DevAx connect

CI/CD なのだから ちゃんとテストを書いてみよう

~ 分散環境のためのコンシューマ駆動契約を添えて ~

Masao Kanamori
Developer Specialist Solution Architect, DevAx
Amazon Web Services Japan
kanamasa@amazon.co.jp, @masaosaan

自己紹介

金森 政雄

- 所属/役職：
  DevAx(Developer Acceleration) チーム
  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 Simulator

今日お話しすること

- CI/CD でなぜテストが必要なのか
- テストコードが書きやすいソフトウェア設計について
- テストコードがないレガシーコードではどうすれば良いのか

今日のゴール

明日から1つでもテスト自動化をしてみようと思ってもらう

まずは質問から

CI/CD やってますか?

自動テストはありますか?

今日のアジェンダ

1. なぜCI/CD に自動化テストが必要なのか
2. 自動テストの作り方 - 理想編 –
3. 仕様化テスト – 現実編 -
4. コンシューマ駆動契約
5. まとめ

今日のアジェンダ

1. なぜCI/CD に自動テストが必要なのか
2. 自動テストの作り方 - 理想編 –
3. 仕様化テスト – 現実編 -
4. コンシューマ駆動契約
5. まとめ

なぜCI/CD に自動テストが必要なのか

aws DevAx connect

CI/CDの目的

Day1 資料より

誰かが良いアイディアを思いついたとき、できるだけ早くユーザーに届けるためのソフトウェア開発手法

- ソフトウェアのビルド・デプロイ・テスト・リリースというプロセスのあらゆる部分が関係者全員から見えるようにし、共同作業をやりやすくすること
- フィードバックを改善し、プロセスにおいてできる限り早い時期に問題が特定されて解決されるようにすること
- ソフトウェアの任意のバージョンを任意の環境に対して、完全に自動化されたプロセスを通じて好きなようにデプロイできるようにすること

CI/CDの目的

Day1 資料より

誰かが良いアイディアを思いついたとき、できるだけ早くユーザーに届けるためのソフトウェア開発手法

- ソフトウェアのビルド・デプロイ・テスト・リリースというプロセスのあらゆる部分が関係者全員から見えるようにし、共同作業をやりやすくすること

- フィードバックを改善し、プロセスにおいてできる限り早い時期に問題が特定されて解決されるようにすること

- ソフトウェアの任意のバージョンを任意の環境に対して、完全に自動化されたプロセスを通じて好きなようにデプロイできるようにすること

リリースプロセスのステージ

Day1 資料より

- ソースコードのチェックイン
- コードのピアレビュー

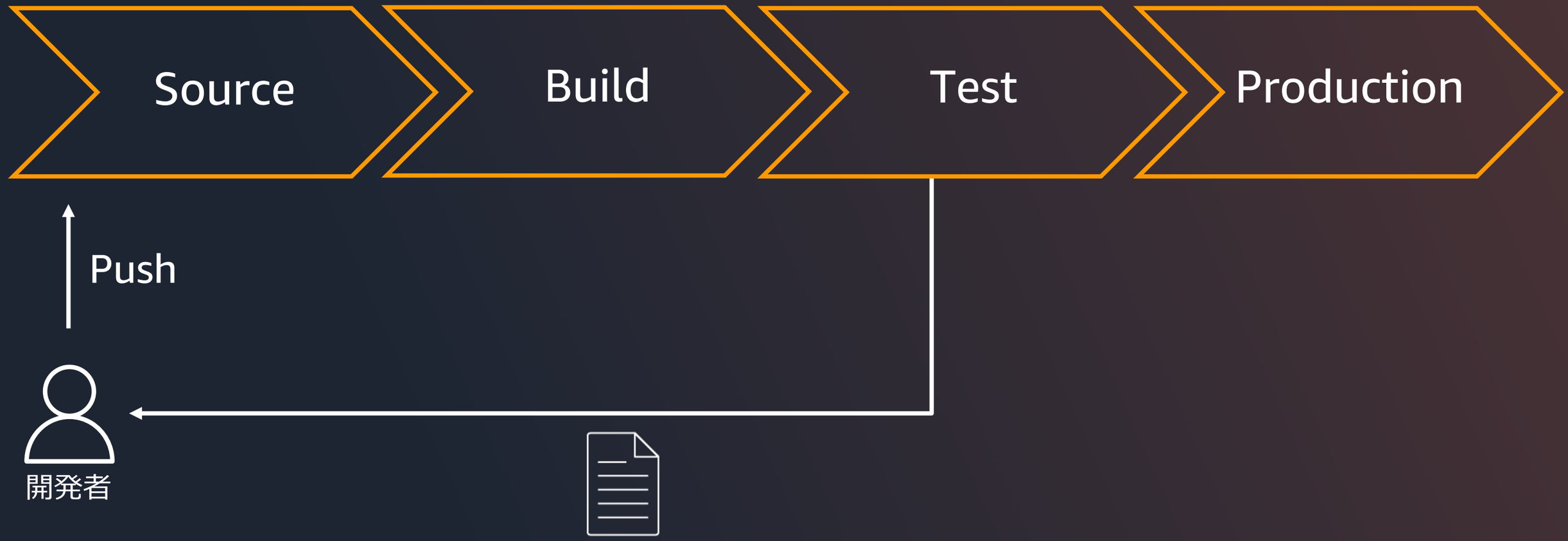
- コードのコンパイル
- ユニットテスト
- スタイルチェッカー
- コンテナイメージ、関数デプロイパッケージの作成

- 周辺システムとの統合テスト
- 負荷テスト
- UIテスト
- セキュリティテスト

- 本番環境へのデプロイ
- エラーを素早く検知するための本番環境のモニタリング

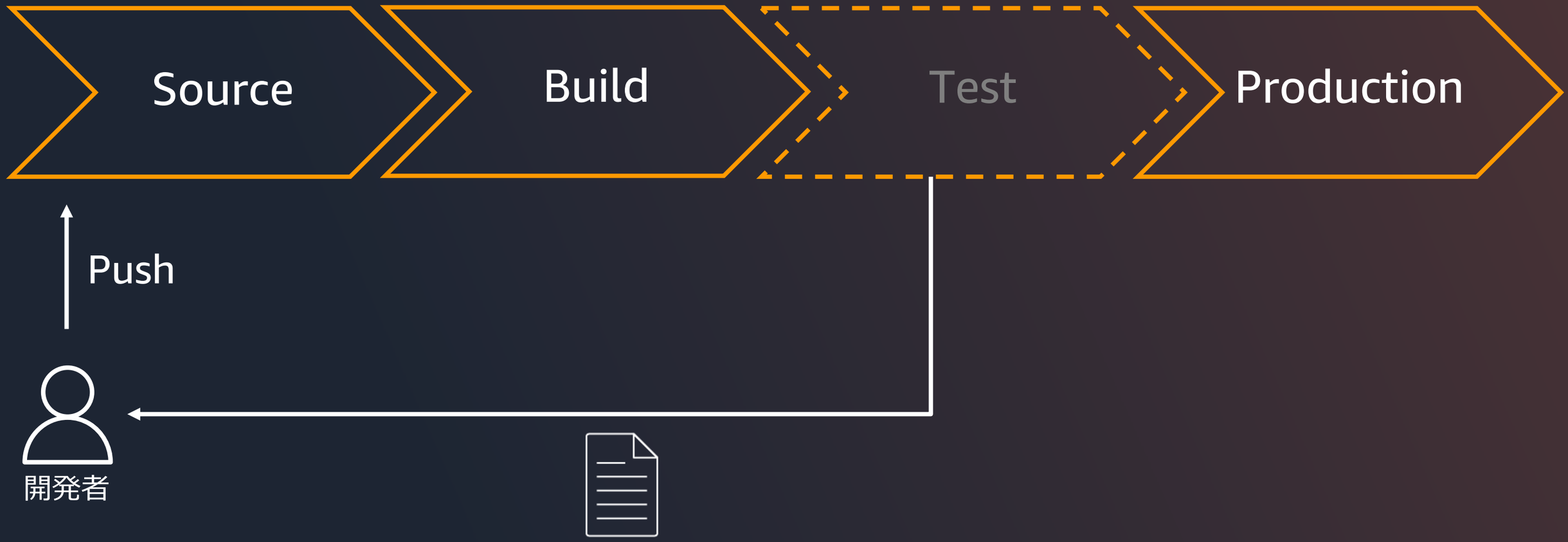
aws DevAx

CI/CD のフィードバック



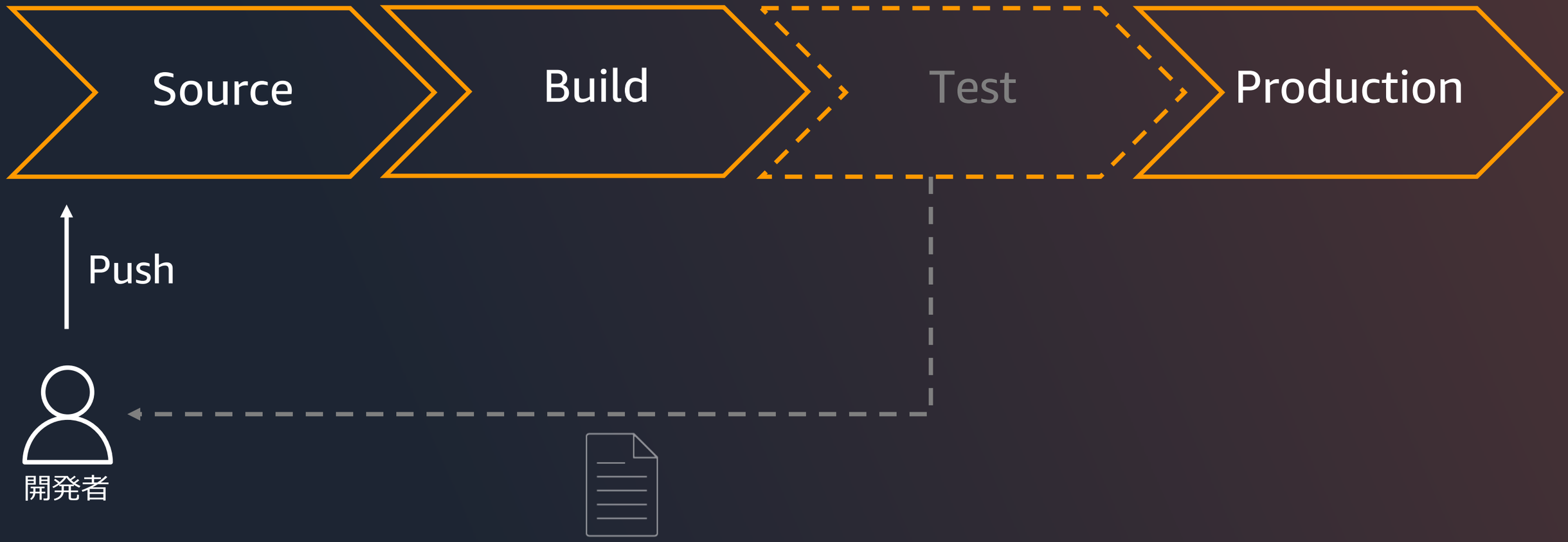
aws DevAx

テストがなければ



aws DevAx

フィードバックもなくなる



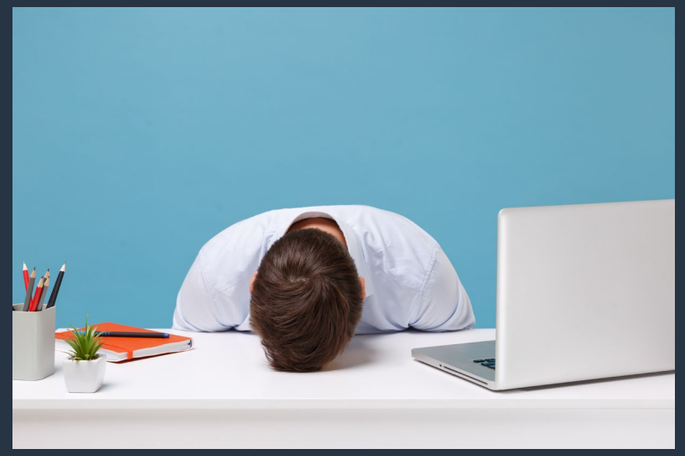
aws DevAx

CI/CD におけるテストの役割

○ テストがある

- フィードバックを早く得る
- 安心して/自信を持って開発できる

✕ テストがない/遅い/不安定



- フィードバックが得られない
- 正しく動くか不安

自動化テストは

開発者のためのガードレール

ガードレールの必要性

「統制」によってブレーキをかけるのではなく、
道の外にはみ出ることだけはないようにして、
柔軟に運転を楽しんでもらおうという考え方

『私にとって、レガシーコードとは、単にテストのないコードです。』

テストのないコードは悪いコードである。
どれだけ上手く書かれているかは関係ない。どれだけ美しいか、
オブジェクト指向か、きちんとカプセル化されているかは関係ない。
テストがあれば、検証しながらコードの動きを素早く変更することができる。テストがなければ、コードが良くなっているのか
悪くなっているのかが本当にはわからない

引用：『レガシーコード改善ガイド』
マイケル・C・フェザーズ著
平澤 章 /越智 典子/稲葉 信之/田村 友彦/小堀 真義 翻訳

なぜテストを書かないのか

前半の質問やアンケートから

- 誰がテストを書くべきなのかわからない
- いつテストを書くべきなのかわからない
- どこまでテストを書くべきなのかわからない
- テストにかかる工数をどうやって説明すれば良いかわからない

その他、よく聞く理由

- 急いでいる案件なので”今回は”書きません
- このコードは変更しない予定(使い捨て）です

なぜテストを書かないのか

前半の質問やアンケートから

- **誰がテストを書くべきなのかわからない** → **もちろん、あなたです!**
- いつテストを書くべきなのかわからない
- どこまでテストを書くべきなのかわからない
- テストにかかる工数をどうやって説明すれば良いかわからない

その他、よく聞く理由

- 急いでいる案件なので”今回は”書きません
- このコードは変更しない予定(使い捨て）です

なぜテストを書かないのか

前半の質問やアンケートから

- 誰がテストを書くべきなのかわからない
- **いつテストを書くべきなのかわからない**
- どこまでテストを書くべきなのかわからない
- テストにかかる工数をどうやって説明すれば良いかわからない

その他、よく聞く理由

- 急いでいる案件なので”今回は”書きません
- このコードは変更しない予定(使い捨て）です

aws DevAx connect

いつ自動化するのか

“Anything that you do more than twice has to be automated.”

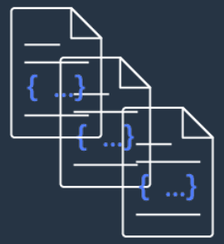
2回以上やることは全て自動化されるべきだ

By Adam Stone CEO, D-Tools

既存のソースコードを変更する手順

よろしく!!

とてもイケてるが
既存のコードを
変更しないと
いけない要件



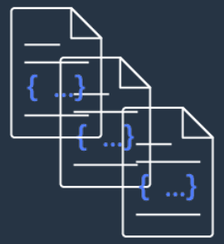
既存のコードたち
(初めて触ります)

どんな手順でコードを変更するか
考えてみてください!!

既存のソースコードを変更する手順

よろしく!!

とてもイケてるが
既存のコードを
変更しないと
いけない要件



既存のコードたち
(初めて触ります)

多分こうするであろう手順

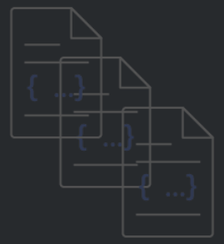
1. 既存のコードの動作確認をする
2. コードを書き換える
3. 期待通りの動作か確認する
4. 1～3を完成するまで繰り返す

既存のソースコードを変更する手順

よろしく!!

動作確認を**2回**実施

→自動化すべき



とてもイケてるが
既存のコードを
変更しないと
いけない要件

既存のコードたち
(初めて触ります)

多分こうするであろう手順

1. 既存のコードの動作確認をする
2. コードを書き換える
3. 期待通りの動作か確認する
4. 1～3を完成するまで繰り返す

既存のソースコードを変更する手順

よろしく!!

動作確認を**2回**実施

→自動化すべき

とてもイケてるが
既存のコードを
変更しないと
いけない要件

既存のコードたち
(初めて触ります)

繰り返しもある

多分こうするであろう手順

1. 既存のコードの動作確認をする

2. コードを書き換える

3. 期待通りの動作か確認する

4.1〜3を完成するまで繰り返す

なぜテストを書かないのか

前半の質問やアンケートから

- 誰がテストを書くべきなのかわからない
- **いつテストを書くべきなのかわからない** → **コードを書き換える時**
- どこまでテストを書くべきなのかわからない
- テストにかかる工数をどうやって説明すれば良いかわからない

その他、よく聞く理由

- 急いでいる案件なので”今回は”書きません
- このコードは変更しない予定(使い捨て）です

なぜテストを書かないのか

前半の質問やアンケートから

- 誰がテストを書くべきなのかわからない
- いつテストを書くべきなのかわからない
- どこまでテストを書くべきなのかわからない
- テストにかかる工数をどうやって説明すれば良いかわからない

その他、よく聞く理由

- 急いでいる案件なので”今回は”書きません
- このコードは変更しない予定(使い捨て）です

2回動作確認はするだろうから自動化しておいた方が良い

なぜテストを書かないのか

前半の質問やアンケートから

- 誰がテストを書くべきなのかわからない
- いつテストを書くべきなのかわからない
- どこまでテストを書くべきなのかわからない
- テストにかかる工数をどうやって説明すれば良いかわからない

その他、よく聞く理由

- 急いでいる案件なので”今回は”書きません
- このコードは変更しない予定(使い捨て）です

aws DevAx connect

今日のアジェンダ

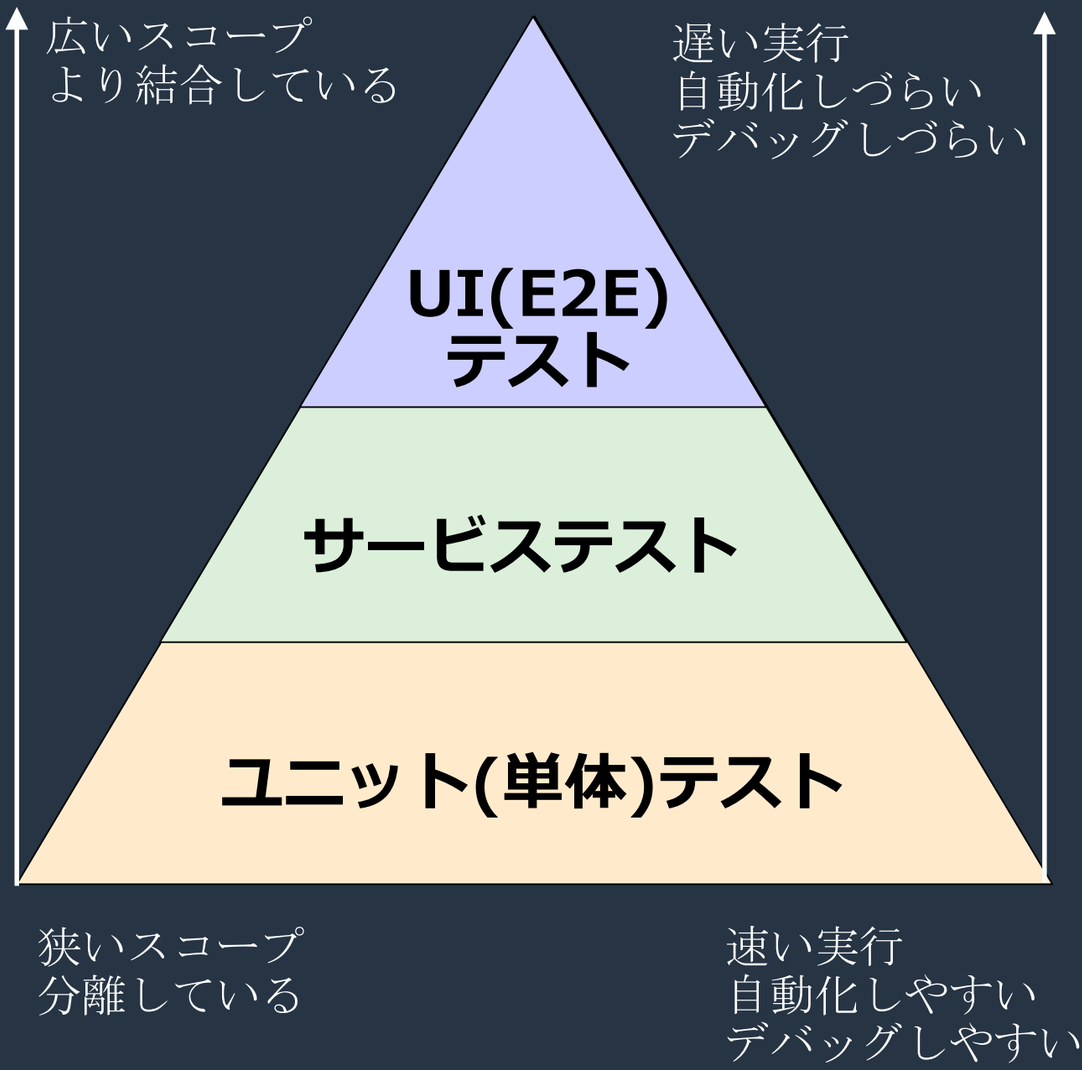
1. なぜCI/CD に自動テストが必要なのか
2. 自動テストの作り方 - 理想編 –
3. 仕様化テスト – 現実編 -
4. コンシューマ駆動契約
5. まとめ

自動テストの作り方 – 理想編

aws DevAx connect

テストピラミッド

- Mike Cohn が
  『Succeeding with Agile』で説明
- 上に行くほどスコープと実行時間↑
  逆に下に行くとスコープ/実行時間↓
- ユニットテストを多くし、
  E2Eテストは重要なジャーニーに絞る
  (逆ピラミッドにならないようにする)



Unit Test対象の関数をTestableにする

- 関数をTestable(テスト可能)にするとは？
  - y = f(x) の形にする
  - xに特定の値を与えるとyの値が決まる

```
# add関数
def add(a, b):
    return a + b

def test_add():
    #unit test
    expected = 10  #期待する値
    assert expected == add(5, 5)
```

Pure Logicに対するUnite Test

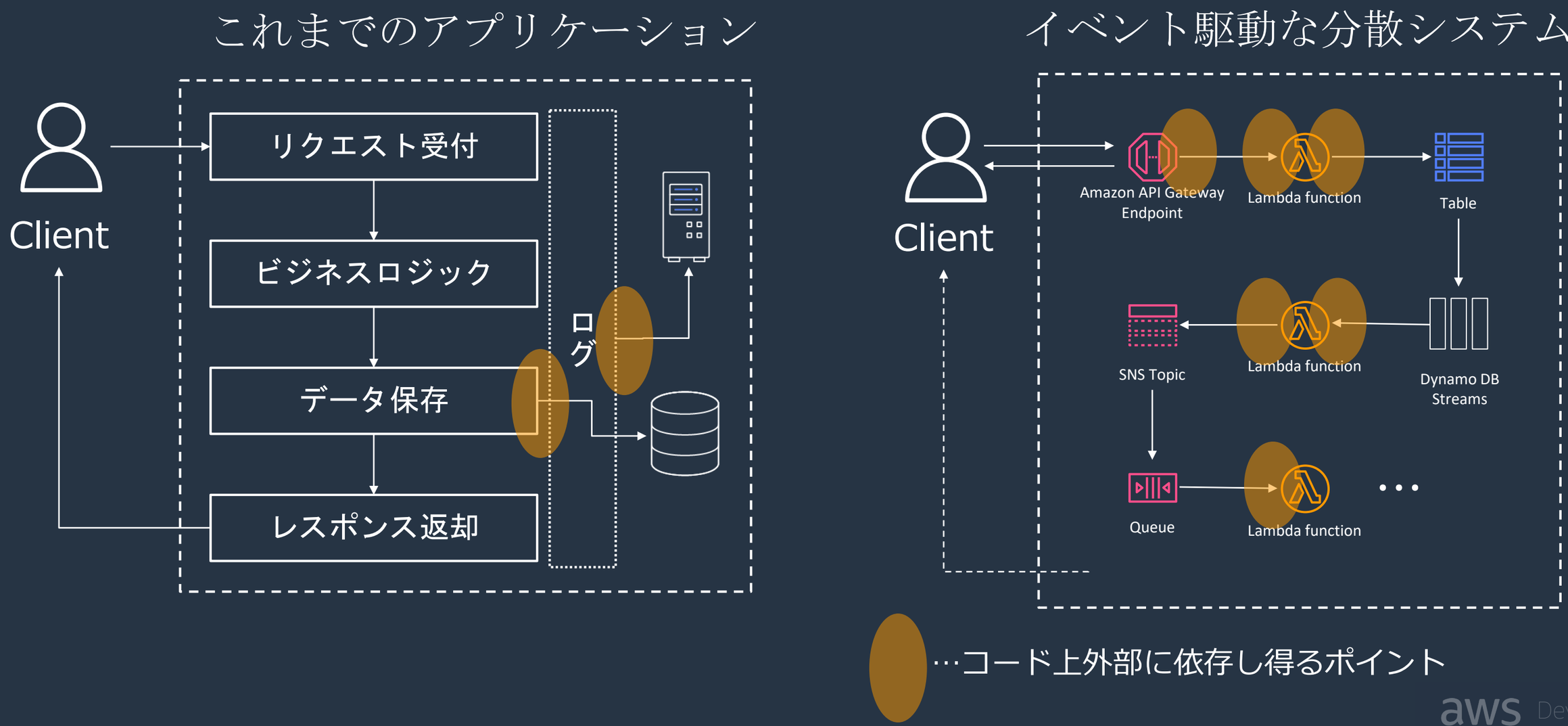
- 純粋な要件に対してテストケースを書く
  他のAWSサービスとの結合方法に対する知識を含まない

- テストが軽くなるため、開発やデリバリーを高速化できる

- 外部のアーキテクチャが変更された時にも、
  テストケースが影響を受けにくい
  - テストの陳腐化が防げる
  - テストのメンテナンスコストの軽減

- Pure Logicなので意図が伝わりやすい、理解しやすい
  - テストケースから仕様を理解出来る

外部の依存関係は増えている



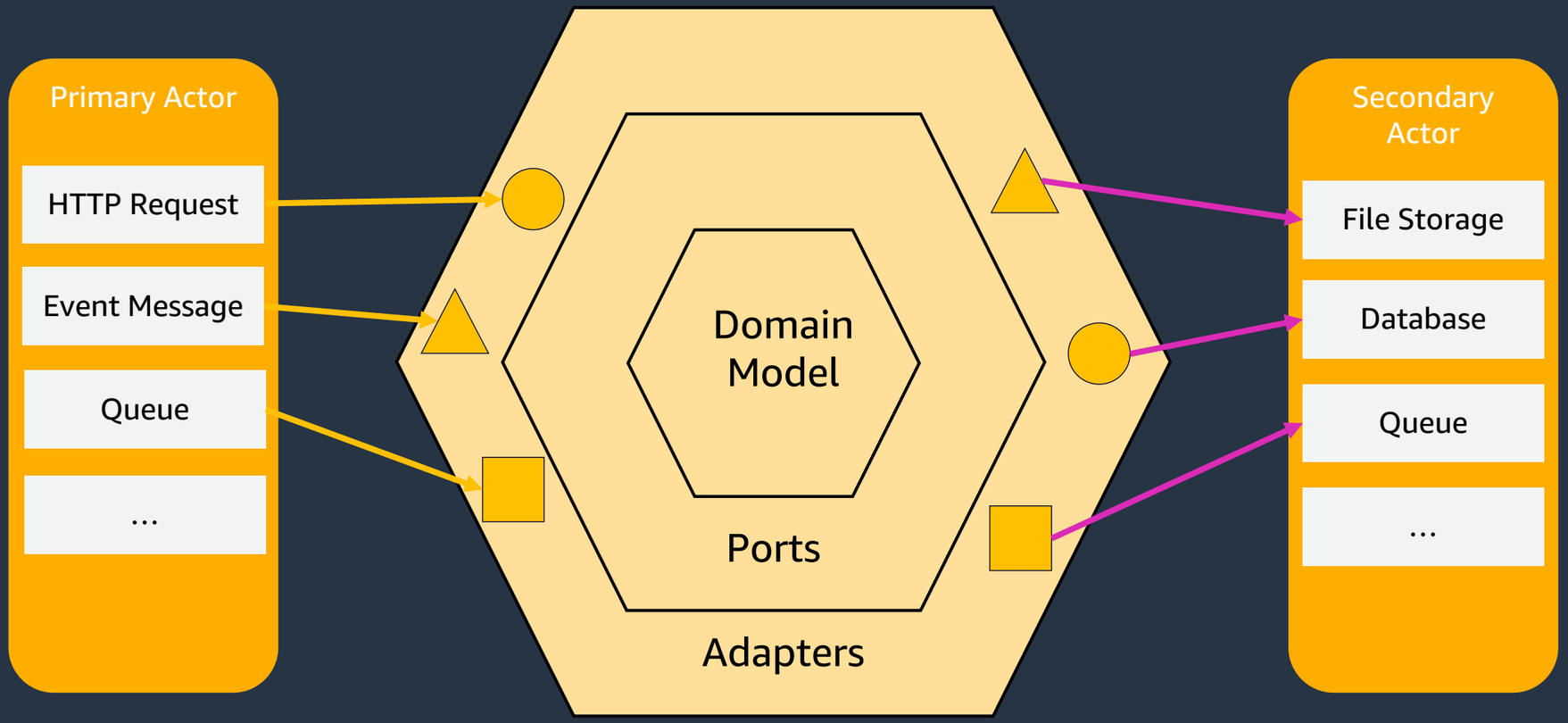
ヘキサゴナルアーキテクチャ

- ヘキサゴナルアーキテクチャは、
  別名ポートとアダプタアーキテクチャとも言われる

- ポートとアダプタによってアプリケーションのコンポーネントが
  容易に環境との間で疎結合に結ばれる
  ソフトウェア設計のアーキテクチャパターン
  https://alistair.cockburn.us/hexagonal-architecture/

- オリジナルは Dr. Alistair Cockburn が提唱
  https://alistair.cockburn.us/coming-soon/

ヘキサゴナルアーキテクチャ



ヘキサゴナルアーキテクチャ

aws-samples/aws-lambda-domain-sample

GitHubでサンプルプロジェクトを公開
https://github.com/aws-samples/aws-lambda-domain-model-sample

README.md

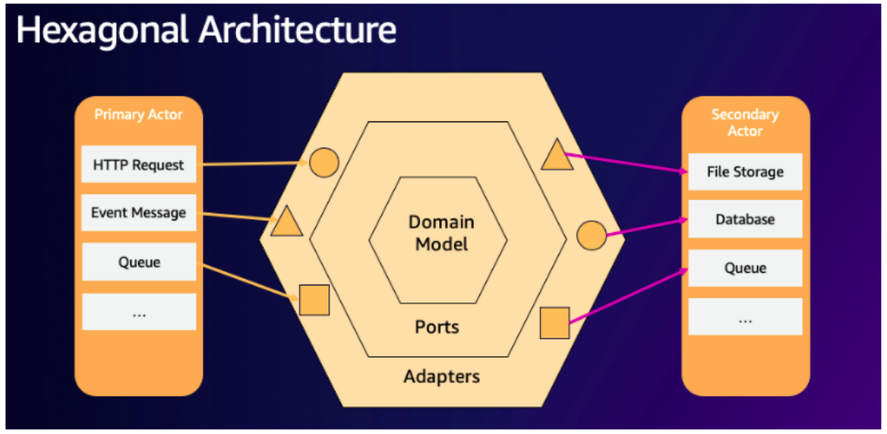
Domain Model objects on AWS Lambda with Hexagonal Architecture Sample

[-Readme in Japanese-]

What is this project?

This project contains a Lambda function with domain model objects. By using Hexagonal Architecture (Ports and Adapters pattern), it separates domain model from other layer code.

The Hexagonal Architecture, or ports and adapters architecture, is an architectural pattern used in software design. The hexagonal architecture was invented by Dr. Alistair Cockburn.



Contributors 2

fatsushi Atsushi Fukui

amazon-auto Amazon GitHub Autom...

Languages

Python 96.3% Shell 3.7%

AWS Summit 2022
Developer Zone のセッションも
ご覧ください!!

https://bit.ly/3xzfm5x

aws DevAx connect

テストしやすい設計、どう作る?

テストしやすいコードを作るには

1. テストしやすい設計をする: **慣れてないと難しい (かも)**

2. 小まめにテストを書く: **自然に**テストしやすい設計に

- おすすめはテストファースト(含むTDD)
- 変更するときに必ずテストをかく

多くの場合は継続案件

- テストがない
- テストがしやすい設計になっていない
- 絡まり合った依存関係
- ドキュメントもない

今日のアジェンダ

1. なぜCI/CD に自動テストが必要なのか
2. 自動テストの作り方 - 理想編 –
3. 仕様化テスト – 現実編 -
4. コンシューマ駆動契約
5. まとめ

仕様化テスト - 現実編 -

aws DevAx connect

Unit Test対象の関数をTestableにする

再掲

- 関数をTestable(テスト可能)にするとは？
  - y = f(x) の形にする
  - xに特定の値を与えるとyの値が決まる

```
# add関数
def add(a, b):
    return a + b

def test_add():
    #unit test
    expected = 10  #期待する値
    assert expected == add(5, 5)
```

Unit Test対象の関数をTestableにする

- 関数をTestable(テスト可能)にするとは？
  - y = f(x) の形にする
  - xに特定の値を与えるとyの値が決まる

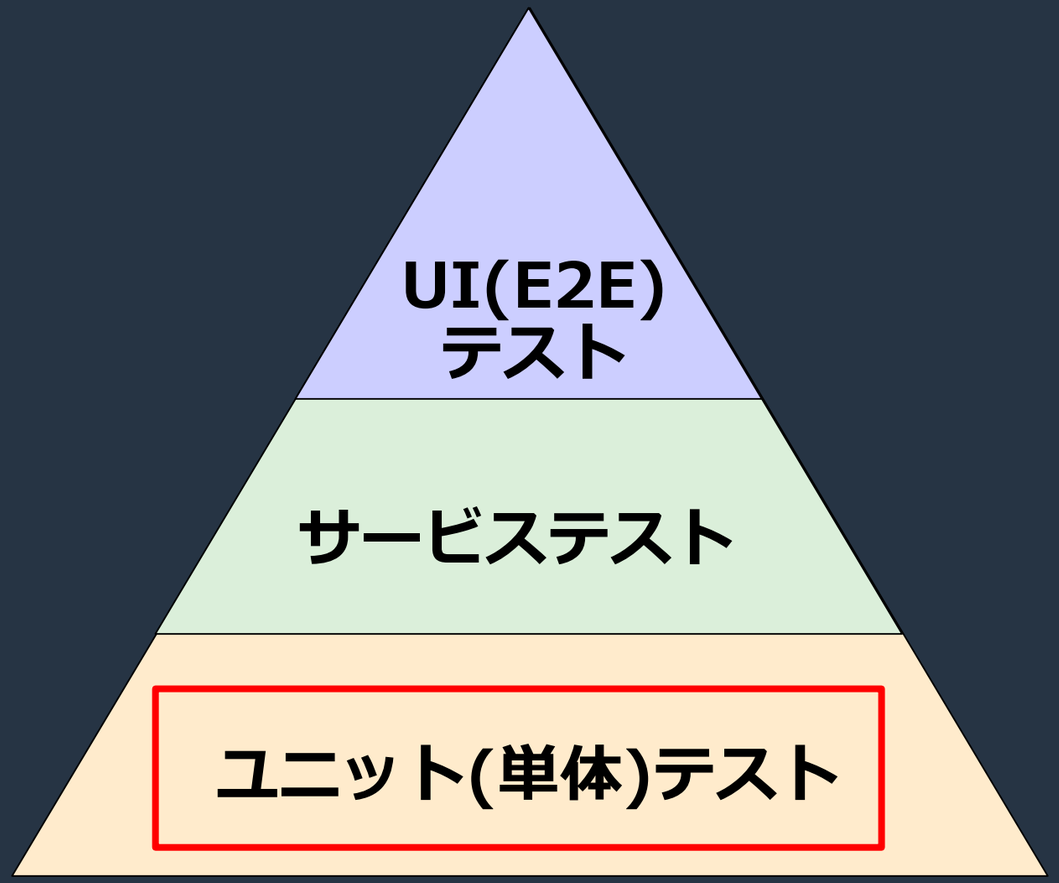
➡ 多くのレガシーコードは Testable ではない

```
# add関数
def add(a, b):
    return a + b

def test_add():
    #unit test
    expected = 10  #期待する値
    assert expected == add(5, 5)
```

```
# add関数
def add_use_xx(a, b):
   nazono_dependencies()
  {…}
   xx(a + b) // 結果は謎のxxへ

def test_add_use_xx():
    #I don’t know how to test…
```

テストとリファクタリング – 鶏と卵

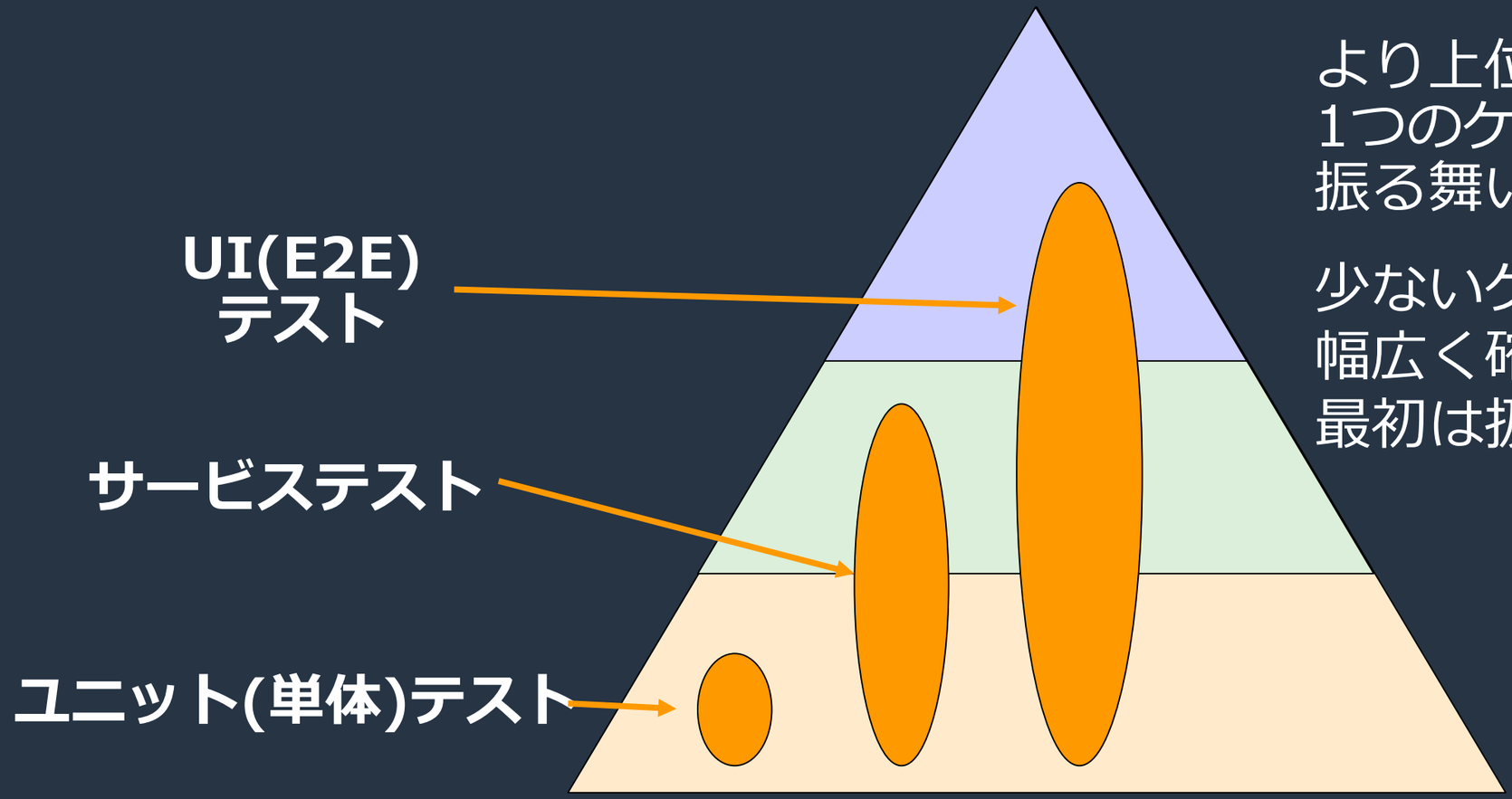
いきなり綺麗なテストピラミッドにできるのか?



レガシーコードはTestable ではないことが多い
→単体テストは特に書きづらい

単体テストを充実させなければいけない?
→すでに大量にあるクラス/関数全てにテストを実装しないといけない??

いきなり綺麗なテストピラミッドにできるのか?



より上位のテストほど
1つのケースで確認できる
振る舞いの範囲は広くなる

少ないケース数で、
幅広く確認できる方が
最初は扱いやすい

思い出そう -> テストはガードレール

<u>カバレッジ恐怖症</u>

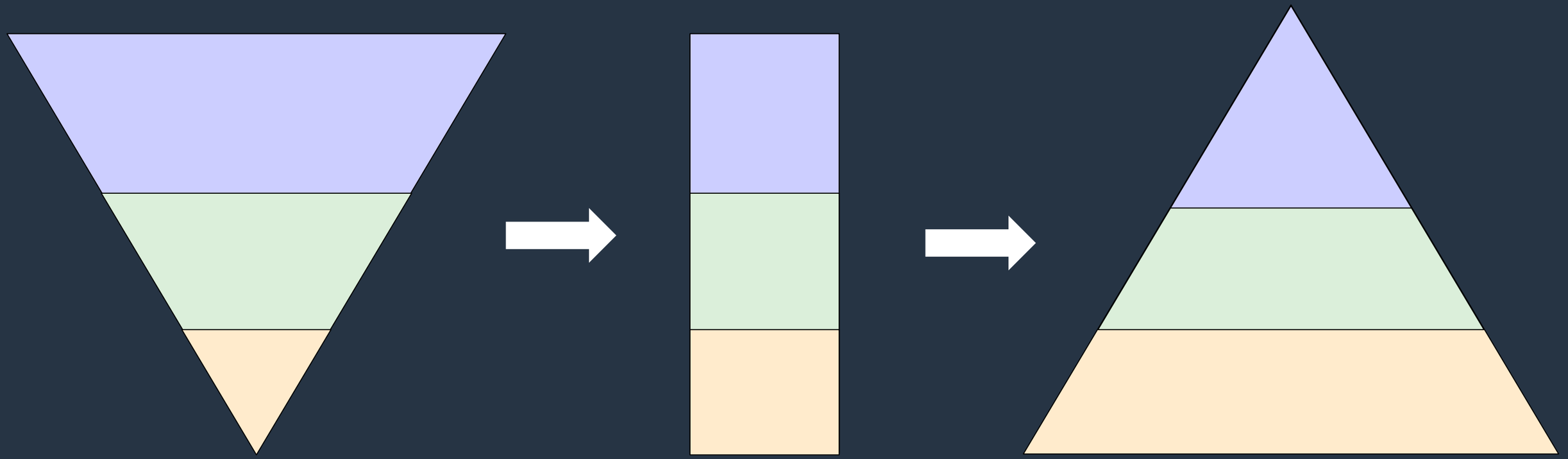
- (この時点では)自動テストは品質を担保するテストでなくて良い
- カバレッジ100%がゴールではない

<u>いきなり健康にはなれない</u>

- 運動不足な人が急に激しい運動はできない(むしろ危ない)
- 少しずつ改善していくしかない

まずは変更する箇所の振る舞いを確認するテストを書く

段々と理想形を目指す



逆ピラミッド

単体テストを
増やしていく

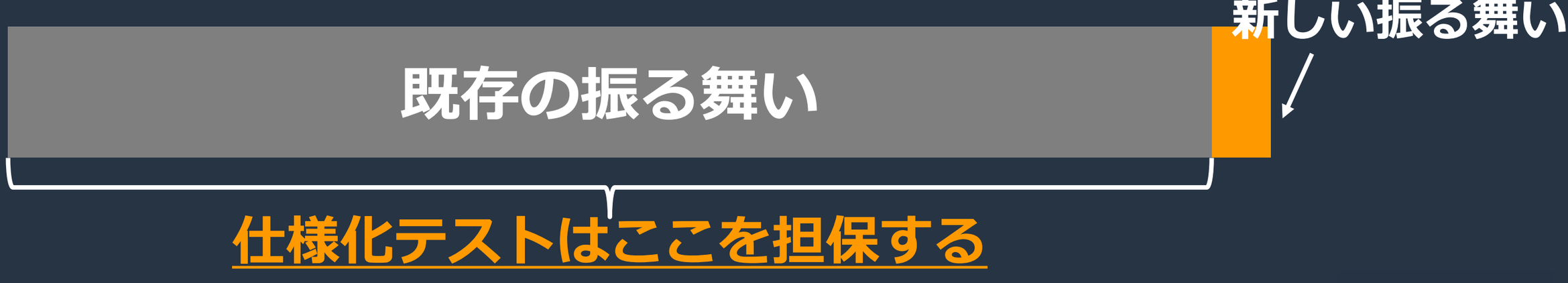
ピラミッド

仕様化テスト

- コードの実際の振る舞いを明らかにするテスト
- レガシーコードにおいては「あるべき論」より「現在の振る舞い」が重要(doc より動いているコード)
- 現在のバグを見つけるのではなく、将来の変更が現在の振る舞いを意図せず変えるという、将来のバグを見つける
- マイケル・フェザーズ(レガシーコード改善ガイドの著者)が考案

重要なのは振る舞いである

- 変更とはソフトウェアの振る舞いを変えること
- リファクタリングは「振る舞いを変えずに設計を改善する」こと
- 新しい変更は対象以外の既存の振る舞いを変更してはいけない、そして、多くの場合保たれる部分の方が多い



仕様化テストを書く手順

1. テスト対象のコードを呼び出す
2. 失敗するとわかっているアサーション(表明)を書く
3. 失敗した結果から現在の振る舞いを確認する
4. テストを変更して、現在の振る舞いで成功するようにする
5. 以上の手順を繰り返す

参考：マイケル・C・フェザーズ著
『レガシーコード改善ガイド』

1. テスト対象のコードを呼び出す

テスト対象のレガシーコード

```
class legacyTarget {
  public String complexMethod() {

  {...} //複雑な処理

  return result
  }
}
```

対象のコードを呼び出すテスト

```
void testComplexMethod() {
  LegacyTarget target = LegacyTarget();
  assertEqual(
   "hoge",
   target.complexMethod());
}
```

2.失敗するとアサーション(表明)を書く

テスト対象のレガシーコード

```
class legacyTarget {
  public String complexMethod() {

  {...} //複雑な処理

  return result
  }
}
```

対象のコードを呼び出すテスト

```
void testComplexMethod() {
  LegacyTarget target = LegacyTarget();
  assertEqual(
    "hoge",
    target.complexMethod());
}
```

3.現在の振る舞いを確認する

テスト対象のレガシーコード

```
class legacyTarget {
  public String complexMethod() {

  {...} //複雑な処理

  return result
  }
}
```

対象のコードを呼び出すテスト

```
void testComplexMethod() {
  LegacyTarget target = LegacyTarget();
  assertEqual(
   "hoge",
   target.complexMethod());
}
```

⬇ テスト実行

テスト結果

```
ERROR: expected: “hoge” but was ”DevAx::connect”
```

4.現在の振る舞いで成功するようにする

テスト対象のレガシーコード

```
class legacyTarget {
  public String complexMethod() {

  {...} //複雑な処理

  return result
  }
}
```

対象のコードを呼び出すテスト

```
void testComplexMethod() {
  LegacyTarget target = LegacyTarget();
  assertEqual(
   "DevAx::connect",
    target.complexMethod());
}
```

実際の振る舞いが仕様化される

CI/CD にこのテストを組み込む

テスト対象のレガシーコード

```
class legacyTarget {
  public String complexMethod() {

  {...} //複雑な処理

  return result
  }
}
```

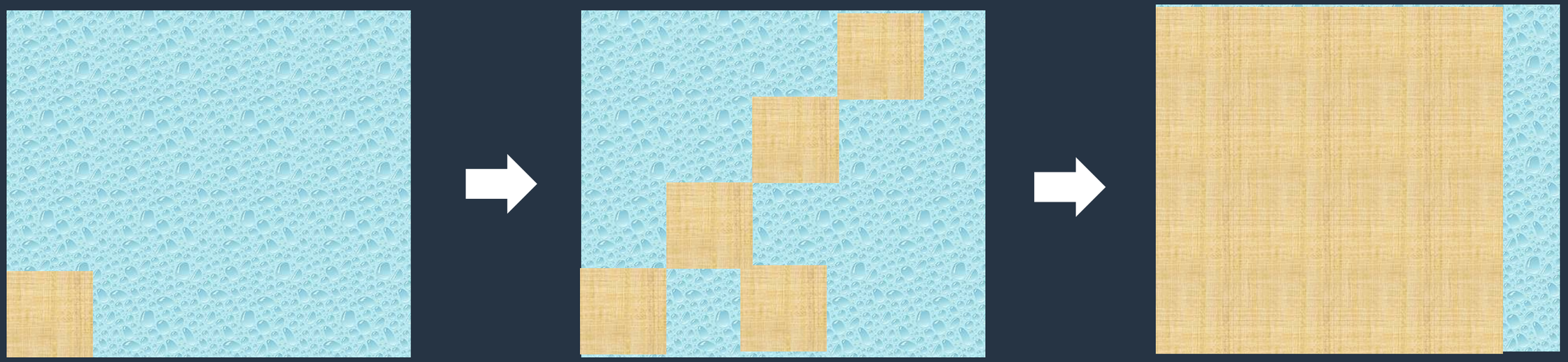
対象のコードを呼び出すテスト

```
void testComplexMethod() {
  LegacyTarget target = LegacyTarget();
  assertEqual(
   "DevAx::connect",
   target.complexMethod());
}
```

ローマは一日にして成らず

テストが足場になって改善していくことができる

- 仕様化テストの数を増やしていく
- このテストを元にリファクタリングを進める
- 段々とテストが書けるコードに



なぜテストを書かないのか

前半の質問やアンケートから

- 誰がテストを書くべきなのかわからない
- いつテストを書くべきなのかわからない
- **どこまでテストを書くべきなのかわからない** → **変更するところから書いていく**
- テストにかかる工数をどうやって説明すれば良いかわからない

その他、よく聞く理由

- 急いでいる案件なので”今回は”書きません
- このコードは変更しない予定(使い捨て）です

なぜテストを書かないのか

前半の質問やアンケートから

- 誰がテストを書くべきなのかわからない
- いつテストを書くべきなのかわからない
- どこまでテストを書くべきなのかわからない
- **テストにかかる工数をどうやって説明すれば良いかわからない**

その他、よく聞く理由

- 急いでいる案件なので”今回は”書きません
- このコードは変更しない予定(使い捨て）です

ところで仕様化テストはどこで作る?

- 振る舞いを呼び出せる縫い目(シーム)を見つける
- UI/E2E テストで作る？
  - 簡単にできるなら良い
  - ただし、変更が多かったり壊れやすいことがある
- 結合テストくらいがちょうど良さそう
- 分散システム間の振る舞いの整合性も確認できると便利そう

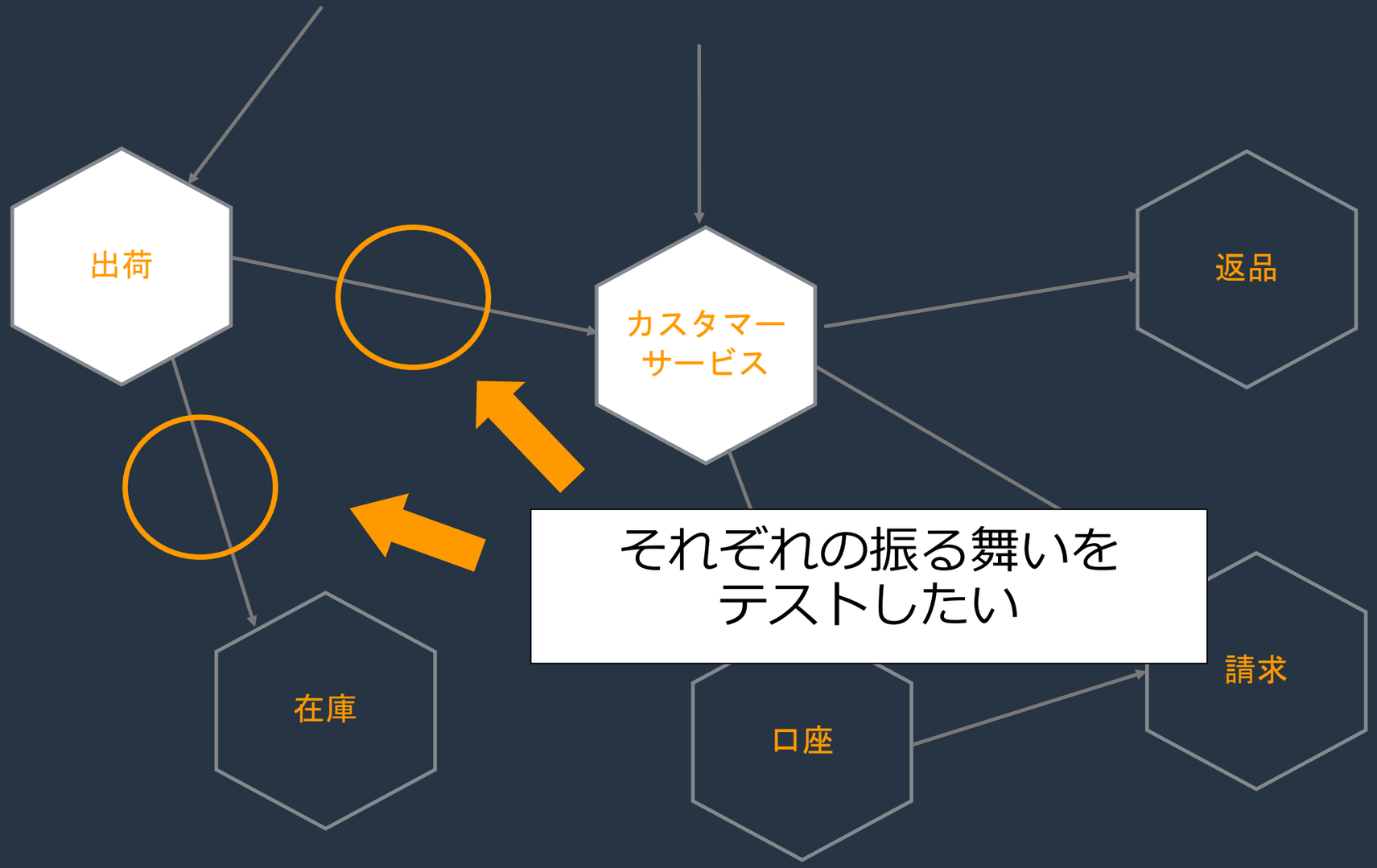
今日のアジェンダ

1. なぜCI/CD に自動テストが必要なのか
2. 自動テストの作り方 - 理想編 –
3. 仕様化テスト – 現実編 -
4. **コンシューマ駆動契約**
5. まとめ

コンシューマ駆動契約

aws DevAx connect

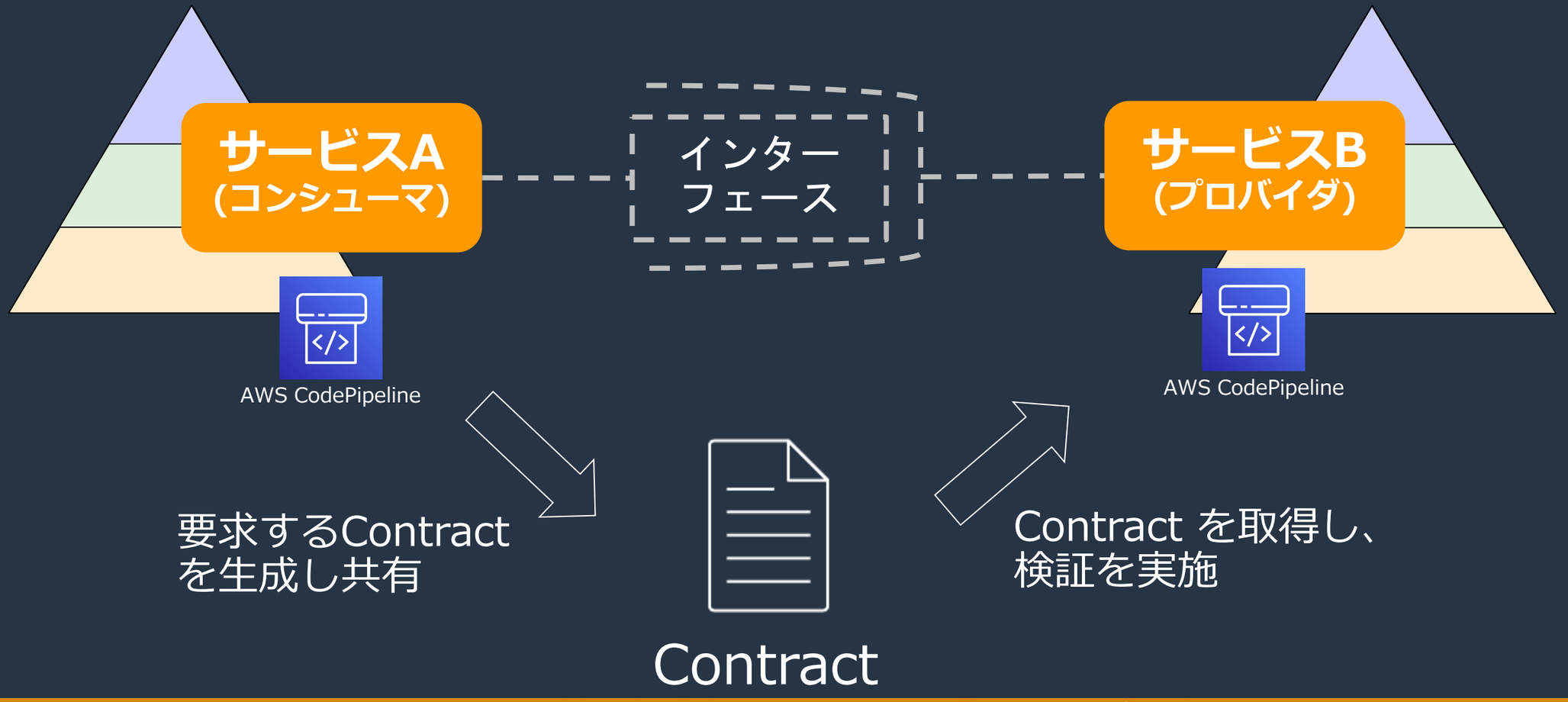
分散システムの整合性をテストできるか



サービス間の連携について

- サービス間のインターフェースは「契約」
- サービス間のテストは「結合テスト」以降になる
- それぞれのサービスが独自に変更、デプロイしていきたい
- が、契約に影響を与える/受ける変更は通知したい/されたい

コンシューマー駆動契約
Consumer-Driven Contracts (CDC)

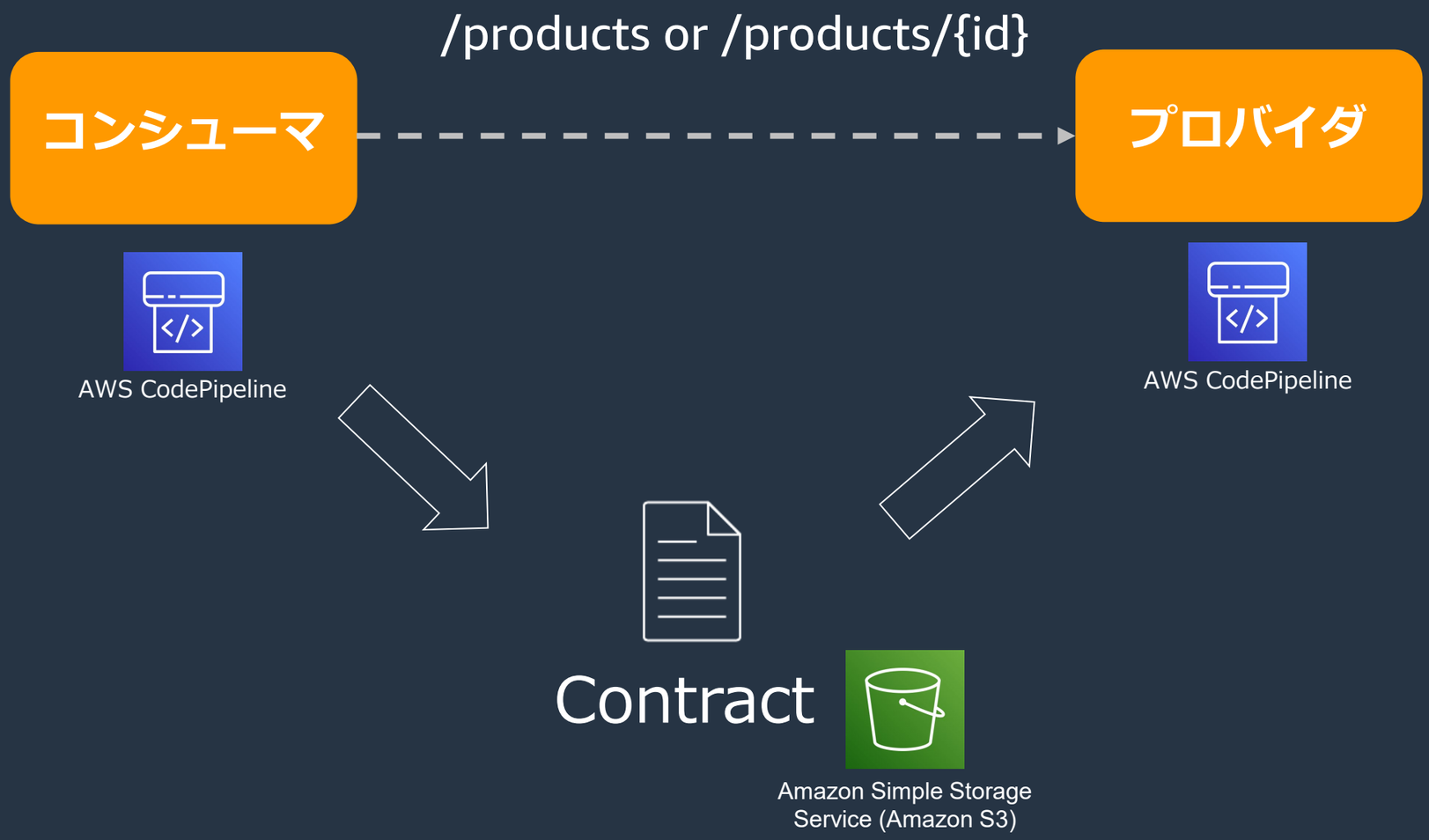


インターフェースの整合性をそれぞれで自動テストできる

PACT

- コンシューマ駆動契約テストのためのツール
- PACT の基本的な動作
  1. コンシューマのテスト(CI)の中で”pact file” (JSON) を生成
  2. プロバイダは”pact file” を利用して、契約を壊していないかテストする
  3. それぞれのテストでMock の機能も提供
- ”pact file”を中継するブローカーも提供
- 対応言語:Ruby, Javascript,JVM,.NET(C#),Go,Python,PHP など主要な言語に対応

Demo



今日のアジェンダ

1. なぜCI/CD に自動テストが必要なのか
2. 自動テストの作り方 - 理想編 –
3. 仕様化テスト – 現実編 -
4. コンシューマ駆動契約
5. **まとめ**

まとめ

aws DevAx connect

まとめ

- 自動テストは開発者のためのガードレール
- テストがないコードは「レガシーコード」
- まずは仕様化テストから書き始めて、増やしていく

明日から変更するときはテストから書きましょう!!

Thank you!

Your name

Your title

Other title or location

Your email, linked in, etc contact details

aws DevAx connect