セルフサービスから自動運転へ
AIネイティブ・プラットフォーム
エンジニアリングの実現方法

Kazuto Kusama
@jacopen

P	Product Evangelist	@PagerDuty Japan
CNIA	代表理事	@一般社団法人クラウドネイティブイノベーターズ協会
SC	理事	@一般社団法人SREコネクト

DASA PLATFORM ENGINEERING CERTIFIED

DASA PLATFORM ENGINEERING ACCREDITED TRAINER 2026

Platform Engineeringとは

プラットフォームは、**開発者や運用者に役立つ機能**、たとえばセルフサービスのAPI、ツール、サービス、ナレッジ、サポートを「**魅力的な社内プロダクト**」として整備した基盤のこと。

プラットフォームは、サポートしようとするシナリオに関連して、「デベロッパープラットフォーム」「デリバリープラットフォーム」「アプリプラットフォーム」あるいは「クラウドプラットフォーム」と名付けられることもある。

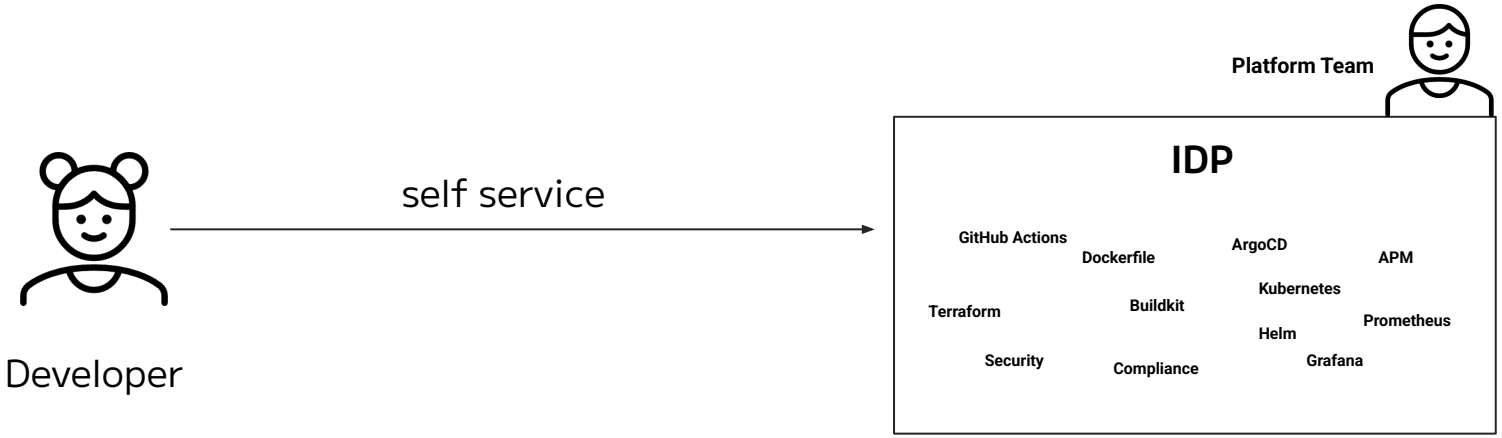
プラットフォームは、**共通の機能を提供し管理する**ことで、開発者やオペレーターがアプリケーションやサービスを**迅速に提供することを可能にする。**
プラットフォームは、プラットフォーム利用者とプラットフォーム機能提供者の間の橋渡しをし、プラットフォームチームによって構築・維持される。

この定義に該当するプラットフォームを設計・構築する分野

Internal Developer Platform

プラットフォームチームによって構築される、開発者のセルフサービスによる利用を可能にする基盤。

さまざまな技術やツールによって構成され、コンテキストや基礎となる技術を抽象化することなく、開発者の認知負荷を軽減していく




perplexity
GROK
Claude Code
Codex
Gemini
CURSOR
大AI时代

従来のプラクティス — ゴールデンパス

ベストプラクティスや標準化されたプロセスにより、開発チームが迷わず素早く安全にアプリケーションを開発・運用できるようにした一連のルートや仕組みのこと

- 開発者の認知負荷を減らす
- セキュリティやコンプライアンスを自然と担保する
- 開発チームがビジネス価値の創出に集中できるよう支援する

例えば

- 新規サービス生成時に使えるテンプレート
- テストやCI/CDパイプラインのセットアップスクリプト
- モニタリングやログ収集などの運用機能のガイド

それ、AIで出来るよ

```
> これからAKSを使ってKubernetesクラスタを作成し、そこにアプリケーションをデプロイします。
  Terraformを使って、AKSおよびNetworkなどの必要なAKSリソースを作成してください。まずはスモールスタートで始めたいので、小さめのリソースで作成してください。ファイルは
  terraformディレクトリに作成してください
  また、作成したAKSクラスタにはArgoCDをデプロイし、GitOpsのアプローチでアプリケーションをデリバリーできるようにします。manifestディレクトリを作成し、Helmを使ってArg
  oCDをデプロイできるようにしてください。

● AKSクラスタとArgoCDのセットアップを行います。まず、タスクを整理します。

● Update Todos
  └ □ Terraformディレクトリを作成し、AKSクラスタのTerraformコードを作成
    □ manifestディレクトリを作成し、ArgoCD用のHelmデプロイメント設定を作成
    □ README.mdにデプロイ手順を記載
```

```
● AKSクラスタとArgoCDのセットアップが完了しました。

  作成した内容：

  Terraformディレクトリ：
  - main.tf: AKSクラスタ、VNet、サブネットの定義
  - variables.tf: 変数定義（スモールスタート向けのデフォルト値設定済み）
  - outputs.tf: 出力値定義
  - terraform.tfvars.example: 設定例

  manifestディレクトリ：
  - argocd-namespace.yaml: ArgoCD用ネームスペース
  - argocd-values.yaml: ArgoCDのHelm設定（リソース制限付き）
  - deploy-argocd.sh: デプロイスクリプト
  - sample-app.yaml: サンプルアプリケーション定義

  特徴：
  - Standard_B2sの小さなインスタンスサイズ
  - 1〜3ノードのオートスケーリング
  - ArgoCDのリソース制限設定
  - GitOpsによる自動同期設定

  READMEに記載の手順に従ってデプロイできます。
```

従来のプラクティス — 抽象化レイヤー

- プロビジョニング抽象化
  - Terraformモジュールとかサービスカタログとか
- ランタイム抽象化
  - Kubernetes + tools
  - Application Platform
- デプロイメント抽象化
  - CI/CDテンプレート
  - アプリケーションマニフェストテンプレート
- 運用抽象化
  - Runbook as code
  - SLOテンプレート
  - アラートポリシーセット

それ、AIで出来るよ

- AI時代だと、面倒なコーディングの多くをAIが代行してくれるため、独自抽象化レイヤーの需要が減る
- むしろ独自の抽象化レイヤーが、AIによる自動生成の妨げになってしまう可能性も

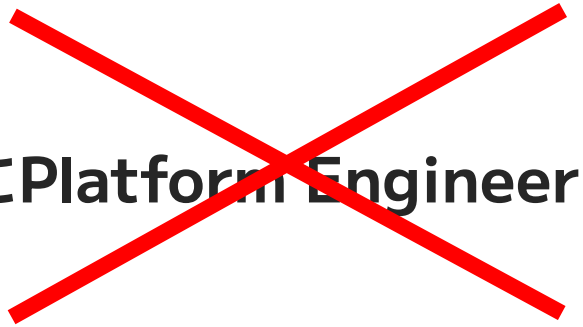
```
> オブザーバビリティとしてNew Relic、インシデント管理にPagerDutyを利用しています。
  推奨されるSLOの設定をTerraformを使ってNew Relicに設定出来るようにしてください。
  また、New RelicからのアラートをPagerDutyで受け取り、担当のチームがオンコールを受けられるようにしてください。

● New RelicとPagerDutyの設定をTerraformで管理できるようにします。

● Update Todos
  L  ☐ Create New Relic provider configuration and SLO settings
     ☐ Create PagerDuty provider configuration and team/schedule setup
     ☐ Configure New Relic alert policies and conditions
     ☐ Set up integration between New Relic and PagerDuty
     ☐ Create example SLO configurations for common services
```

つまり、AI時代において
Platform Engineeringは
従来のやり方が通用しなくなってくる

AI時代にPlatform Engineeringは不要？



AI時代だからこそ
Platform Engineeringが必要

AI時代だからこその問題

- AIにより開発はボトルネックではなくなった
  - 一方で他の要素がボトルネックになりつつある
  - 設計
  - レビュー
  - **デリバリー**
- ボトルネックを下流(デリバリー、運用)に押しつけているのが現状

AIは良いところも悪いところも増幅する

インテグレーション・デプロイ・運用が新たなボトルネック

- フィーチャーブランチのスループットは平均59%増
- メインブランチのスループットは7%減少
- メインブランチのCIの成功率は70.8% (推奨値は90%)

The delivery gap is the strategy gap
https://www.thoughtworks.com/en-us/insights/blog/generative-ai/a-thoughtworks-perspective-on-circleci-s-2026-state-of-software-

AI は鏡であり、増幅器である

State of AI-Assisted Software Development
https://cloud.google.com/resources/content/2025-dora-ai-assisted-software-development-report

- 優れた文化・プロセスを持っている組織は、AIでその強さを増幅できる
- ボトルネックがある組織は、AIによってより顕在化する

→ だからこそ、優れたプラットフォームに取り組む必要がある

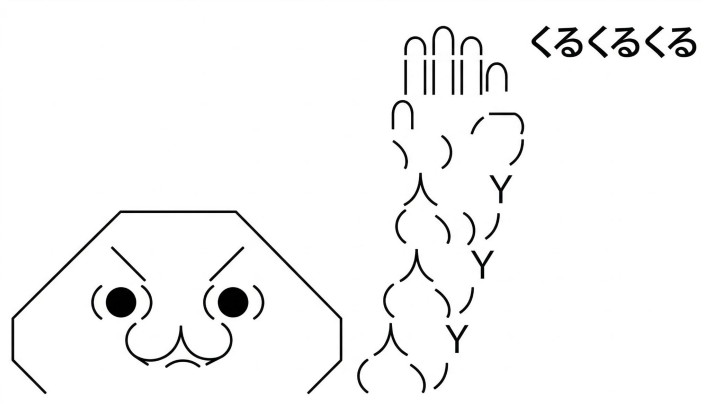
AIの進化が早すぎる問題

毎日のようにさまざまなモデルやツールが出てくるが、結局どうすればええんや問題

AIツールの進化が新たな認知負荷に

個人の趣味ならそれでもいいが、組織として取り組むには選択肢がコロコロ変わることは普及の阻害要因になる

→ AI CoEとしてのプラットフォーム



Platform Engineering Kaigiで実証実験します

9/26(土)
Platform Engineering Kaigi 2026

プラットフォームエンジニアリングの
カンファレンス。今年のテーマは
「**AI Native Platform Engineering**」

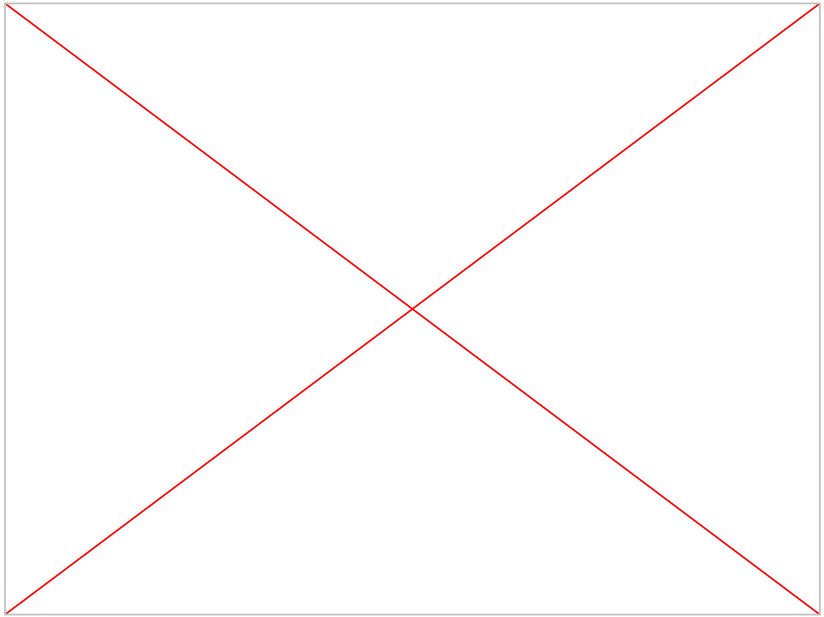
このイベントで、実行委員がつくったAI
ネイティブプラットフォームである
「Platform City」を参加者に公開しま
す。

Platform Engineering Kaigiで実証実験します

AIをフル活用したプラットフォームはこうなるはず！　というアイディアをふんだんに盛り込んだプラットフォームを提供。

イベント参加者はコーディングエージェント等で自由に開発し、Platform City上にデプロイができます。

デプロイされた参加者のアプリは、UI上で建物として表現され、それらが連携し合って一つの街を創りあげていく視覚化を取り入れています。



仮説: AIの力を借りればオンボーディング爆速では

- 新たに着任した人のオンボーディングを半日以下に短縮できるのでは
  - スキル
  - MCP
- 「このチームにオンボーディングしたい」とAgentに言えば、あとは良い感じにやってくれる

仮説: ガードレールこそがAI時代の根幹では

- 「これを使え」という型を強制するのではなく、「ここまでの範囲なら好きにやっていいよ」というガードレールを作ることが重要では
- Policy as Codeの3層構造
  - 生成時
    - エージェントに供給するContextとInstructionで制御
  - 計画時
    - Checkov, Sentinel, tflint
  - 実行時
    - OPA, Azure Policy
- 統制の単位がArtifactからActorへ
  - AIエージェントは成果物を検査しても実行時の挙動を保証できないため、統制の単位を「何がデプロイされたか」から「誰が・何の権限で・何をしたか」へ移す

→ Platform CityでもプラットフォームにAIを考慮したガードレールを作成

Self-ServiceからSelf-Drivingへ

- これまでIDPに求められていたのは、Self-Service要素だった
  - ポータルから必要なものをクリックしてパラメータを入れると
    セルフサービスでリソースを調達できる
  - 提供されるテンプレートを元にカスタマイズ
  - 抽象化により、対象への知識がなくても自身で調達ができるようになる
- AIに「ボタンは要らない」
  - 多少コードの記述量が多くてもAIは苦にしない
  - 汎用的な知識は豊富に持ち合わせているため、抽象化を行わなくても
    スムーズに動ける
  - むしろ十分なコンテキストが共有されない独自の抽象化のほうがAIにとって難しい

⇨ AI時代に求められるのは、Self-ServiceなPlatformではなく、AIが自ら判断して自由に動ける**Self-Driving**なPlatform

仮説: GitOpsは必ずしも必要ではないのでは

- Kubernetes登場以降、GitリポジトリをSSoTとしてインフラおよびアプリケーションを管理する方法が有効とされた
  - Gitを見れば状況が分かる
  - Pull Requestを通じて品質の担保
- AI Native時代では必ずしも有効ではない可能性
  - AIでオンボーディング→リソース自動生成　とするとき、わざわざGitにコミットを積むのは無駄が多い
  - 人間による承認ゲートがAI時代のいちばんのボトルネック
  - Gitを経由せず、ガードレールの範囲内でAPIを直叩き or MCP経由 で良いのでは

→ Platform Cityはk8sだがユーザーのオンボーディングに GitOpsは取り入れず、APIを叩かせる

仮説: 過度な抽象化は不要では

- たとえばTerraformのモジュールなど、複数のリソースをパッケージにすることで認知負荷を下げる取り組みは不要では
  - AIに任せるのであれば、あえてパッケージ化せずにそれぞれのリソースを作らせても負担ではない
  - コンテキストが不足することにより正しい動作の妨げになる可能性
  - むしろ抽象化が無い方が小回りが効くし既存ノウハウも使える
- 一方で、決定論的な挙動を期待するための抽象化は必要かもしれない
  - AIに任せると、時と場合によって作ったり作らなかったりということがあり得る
  - 抽象化レイヤーを挟むことで確実な挙動が期待できる
- ガバナンスの実装点としての抽象化
  - GuardrailとPolicyが有効な抽象化により、適切なAIの制御ができる可能性

→ **抽象化が不要になるわけではないが、目的が変わってくる可能性**

仮説: IaCの役割が変わってきたのでは？

- 従来のIaCは、コード化することにより以下のメリットを得ようとしていた
  - 自動化
  - バージョン管理
  - 人間による承認ゲート
  - 宣言的デプロイ
  - ドリフト検知とレビュー可能性
- AI時代においてもIaCは重要だが、主目的が変わってきた
  - エージェントが生成する中間表現
    - エージェントとしても、決まった表現があったほうが動きやすい
  - 非決定性を統制するための仕組み

→ 引き続きIaCは重要だが、従来の運用に囚われすぎないようにする

仮説: LLMがガッツリ組み込まれたプラットフォーム

- プラットフォーム自体にLLMをガッツリ取り入れて高度な運用をできるようにするのもあり？
  - コードレビュー
  - 監査
  - ガバナンス
  - 運用フィードバック
  - 障害対応
- プラットフォーム内でやるべきか、外部でやるべきかの見極めが重要

まとめ

- AI開発の普及により、従来のプラットフォームエンジニアリングではカバーしきれない領域が出てきている
- プラットフォームの存在価値は高く、AI開発に特化したプラットフォームが求められている
- これまで良いとされてきたSelf-Serviceなプラットフォームから、AIが自身で判断して動けるSelf-Drivingを実現するプラットフォームへの変化が求められる