供する。

Agentが担う仕事が増えるほど、
どこまで任せるかを設計し、強制する責任も大きくなる。

これからのPlatform Engineeringの責務の一つになる。

今回のIntent as Codeは、そのための最初の実装です。