JAPAN | 2024

aws SUMMIT

AWS-09

安全とスピードを両立するために ~ DevSecOps とプラットフォームエンジニアリグ ~

金森 政雄

アマゾン ウェブ サービス ジャパン合同会社
プロトタイプ＆カスタマーエンジニアリング本部
Developer スペシャリストソリューションアーキテクト

自己紹介

金森 政雄

- 所属/役職：
  Developer スペシャリスト
  ソリューションアーキテクト
- 好きなサービス

Amazon Elastic Container Service

AWS Step Functions

AWS Fault Injection Service

アジェンダ

1. 安全とスピードがなぜ求められるのか
2. DevSecOps
3. プラットフォームエンジニアリングと運用モデル
4. まとめ

今日の話の概要

アジリティのために
小さいチームで開発

自動化が必要

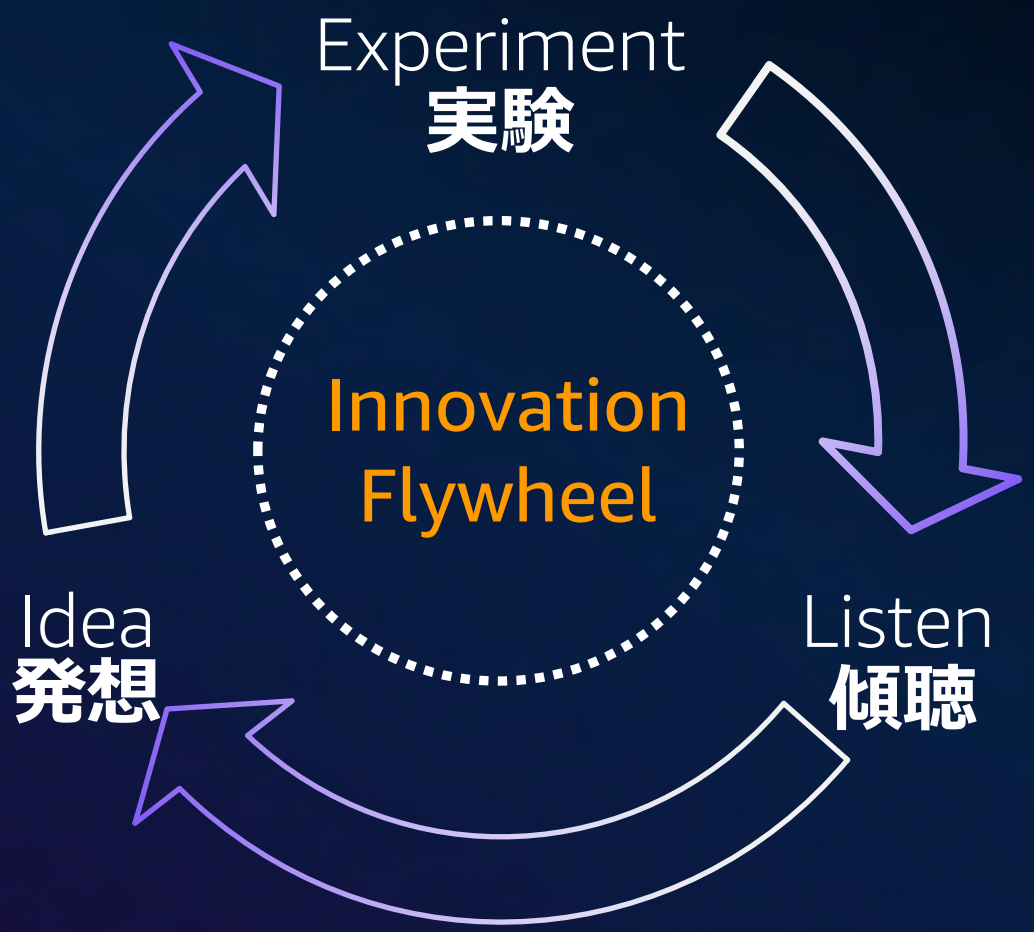
スケールするための
仕組み

安全とスピードが
なぜ求められるのか

現代のアプリケーション

- 独立してスケーラブルなマイクロサービス
- API による接続
- アップデートを継続的にデリバリ
- 変化に迅速に適応
- グローバルにスケール
- フォールトトレランス
- 状態の慎重な管理と永続化
- 組み込まれたセキュリティ

実験のサイクルを早く回したい



スピードと品質は比例する

	Low	Medium	High	Elite
デプロイの頻度	週毎~月毎	週毎~月毎	日毎～週毎	必要な時に
変更のリードタイム	1 週間–1 月	1週間–1月	1–7日	1日未満
変更の失敗率	64%	15%	10%	5%

Source: 2023 State of DevOps Report, DORA

安全性とスピード – Amazon の場合

Amazonは年間 **1 億 5 千万回** 以上本番環境にデプロイ

安全性とコンプライアンスを維持しながら

ISMAP

PCI-DSS

GDPR

FIPS

SOC

NIST

HIPAA

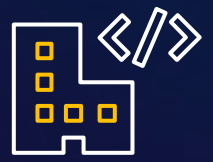
Amazon が学んだソフトウェアの開発とデリバリに関する教訓

アジリティのための分割
(マイクロサービス、2 ピザチーム)

すべてを自動化する

標準化されたツール

ベルトとサスペンダー
(ガバナンス、テンプレート)



コードとしての
インフラストラクチャ

IaC についてはこちらのセッションを！

チームのつながりを Infrastructure as Code でデザインする

高野 賢司
アマゾン ウェブ サービス ジャパン合同会社
技術統括本部 ストラテジック製造グループ
中部ソリューション部
シニア ソリューション アーキテクト

日時: 6/21 14:50 ~

ソフトウェア開発においてチーム間のつながりが不適切だと、俊敏性が減少し、開発コストが増大します。
チーム内においても、異なるスキルやモチベーションを持つメンバー間の壁を取り払うにはメカニズムが必要です。

Infrastructure as Code (IaC) はアプリケーション全体の状態と開発者の意図を明確化するため、チームの共通言語としてコラボレーションの基点になります。
このセッションでは、日本のソフトウェア開発の現場におけるチームトポロジーや認知負荷について検討し、ビジネス価値を迅速に提供できるチームを作るために IaC を戦略的に活用する方法を学びます。

aws

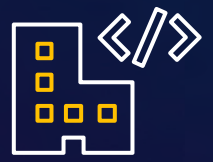
Amazon が学んだソフトウェアの開発とデリバリに関する教訓

アジリティのための分割
(マイクロサービス、2 ピザチーム)

すべてを自動化する

標準化されたツール

ベルトとサスペンダー
(ガバナンス、テンプレート)



コードとしての
インフラストラクチャ

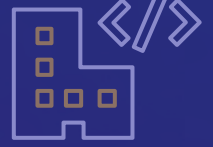
Amazon が学んだソフトウェアの開発とデリバリに関する教訓

アジリティのための分割
(マイクロサービス、2 ピザチーム)

すべてを自動化する

標準化されたツール

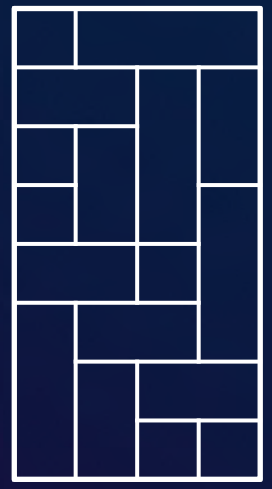
ベルトとサスペンダー
(ガバナンス、テンプレート)



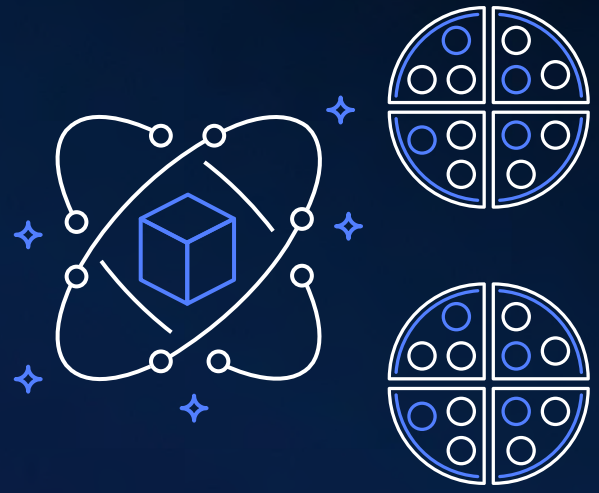
コードとしての
インフラストラクチャ

Amazon のトランスフォーメーション

2001 → 2002

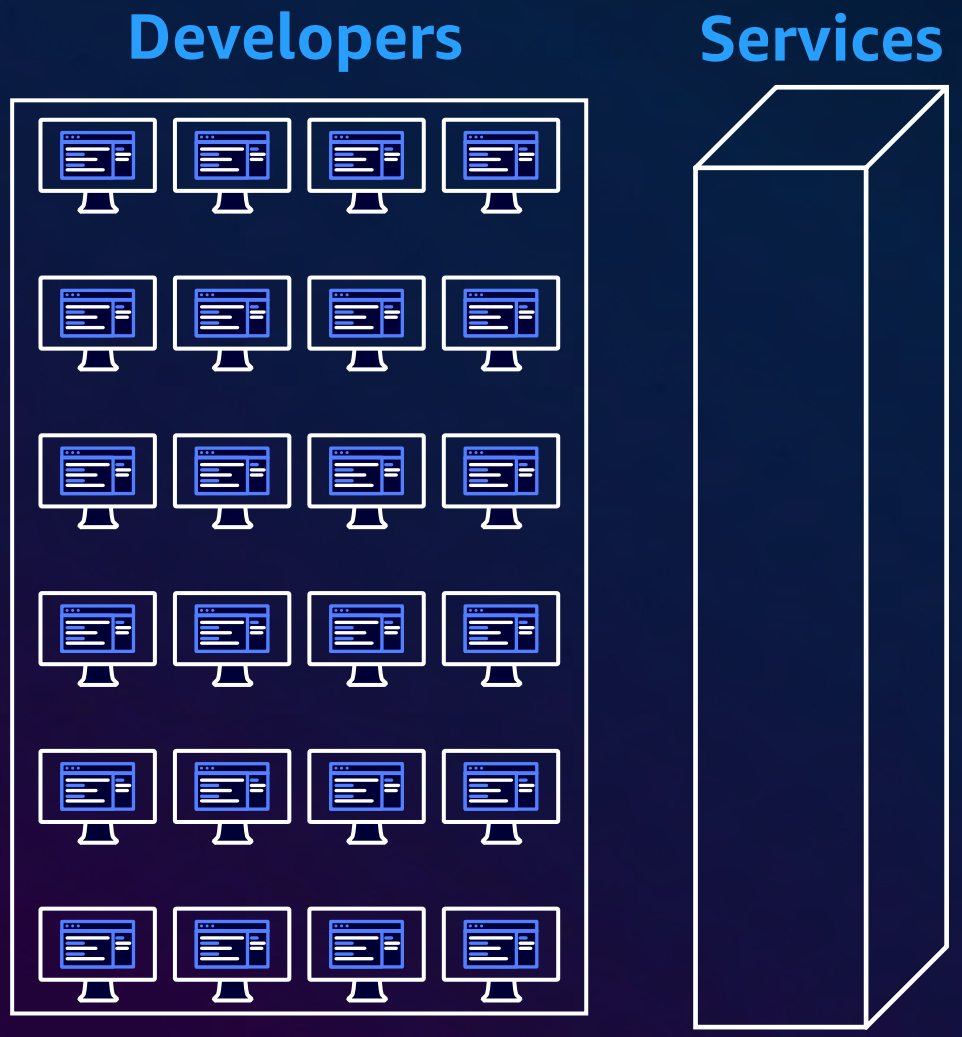


モノリシックアプリ
＋ チーム



マイクロサービス
＋ 2ピザチーム

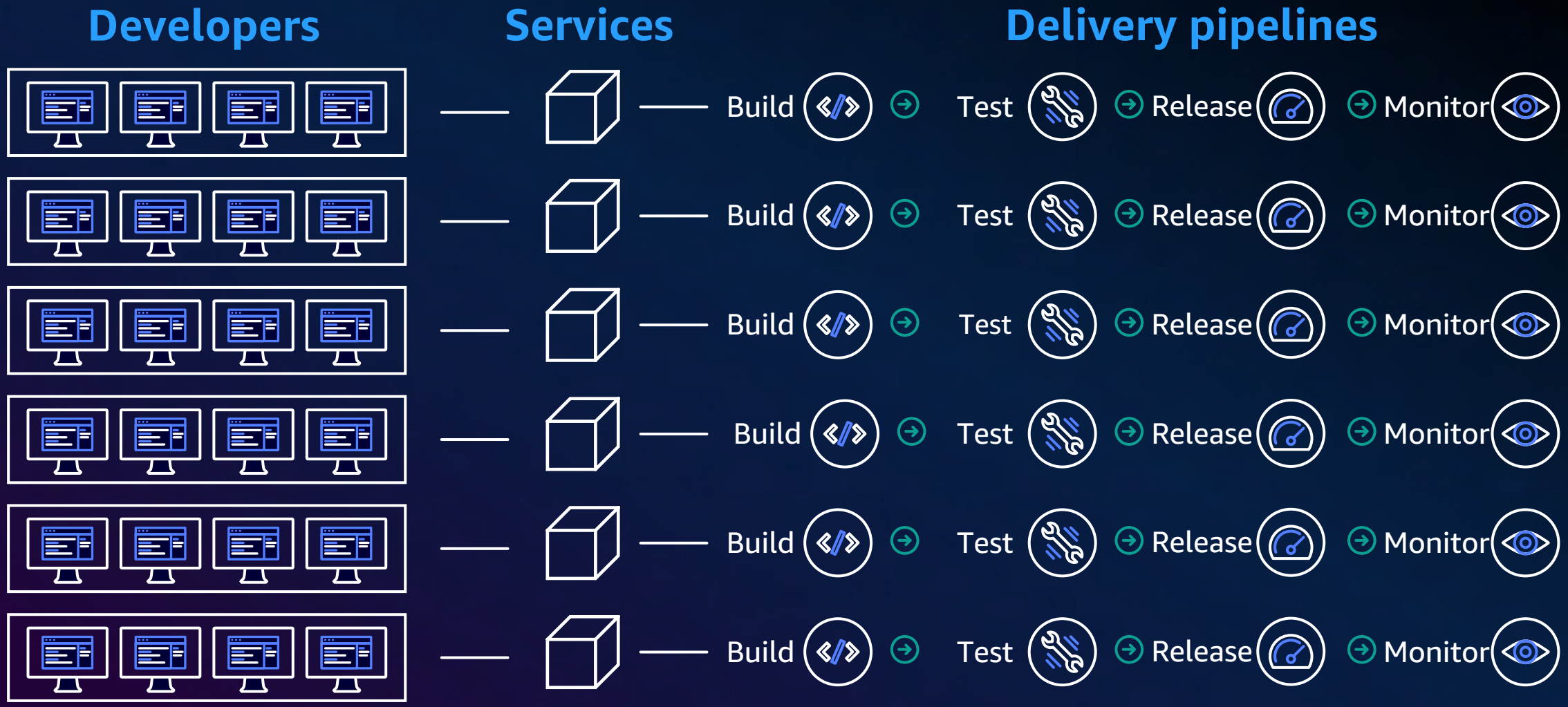
モノリスの開発ライフサイクル



Delivery pipelines

Build → Test → Release → Monitor

マイクロサービスの開発ライフサイクル



チームがすべてを所有する

- 計画
- セキュリティ
- パフォーマンス
- スケーラビリティ
- デプロイ
- オペレーション
- バグ対応
- ドキュメント
- テスト

1つのチームがこれらすべてを行うには?

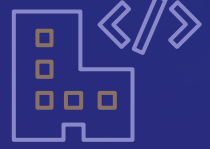
Amazon が学んだソフトウェアの開発とデリバリに関する教訓

アジリティのための分割
(マイクロサービス、2 ピザチーム)

すべてを自動化する

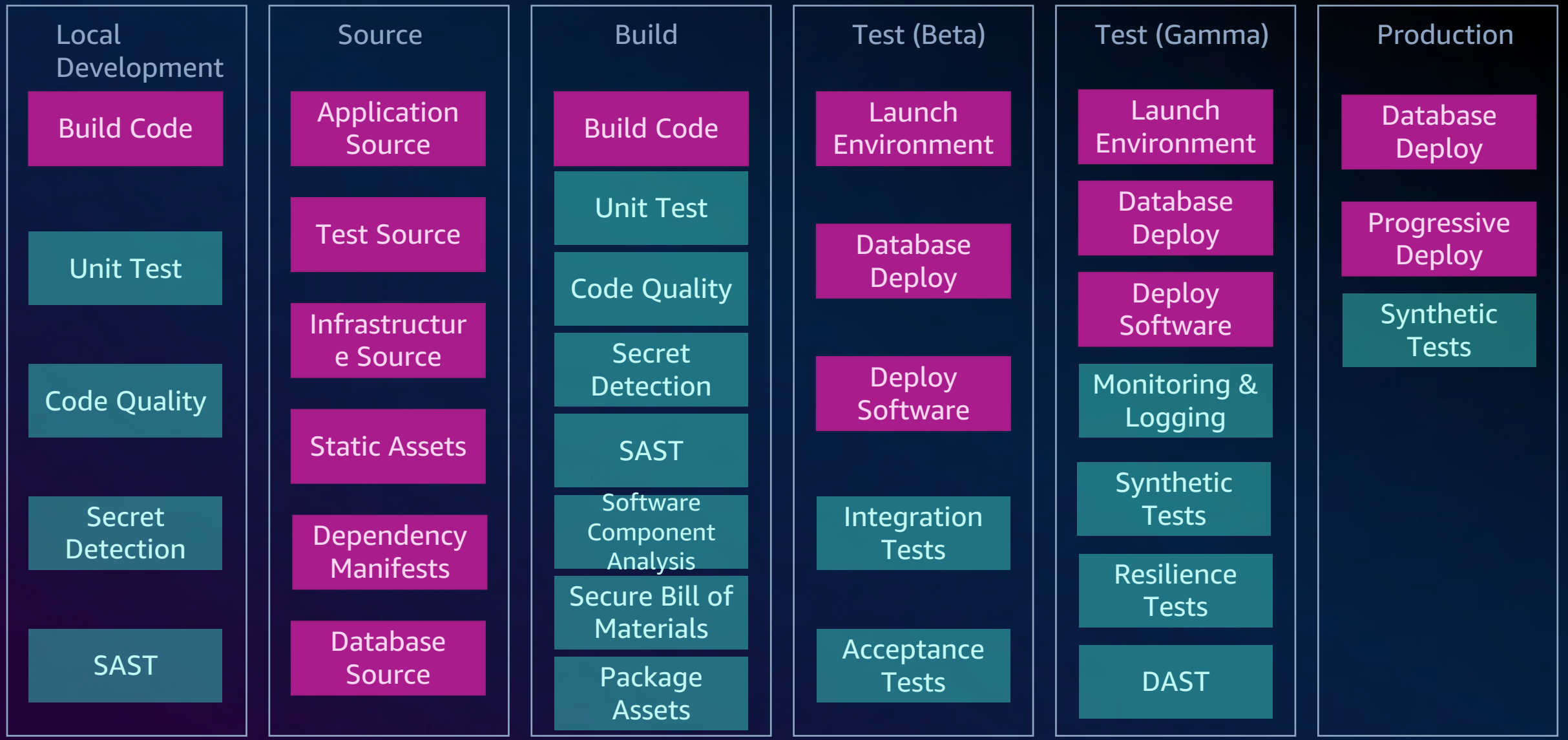
標準化されたツール

ベルトとサスペンダー
(ガバナンス、テンプレート)

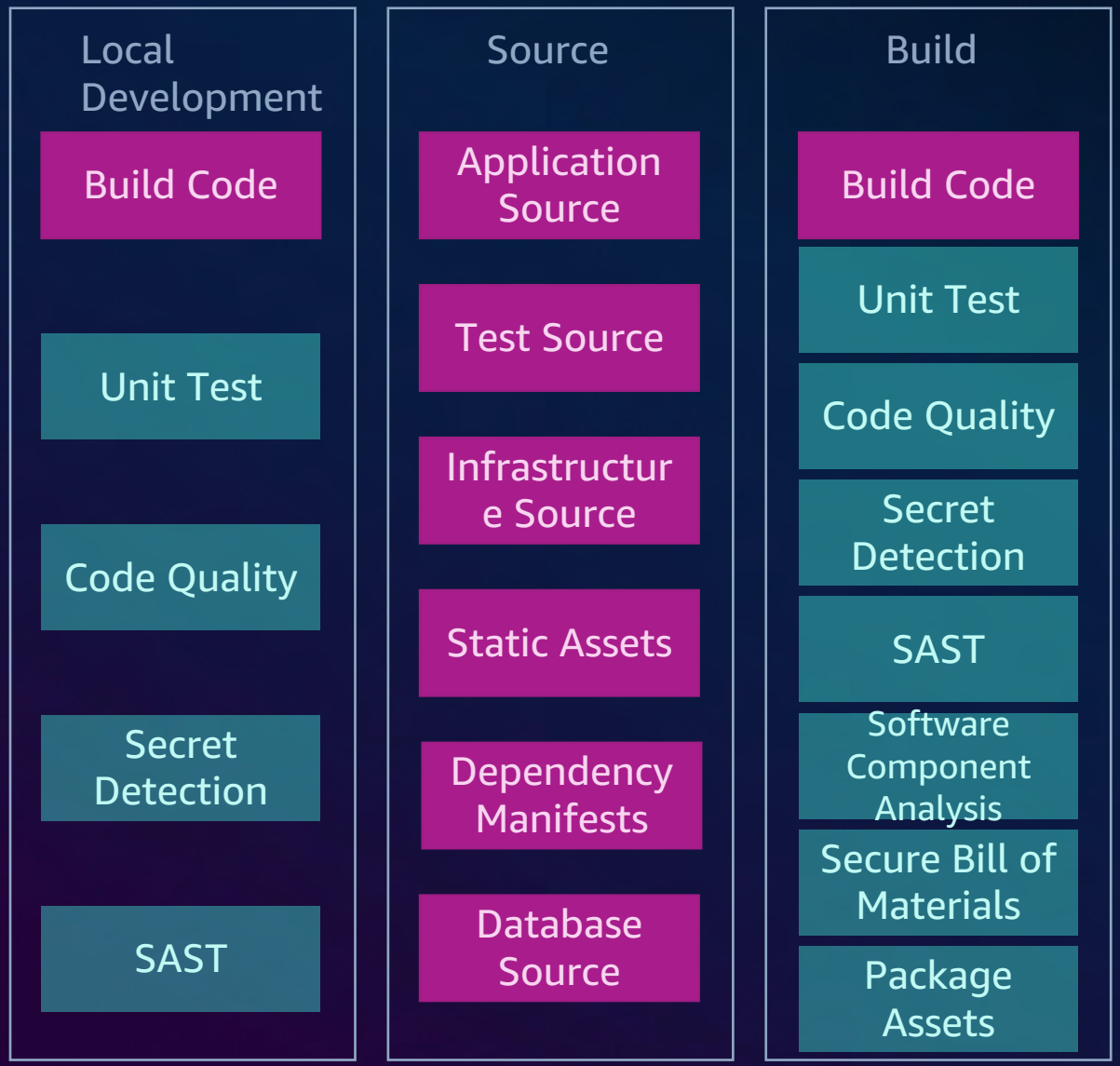


コードとしての
インフラストラクチャ

AWS Deployment Reference Architecture



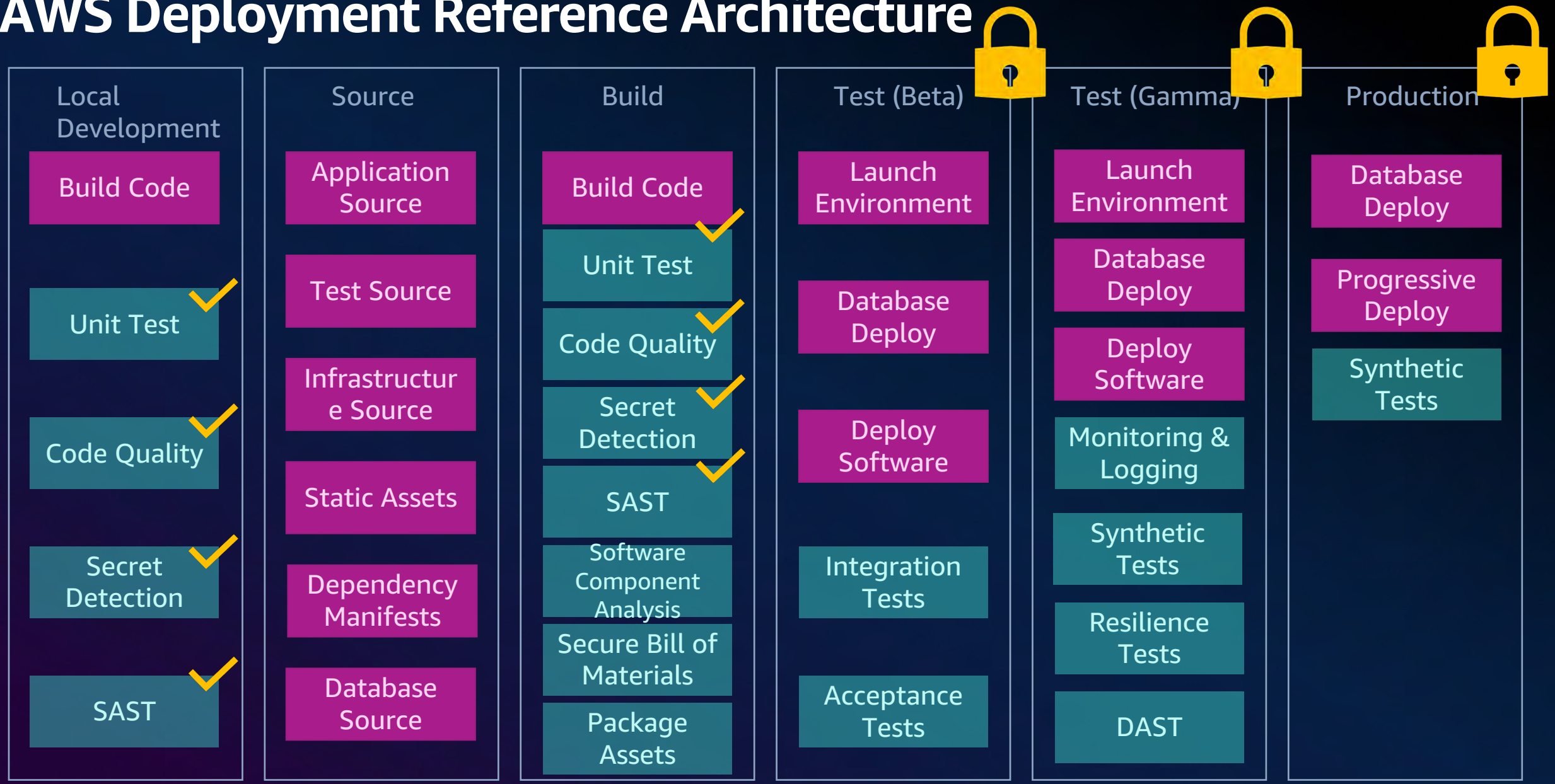
AWS Deployment Reference Architecture



https://pipelines.devops.aws.dev

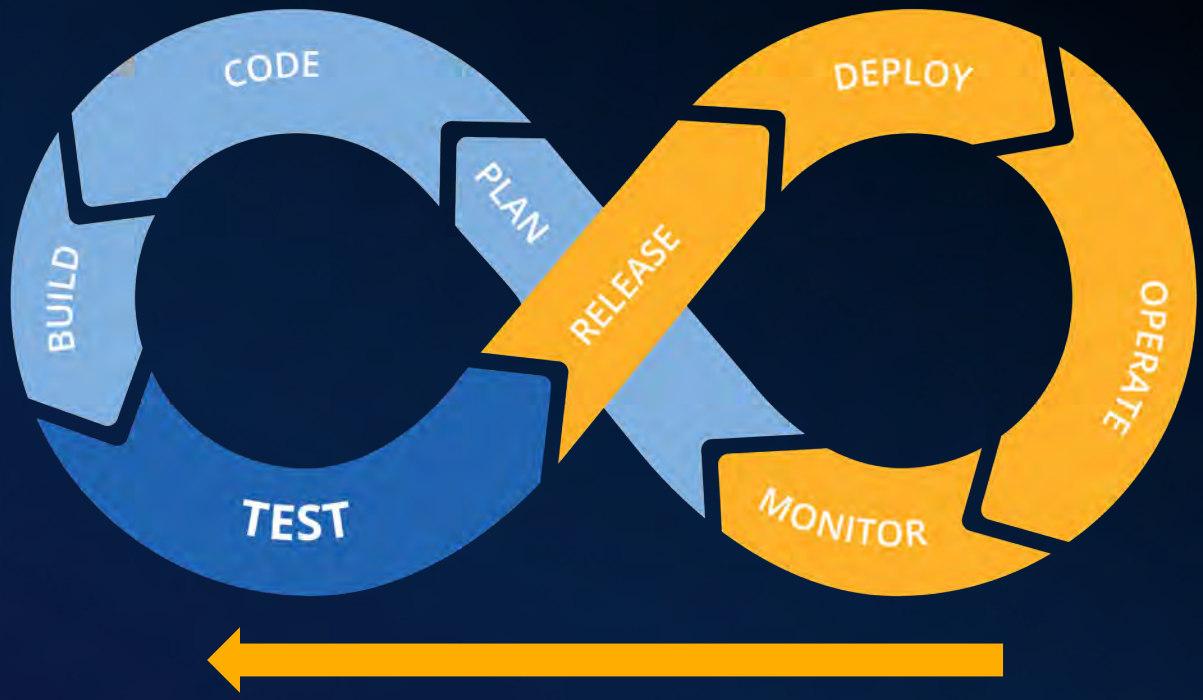
DevSecOps

AWS Deployment Reference Architecture

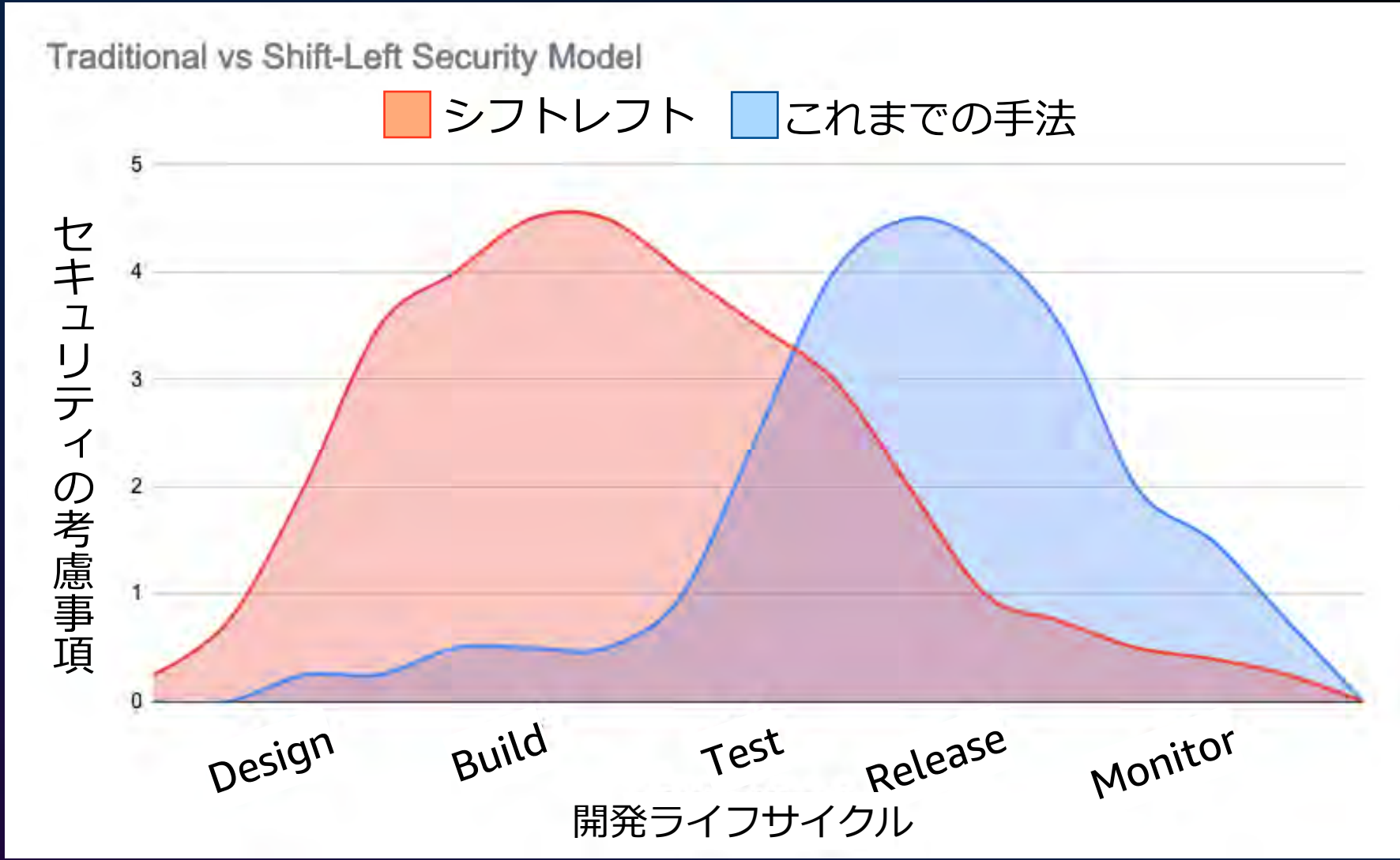


Shift-Left (シフトレフト)

セキュリティへの取り組みを
ソフトウェア開発ライフサイクルの
(可能な限り)早い段階に移すこと。



「セキュリティ」を最初から考慮する



※セキュリティに関する考慮事項と開発ライフサイクルの関係性のイメージ図

FAQ: 開発チームの工数が増えるのでは?

- ソースコードの修正は早期に実施するほどコストが低い [※]
- セキュリティへの意識付け
  - セキュリティチームへ都度依頼しないで良くなる
  - セキュリティチームとコミュニケーションしやすく
- 多くのプロセスは様々な方法で自動化できる

[※] プロジェクトのライフサイクルを通じたエラーコストのエスカレーション- https://ntrs.nasa.gov/citations/20100036670

脅威モデリングとは

- コードを記述する前に、潜在的なセキュリティ問題を特定するための**設計時のアクティビティ**

- 以下のような情報を定義する[※]：
  - アセット、アクター、エントリポイント、コンポーネント、信頼レベル
  - 脅威のリスト
  - 脅威ごとの軽減策
  - リスクマトリックスの作成と確認

[※] 脅威モデリングへのアプローチ方法- https://aws.amazon.com/jp/blogs/news/how-to-approach-threat-modeling/

脅威モデリングの例

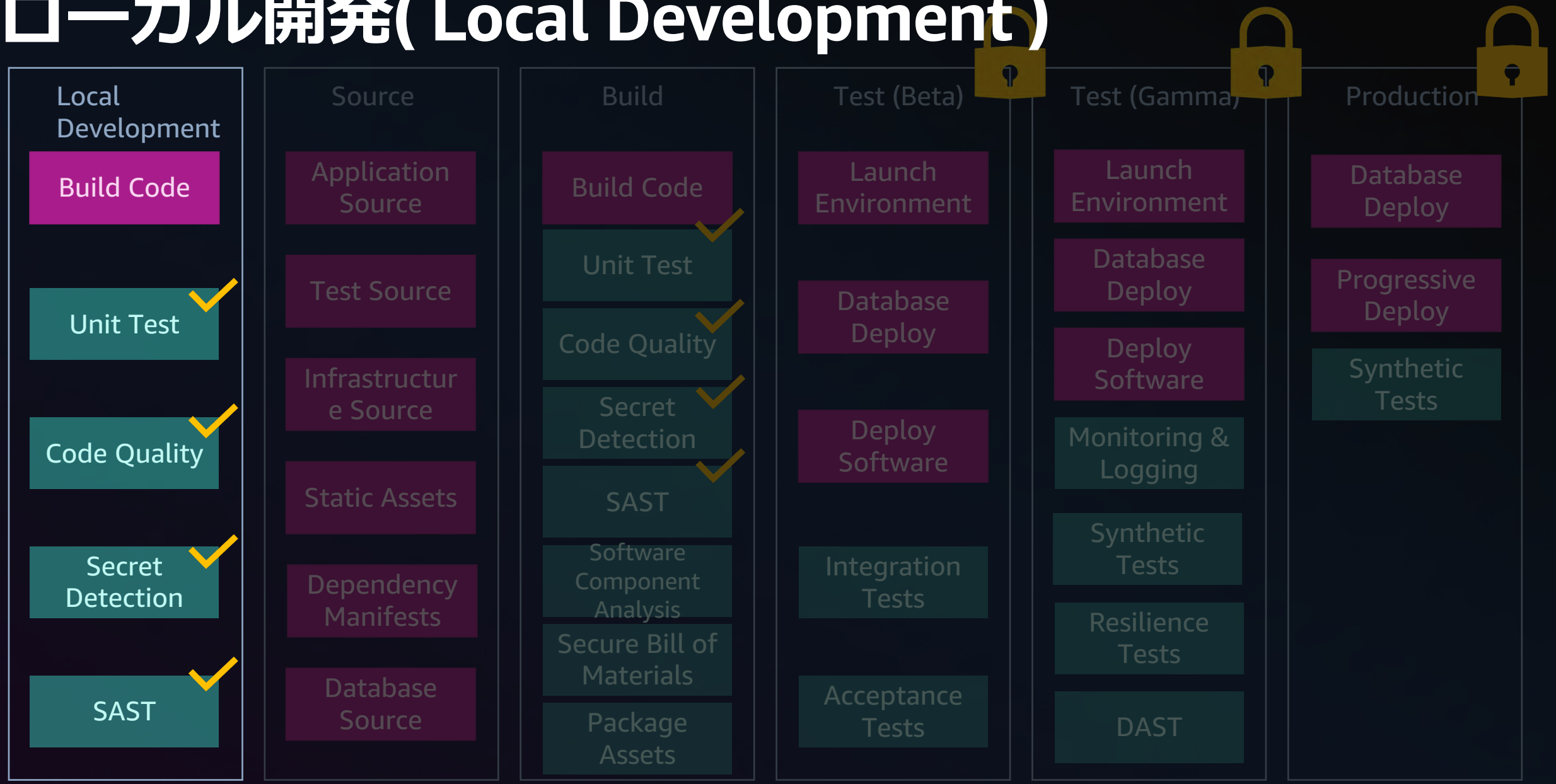
脅威モデリングには多くの方法があるが、最も成熟した方法の1つは STRIDE[1] [2]
開発前に**セキュリティ要件**を特定するために、脅威と緩和策をリストにまとめる

脅威 ID	アセット	エントリポイント	攻撃サマリー	テクニック
TR-01	ユーザー アカウント	ログイン ページ	誰かのアカウントをハイジャックし、悪意のある活動を行うことができること。	ブルートフォース パスワードダンプ セッションハイジャック フィッシング

脅威 ID	予防的/発見的コントロール
TR-01	• 多要素認証 • パスワードのライフサイクルとルール • セッション ID の安全な生成 • セッションライフサイクル • セッションクッキーの安全なストレージ • 顧客への定期的なセキュリティリマインダー

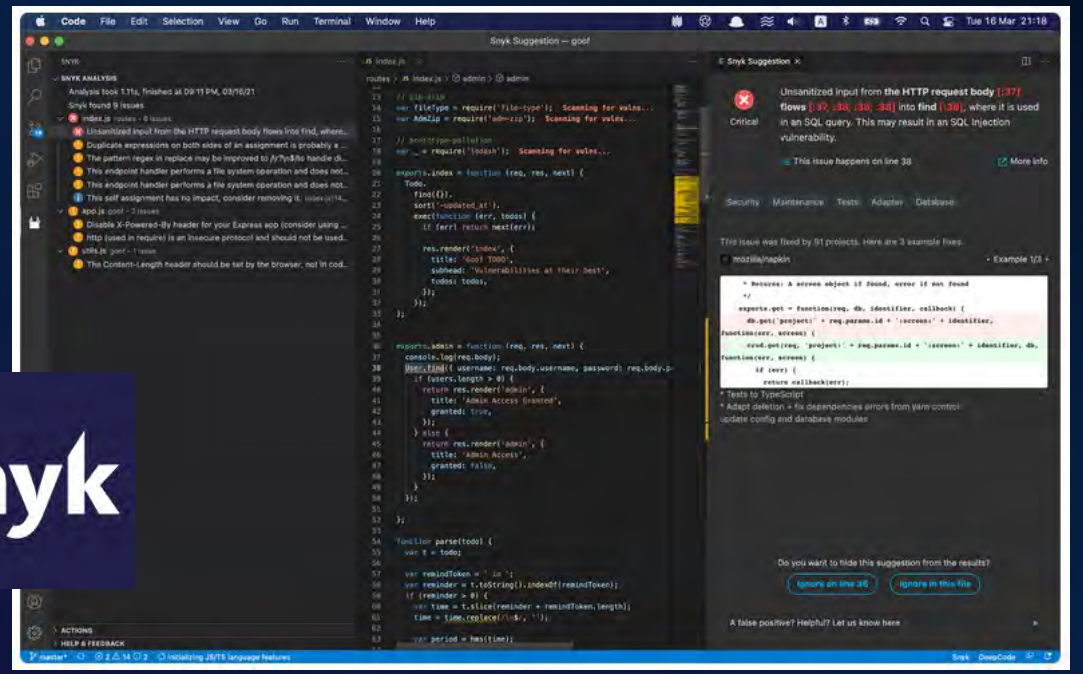
[1] STRDE 脅威モデリング- https://en.wikipedia.org/wiki/STRIDE_%28security%29
[2] Threat modeling the right way for builders - https://catalog.workshops.aws/threatmodel/en-US

ローカル開発(Local Development)



IDE セキュリティプラグイン

- IDE プラグインは、開発者へ迅速に実行可能な対応策を提供
- コーディングの段階から潜在的リスクを防止する
- 正規表現ベースのアプローチ or AI 活用
- 迅速なフィードバックのために設計されており、他のフェーズでのアプローチと組み合わせて利用することを推奨



Amazon Q Developer
Code security scanning

snyk

Pre-Commit フック

- アクセスキー、アクセストークン、SSH キーなどの機密情報は、偶発的な git コミットにより誤って漏洩することがある
- Pre-Commit フックは、開発者のマシンまたはクラウド IDE にインストールし、機密データをフィルタリングする
- 通常、正規表現ベースのアプローチ

Talisman

git-secret

trufflesecurity / truffleHog

awslabs / git-secrets

GitHound

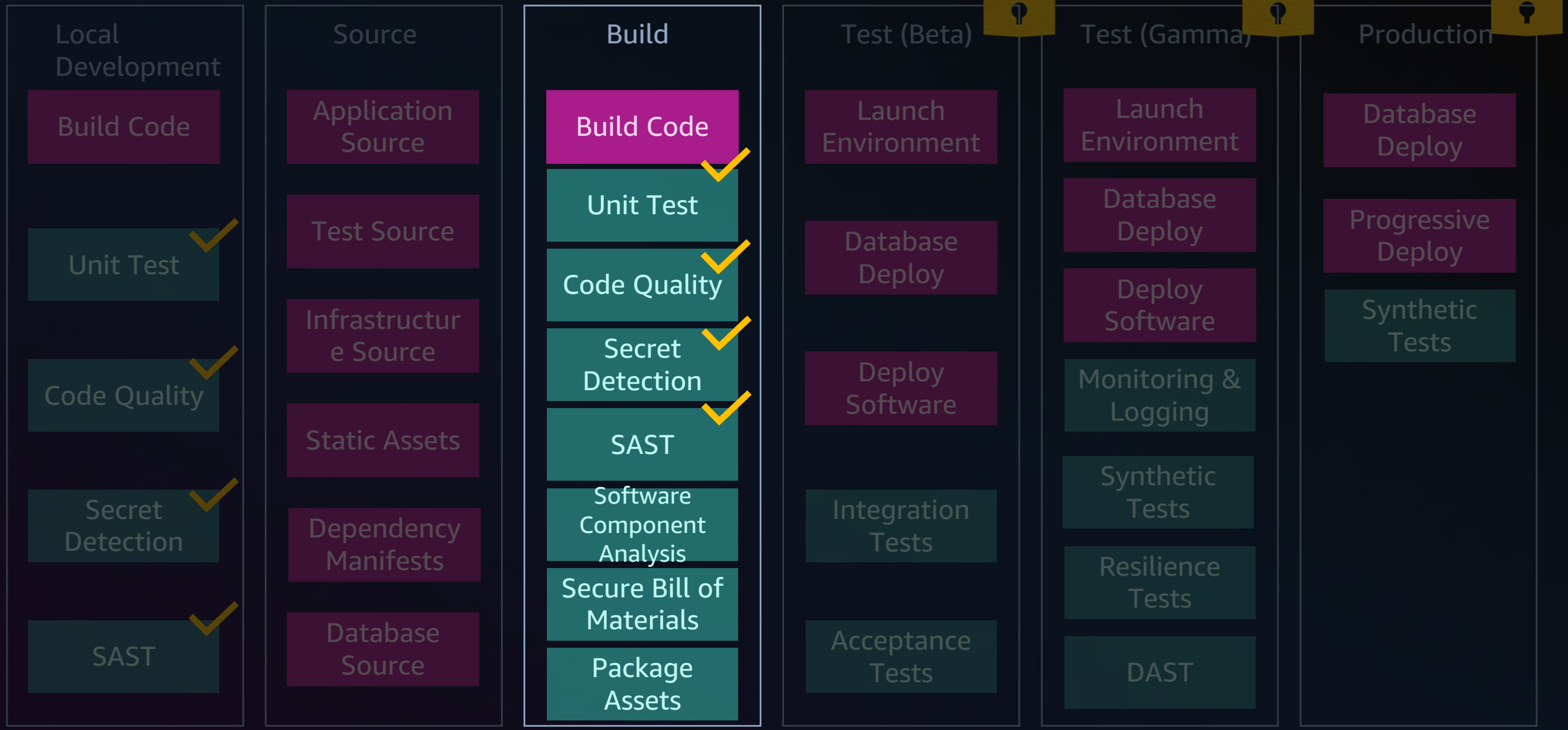
コードレビューの実施

- コードレビューで防げる脆弱性も多い(バグを減らすことも効果的)
- 必要に応じてコーディング規約を策定/参照する
- フレームワークに沿った正しい実装ができているかをチェックする

公開されている各言語におけるコーディング規約の一例

言語	説明	URL
Python	Pythonの公式コーディング規約 PEP8	https://pep8-ja.readthedocs.io/ja/latest/
Java	JPCERTによるJava コーディング スタンダード	https://www.jpcert.or.jp/java-rules/
Ruby	Ruby AssociationによるRuby コーディング規約	https://www.ruby.or.jp/ja/tech/development/ruby/050_coding_rule.html

Build



静的アプリケーションセキュリティテスト (SAST)

- 開発したコードにおける セキュリティの脆弱性を分析
- 例 - SQLインジェクション、XSS、不安全なライブラリなど
- 誤検知の管理は対応が必要

AWSのサービス

Amazon CodeGuru Security (Preview)

AWS以外のサービス

- **GitHub Code Scanning**
- **SonarQube**
- **Checkmarx**

など

参考: OWASP「Source Code Analysis Tools」
https://owasp.org/www-community/Source_Code_Analysis_Tools

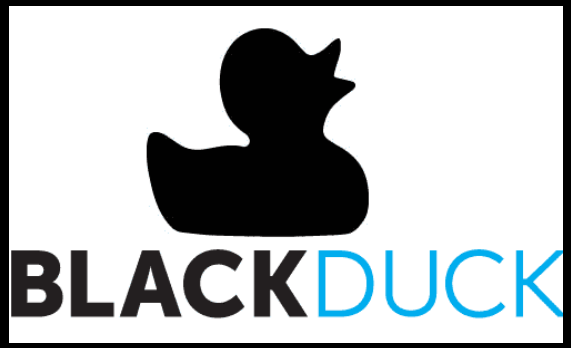
ソフトウェア構成解析 (SCA)

ソフトウェアの大部分はサードパーティのライブラリであり、
多くのセキュリティ脆弱性の由来となっている [※]

OSS ライブラリは継続的な更新が必要

- pip、npm、bundler、go get、composer など

Software Composition Analysis (SCA) では、
脆弱なサードパーティライブラリを識別するためのチェックを行う



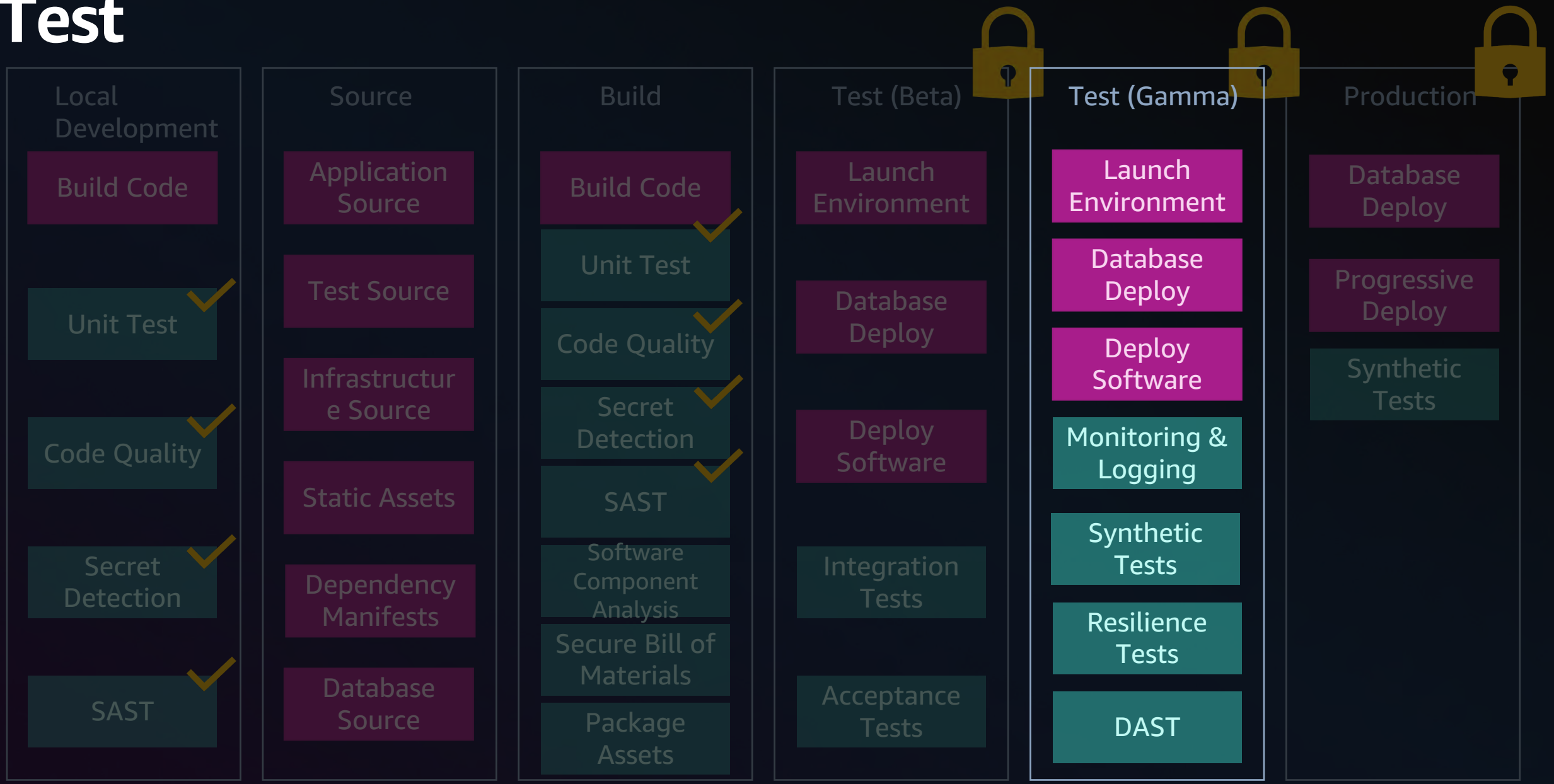
[※] Veracode State of Security Report- https://www.veracode.com/state-of-software-security-report

ライセンスチェック

組織によっては、会社の知的財産を保護するために特定のライブラリを使用しないよう求められる場合がある

GNU (General Public License) のようなコピーレフトライセンスを用いる場合、アプリケーションも同ライセンスで配布する必要がある

Test



動的アプリケーションセキュリティテスト (DAST)

- ツールを使用したブラック/グレーボックスのセキュリティテスト
- デプロイメント固有の問題の特定に役立つ
- SAST の結果と比較して、誤検出を除外

OWASP ZAP (Zed Attack Proxy)

- OWASPが開発しているアプリケーション脆弱性診断ツール
- 完全無償で利用可能

そのほかの脆弱性スキャンツール (DAST)

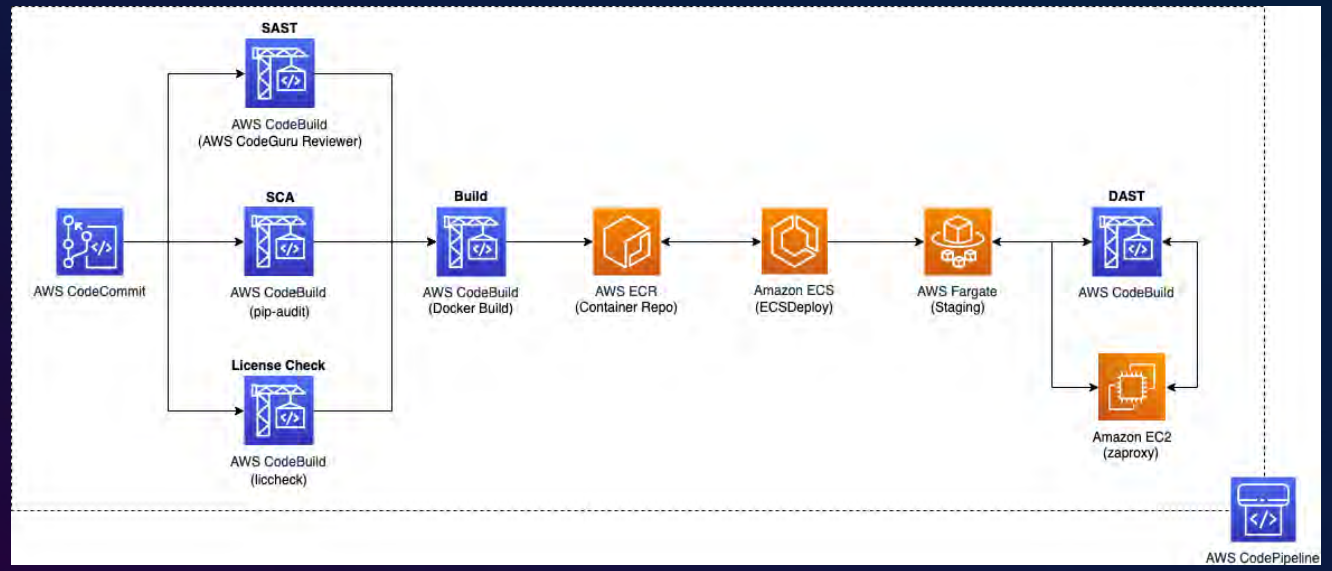
- Burp Suite
- Nikto
- Fiddler

など

参考: OWASP「Vulnerability Scanning Tools」
https://owasp.org/www-community/Vulnerability_Scanning_Tools

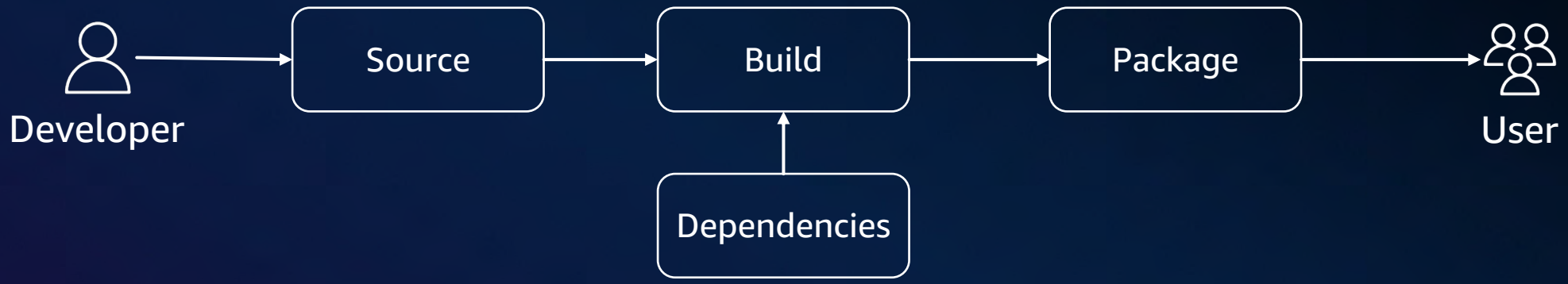
Security for Developers ワークショップ

シンプルで脆弱性のあるPython アプリをCI/CD にセキュリティテストを組み込んで改善していくワークショップ



https://catalog.workshops.aws/sec4devs/ja-JP

ソフトウェアサプライチェーンとは



ソフトウェアが開発され利用者に届くまで(ソフトウェア開発ライフサイクル)に関わる全ての要素

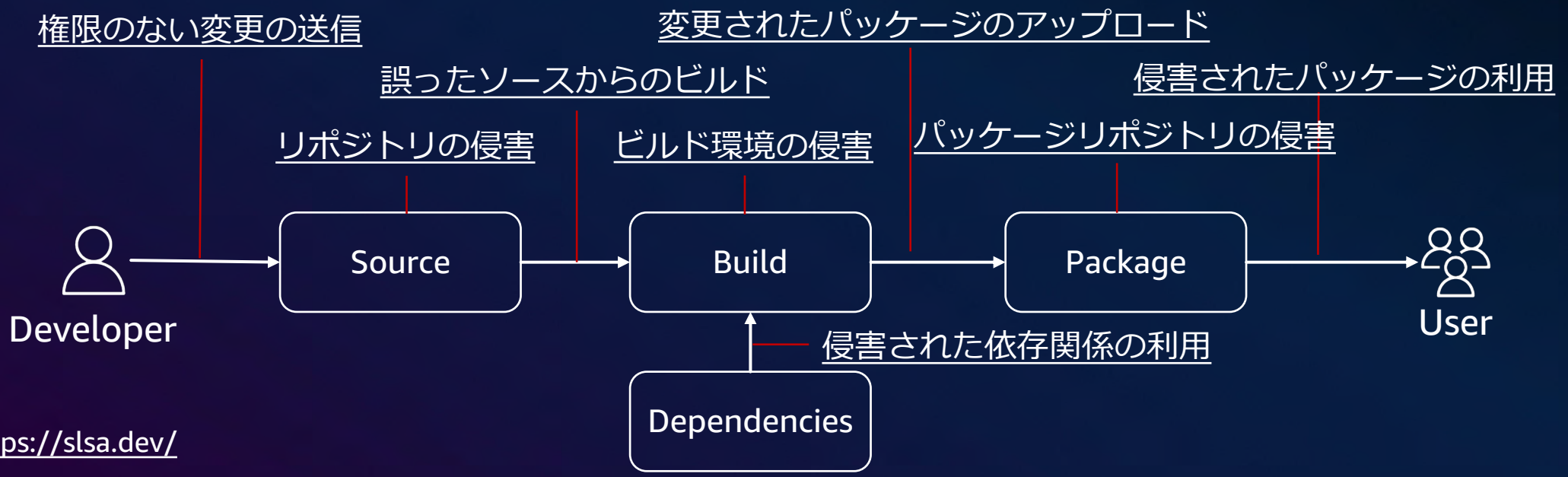
コード、設定ファイル、依存するソフトウェア、ビルド実行環境、ツール、開発組織/人、プロセスなど幅広い要素があり、複雑に絡まっている

ソフトウェアサプライチェーンに潜む脅威

ソフトウェア開発ライフサイクルのどこかで侵害が成功すると、
サービスを利用するユーザーも影響を受ける

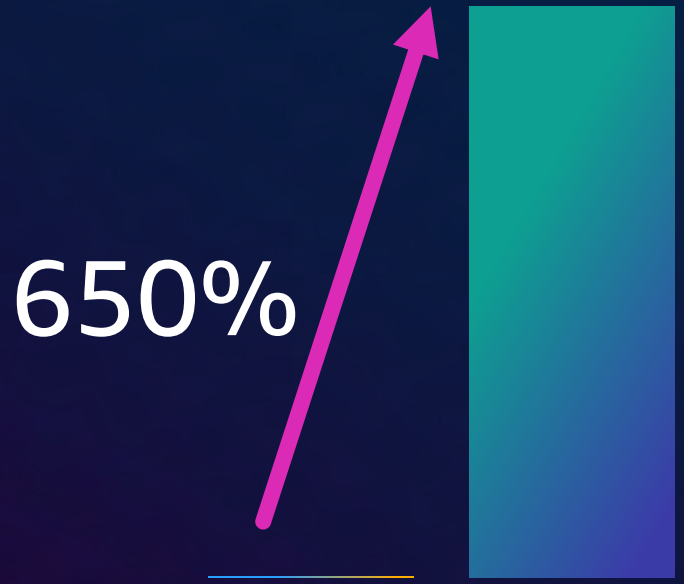
各工程において、様々な脅威が発生する

SLSA framework（※1）



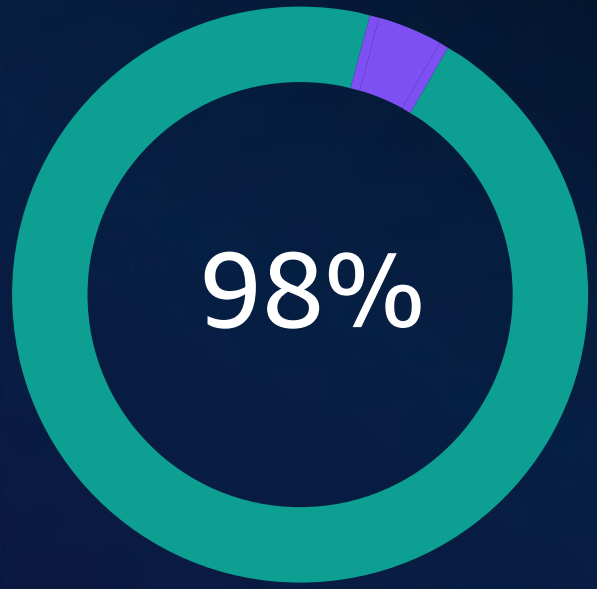
※1 https://slsa.dev/

ソフトウェアサプライチェーンの安全性



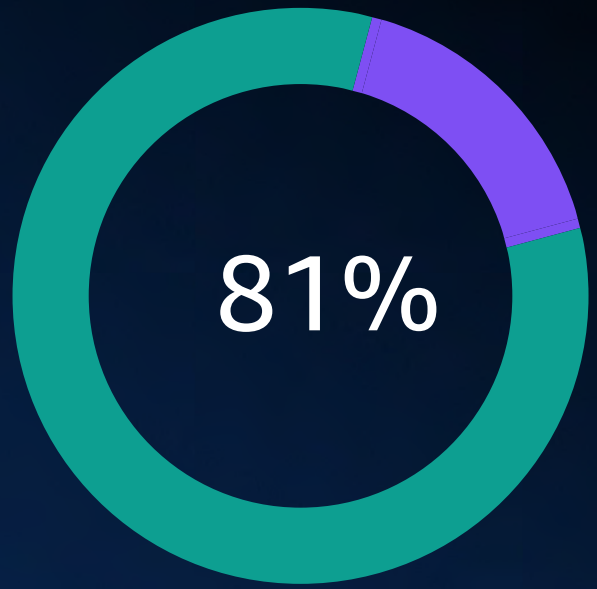
Software supply chain への攻撃

Sonatype, *2021 State of the Software Supply Chain*, https://bit.ly/3ooE5UJ



Open source software (OSS) 利用率

The Linux Foundation, *Software Bill of Materials (SBOM) and Cybersecurity Readiness*, 2021, https://bit.ly/3yYeGpP



コードベースが脆弱性を含む

Synopsys, *2022 Open Source Security and Risk Analysis Report*, https://bit.ly/3ctqHMp

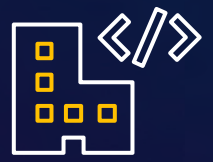
ガバナンスの自動化の動向

変更管理ボード

プラットフォームエンジニアリング

標準化

Supply-chain Level for Software Artifact (SLSA)



Secure by Design

Supply chain Levels for Software Artifacts (SLSA)

SLSA はサプライチェーンのセキュリティに関するガイドラインを提供するOpen SSF (Open Source Security Foundation) のプロジェクト

- ソフトウェアサプライチェーンに関する共通言語
- 使用するアーティファクトの信頼性を評価し、サプライチェーンを保護する方法
- ソフトウェアのセキュリティを強化するための実践的なチェックリスト
- NISTセキュアソフトウェア開発フレームワーク（SSDF) の今後の大統領令基準への準拠に向けた取り組みを測定する方法

SLSA

OpenSSF
OPEN SOURCE SECURITY FOUNDATION

参考: https://slsa.dev/spec/v1.0/about

SLSA の3 つの Build レベル

Build L1
ビルドの来歴 (provenance)

要件

- 一貫したビルドプロセス
- どのようにビルドされたかを示す”来歴”が存在
- 来歴を配布
  (可能ならパッケージエコシステムを使用)

Build L2
ホスト型のビルドプラットフォーム

要件

Build L1 に加えて。。。

- ビルドは専用のインフラで実行
- 来歴はビルドのインフラに紐づけて署名
- 検証には来歴の信頼性の検証を含む

Build L3
堅牢なビルド

要件

Build L2に加え、プラットフォームが強い統制を実装

- 同じプロジェクト内でもビルド実行が互いに影響を与えないこと
- 来歴の署名用シークレットにはユーザ定義ステップのアクセス不可

参考: SLSA の検証を試してみたい方へ

AWS IATK とPowertools for AWS Lambda はSLSA Verifier による署名されたビルドの検証をサポート(以下はAWS IATK の手順)

1. 以下をインストール/ダウンロード
   - SLSA Verifier tool
   - 最新のリリースアーティファクト
   - attestation ファイル (*python-client.multiple.intoto.jsonl*)

2. 下記のコマンドを実行して検証(1. のファイルがあるディレクトリで実行)

```
slsa-verifier verify-artifact ¥
--provenance-path "python-client.multiple.intoto.jsonl" ¥
--source-uri github.com/awslabs/aws-iatk ¥
aws-iatk-0.1.0.tar.gz
```

参考: AWS IATK https://awslabs.github.io/aws-iatk/security/#how-to
Powertools for AWS Lambda https://docs.powertools.aws.dev/lambda/python/latest/security/

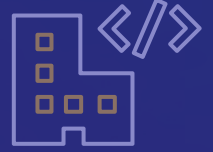
Amazon が学んだソフトウェアの開発とデリバリに関する教訓

アジリティのための分割
(マイクロサービス、2 ピザチーム)

すべてを自動化する

標準化されたツール

ベルトとサスペンダー
(ガバナンス、テンプレート)



コードとしての
インフラストラクチャ

プラットフォームエンジニアリングと運用モデル

Amazon の内部で利用しているツール

Build System

Build CLI

Provenance Database

Policy Engine

プラットフォームエンジニアリング

開発チーム

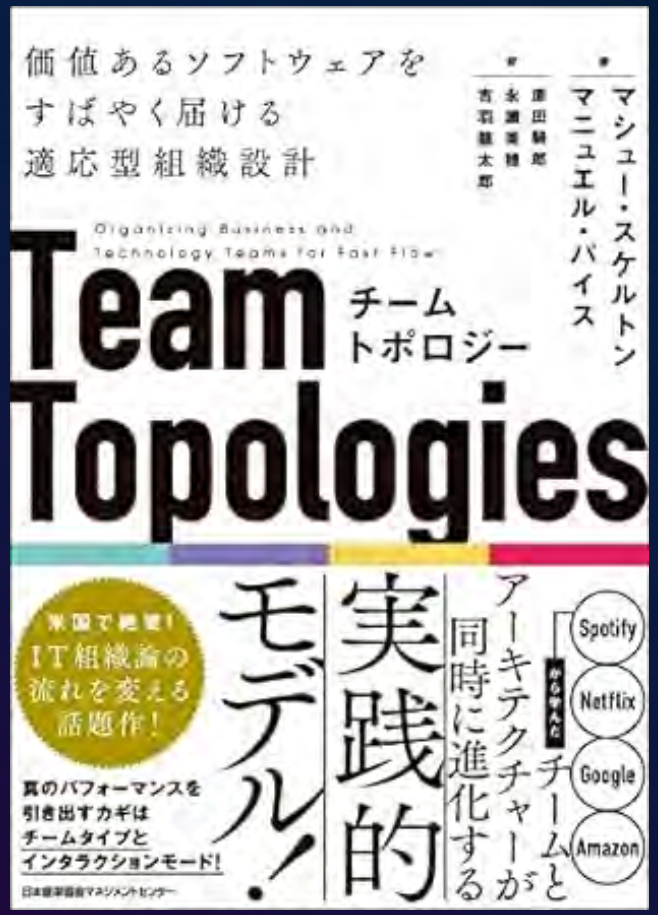
自動化

再利用可能な
コンポーネント

開発者ツール

セルフサービス
開発者ポータル

チームトポロジー



日本語版: チームトポロジー
価値あるソフトウェアをすばやく届ける適応型組織設計
https://www.amazon.co.jp/dp/4820729632

4つのチームタイプと
3つのインタラクションモード

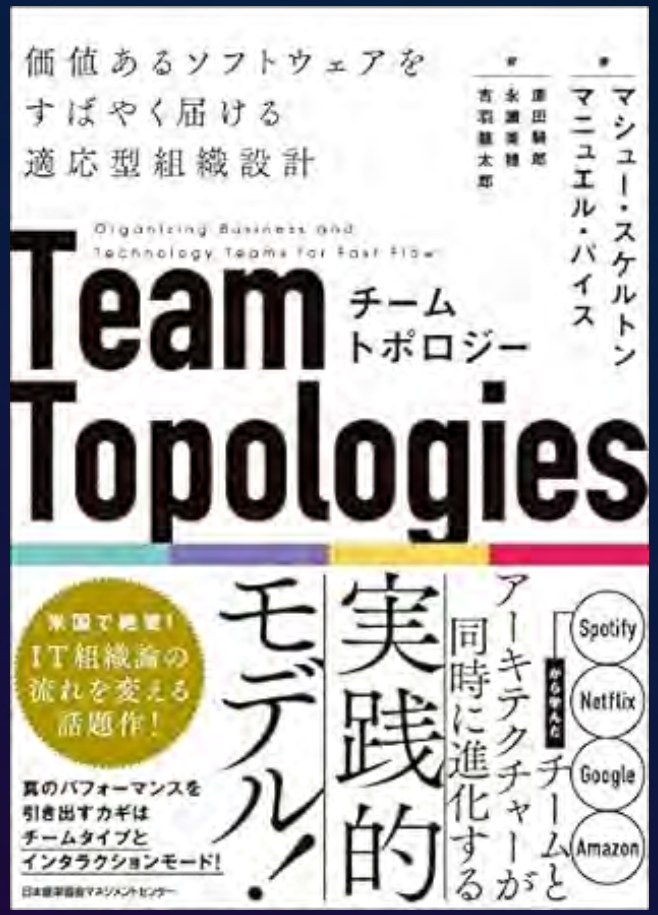
チームタイプ

ストリームアラインドチーム
→ビジネスドメインの業務フローに合わせ、
デリバリを行うチーム

他の3つのチームタイプはストリーム
アラインドチームの負荷を減らすことが目的

- イネイブリングチーム
- コンプリケイテッドサブシステムチーム
- プラットフォームチーム

チームトポロジー



日本語版: チームトポロジー
価値あるソフトウェアをすばやく届ける適応型組織設計
https://www.amazon.co.jp/dp/4820729632

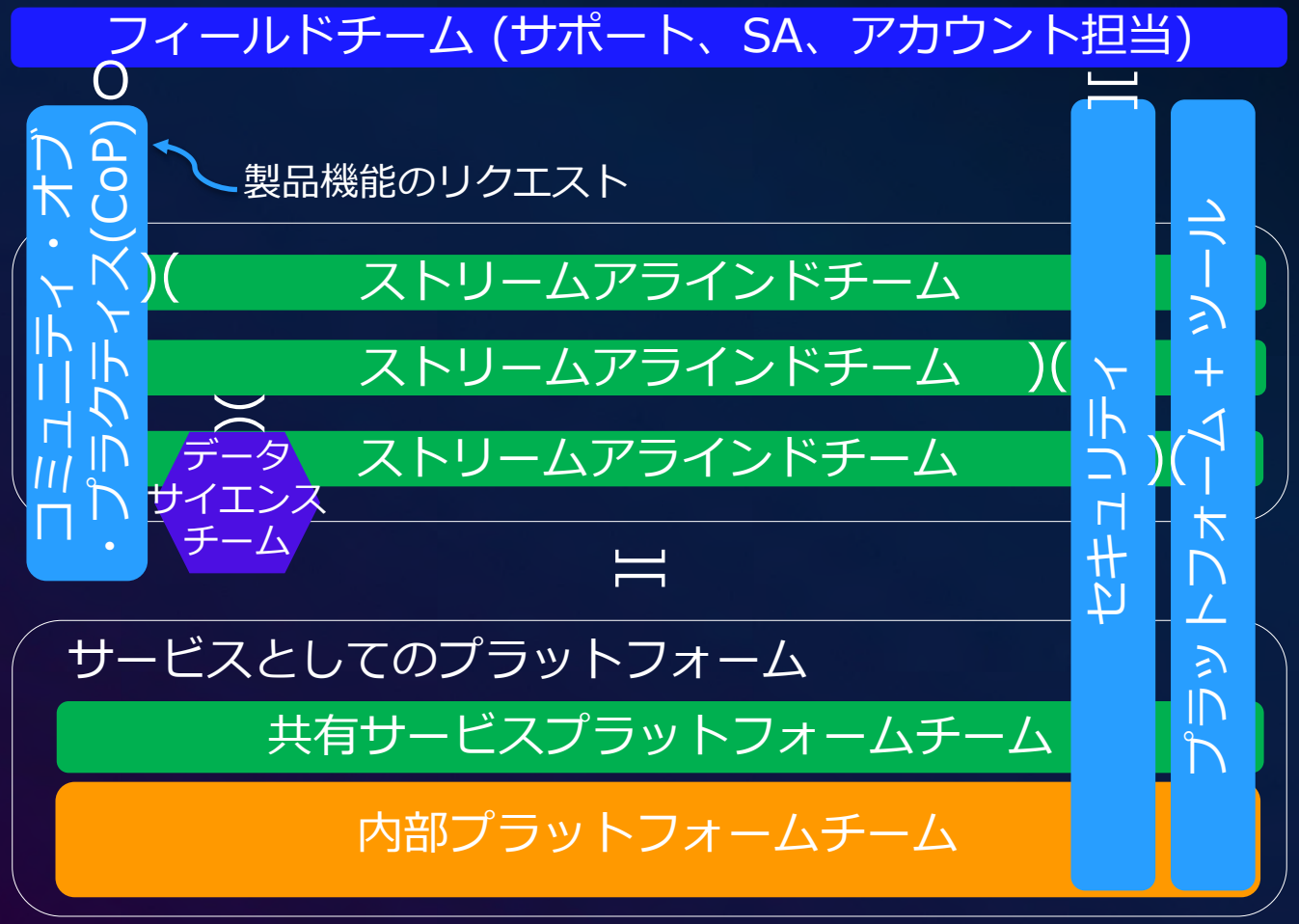
4つのチームタイプと **3つのインタラクションモード**

インタラクションモード

チーム間のインタラクション(交流) は
3つのモードに分類される

- コラボレーション
- ファシリテーション
- X-as-a-Service

AWS の組織をチームトポロジーで見てみる



キー:

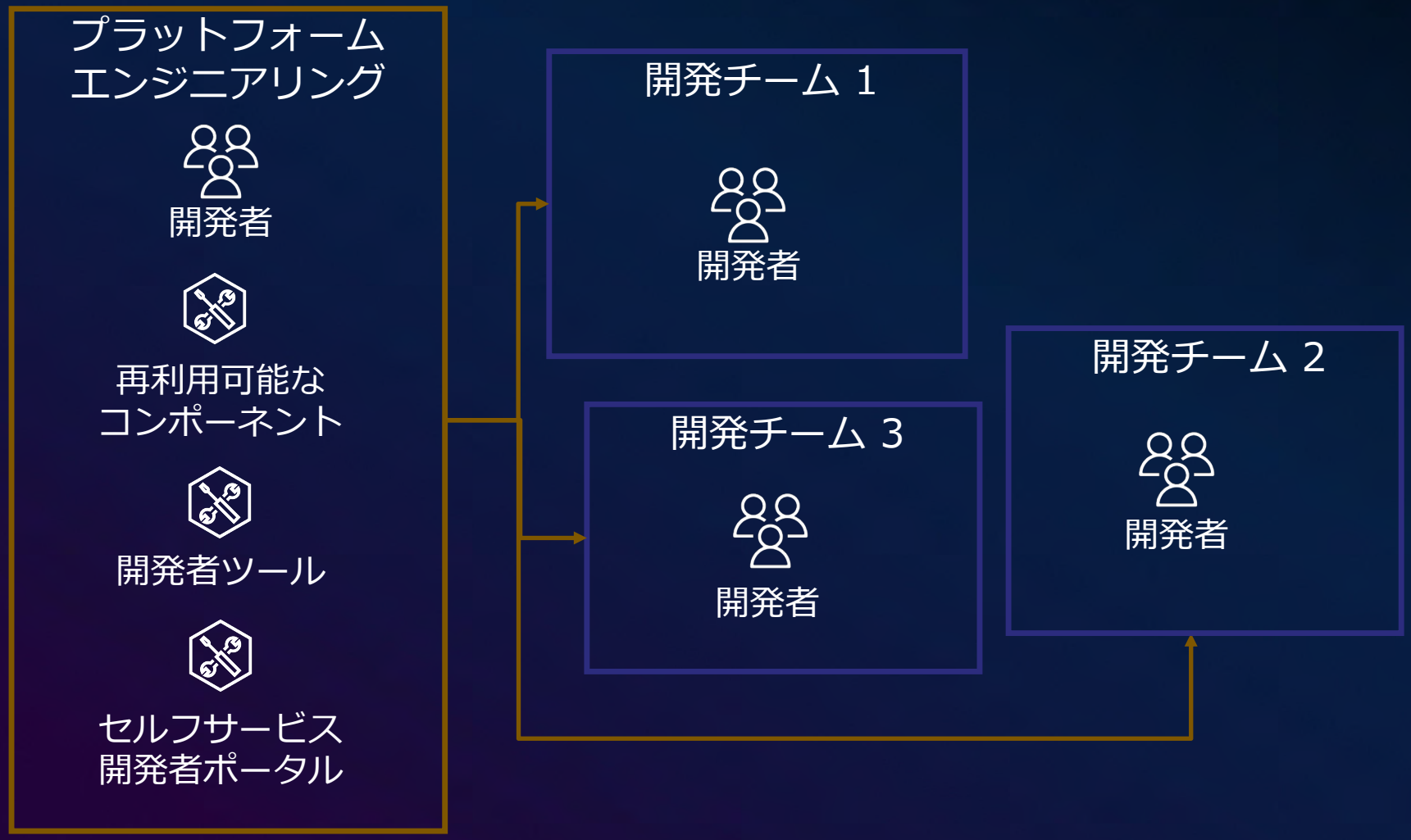
- ストリームアラインドチーム
- イネーブリングチーム
- コンプリケイテッド・サブシステムチーム

O コラボレーション

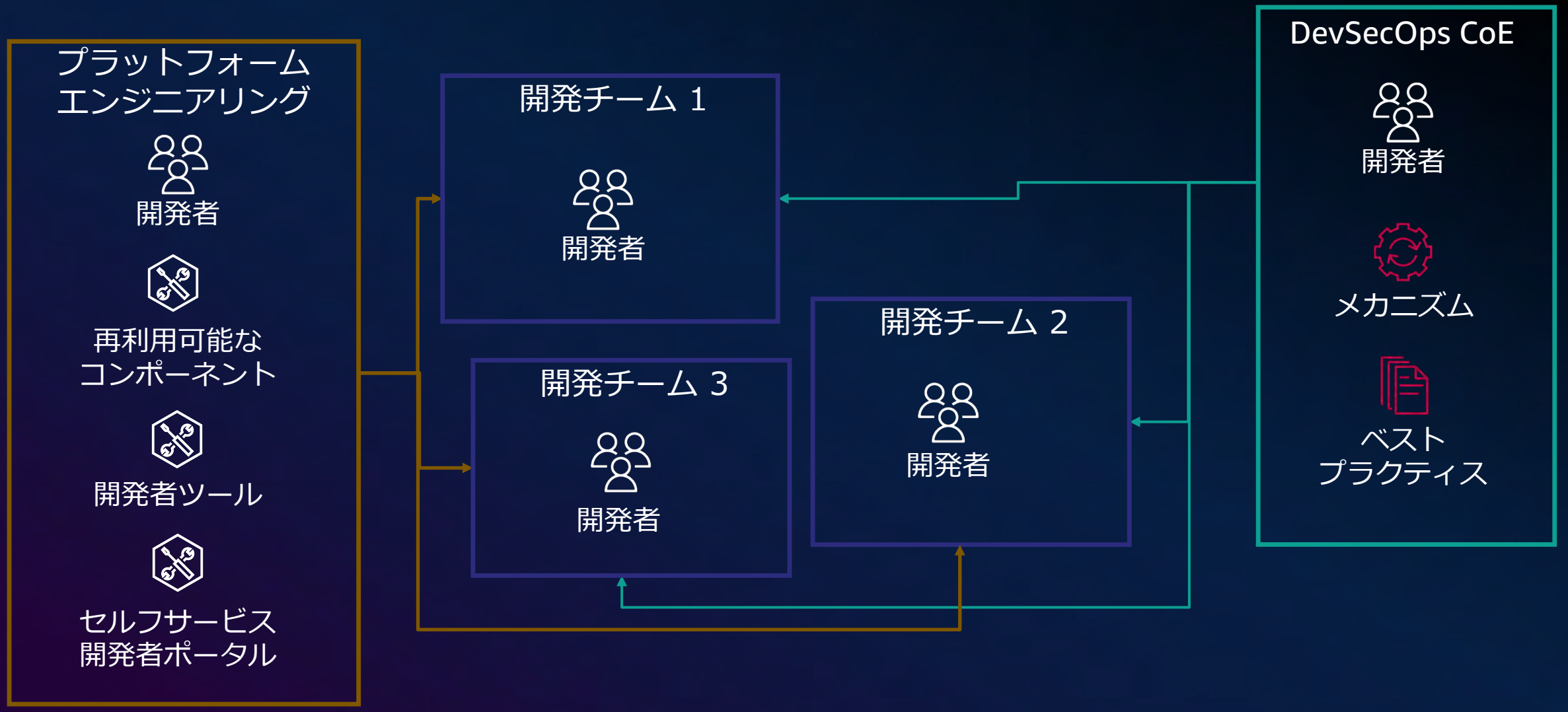
)(ファシリテーション

][X-as-a-Service

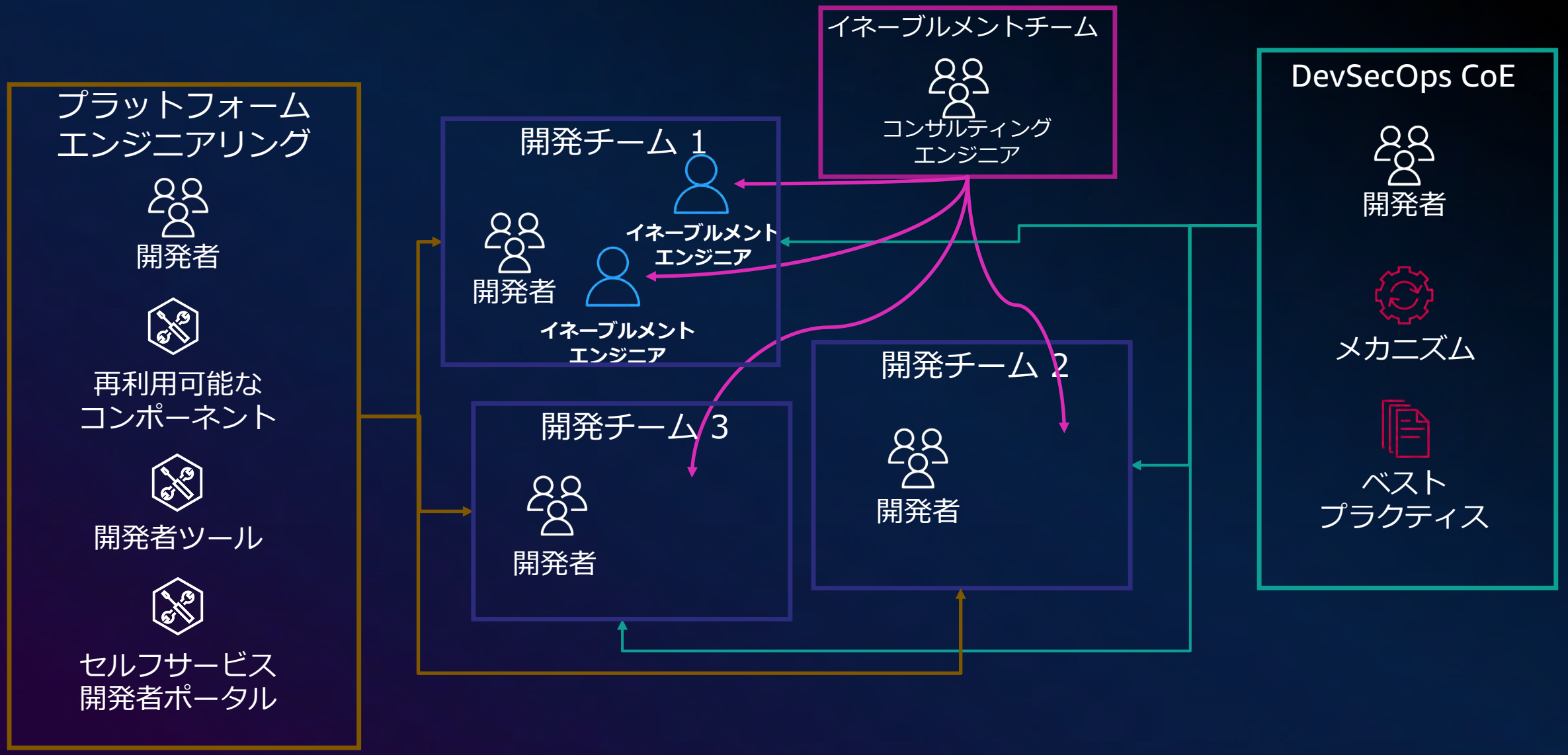
DevSecOps のスケーリング



DevSecOps のスケーリング



DevSecOps のスケーリング



運用モデル

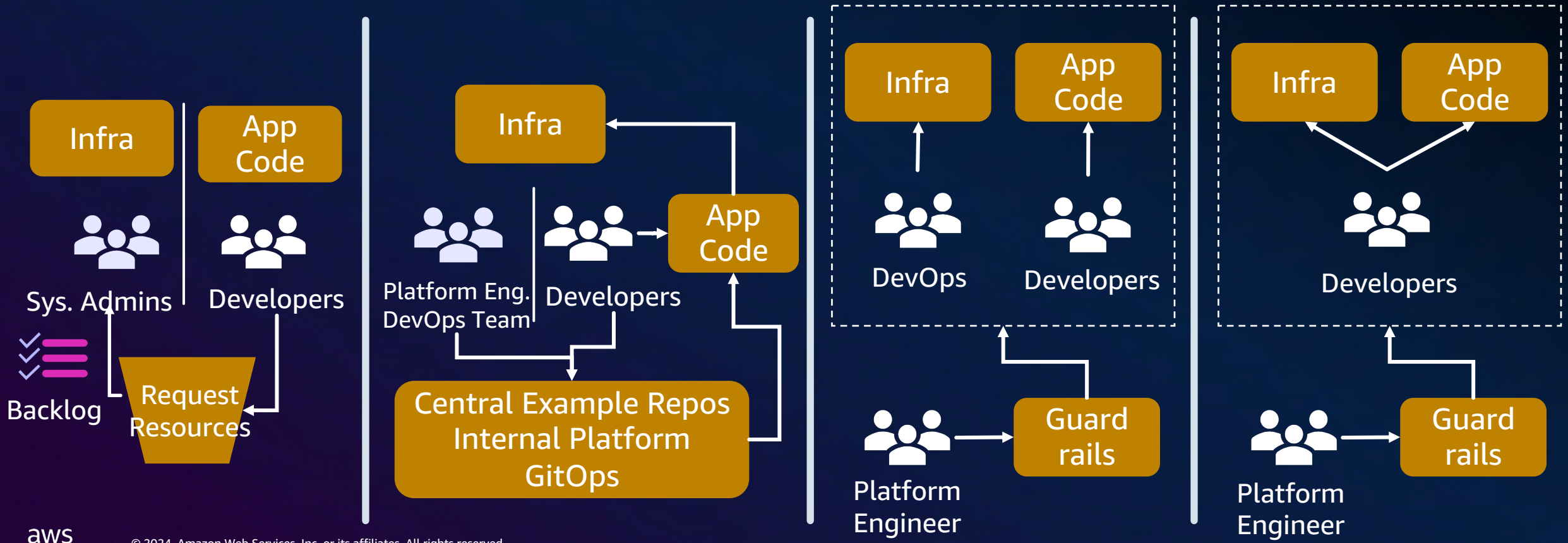
参考：組織のクラウドオペレーションをいかにモダナイズするか
https://aws.amazon.com/jp/blogs/news/how-organizations-are-modernizing-for-cloud-operations/

中央集権型
プロビジョニング

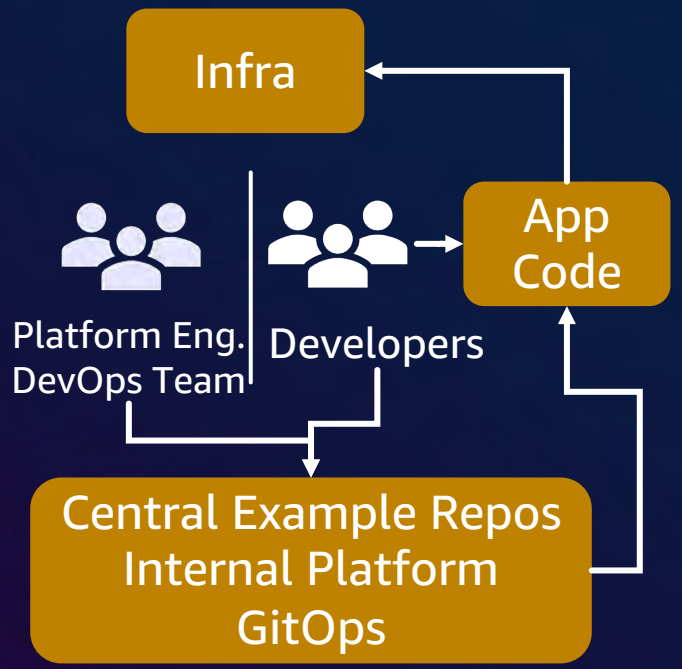
プラットフォーム型
ゴールデンパス

組み込み型
DevOps

分散型
DevOps



プラットフォーム型ゴールデンパス



プラットフォームチームが「好ましい標準」を示す
(適切なデフォルト、ガードレール、グッドプラクティス)

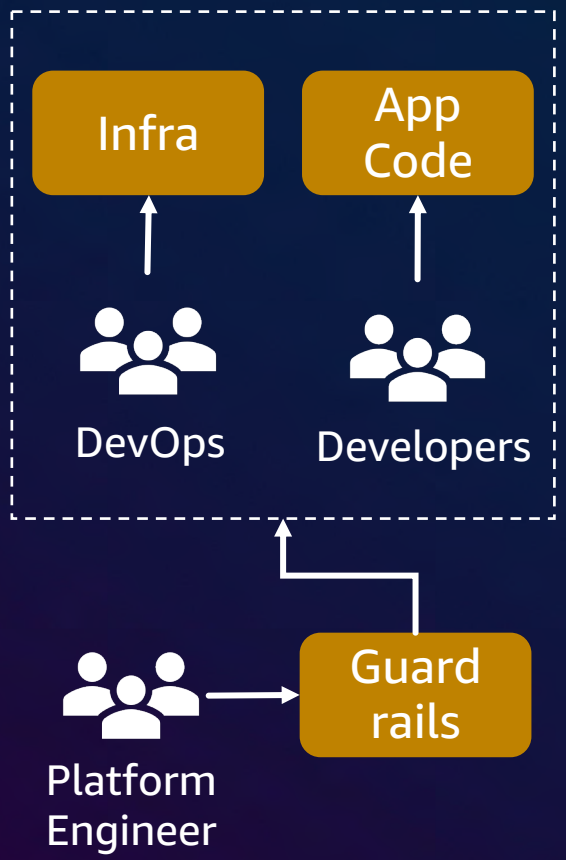
メリット

- 一貫性と柔軟性のバランスが良い
- 標準を維持しながら、カスタマイズ可能

デメリット

- 組織全体の管理を維持するのが難しい
- 標準の更新をリソース全体に伝播させるのが難しくなる

組み込み型 DevOps



DevOps エンジニアが開発チームと直接連携してインフラの対応を行うモデル

- フローティングモデル (構築完了したらチームを移動)
- 常任組み込みモデル

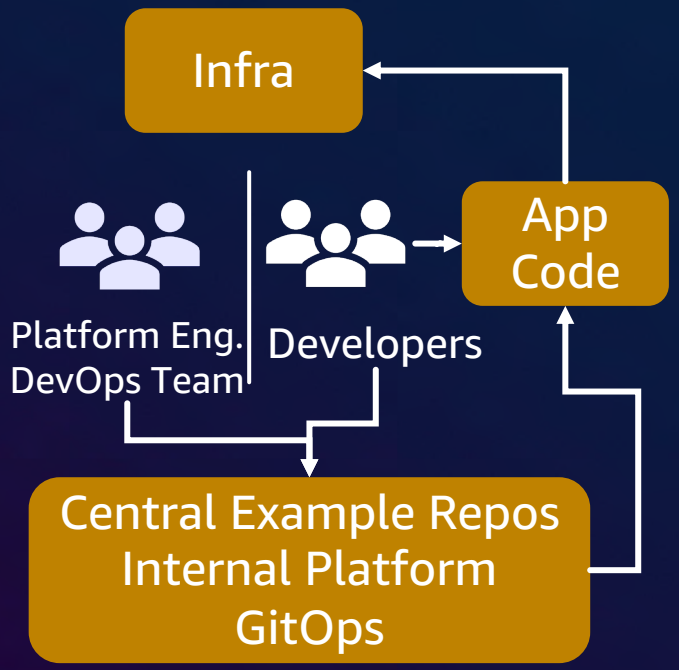
メリット

- 俊敏性と柔軟性が向上

デメリット

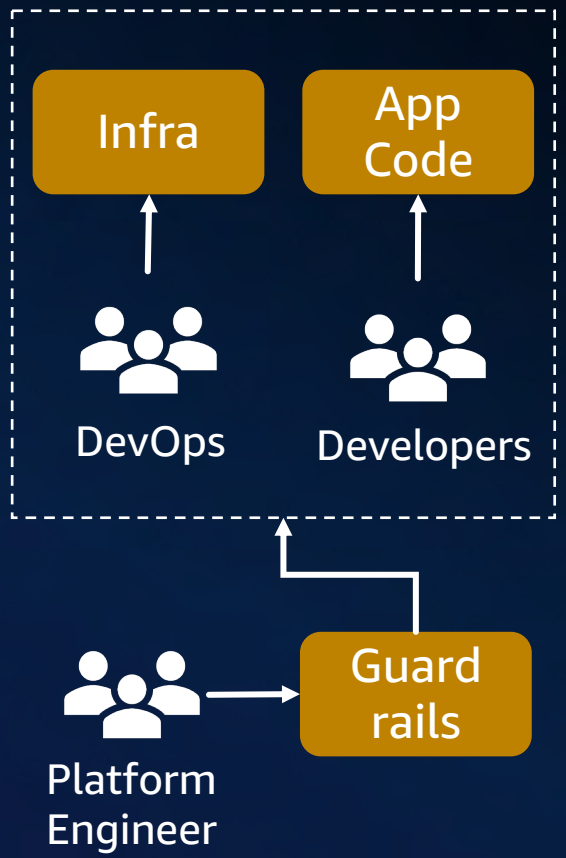
- 規模が拡大するとコストがかさむ可能性
- DevOps エンジニア間での情報共有を行わないと、アプローチにズレが出る

プラットフォーム型ゴールデンパス

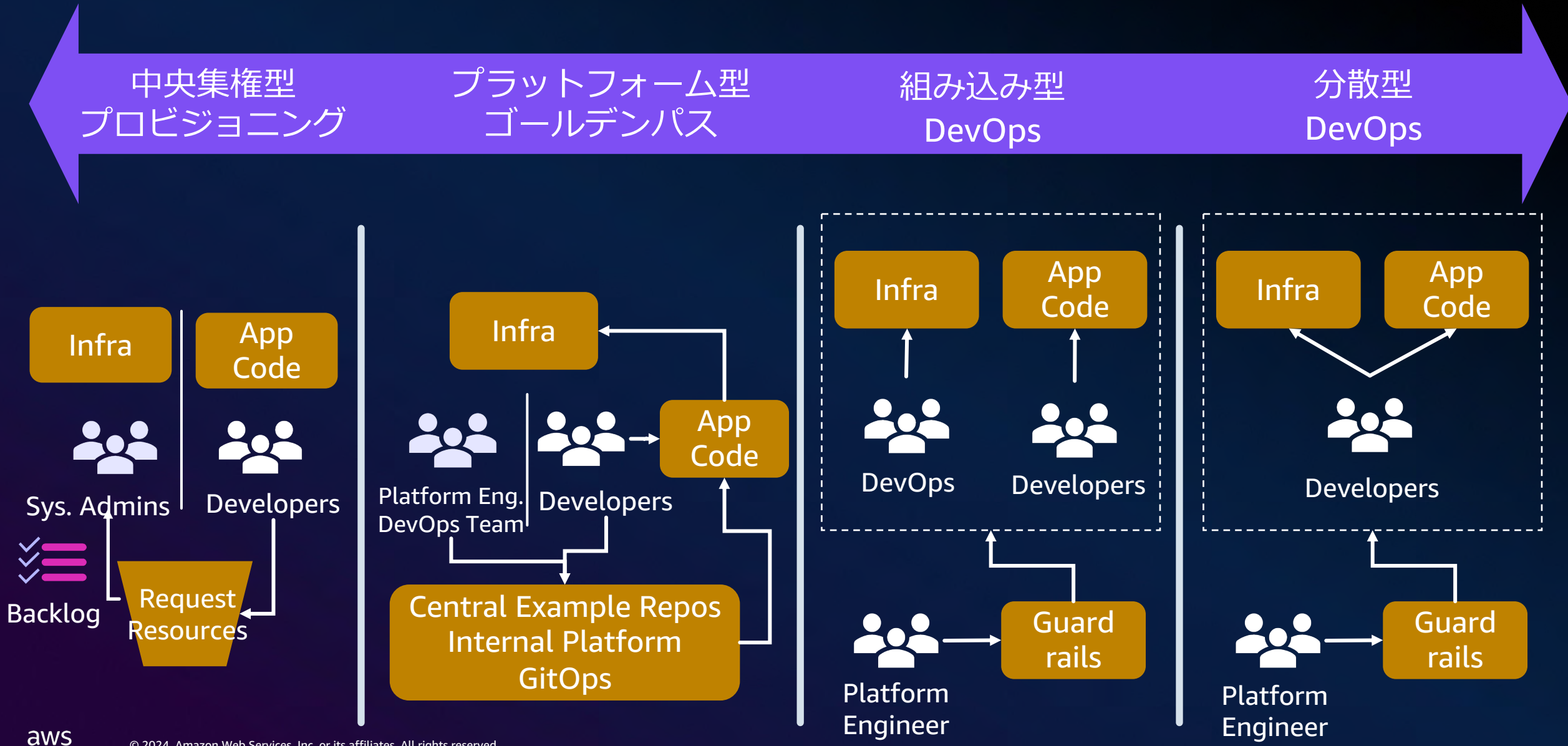


プラットフォームチームは
この2つを行き来する
ことが多い

組み込み型DevOps



皆さんの組織の現在と未来は?



まとめ

今日の話の概要(再掲)

アジリティのために
小さいチームで開発

自動化が必要

スケールするための
仕組み

Amazon が学んだソフトウェアの開発とデリバリに関する教訓

アジリティのための分割
(マイクロサービス、2 ピザチーム)

すべてを自動化する

標準化されたツール

ベルトとサスペンダー
(ガバナンス、テンプレート)

コードとしての
インフラストラクチャ

安全とスピードを両立するために

すべてを自動化する

標準化されたツール

ベルトとサスペンダー
(ガバナンス、テンプレート)

Shift Left してセキュリティのテストも自動化する

サプライチェーンのセキュリティも考慮する

自分たちにあった運用モデルを検討する

運用モデルに沿ったプラットフォームエンジニアリングを実現する

最後に、明日からやれること

- Pre-Commit フック入れてない人は入れましょう!!
- セキュリティの自動化ツール試してみましょう
  - ワークショップ: https://catalog.workshops.aws/sec4devs/ja-JP
  - Amazon Q Developer は無料ですぐに試せます
- 今の運用モデルのあるべき姿について、話し合ってみましょう

Thank you!

Masao Kanamori

kanamasa@amazon.co.jp

Appendix: 運用モデル

運用モデル

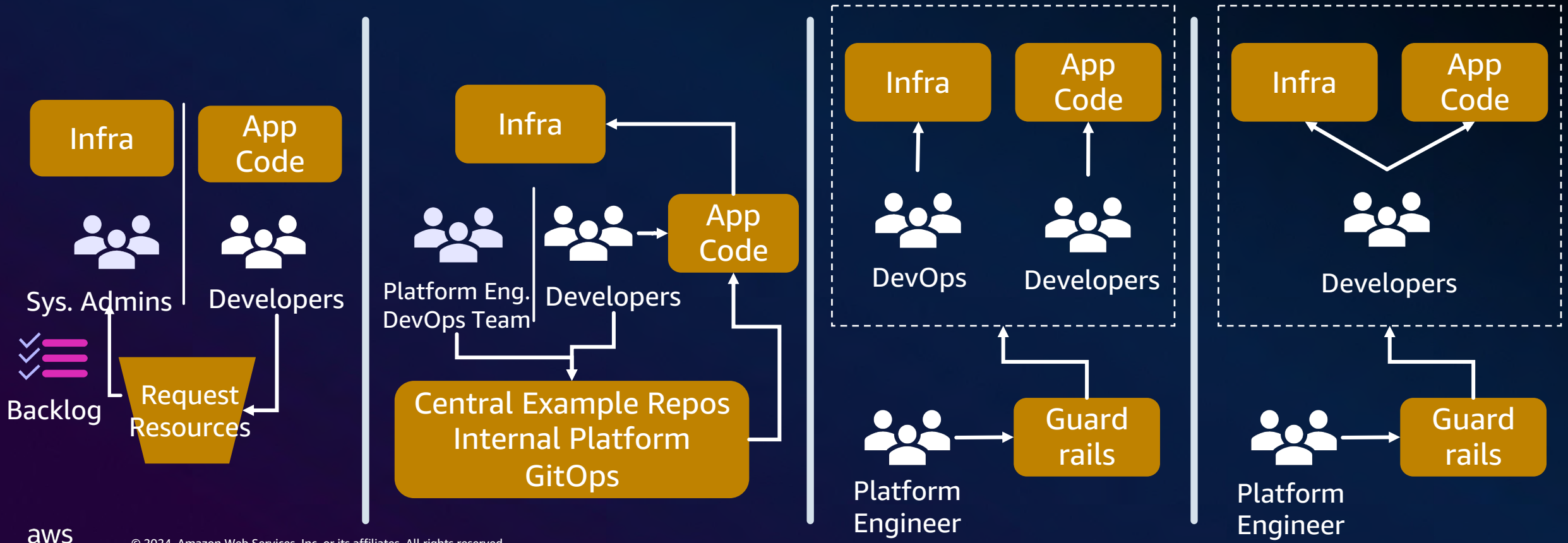
参考：組織のクラウドオペレーションをいかにモダナイズするか
https://aws.amazon.com/jp/blogs/news/how-organizations-are-modernizing-for-cloud-operations/

中央集権型
プロビジョニング

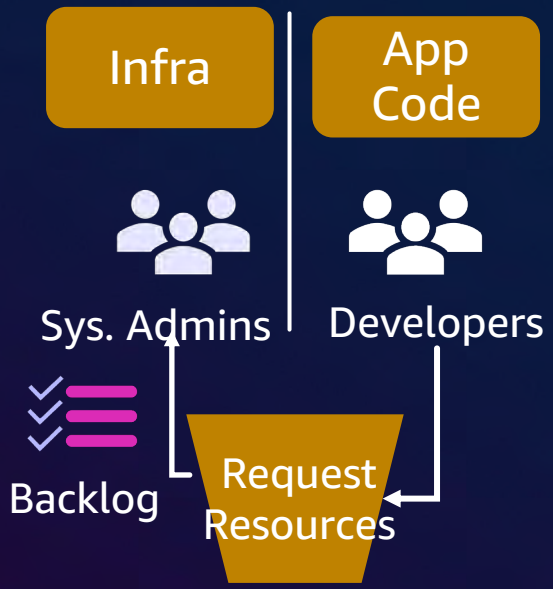
プラットフォーム型
ゴールデンパス

組み込み型
DevOps

分散型
DevOps



中央集権型プロビジョニング



インフラの設計/管理は一元化された中央チームが行う。
開発チームはリクエストを送り、
プロビジョニングされるのを待つ。

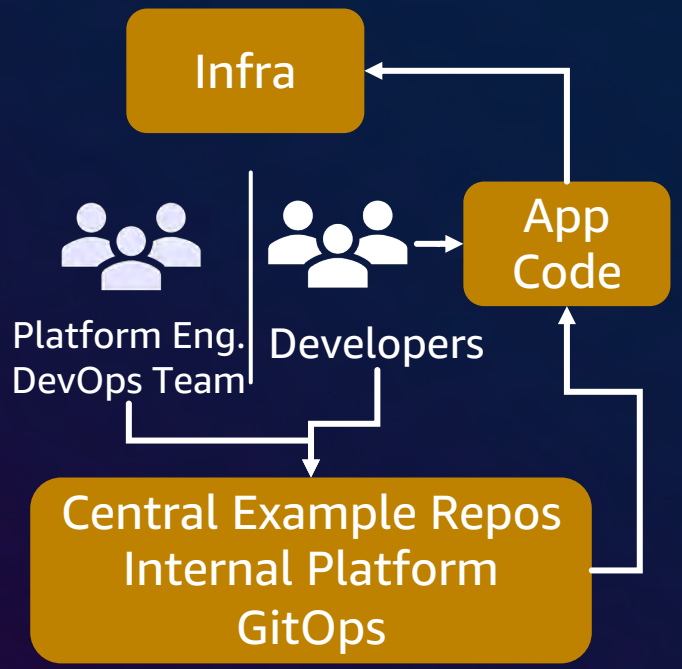
メリット

- 中央集権的な制御できる

デメリット

- デプロイサイクルとフィードバックループが遅い
- ユースケースが異なる多数の開発チームのサポートが困難

プラットフォーム型ゴールデンパス



プラットフォームチームが「好ましい標準」を示す
(適切なデフォルト、ガードレール、グッドプラクティス)

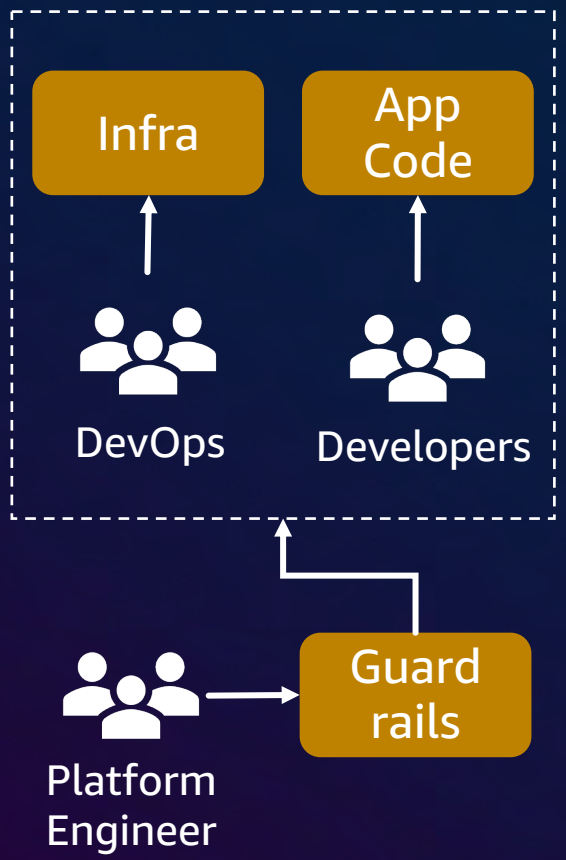
メリット

- 一貫性と柔軟性のバランスが良い
- 標準を維持しながら、カスタマイズ可能

デメリット

- 組織全体の管理を維持するのが難しい
- 標準の更新をリソース全体に伝播させるのが難しくなる

組み込み型 DevOps



DevOps エンジニアが開発チームと直接連携してインフラの対応を行うモデル

- フローティングモデル (構築完了したらチームを移動)
- 常任組み込みモデル

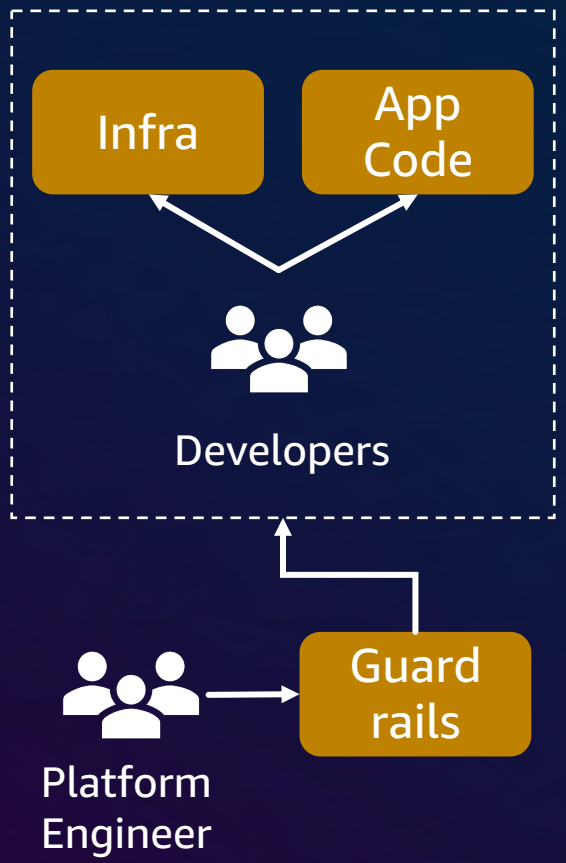
メリット

- 俊敏性と柔軟性が向上

デメリット

- 規模が拡大するとコストがかさむ可能性
- DevOps エンジニア間での情報共有を行わないと、アプローチにズレが出る

分散型 DevOps



開発チームが、インフラとパイプラインの全ての所有権と責任を持つ

プラットフォームチームはガードレールの構築に集中

メリット

- 俊敏性と柔軟性が最大に

デメリット

- 不整合、エラー、脆弱性のリスクが最も高い
- 開発チームの責任が増える(文化の変化も必要)