
aws
SUMMIT
TOKYO | APRIL 20-21, 2023

AWS-12

プロトタイピングのススメ 手早くサービスを作って検証するための実践的ノウハウ

友岡 雅志
アマゾン ウェブ サービス ジャパン合同会社
デジタルトランスフォーメーション本部 プロトタイピングエンジニア

本日の内容について

お話すること

✓ AWS でプロトタイピングを効率的に行うノウハウ

✓ 業界に依らず普遍的に役立つ知識が多めです

お話しないこと

× 特定のユースケースに対する具体的技術の紹介・比較

前提知識

- モダンな開発手法に関する基礎的な知識 (CI/CD、自動テストなど)
- AWS サービスを使った開発の基礎的な経験

自己紹介

Masashi Tomooka

Prototyping Engineer, Amazon Web Services Japan

経歴

(大学) 人工衛星の開発運用

(前職) モバイルゲームクライアントの開発

→ モバイルゲームサーバーサイドの開発運用

(現職) お客様のプロトタイプ開発支援

tmokmss

プロトタイピングとは (本セッションでの定義)

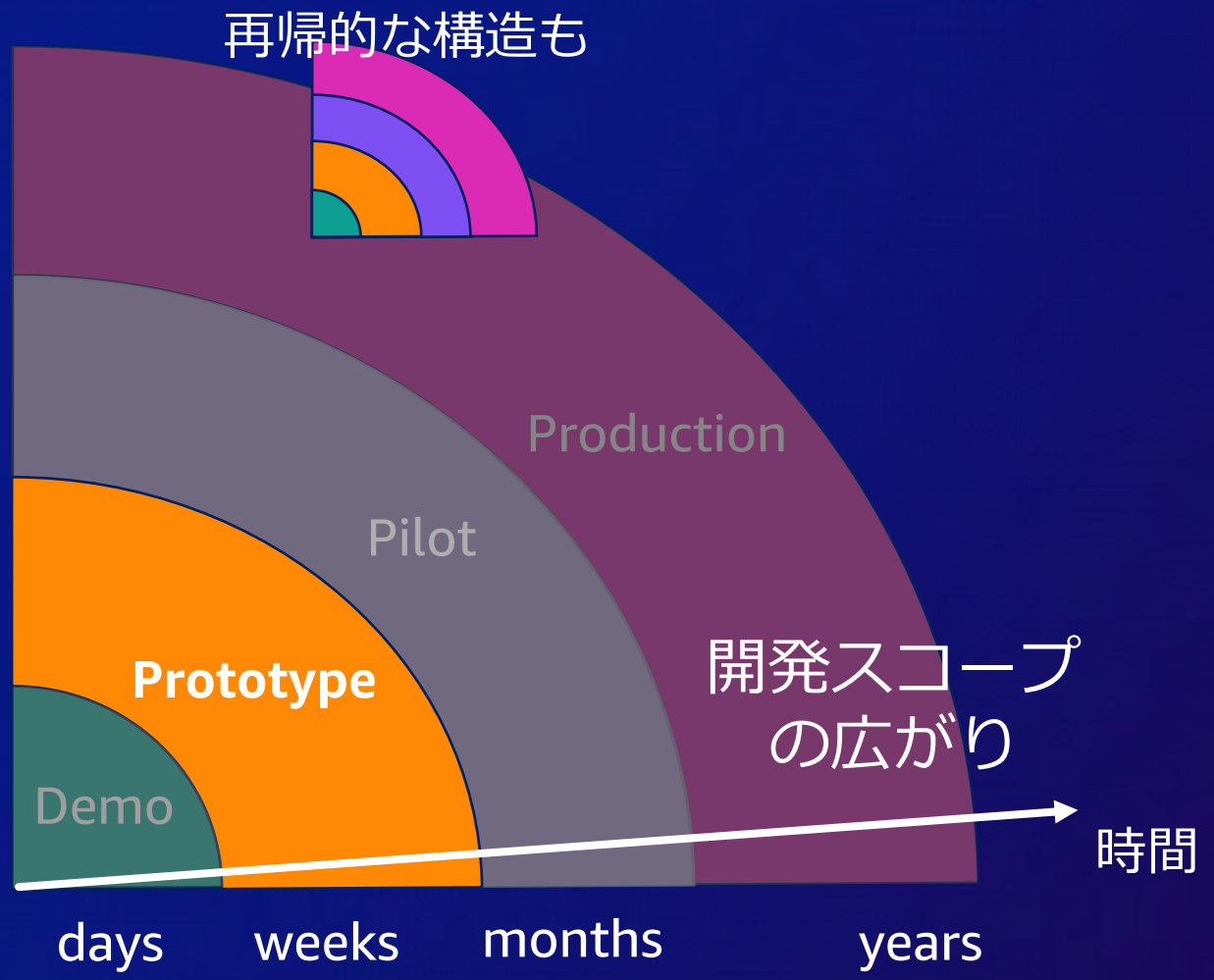
プロトタイピング = プロトタイプをつくる営み

実際にリリースされるシステム

限定された規模のユーザーで検証する

最重要のスコープに限り
短期間で必要な検証を行う

サービスの実用性を示す最低限のシステム
時にコードすら伴わない



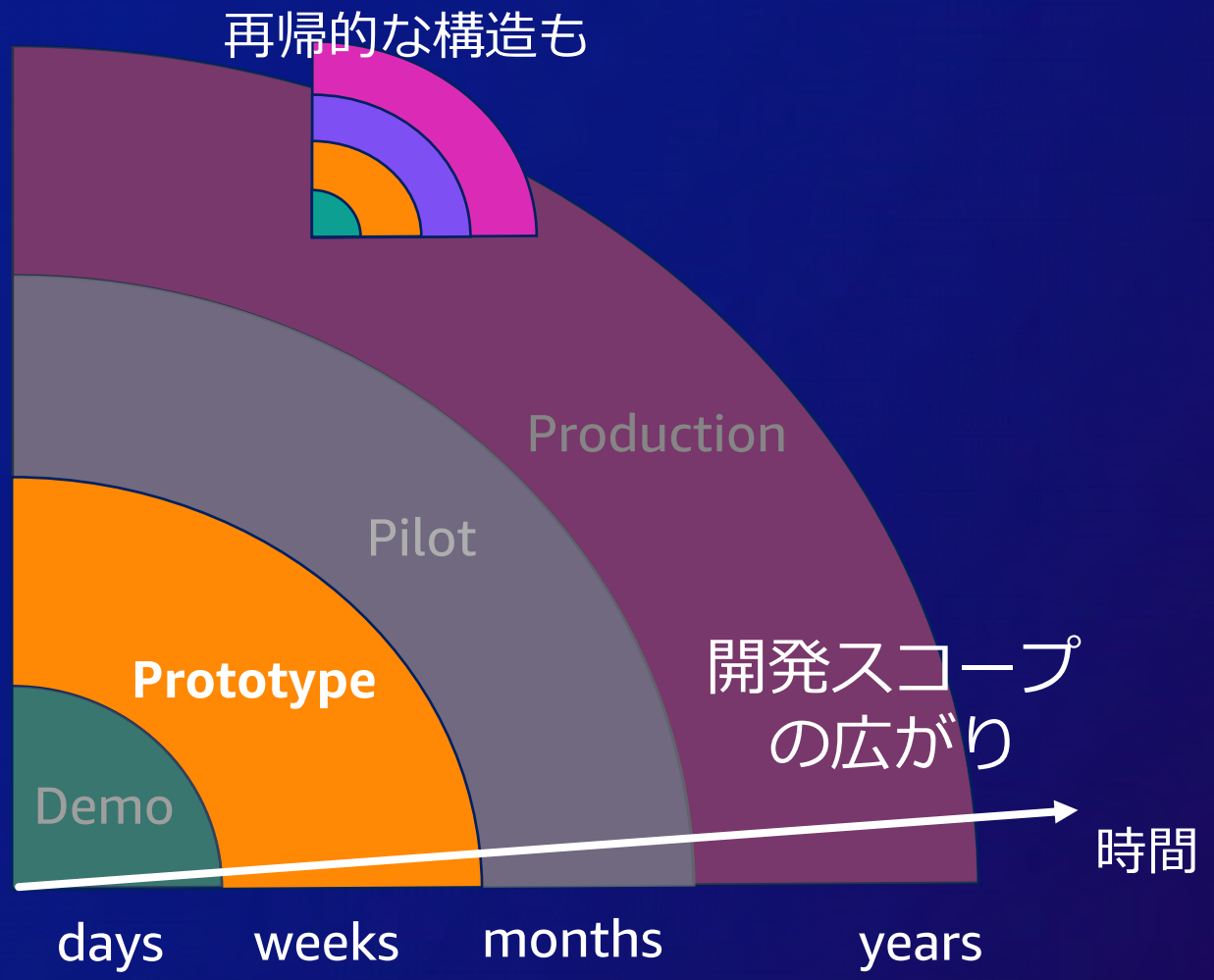
プロトタイピングとは (本セッションでの定義)

プロトタイピング = プロトタイプをつくる営み

実際にリリースされるシステム

多くの開発者は、この意味での
プロトタイピング活動に
関わる機会があるはず

サービスの実用性を示す最低限のシステム
時にコードすら伴わない



プロトタイピングが役立つお悩み

エンジニア
Dave

机上で練り上げた最高のアーキテクチャ、満を持して実装に取り組んだところ全然期待通りに動かなかった

本番向けに開発を進めていたシステム、考慮外の問題が起き技術選定からやり直しになった

“Fail fast”
プロトタイピングにより早期に問題を見出す

プロトタイピングが生みがちなお悩み

プロトタイプの開発に着手したが、思いの外素早く進まない
どうすればもっと早められる？

AWS のサービスは抽象度が低めなものも多いけど、
雑に良い感じに動けば十分なときにも向いてるの？

エンジニア
Eve

プロトタイピングチームのノウハウを詳しくお伝えします

プロトタイピングのノウハウ

1. 明確な目的と最小限の「スコープ」
2. 「かっこいい」アーキテクチャ、禁止
3. AWS サービスの抽象度を「上げる」
4. 「完璧に」進める必要はありません
5. 「爪を研ぐ」ことの重要さ

プロトタイピングのノウハウ

1. 明確な目的と最小限の「スコープ」
2. 「かっこいい」アーキテクチャ、禁止
3. AWS サービスの抽象度を「上げる」
4. 「完璧に」進める必要はありません
5. 「爪を研ぐ」ことの重要さ

プロトタイピングの目的は？

前提を明確にし、チームで合意しましょう

- なぜプロトタイプを作るのか
- プロトタイプで検証したい項目
- 項目ごとの優先度　など

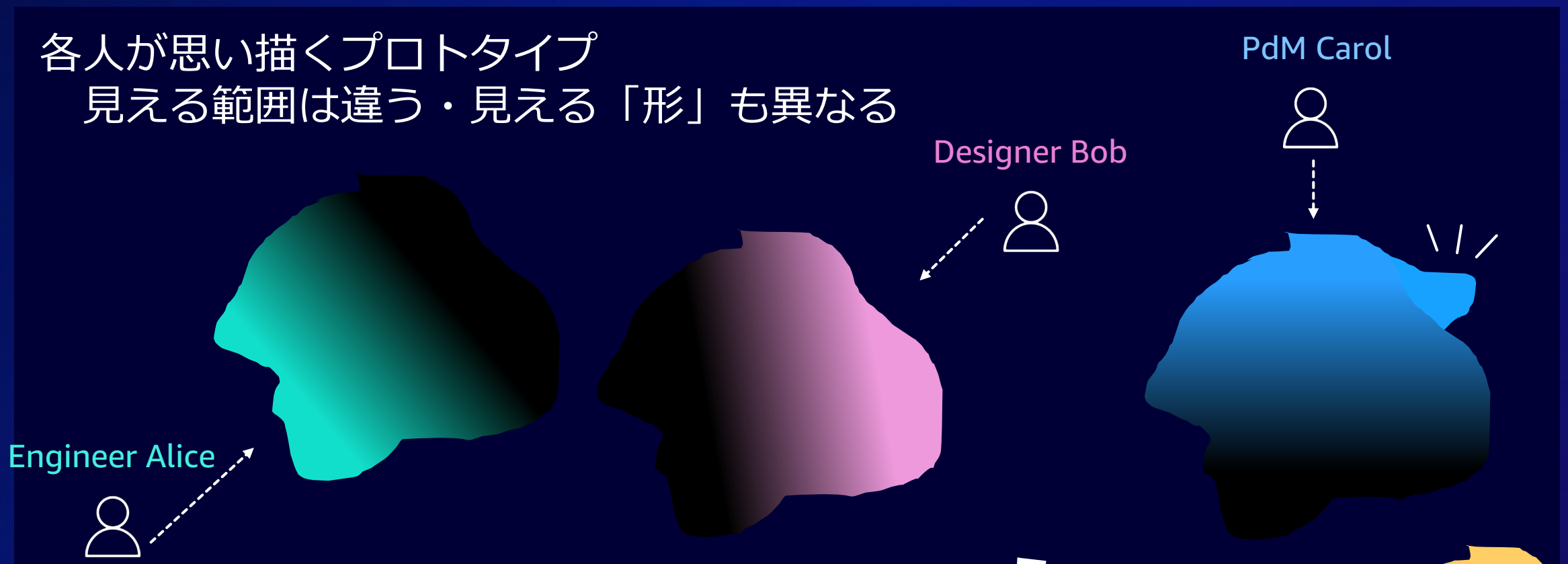
Why？

✅ 後の意思決定が迅速になる

✅ 方針のブレによる手戻りを防ぐ

プロトタイプの「形」を明らかにする

各人が思い描くプロトタイプ
　見える範囲は違う・見える「形」も異なる



恐れずに質問を重ね、すり合わせ・合意に至る
今明確にできる部分をウヤムヤにしないこと

共通認識へ

目的
要件
優先度
etc…

後の意思決定を迅速にし、手戻りを防ぐため

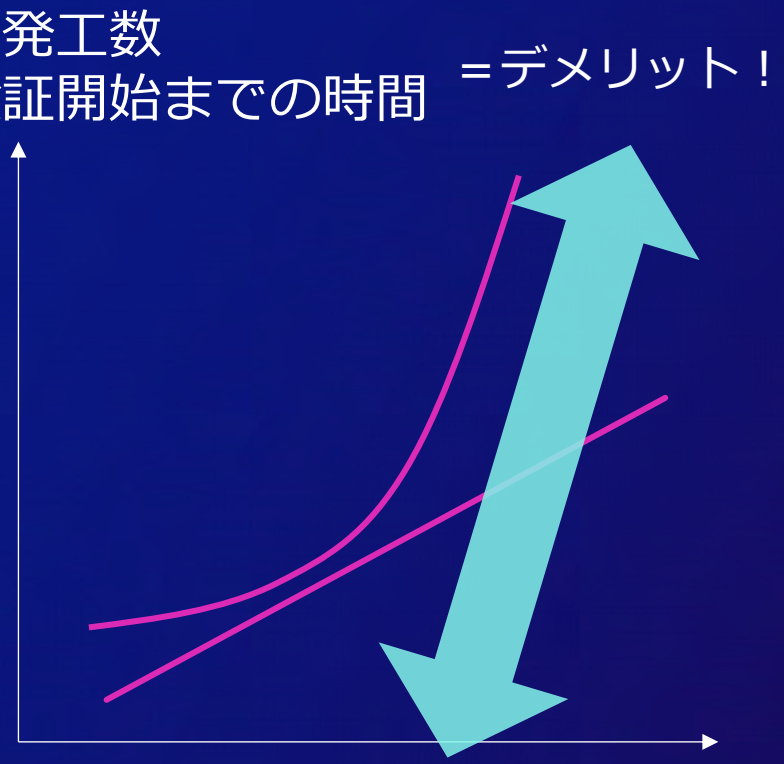
目的から開発スコープが決まる

目的に対して**最小限の**スコープを定める

開発スコープ

開発スコープを広げるほど検証開始が遅くなる

いたずらに広げるべきではない



II

メリット？

プロトタイピングのノウハウ

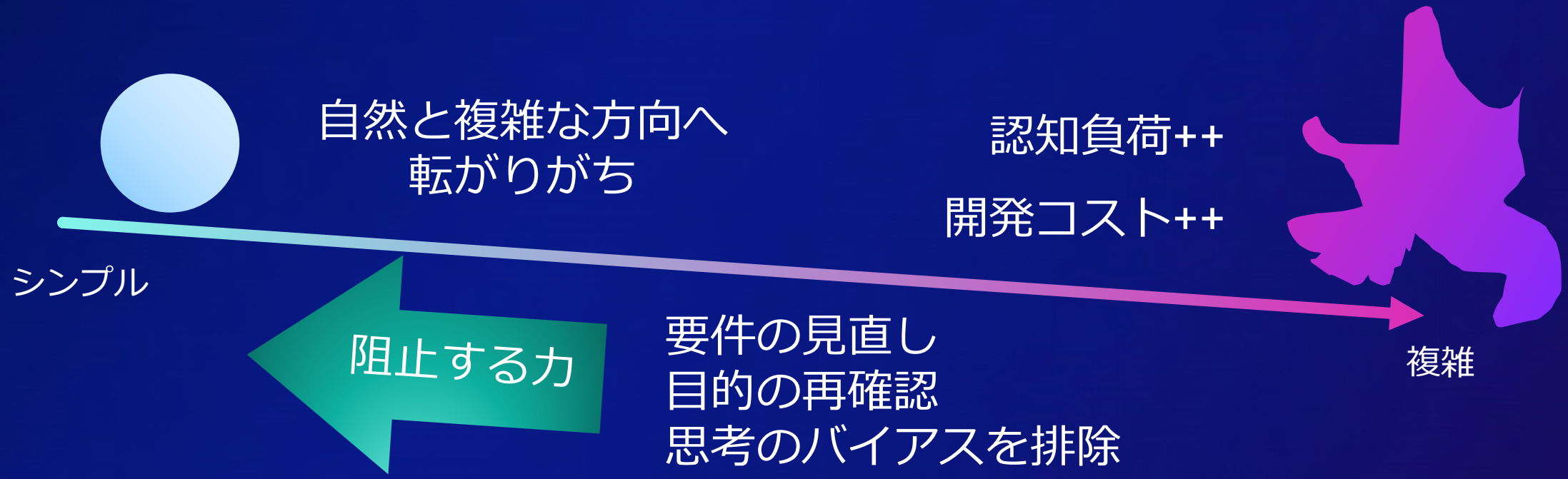
1. 明確な目的と最小限の「スコープ」
2. 「かっこいい」アーキテクチャ、禁止
3. AWS サービスの抽象度を「上げる」
4. 「完璧に」進める必要はありません
5. 「爪を研ぐ」ことの重要さ

設計: たくさんの意思決定が必要

- Lambda, ECS, EKS, or EC2
- RDBMS or NoSQL
- モノリス or マイクロサービス
- アプリのレイヤー分け方？
- 共通化 or 専有化
- CDKスタックの分割方法

プロトタイプ設計のコツ: 不要な複雑化はしない

- 必要のない限りはシンプルに保つのが得策
  - シンプル = 低い開発コスト・認知負荷
- 何がシンプルかは状況によるため、それぞれで議論は必要



プロトタイプ設計におけるアンチパターン？

- 先読みをし「すぎ」た技術選定
- 早すぎる最適化
- 明確な課題意識なく「かっこいい」パターンを導入する

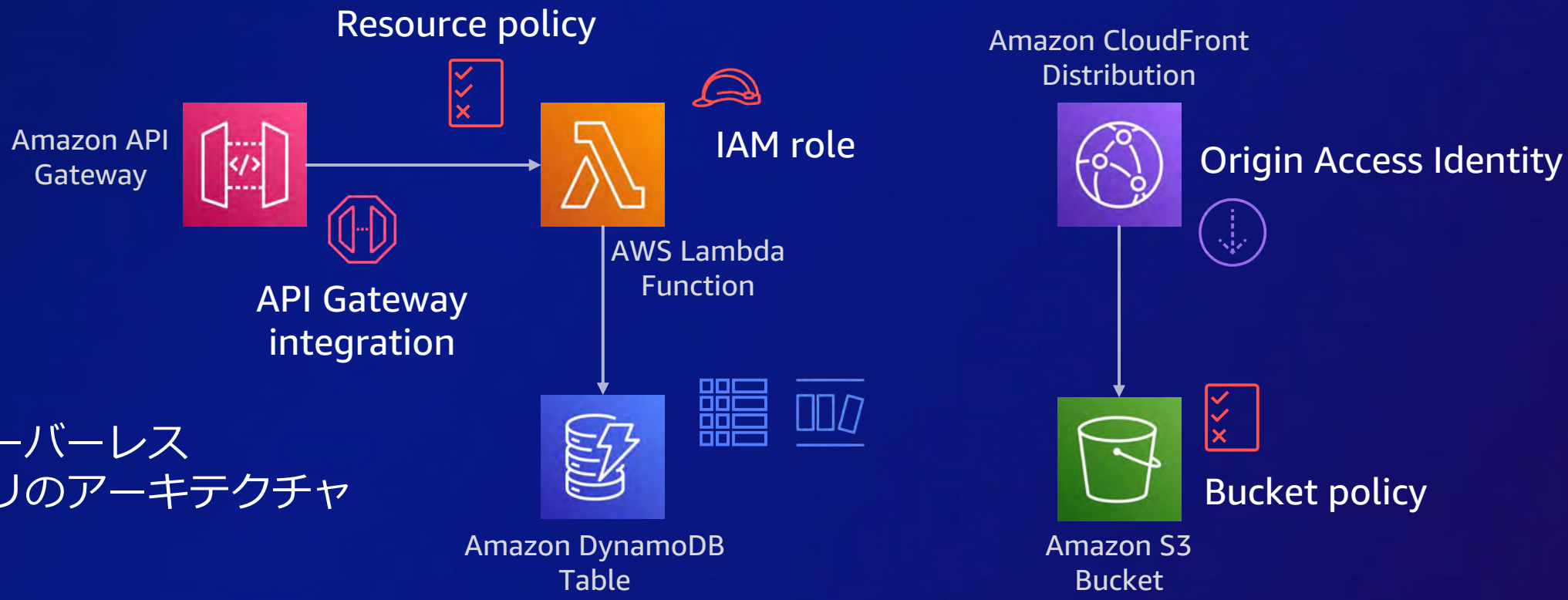
※ 特定技術の検証がプロトタイプの目的である場合はこの限りではない

プロトタイピングのノウハウ

1. 明確な目的と最小限の「スコープ」
2. 「かっこいい」アーキテクチャ、禁止
3. AWS サービスの抽象度を「上げる」
4. 「完璧に」進める必要はありません
5. 「爪を研ぐ」ことの重要さ

AWS の特徴

複数のサービスをビルディングブロックとして組み合わせ、
所望のワークロードを実現する　細やかに設定できる 🤷

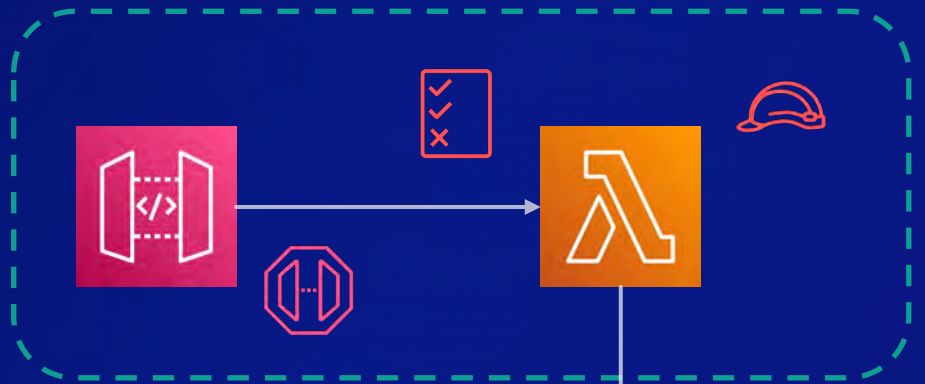


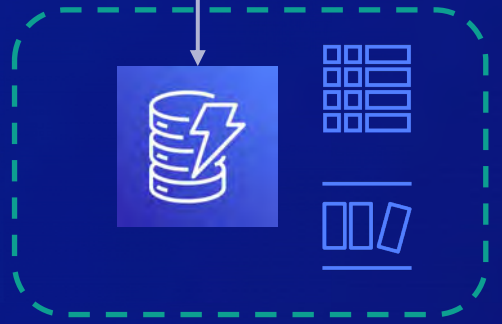
典型的なサーバーレス
ウェブアプリのアーキテクチャ

さらに便利にしたい？

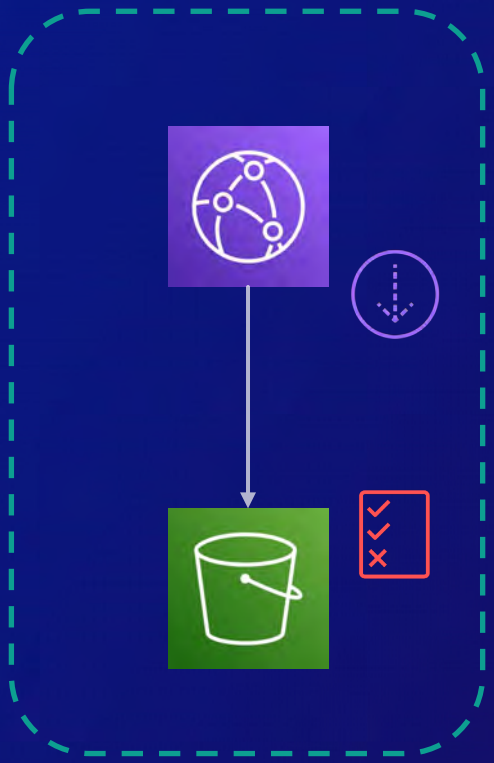
抽象度が上がれば学習・構築コストは下がる
個々のリソースではなく、パターンとして抽象化したら？

API Gateway + Lambda
のパターン





DynamoDB + 周辺リソース
(GSI, DDB Stream, etc.) のパターン



S3 + CloudFront
のパターン

一つの方法: AWS Cloud Development Kit (CDK)

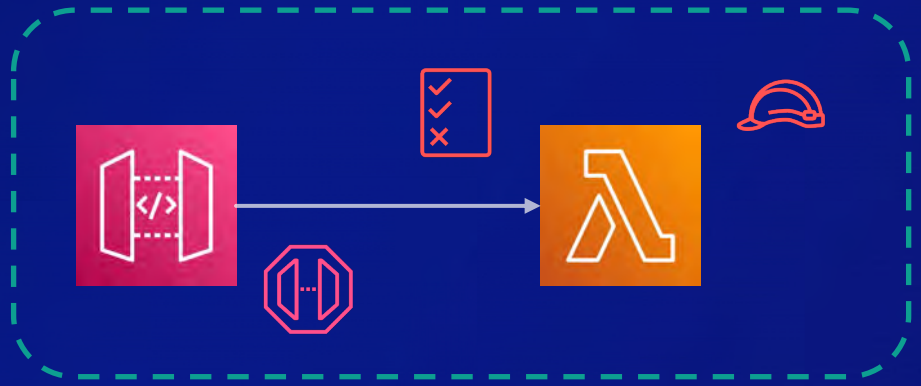
AWS 公式の Infrastructure as Code (IaC) フレームワーク

汎用言語のコードから CloudFormation テンプレートを生成する

多言語の対応
TypeScript など5+言語

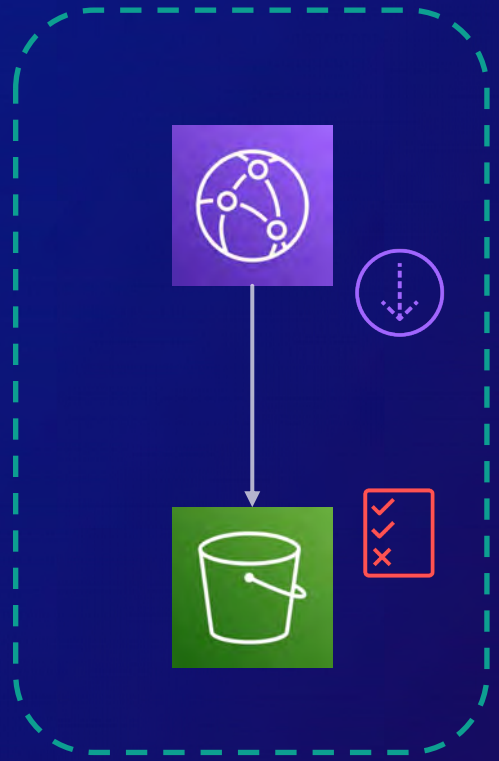
エディタとの連携
自動補完
インラインのドキュメント

抽象化
よくあるパターンを定義
再利用性



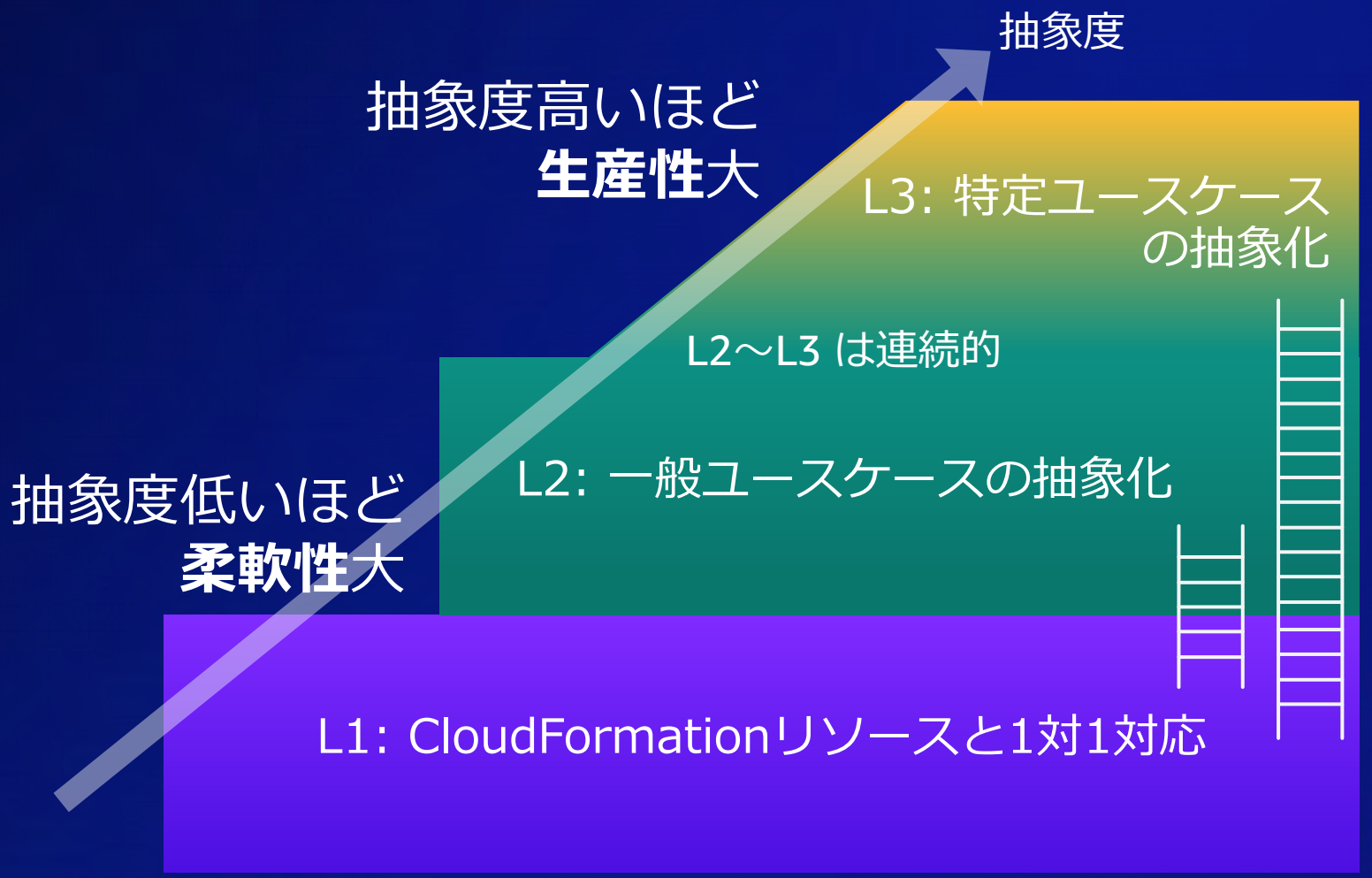
```
const handler = new NodejsFunction(this, "Handler", {
  runtime: Runtime.NODEJS_18_X,
})

new LambdaRestApi(this, "Api", {
  handler: handler
})
```



```
new CloudFrontToS3(this, "FrontendDistribution", {})
```

AWS CDK で抽象度の階段を自由に往来する



例: EC2 インスタンスの CDK コード

```
new BastionHostLinux(this, "Instance", {
  vpc,
});
```

```
new Instance(this, "Instance", {
  vpc,
  instanceType: new InstanceType("t4g.medium"),
  machineImage: MachineImage.latestAmazonLinux({
    generation: AmazonLinuxGeneration.AMAZON_LINUX_2
  })
});
```

```
new CfnInstance(this, "Instance", {
  instanceType: "t4g.medium",
  iamInstanceProfile: "instanceProfile",
  subnetId: "subnet-xxxxx",
  imageId: "ami-xxxxx"
});
```

抽象度を上げて下げる例

L2: 一般的なユースケースの抽象化 (EC2インスタンスの例)

```
1  const instance = new Instance(this, "Instance", {
2    vpc,
3    instanceType: new InstanceType("t4g.medium"),
4    machineImage: MachineImage.latestAmazonLinux({
5      generation: AmazonLinuxGeneration.AMAZON_LINUX_2
6    })
7  });
8
9  (instance.node.defaultChild as CfnInstance).disableApiTermination = true;
```

L1層に下りる

AWS CDK なら
生産性と柔軟性の両取りが可能

詳細: Abstractions and escape hatches

その他 AWS CDK の魅力

- ✅ 柔軟性を損なわない抽象化
- ✅ 一度作ったコードを整備し、ナレッジを蓄積
- ✅ hotswap により変更を高速にデプロイ — CloudFormation を超越
- ✅ 好きな言語で書ける　ライブラリの活用も自由
- ✅ OSS ゆえの透明性

Next...

cdkworkshop.com/ja

プロトタイピングのノウハウ

1. 明確な目的と最小限の「スコープ」
2. 「かっこいい」アーキテクチャ、禁止
3. AWS サービスの抽象度を「上げる」
4. 「完璧に」進める必要はありません
5. 「爪を研ぐ」ことの重要さ

プロトタイプで目指すクオリティ

- 単体テストのカバレッジを 100% に保つ？
- CI/CD を整備する？
- リーダブルで保守性の高いコードを書く？
- Infrastructure as Code ですべてのインフラを管理する？

巷の意見に振り回されてないですか？
プロトタイピングのゴールはすでに決めましたよね？

プロトタイプと完璧主義

- プロトタイピングと完璧主義の相性 → 良くはない
- 実装そのものより、検証結果に価値があることが多いため
- 一方で全く後先を考えないというのも悪手でありがち

トレードオフ・投資対効果を考える

ベストプラクティス	導入コスト	運用コスト	導入しないリスク	導入によるリターン	リスク・リターンが顕出する状況・時期
自動E2Eテスト	中	大	開発が鈍化 バグ発生しやすく	コード品質向上 バグの減少	リファクタ時 機能が増えてきた時

導入によるデメリット　　導入によるメリット　　メリットはいつ享受できる？

- ベストプラクティスに背くことで一定のコスト減は可能
- メリットを享受できないほど短命なプロトタイプもある
- 都度天秤に掛けて判断する

※ 上記コスト評価は一例。例えば導入コストはメンバーの経験に依存して大きく変わる

トレードオフ・投資対効果を考える (練習)

ベストプラクティス	導入コスト	運用コスト	導入しないリスク	導入によるリターン	リスク・リターンが顕出する状況・時期
自動E2Eテスト	中	大	リリースが鈍化 バグの発生	コード品質向上 バグの減少	リファクタ時 機能が増えてきた時
デプロイの自動化					
Linter/formatter					
Infrastructure as Code					
…					

※ 上記コスト評価は一例。例えば導入コストはメンバーの経験に依存して大きく変わる

技術的負債 = 悪者？

- 完璧を目指す・ベストプラクティスを踏襲することが常に最善ではないことは明らか
- 一方、将来その選択の是正に迫られることもある
  - このためしばしば「負債」と例えられる
- プロトタイプの 目的 に沿えば、負債が理に適うことも
  - + 本当に「負債」となるかは不確実

設計の話にも通ずるね

ベストプラクティス ≠ 悪 nor 神託

鵜呑みにせず、常に考える

プロトタイピングのノウハウ

1. 明確な目的と最小限の「スコープ」
2. 「かっこいい」アーキテクチャ、禁止
3. AWS サービスの抽象度を「上げる」
4. 「完璧に」進める必要はありません
5. 「爪を研ぐ」ことの重要さ

経験が物を言う

プロトタイプの成功のために:

- 様々な意思決定を素早く行う必要あり
- 意思決定に自信がないと徐々にブレる

自信を持った意思決定には、経験が重要

日頃から爪を研ごう

とりあえず
まずは試すべし

弊チームでの活動例:

1. 「プレ・プロトタイプ」活動
   - GraphQL で簡単なウェブアプリをつくってみる
   - CloudFormation/Terraform/CDK で同じ構成を書いてみる

「やったことある」を積み重ねる

2. クロージングレビュー
   - あるプロトタイピングでの教訓を共有する会
   - 同じ轍を踏まない / 良いパターンはさらに追究する

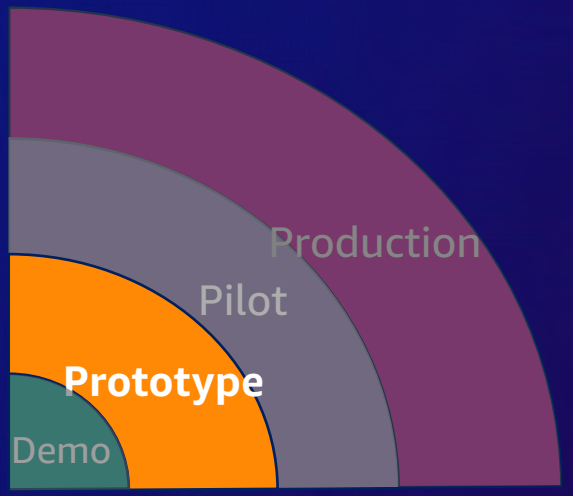
より大きなフィードバックループを作る

プロトタイピングのノウハウ

1. 明確な目的と最小限の「スコープ」
2. 「かっこいい」アーキテクチャ、禁止
3. AWS サービスの抽象度を「上げる」
4. 「完璧に」進める必要はありません
5. 「爪を研ぐ」ことの重要さ

但し書き

- 様々知見を紹介したが、結局は「とりあえずやってみる」のが最強
  - 初期フェーズでの失敗は取り返しが効きやすい
- 本日の話もある種の「ベストプラクティス」集
- まずは「やってみて」経験を積む　これがスタート



Key Takeaways

1. プロトタイピングで早期に問題に気づく
   - Fail fast の手段として活用
2. 抽象化が高速なプロトタイピングの鍵
   - それを実現する一つの方法が AWS CDK
3. 経験こそ成功の礎 (いしずえ)
   - 「とりあえず試してみる」の心がけを保つ

Call to Action

ベストプラクティス	導入コスト	運用コスト	導入しないリスク	導入によるリターン	リスク・リターンが顕出する状況・時期
自動テスト					

- あなたの関わっているプロジェクトで、より Fail fast なアプローチを取り入れられないか、チームで議論してみましょう
- 「ベストプラクティス」のトレードオフ表を、あなたのチームの状況に沿って埋めてみましょう

Announcement

関連セッション

- MVP から始める、ビジネスのスケールに連動したアーキテクチャの遷移について (AWS-11)

チームブログ　prototyping-blog.com

Recent posts

Glue Sparkを使ってRedshiftのデータをDynamoDBへ書き出す

最新の nvidia-driver 対応の Amazon Linux 2 ベースの EKS AMI の作り方

cdk destroy しても残ってしまうリソースへの対処法

Amazon EKS の 複数 Pod 間で GPU を共有する (Time-Slicing 編)

OpenAI API を使って Discord Bot を作る

Thank you!

友岡 雅志 (Masashi Tomooka)
アマゾン ウェブ サービス ジャパン合同会社
デジタルトランスフォーメーション本部 プロトタイピングエンジニア
tmokmss