DevAx
connect

トランクベース開発で信頼できる CI を実践する

DevAx::connect シーズン3 rethink CI/CD

Masatoshi Hayashi
Specialist Solutions Architect, Containers

@literalice

自己紹介

林 政利, @literalice
Specialist Solutions Architect, Containers / AWS Japan

好きな AWS のサービス
Amazon Elastic Kubernetes Service (Amazon EKS)
AWS Certificate Manager

SIer

フリーランス

Java/Ruby 開発者
Kubernetes インフラ設計
(Web 企業)

Containers SA, Support Engineer
(クラウド製品ベンダー)

AWS Japan
Containers SA

セッションの対象者

- CI/CD を導入しているが、品質やアジリティが上がらない
- トランクベース開発は知っているが、どう始めたらいいかわからない
- ブランチ戦略に悩んでいる

セッションのゴール

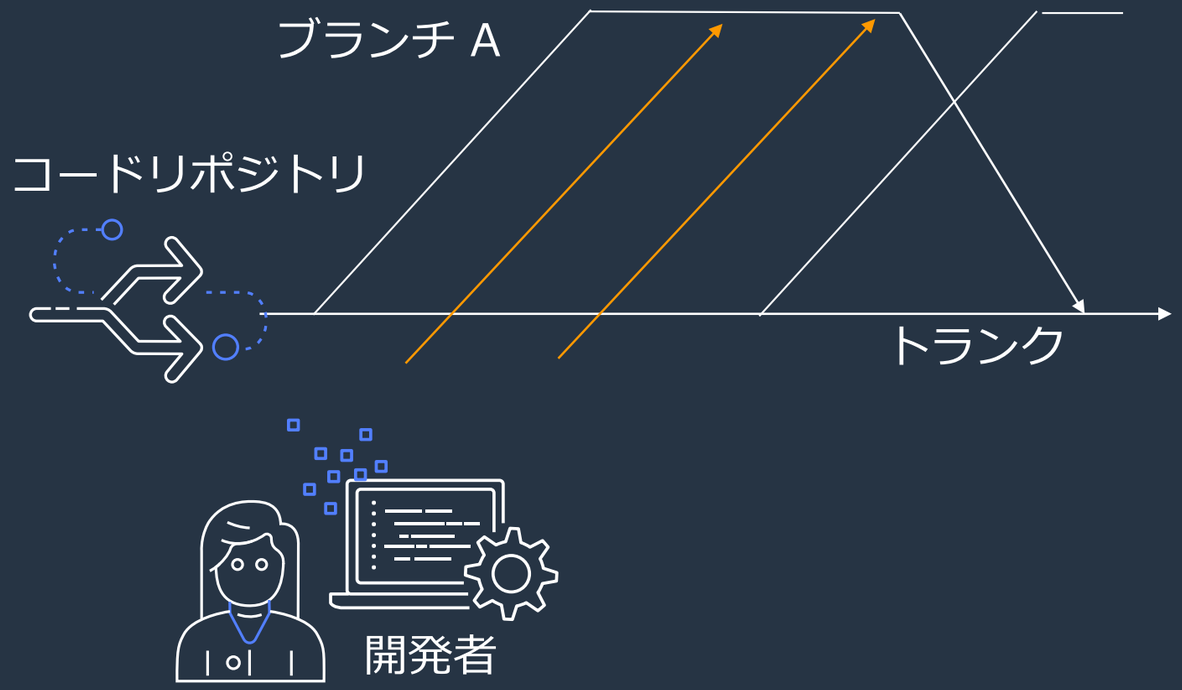
- 適切な CI/CD の導入を目指す
- トランクベース開発を導入するためのアクションを検討する
- ブランチ戦略を決定する

アジェンダ

- トランクベース開発とは
- なぜ、トランクベース開発が必要なのか
- トランクベース開発のために必要なプラクティス
  - 自動テスト、INVEST なバックログ
- トランクベース開発のブロッカーとその解消
  - フィーチャーフラグ、抽象ブランチ、リリースブランチ
- Getting Started

トランクベース開発とは

ブランチ戦略

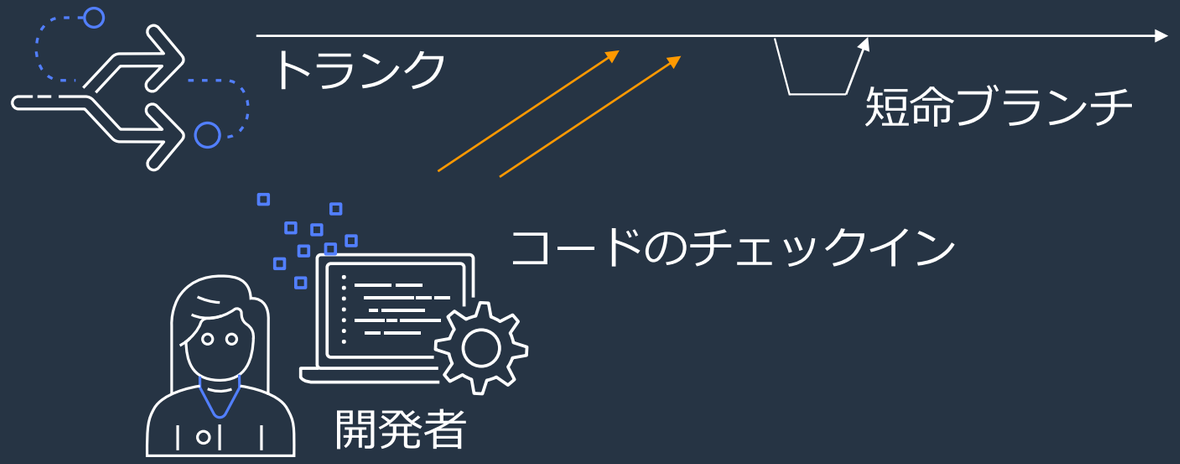


ブランチを何のために、どういうタイミングで作成して、いつ何のためにマージするのか？

- Gitflow
- GitHub flow
- Gltlab flow
- etc..

ブランチ戦略 - トランクベース開発

開発者が main に頻繁に変更をマージする戦略



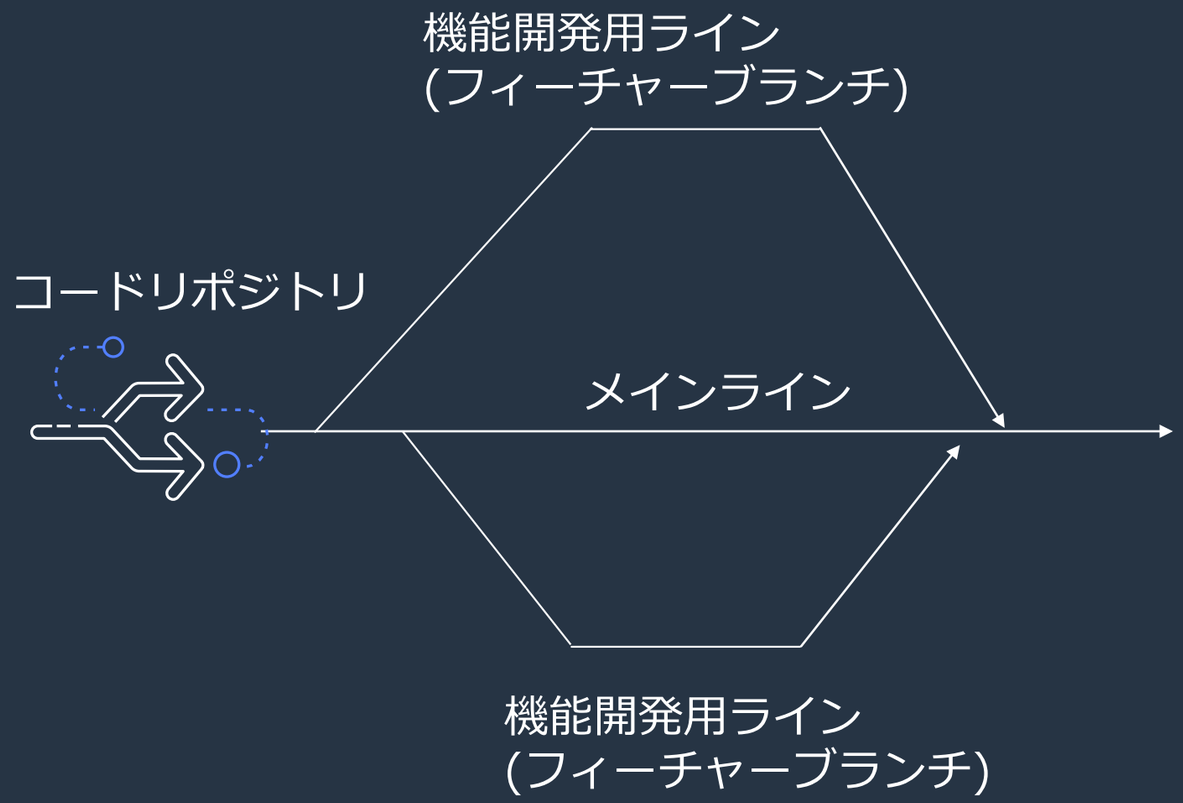
コードをメインのブランチ(トランク)にマージし続けるプラクティス

一日単位の短命なブランチを活用することもある

https://trunkbaseddevelopment.com/

非トランクベースのブランチ戦略

長命なフィーチャーブランチ、環境毎のブランチ、 Gitflowなど



フィーチャーブランチをリリース
まで育てる

リリースが決まったら組み込む
フィーチャーブランチを選択

マージ、リリース

Gitflow

2010年に提唱された著名なブランチ戦略

メイン

* develop ブランチ (ナイトリー)

* main ブランチ (最新の本番)

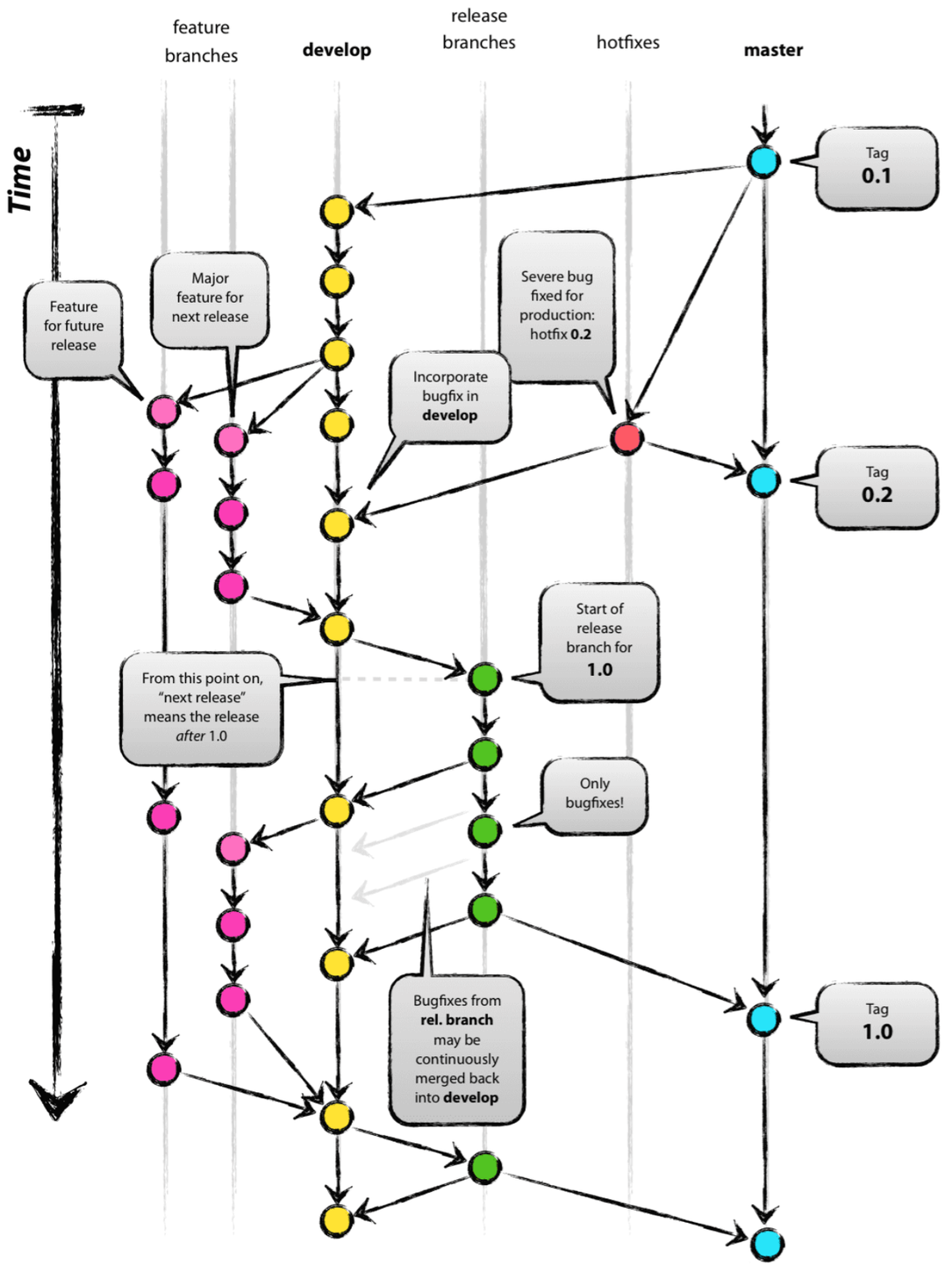
リリースブランチ

ホットフィックスブランチ

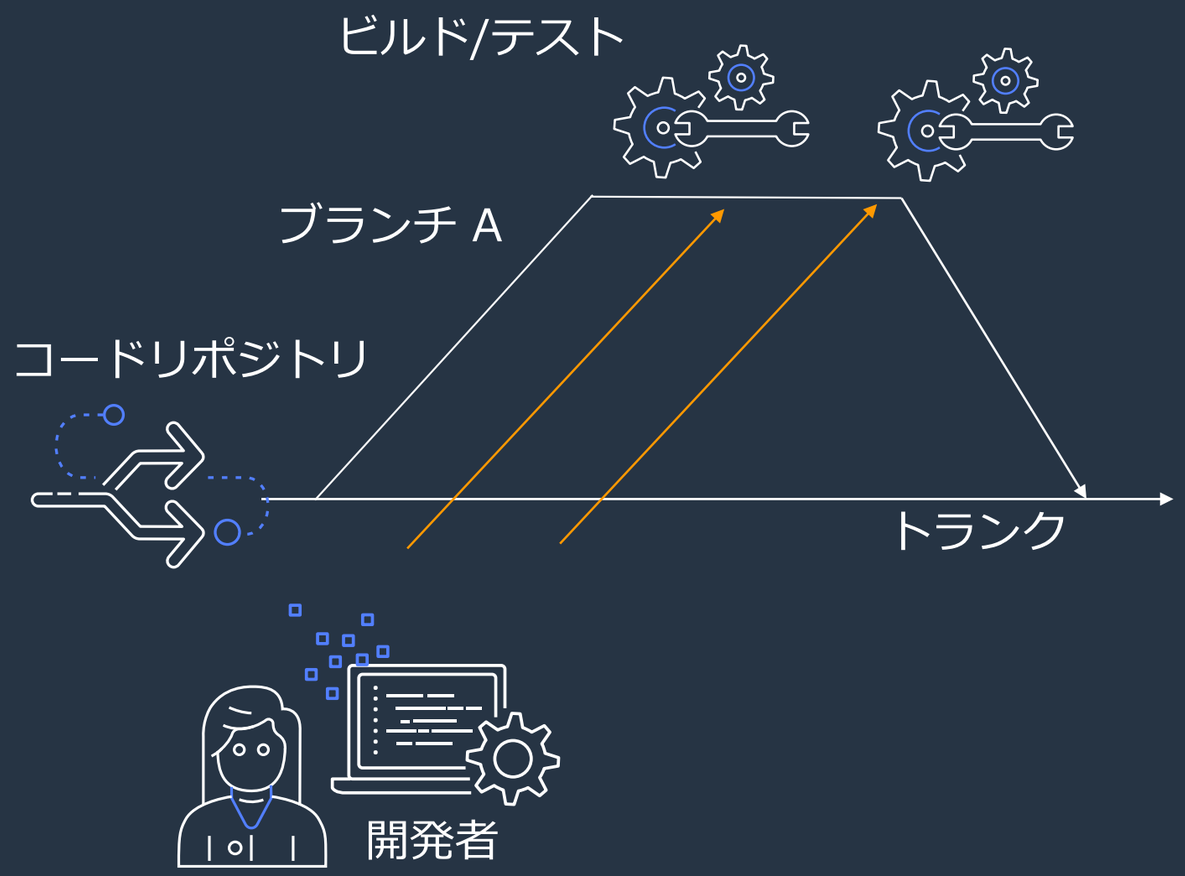
フィーチャーブランチ

オンラインサービスのように継続的にデリバリーするコードを想定していない

https://nvie.com/posts/a-successful-git-branching-model/

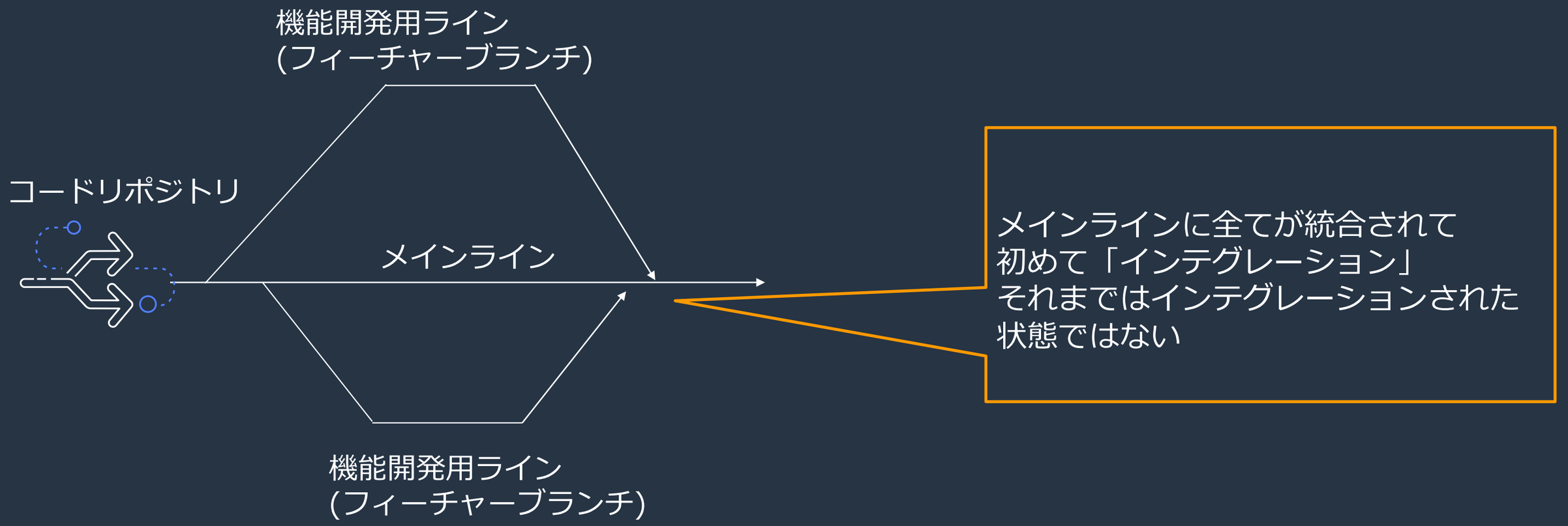


CI/CDを再考する ～ インテグレーションって何？



- コードのチェックインを検知、検証し続けるサーバーがある
  - ソースを統合してコンパイルしているので、インテグレーションしている
  - コンパイルが通るだけで十分か？
  - 実行時のエラーや静的解析も実施しないとインテグレーションではない？
- もっと広い意味でのインテグレーションもある

ブランチのインテグレーション



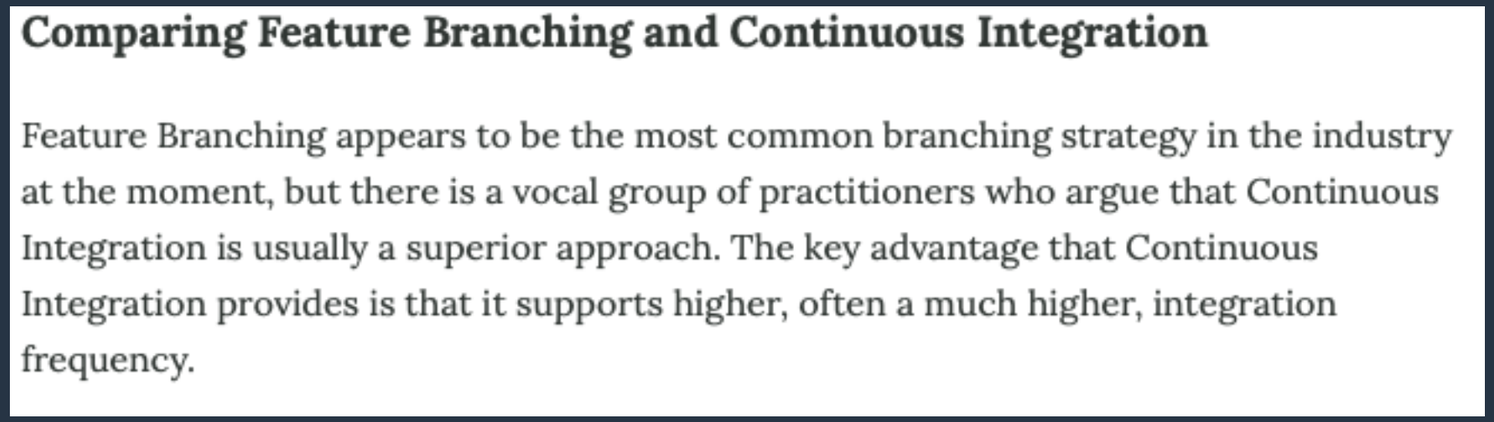
継続的インテグレーションとは?

AWS CodePipeline を試用する

継続的インテグレーションについて

継続的インテグレーションは、開発者が自分のコード変更を定期的にセントラルリポジトリにマージし、その後に自動化されたビルドとテストを実行する DevOps ソフトウェア開発の手法です。継続的インテグレーションという用語が最もよく使われるのは、ソフトウェアのリリースプロセスのビルド段階または統合段階を指す場合で、オートメーションの要素 (CI やビルドサービスなど) と啓発の要素 (頻繁に統合する必要性を学習することなど) の両方が含まれます。継続的インテグレーションの主な目的は、バグを早期に発見して対処すること、ソフトウェアの品質を高めること、そしてソフトウェアの更新を検証してリリースするためにかかる時間を短縮することです。

https://aws.amazon.com/jp/devops/continuous-integration/



"フィーチャーブランチと継続的インテグレーション - フィーチャーブランチが最も一般的かもしれないが、継続的インテグレーションの方がより優れたアプローチだと主張している人もいる"

https://martinfowler.com/articles/branching-patterns.html#ComparingFeatureBranchingAndContinuousIntegration

トランク ベース開発とは何か

トランク ベースの開発とは、開発者が細かく頻繁なアップデートをコア「トランク」または main ブランチにマージするバージョン管理手法です。これはマージ フェーズと統合フェーズの流れを合理化するため、DevOps チームの間では一般的なプラクティスであり、DevOps ライフサイクルの一部となっています。事実、トランク ベースの開発は CI/CD で必須のプラクティスです。他の長く使用されている機能のブランチング戦略と比べて、開発者はわずかなコミットで短期間のブランチを作成できます。コード ベースの複雑さとチーム規模が拡大するにつれて、トランク ベースの開発は本番リリースのフローを維持するのに役立ちます。

"トランクベースの開発は CI/CD で必須のプラクティスです"

https://www.atlassian.com/ja/continuous-delivery/continuous-integration/trunk-based-development

Gitflow ワークフロー

Gitflow とは、元来は Git ブランチを管理するための破壊的で斬新な戦略のレガシー Git ワークフローです。Gitflow の需要は落ち込み、トランク ベースのワークフローが利用されるようになっています。現在ではこれが最新の継続的なソフトウェア開発のベスト プラクティスおよび DevOps プラクティスと見なされています。Gitflow はまた、CI/CD と使用することも困難な場合があります。この投稿では、歴史的な目的で Gitflow の詳細をご説明します。

"Gitflow はまた、CI/CD と使用することも困難な場合があります。
この投稿では、歴史的な目的で Gitflow の詳細をご説明します"

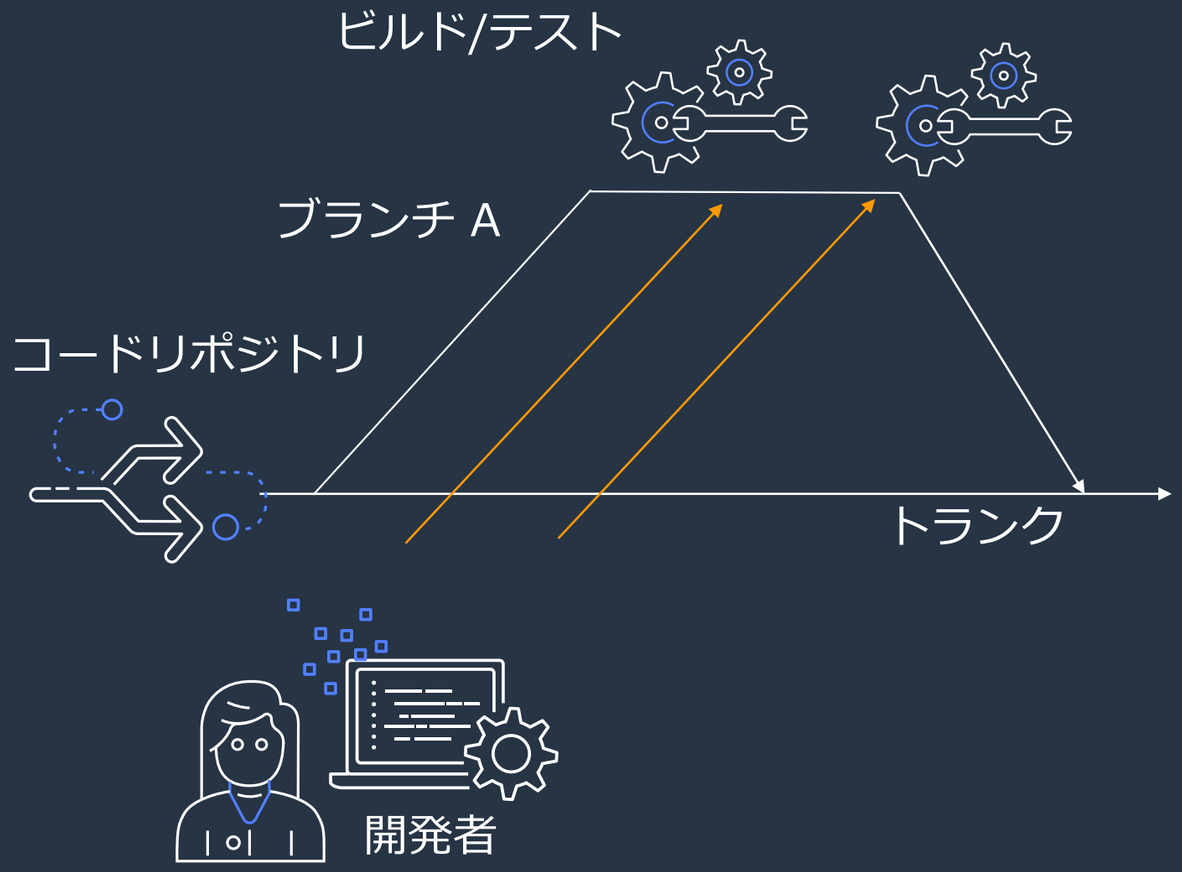
https://www.atlassian.com/ja/git/tutorials/comparing-workflows/gitflow-workflow

トランクベース開発は CI のキモ
むしろトランクベース開発は CI の前提

このセッションでの CI/CD の位置づけ

- CB 継続的ビルド (Continuous Build)
- CI 継続的インテグレーション (Continuous Integration)
- CD 継続的デリバリー/デプロイ (Continuous Delivery)

継続的ビルドのみの開発プロセス

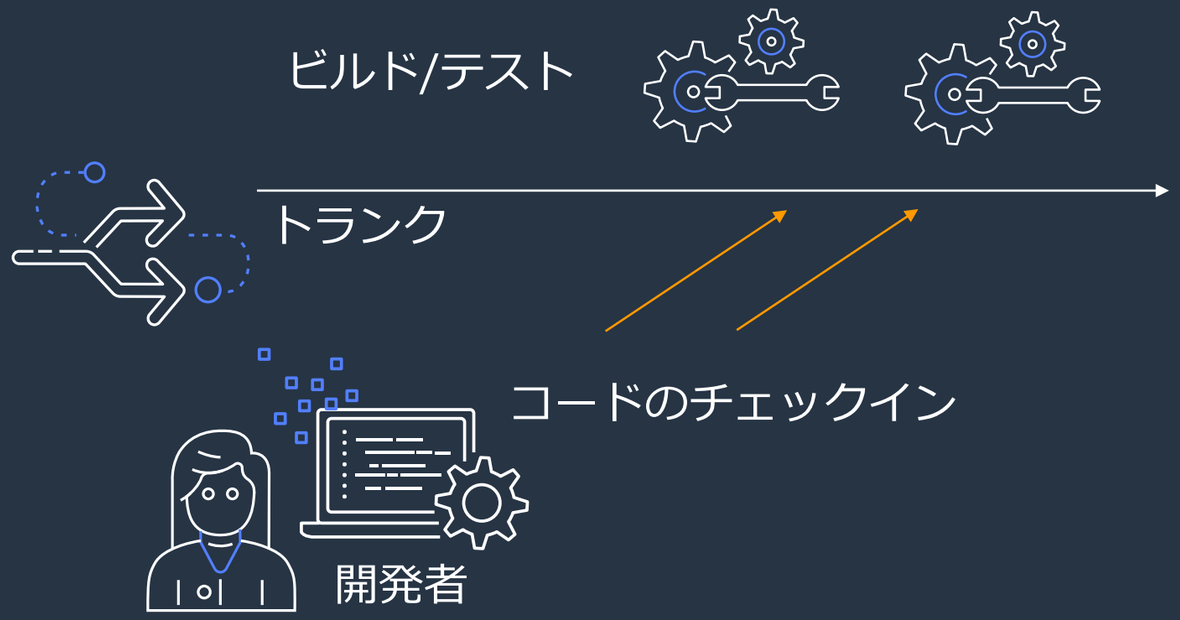


継続的ビルド

どこかのブランチにチェックインしたら自動的にビルドやテストが走る

どこかでマージしてデリバリー

CI 継続的インテグレーション



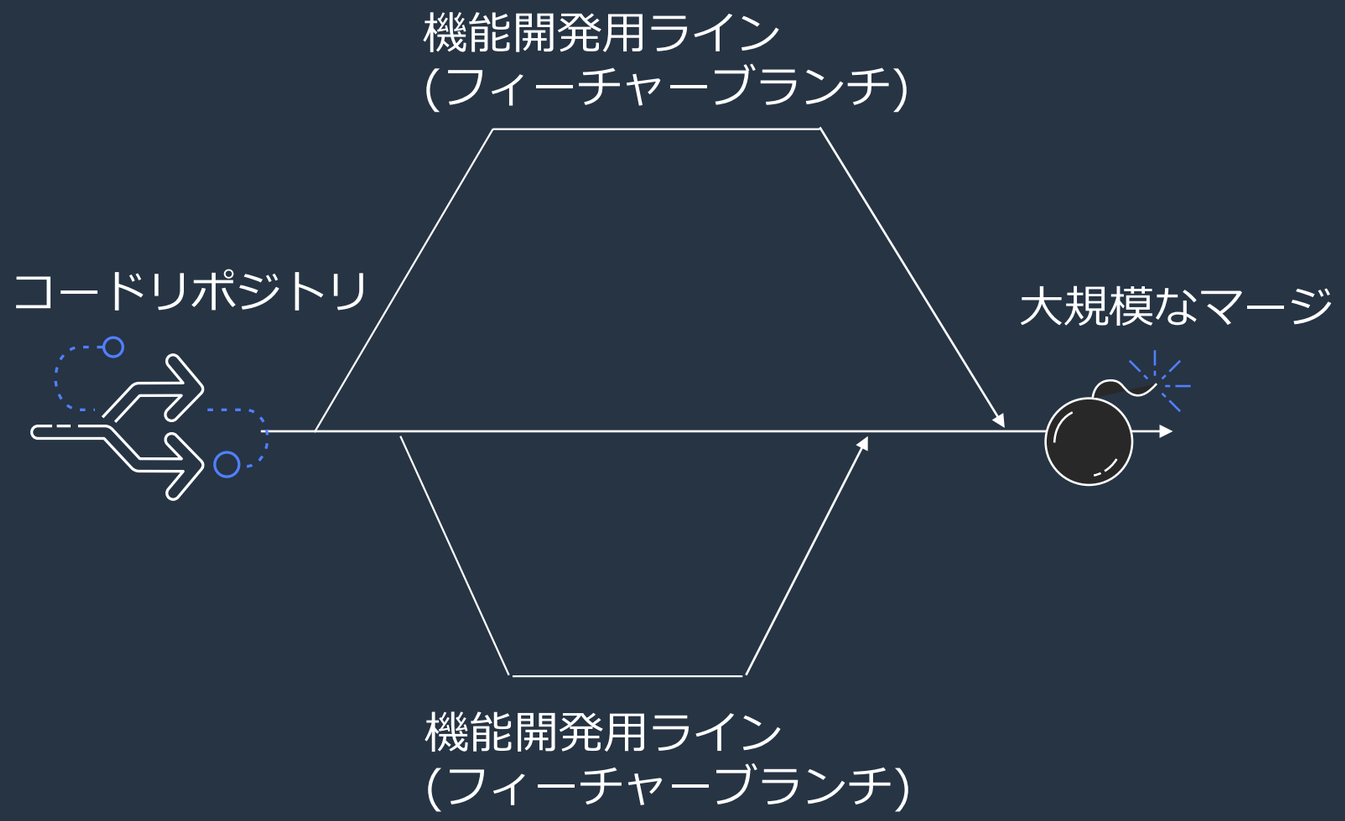
コードをメインのブランチ(トランク)にマージし続けるというプラクティス

トランクを「正しく動く」状態に保ち続ける

なぜ、トランクベース開発が必要なのか

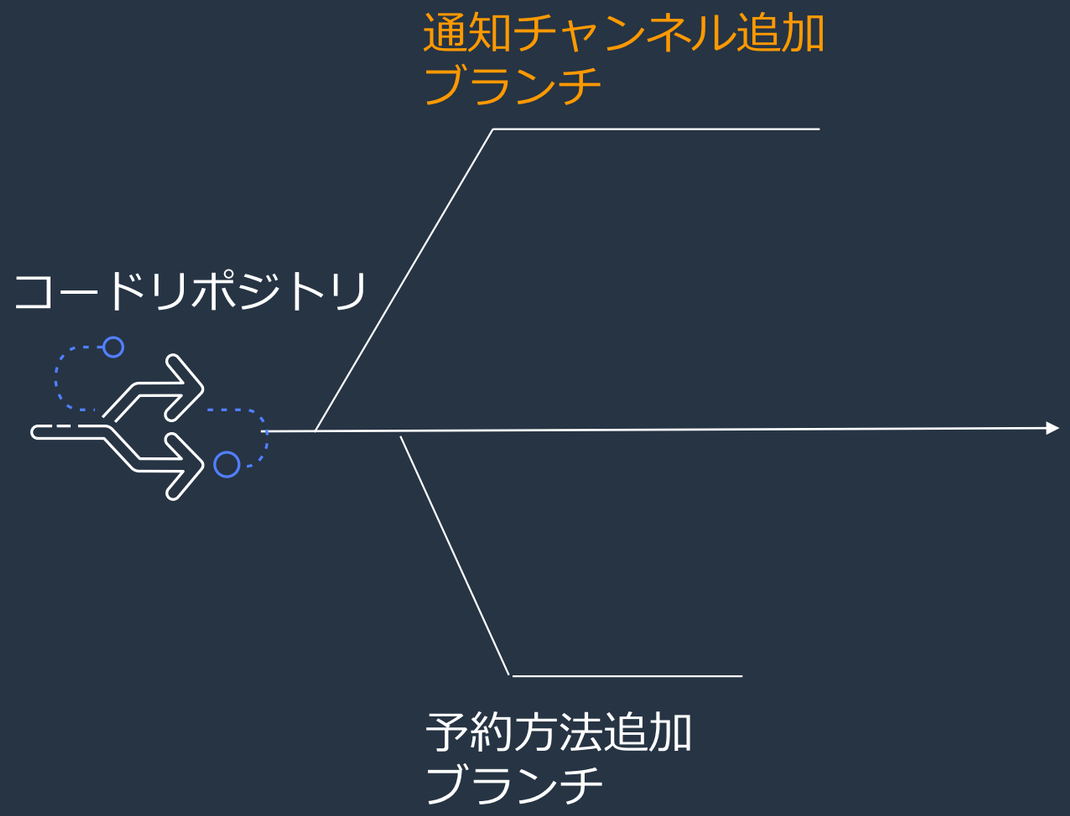
なぜ、トランクベース開発が必要なのか

ソースコードの大規模なマージを避け、小さなマージに分割する



- リリースの安定性
- 継続的な改善
(リファクタリング)

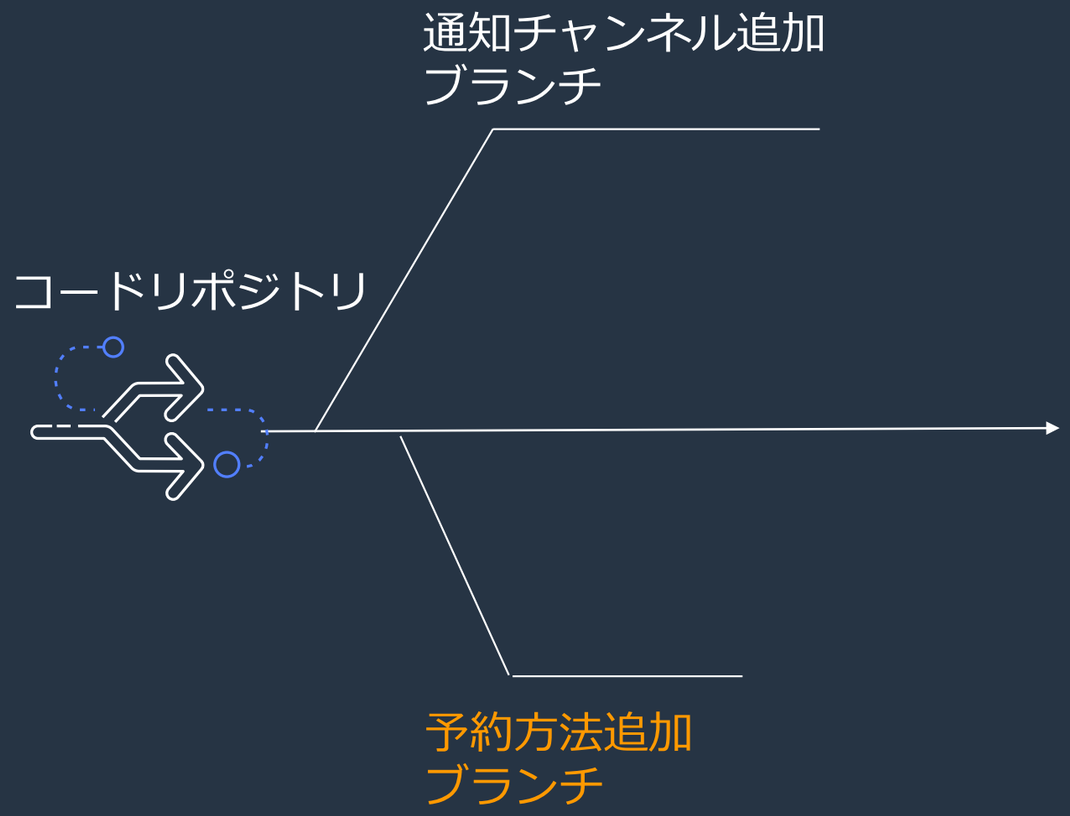
リリースの安定性が損なわれる例




```
reserveSeet(User user) {
  r = new Reservation(user, new Date());
  notify(r);
  return r;
}
```

```
reserveSeet(User user) {
  r = new Reservation(user);
  return r;
}
// ...
r = reserveSeet(user);
notify(r);
```

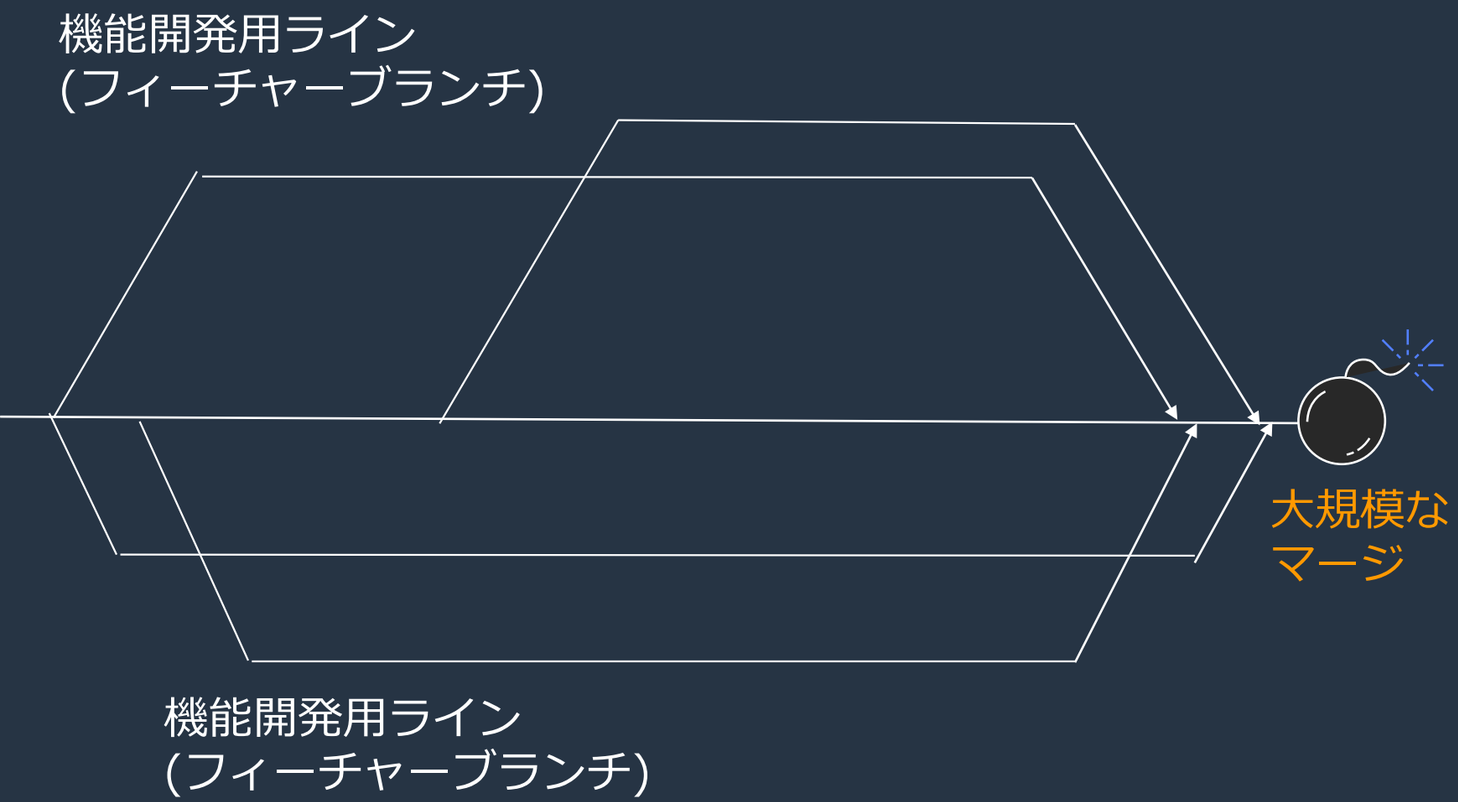
リリースの安定性が損なわれる例




```
reserveSeet(User user) {
  r = new Reservation(user, new Date());
  notify(r);
  return r;
}
```

```
reserveFromSNS() {
  r = reserveSeet(user);
}
```

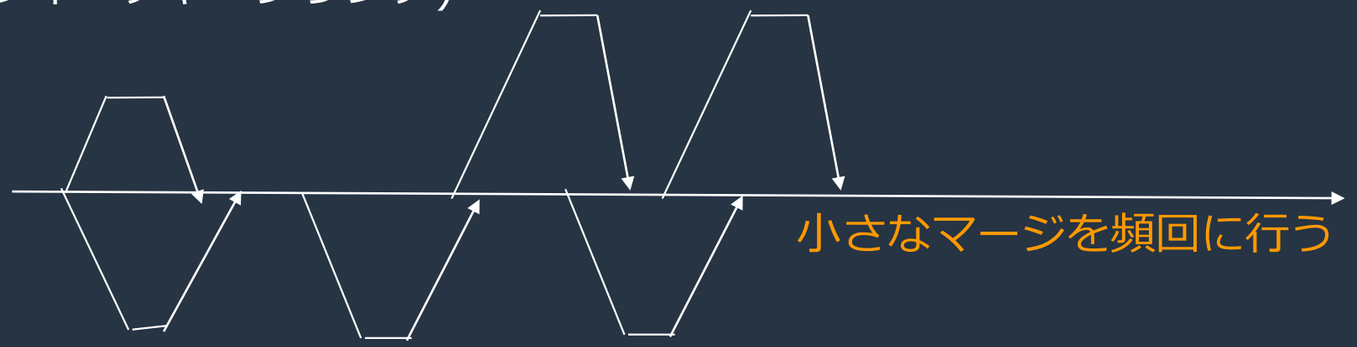
リリースの安定性が損なわれる例



- リリースまで挙動が不明
- コンフリクトが起きなければいいのか？
- 手作業の修正
- 解消の規模、工数の予測ができない
- リリースの不確実性が高い

トランクベース開発でリリースの安定性を向上

機能開発用ライン
(フィーチャーブランチ)

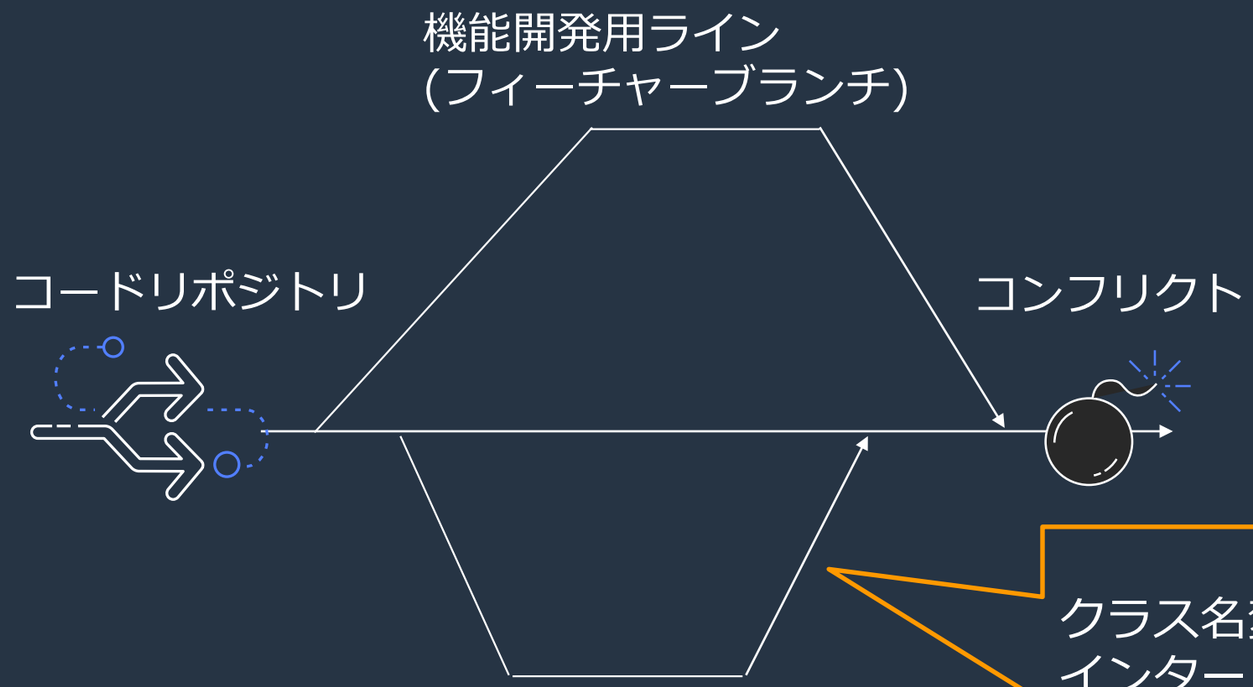


機能開発用ライン
(フィーチャーブランチ)

- マージを特別イベントにしない
  - 痛みを伴う作業こそ頻繁に
- フィードバックが早く返る
- リリースの不確実性が低い

なぜ、トランクベース開発が必要なのか

継続的な改善 (リファクタリング) が容易



- 長命なブランチの運用だと
  リファクタリングが難しい
- 技術的負債が積み上がる

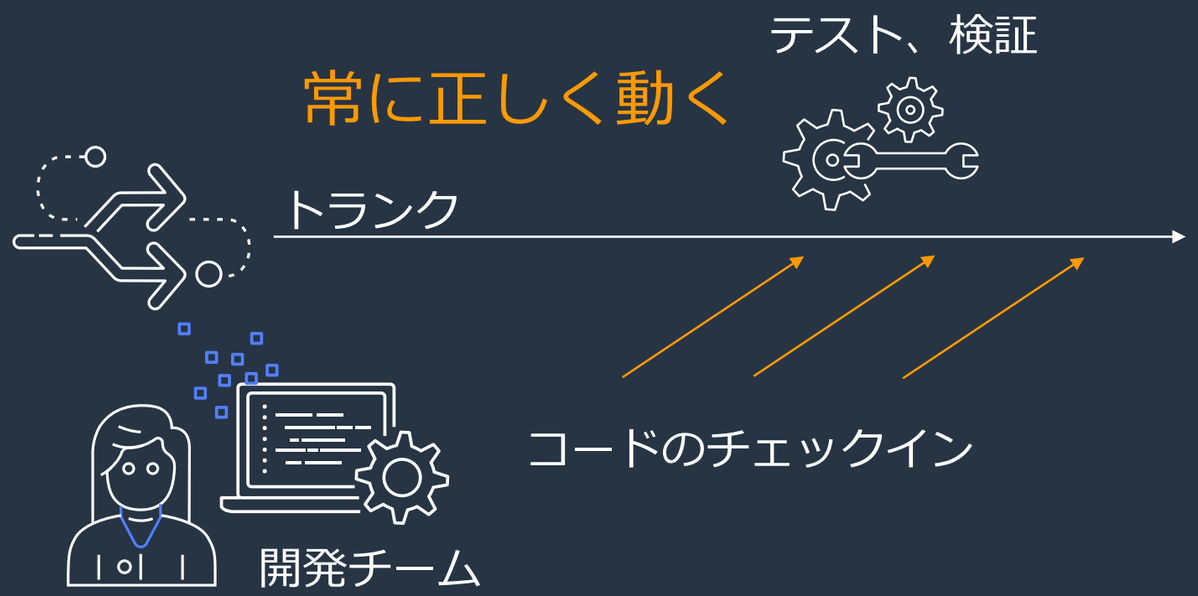
クラス名変更
インターフェース、抽象の抽出
メソッドの移動
変数名の変更
etc. etc.

トランクベース開発は 低頻度なリリース であっても有効

トランクベース開発に必要なプラクティス

aws DevAx connect

トランクベース開発に必要なプラクティス



自動テスト

正しく動くが分からないと頻繁に
トランクにチェックインし続けること
はできない

コードレビュー

重厚長大なマージプロセスでは頻繁に
トランクにチェックインし続けること
はできない

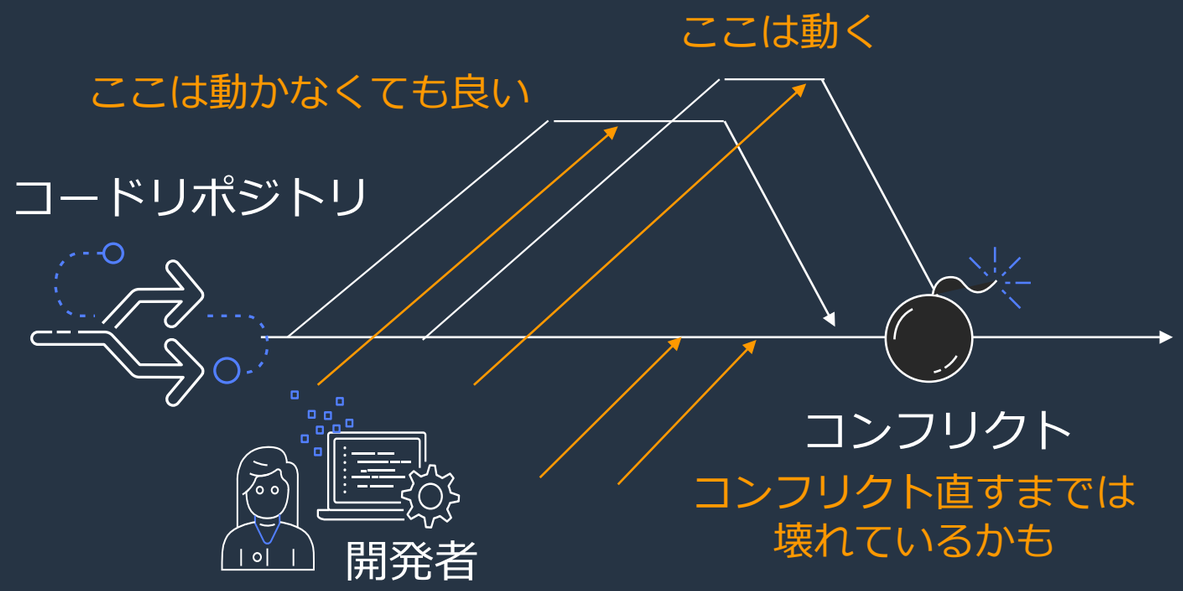
小さいチャンクでの機能開発

小さい変更でなければ、頻繁に
トランクにチェックインし続けること
はできない

aws DevAx connect

非トランクベース開発でのテスト環境

ビルド、テストが失敗したら即修正というプラクティスが適用されにくい

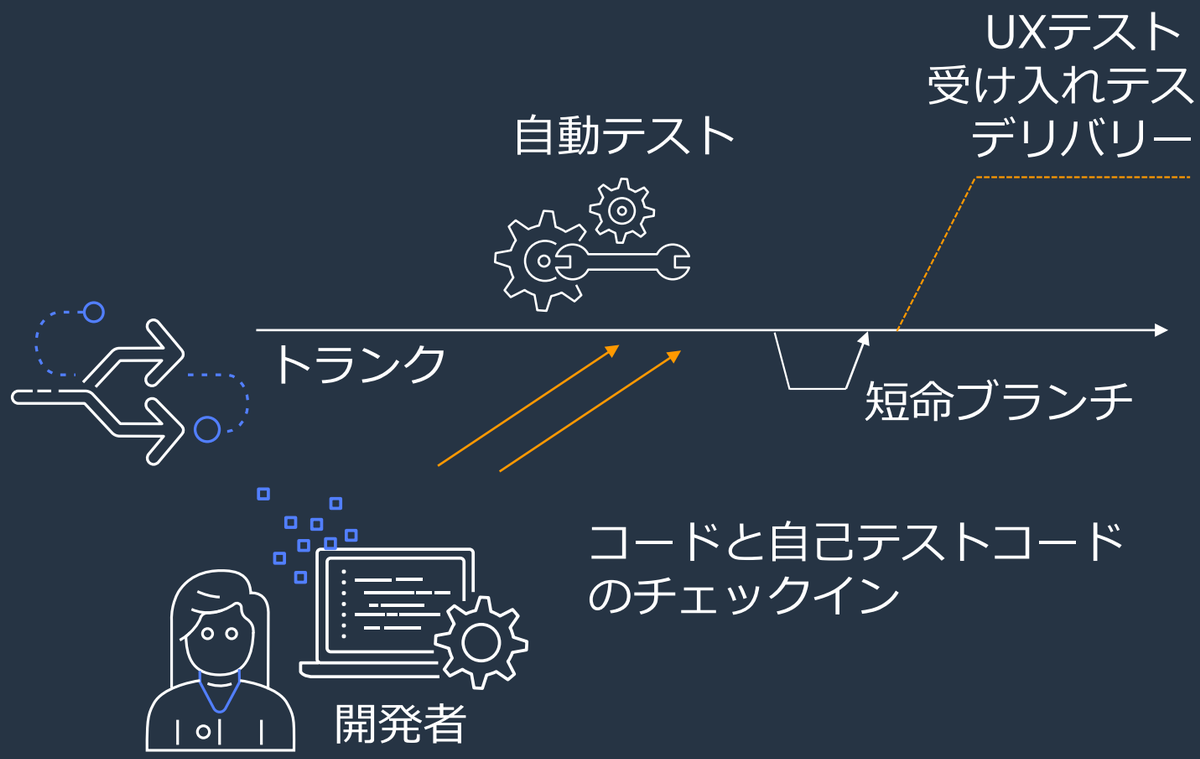


- 開発中のブランチだからとりあえず壊れてても良い
- ナイトリーだからとりあえず壊れてても良い
- マージしたときにどうせ壊れるから(ry
- 後の手動テストで確認する運用

- テストがあっても信頼しにくい

トランクベース開発で重要な自動テスト環境

自己テストコードで安心してトランクにチェックインできる環境を作る



- CI でエラーが発生したら開発を即止めて修正
- 変更したコードが「正しいか」開発チームがすぐにフィードバックを受ける
- リグレッションテストの蓄積
- 安心してコードをトランクにチェックインできる環境の構築

"自動テストがなければ、
継続的インテグレーションは、正しくコンパイル、
実行できないゴミの山を最も
早く築き上げる方法になってしまいます"

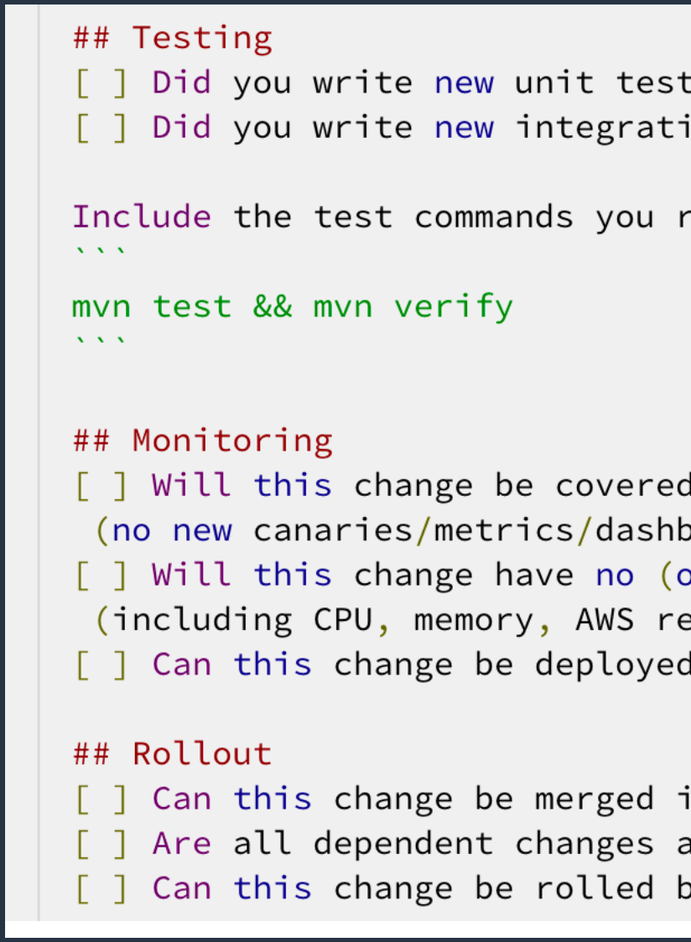
Gruver (Hewlett-Packard)
The DevOps ハンドブック

コードレビュー

- レビュープロセスを重くしない
  - XXX の承認とxxxの承認とxxxの承認が必要で、xxxで、etc..
  - 重いとなるべくコードをまとめようとしがち
  - 逆に問題点が見つけにくくなる

- マージ前にレビューを設ける場合は、依頼を受けたら即対応する
  - マージ後にレビュー、というケースもある
    - 自動テストが厚く、チームの性質によってはマージしたあとに気になる場合だけレビュー依頼を投げる
    - あとからリファクタリング

- ペアプロ



Amazon CodeGuru

- レビューを省力化
- 機械学習で自動コードレビュー
- 最終的には人の手で判断できる

```
59 +                return "SUCCESS";
60 +            } catch (final Exception ex) {
```

literalice 10 minutes ago

Recommendation generated by Amazon CodeGuru Reviewer. Leave feedback
replying to the comment or by reacting to the comment using emoji.

Problem: While wrapping the caught exception into a custom one, informatio
being lost, including information about the stack trace of the exception.

Fix: If the caught exception object does not contain sensitive information, co
"rootCause" or inner exception parameter to the constructor of the new exce
exception. (Note that not all exception constructors support inner exceptions
supports inner exceptions.)
Learn more

Reply...

Resolve conversation

```
61 +                logger.log(String.format("Failed to process shipment U
   ex.getMessage()));
62 +                throw new RuntimeException("Hiding the exception");
63 +            }
64 +        }
```

Amazon / AWS のコードレビュー例

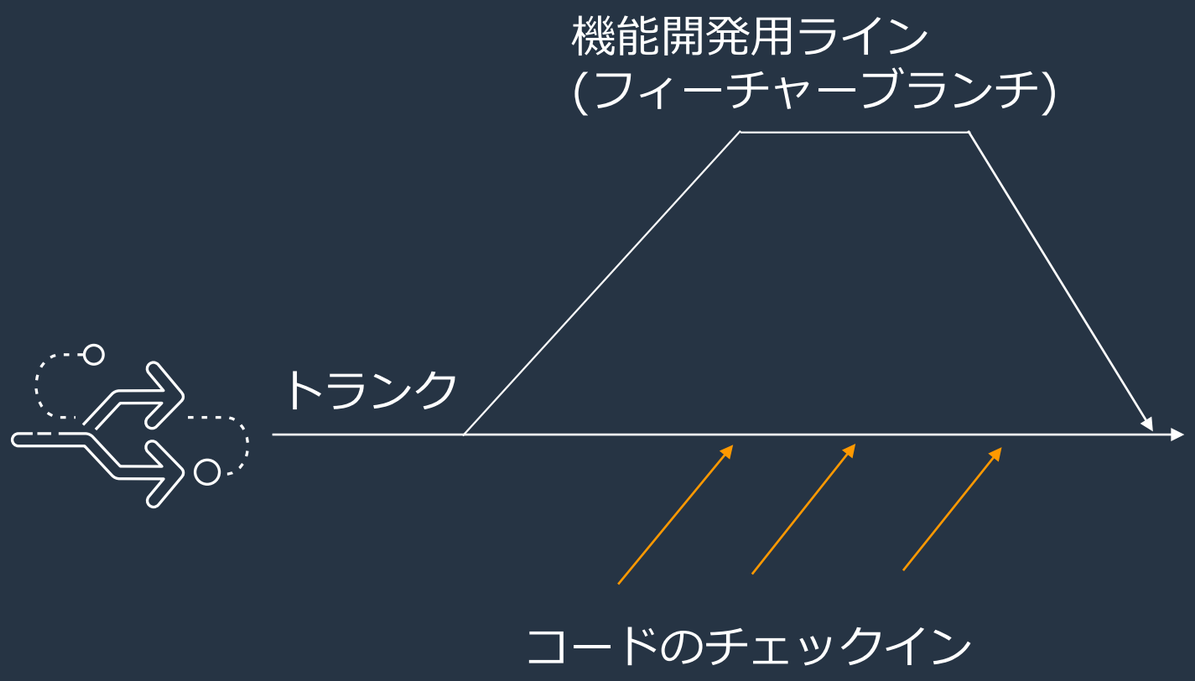
- ユニットテスト、結合テストが含まれているか

- 既存の監視システムで、この変更がカバーできるか
- CPUやメモリ、AWSのリソースの利用量に影響しないか
- アラームを発生させずに本番環境にデプロイできるか

- 依存する全ての変更は本番環境にデプロイ済みか
- 本番環境デプロイ後、ロールバックできるか

```
## Testing
[ ] Did you write new unit test
[ ] Did you write new integrati

Include the test commands you r
```
mvn test && mvn verify
```

## Monitoring
[ ] Will this change be covered
 (no new canaries/metrics/dashb
[ ] Will this change have no (o
 (including CPU, memory, AWS re
[ ] Can this change be deployed

## Rollout
[ ] Can this change be merged i
[ ] Are all dependent changes a
[ ] Can this change be rolled b
```

トランクベース開発のためのバックログ



× リリースできないコードを完成するまで
別ブランチで育ててリリース直前にマージ

○ 小さな単位でコード変更を行い、
リリースできる状態を保ちながら
マージし続ける

INVEST なバックログ

“I” ndependent (of all others)
独立している
“N” egotiable (not a specific contract for features)
交渉できる
“V” aluable (or vertical)
価値がある
“E” stimable (to a good approximation)
見積もることができる
“S” mall (so as to fit within an iteration)
小さい
“T” estable (in principle, even if there isn’t a test for it yet)
検証できる

https://www.agilealliance.org/glossary/invest/

トランクベース開発のブロッカーとその対策

aws DevAx connect

トランクにチェックインし続けることができない

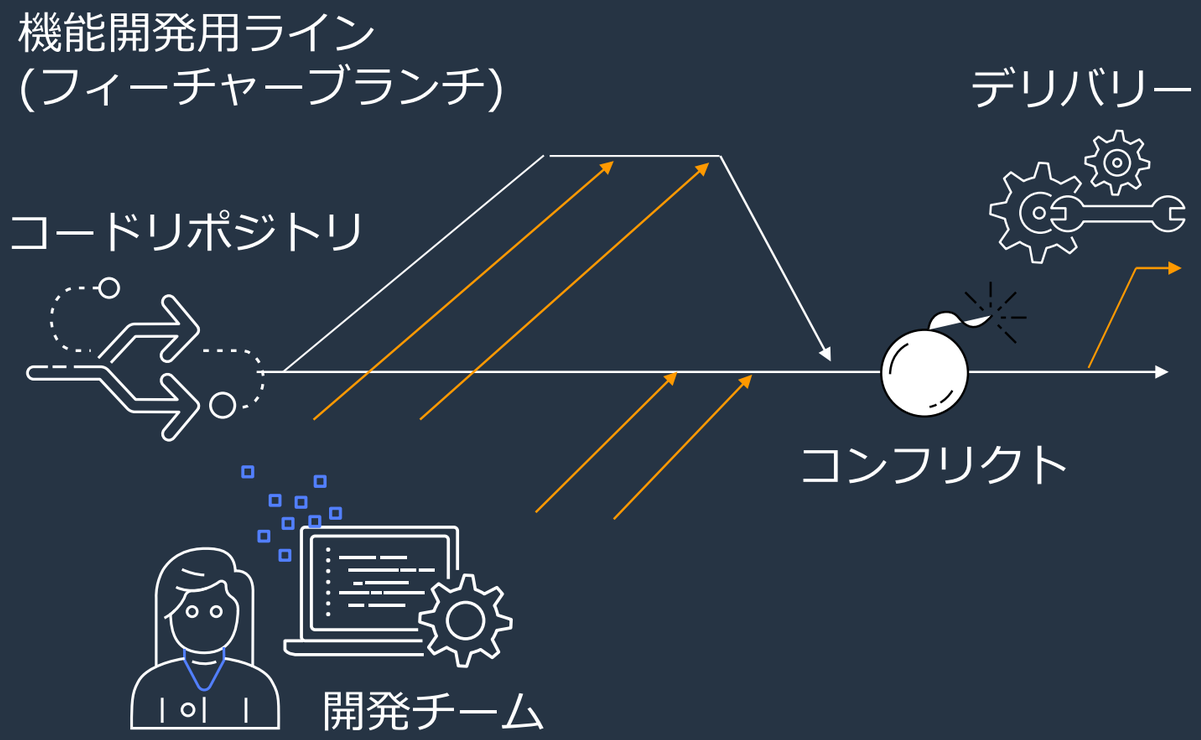
- リリースできないコードをトランクにチェックインしたくない
  - 開発中の機能は承認がないとリリースできない
  - 大規模な変更が多い、INVESTなバックログに慣れておらず分割が難しい
- 緊急リリース (hotfix 対応) にトランクのコードを含めたくない
  - かつ、リリースの頻度が高くない
- テストに3時間かかる

トランクにチェックインし続けることができない

- リリースできないコードをトランクにチェックインしたくない
  - 開発中の機能は承認がないとリリースできない
  - 大規模な変更が多い、INVESTなバックログに慣れておらず分割が難しい
- 緊急リリース (hotfix 対応) にトランクのコードを含めたくない
  - かつ、リリースの頻度が高くない
- テストに3時間かかる

デプロイとリリースの分離

ビジネス判断により、リリースできない機能の開発について



デプロイ = リリース

デプロイとともに、開発中の
機能がリリースされてしまう

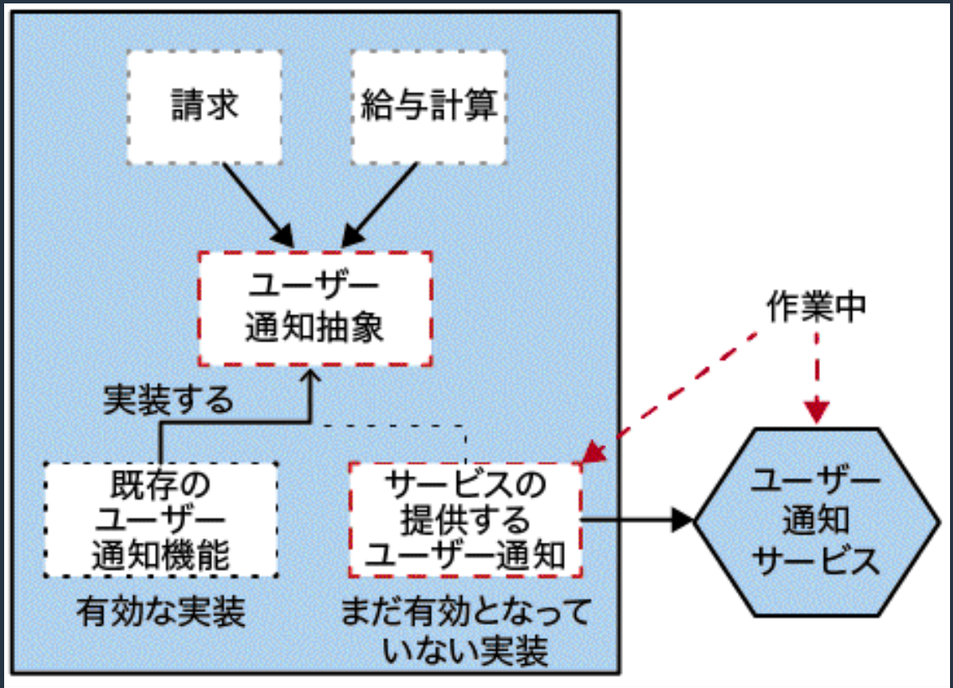
そのため、リリースできない機能を
別ブランチに避ける

リリースせずにデプロイする手法

- 抽象ブランチ
- API 先行の開発
- フィーチャーフラグ

デプロイとリリースの分離

抽象ブランチ



新機能を抽象として切り出す
具体的な実装をトランク上で開発

DIやフィーチャーフラグを活用して
ビジネス上の判断、検証をもってリリース

Sam Newman. モノリスからマイクロサービスへ

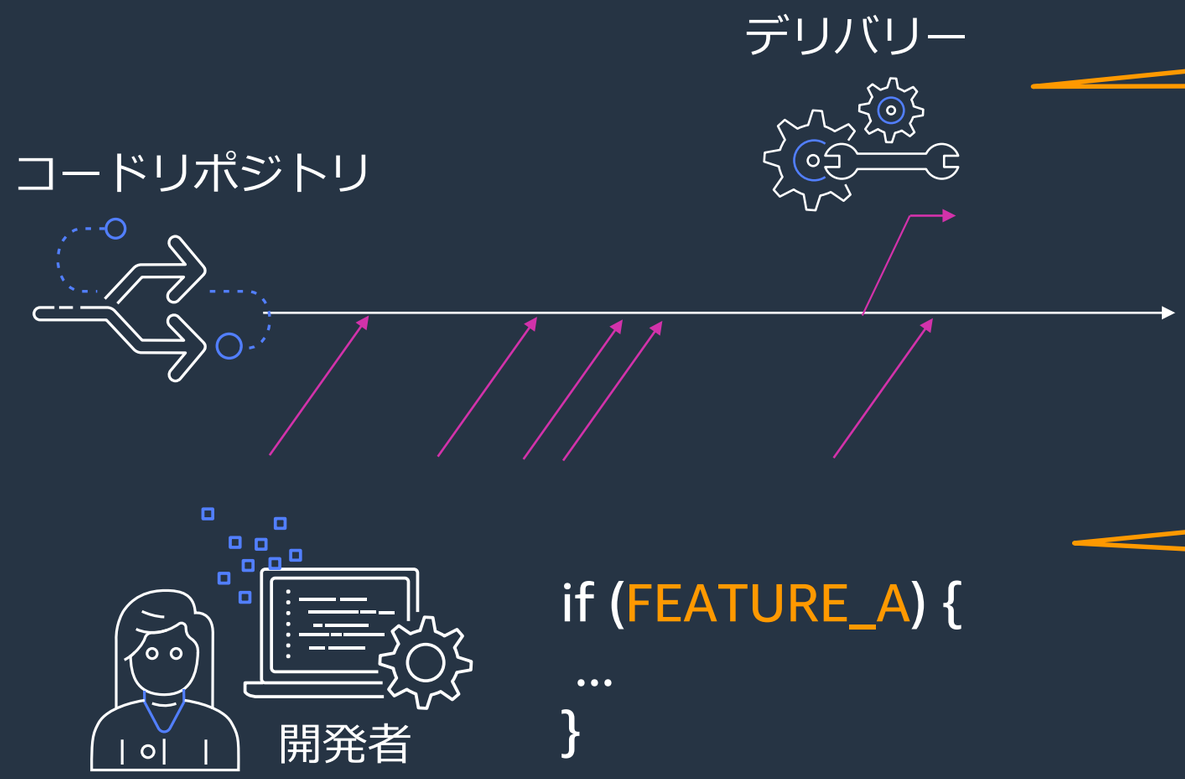
デプロイとリリースの分離

API 先行の開発

- API呼出を介したコンポーネントに分けて開発
- API に対してトランク上で開発する
- 機能が完成して承認を受けたらそのAPIを呼び出すようにする

デプロイとリリースの分離

フィーチャーフラグ



リリース

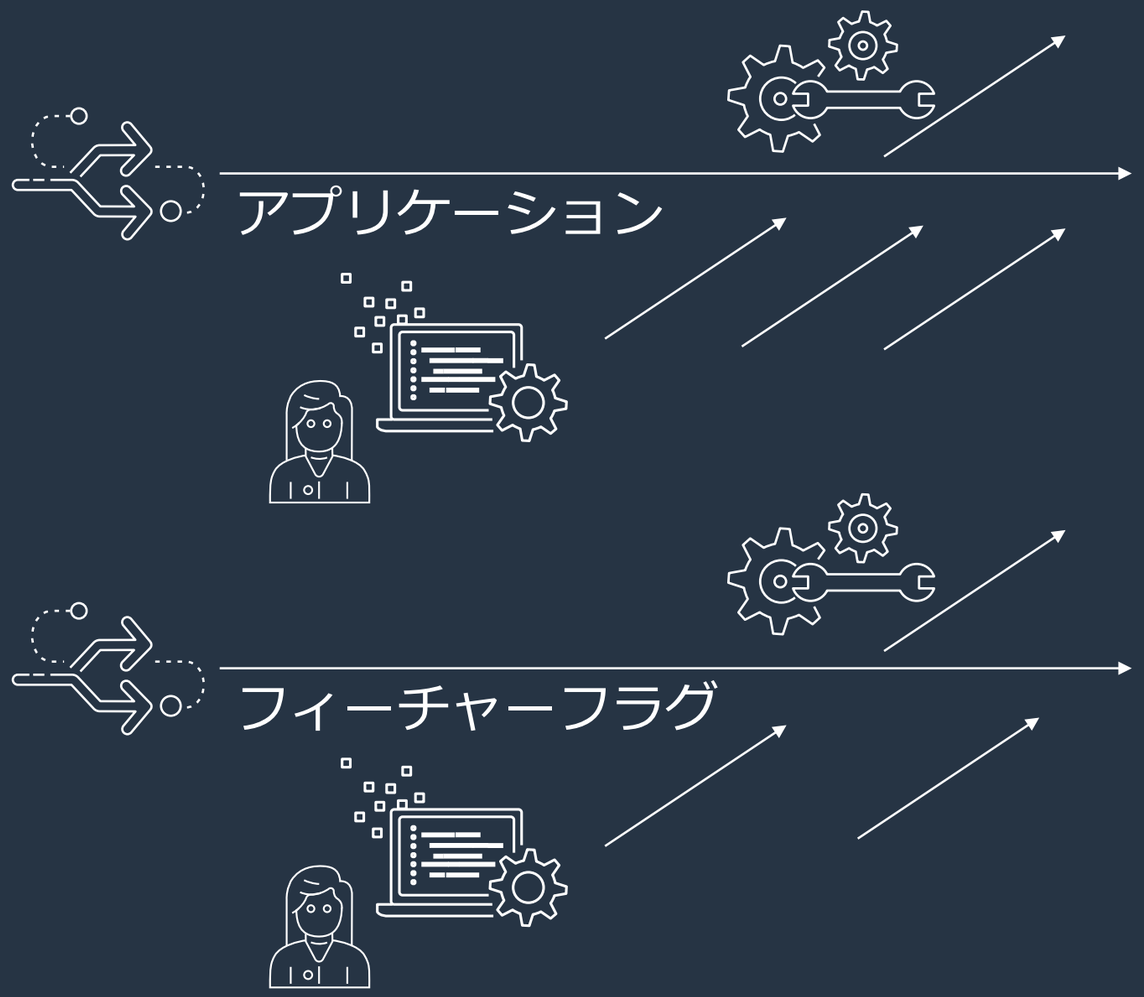
必要な環境・ユーザーにのみフラグを有効にして機能をリリース

デプロイ

開発中の機能をフラグでガードコードはメインラインに継続的にインテグレーション

デプロイとリリースの分離

フィーチャーフラグ

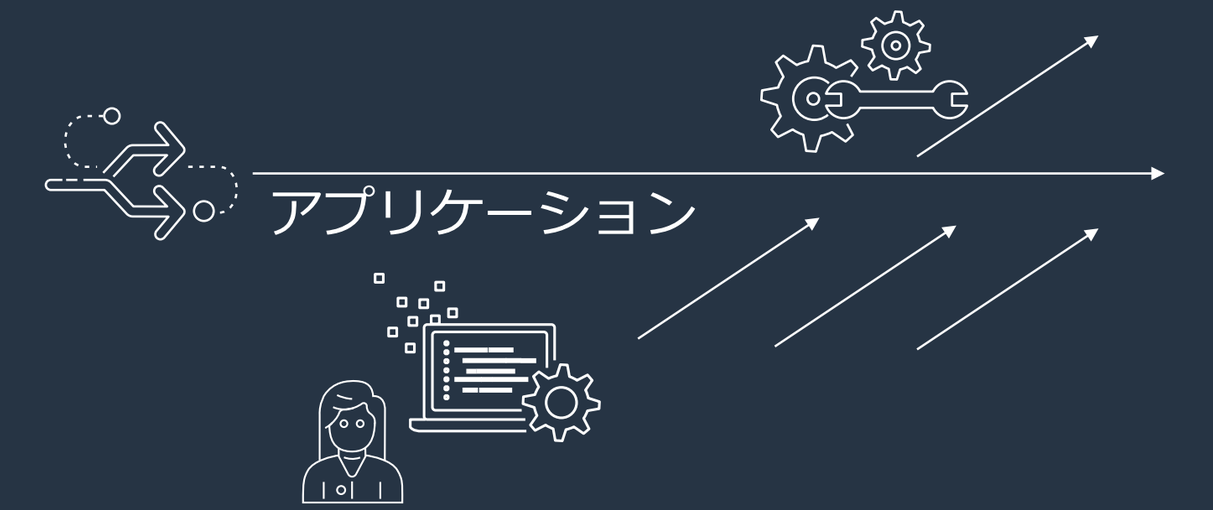


日常の開発
トランクに継続的にチェックイン

機能のリリース
フィーチャーフラグを変更する
QA、承認を入れるプロセス

デプロイとリリースの分離

AWS AppConfig Feature Flag

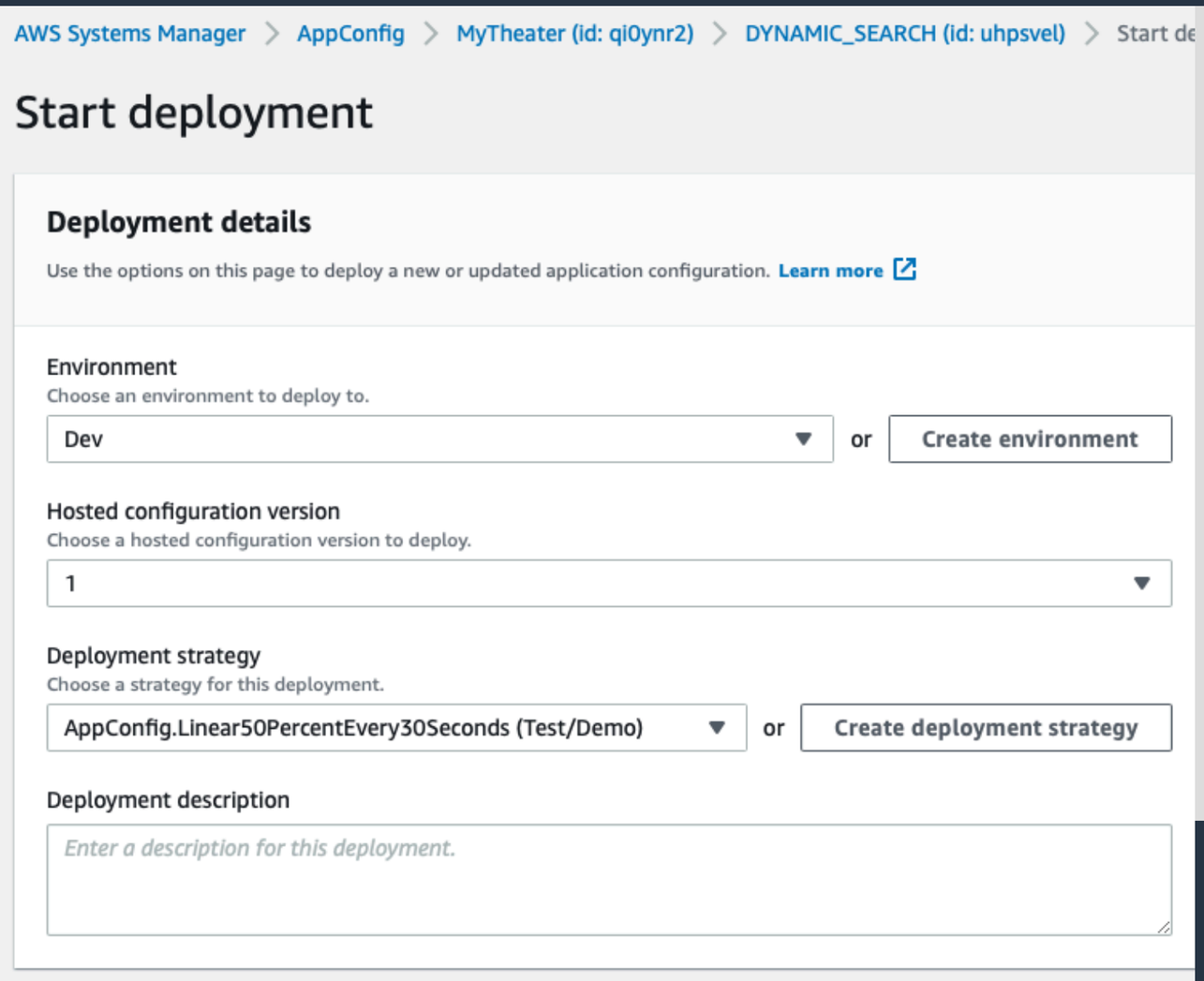


日常の開発
トランクに継続的にチェックイン

機能のリリース
AppConfig でフィーチャーフラグ管理
どの環境にXX%づつデプロイ

デプロイとリリースの分離

AWS AppConfig Feature Flag



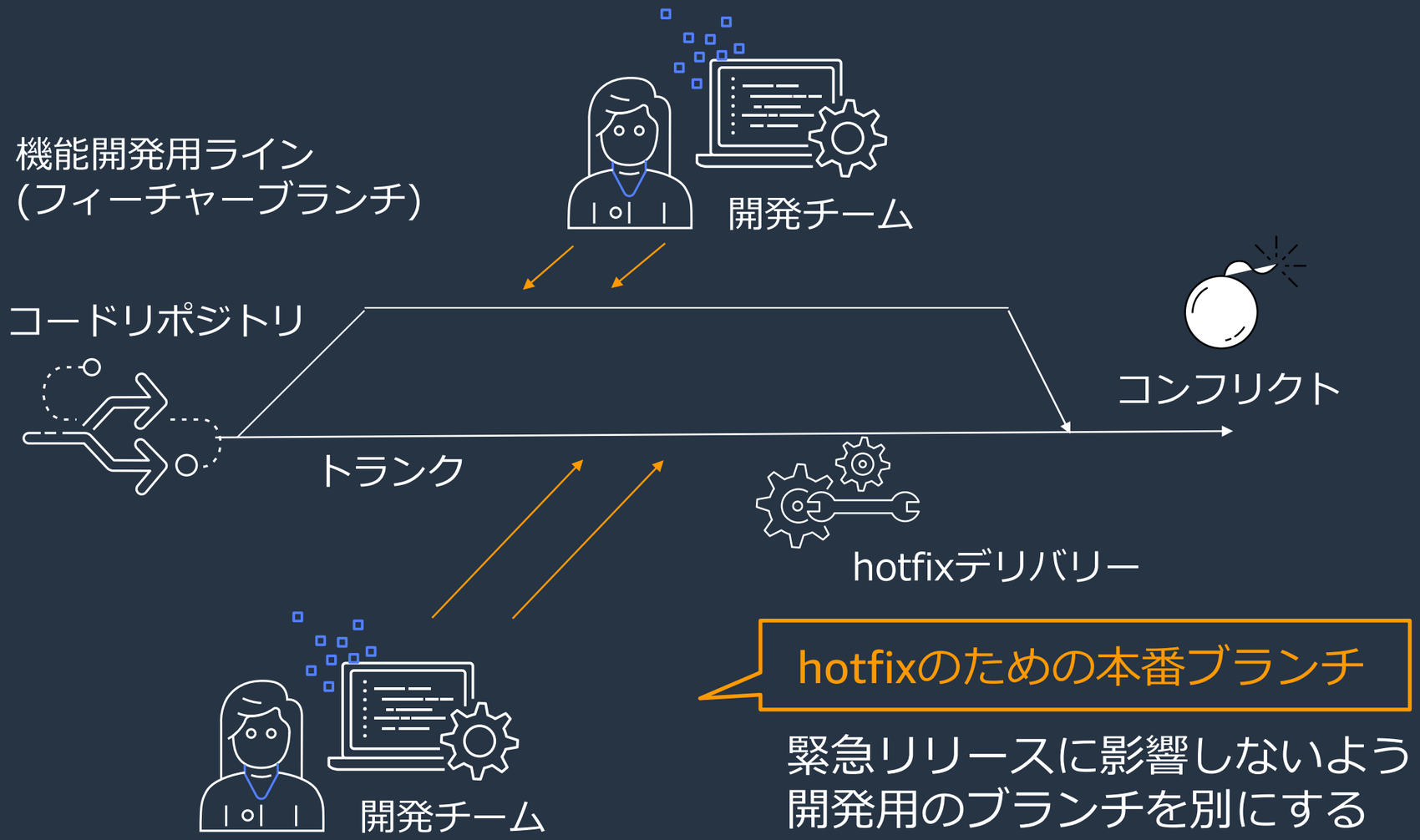

```
client =
boto3.client("appconfigdata",..)

res =
client.start_configuration_session(
ApplicationIdentifier="MyTheather",
EnvironmentIdentifier="Dev",
ConfigurationProfileIdentifier="DYNAMIC_
SEARCH", )

// ...
```

トランクにチェックインし続けることができない

- リリースできないコードをトランクにチェックインしたくない
  - 開発中の機能は承認がないとリリースできない
  - 大規模な変更が多い、INVESTなバックログに慣れておらず分割が難しい
- 緊急リリース (hotfix 対応) にトランクのコードを含めたくない
  - かつ、リリースの頻度が高くない
- テストに3時間かかる

非トランクベース開発の hotfix リリース

緊急リリース (hotfix) に開発中の機能を含めたくない

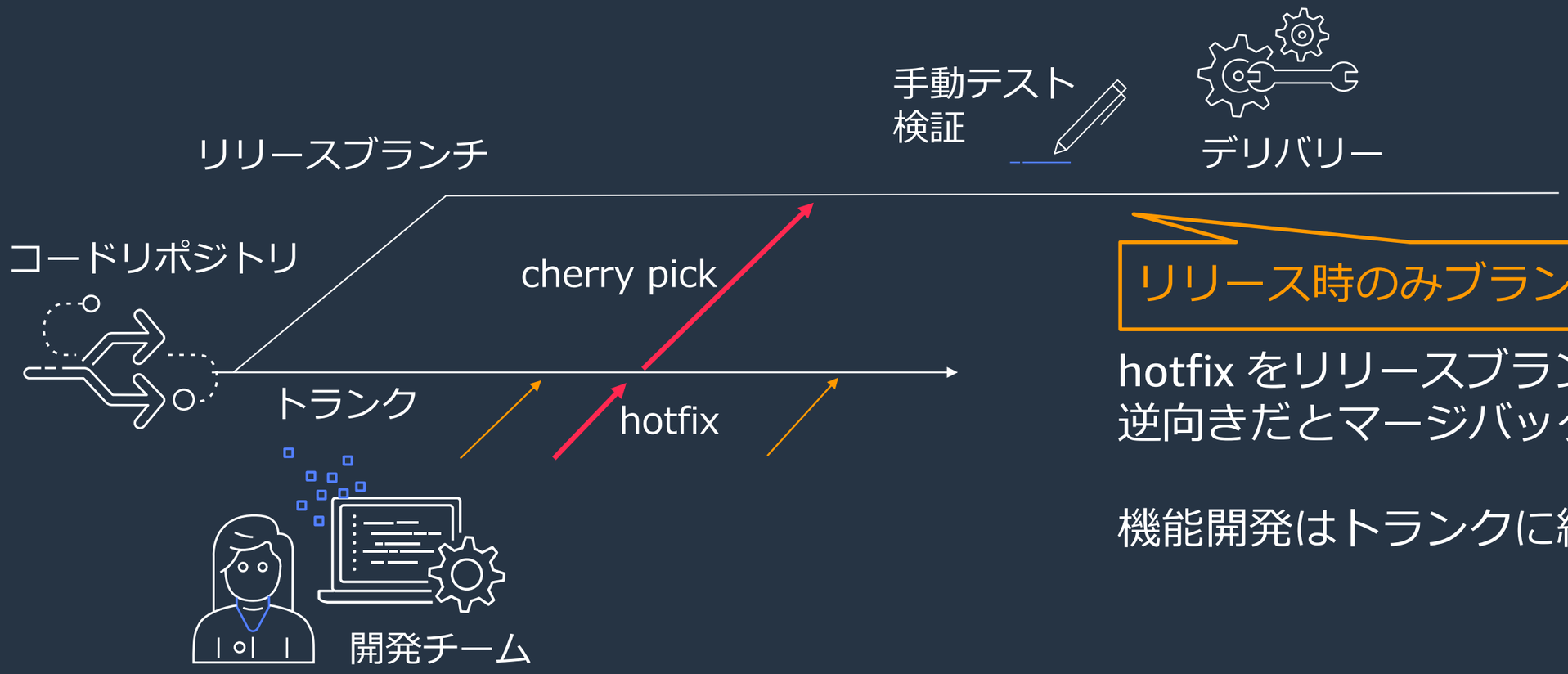


hotfixのための本番ブランチ

緊急リリースに影響しないよう
開発用のブランチを別にする

リリースブランチの活用

マージしないブランチで対策する



リリース時のみブランチを作成

hotfix をリリースブランチに適用
逆向きだとマージバックを忘れやすい

機能開発はトランクに継続的にチェックイン

トランクにチェックインし続けることができない

- リリースできないコードをトランクにチェックインしたくない
  - 開発中の機能は承認がないとリリースできない
  - 大規模な変更が多い、INVESTなバックログに慣れておらず分割が難しい
- 緊急リリース (hotfix 対応) にトランクのコードを含めたくない
  - かつ、リリースの頻度が高くない
- テストに3時間かかる

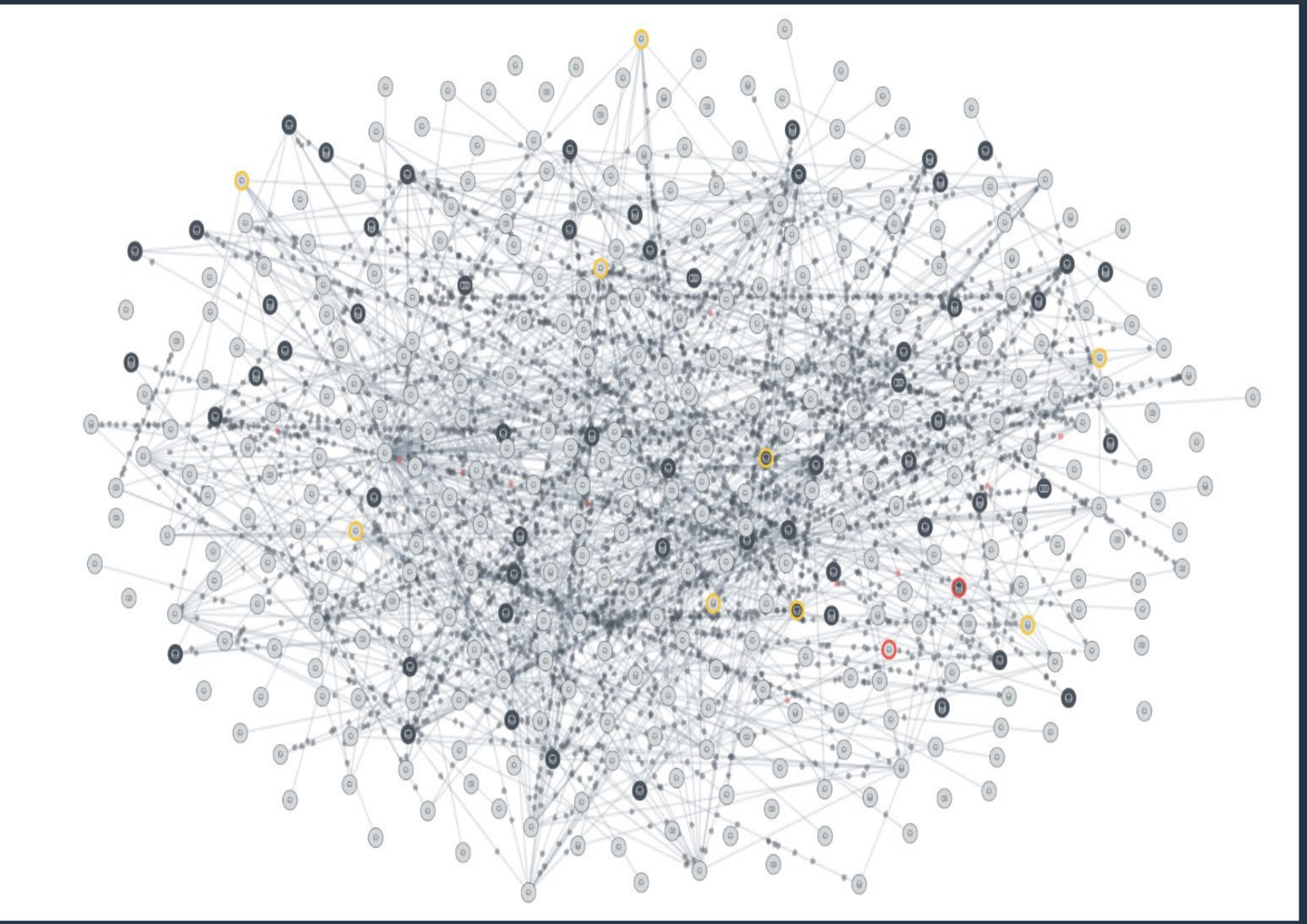
テストに時間が掛かりすぎる

- ユニットテストをデータベースとか他コンポーネントに依存させない
  - テストをステートレスにして、コンテナ環境で並列実行
- ローカルで実行できるようにする
- 他の機能開発を待たないでも開発できるようにする
  - コードをチェックインしたら忘れて次に行く
  - INVEST なバックログ

トランクにチェックインし続けることができない

- リリースできないコードをトランクにチェックインしたくない
  - 開発中の機能は承認がないとリリースできない
  - 大規模な変更が多い、INVESTなバックログに慣れておらず分割が難しい
- 緊急リリース (hotfix 対応) にトランクのコードを含めたくない
  - かつ、リリースの頻度が高くない
- テストに3時間かかる

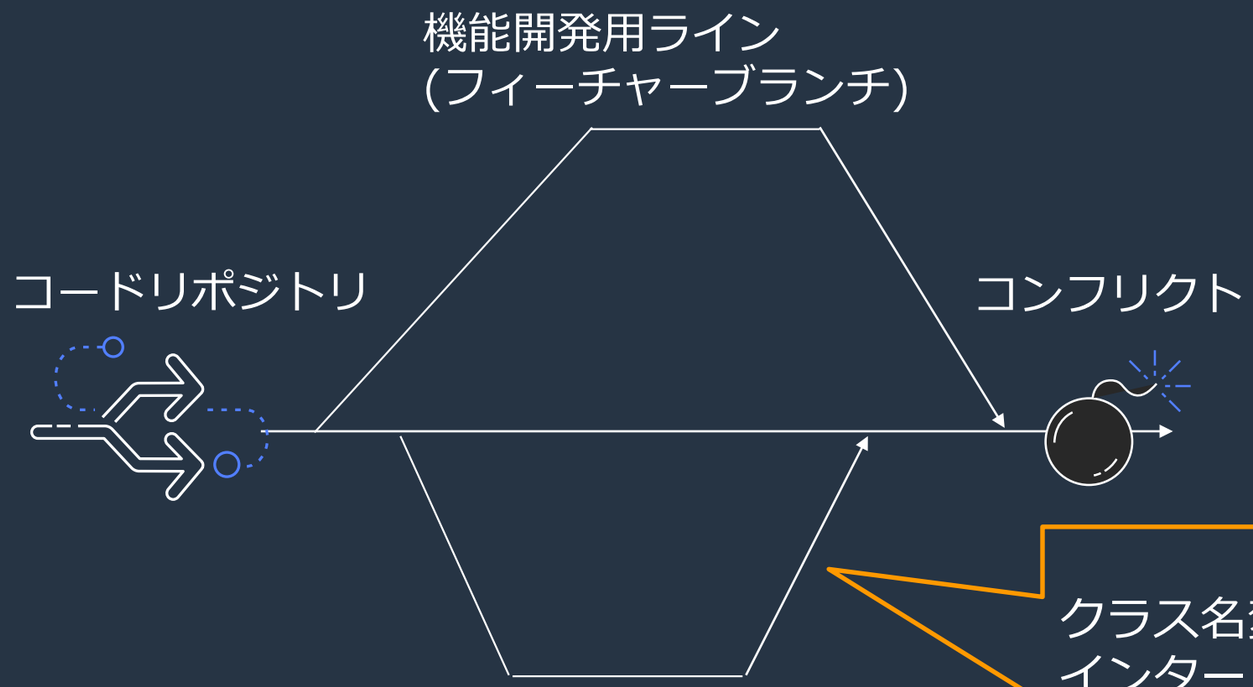
トランクに自信を持ってチェックインできない



- コードが疎結合でない、整理されていない
  - 抽象ブランチもフィーチャーフラグも導入しにくい
  - モジュール化されていないので影響範囲が分からない
- 長期間の手動QAがないと安心できない

なぜ、トランクベース開発が求められるか？(再掲)

継続的な改善 (リファクタリング) が容易



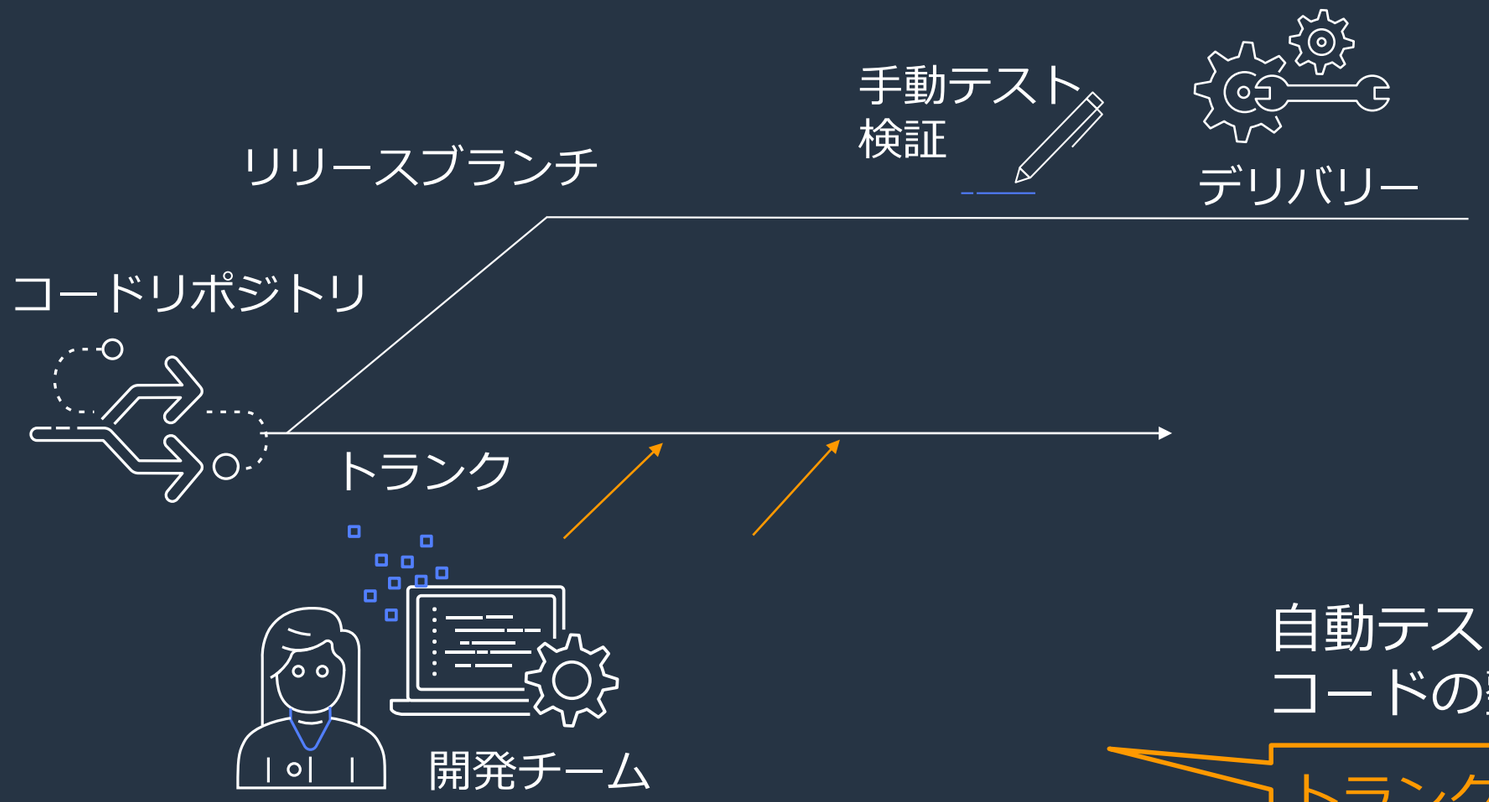
- 長命なブランチの運用だと
  リファクタリングが難しい
- 技術的負債が積み上がる

クラス名変更
インターフェース、抽象の抽出
メソッドの移動
変数名の変更
etc. etc.

Getting Started

リリースブランチを活用したリリース

トランクベース開発を先行させ、継続的インテグレーションを整備



リリースプロセス実施

手動テスト
承認
本番デプロイ
hotfix

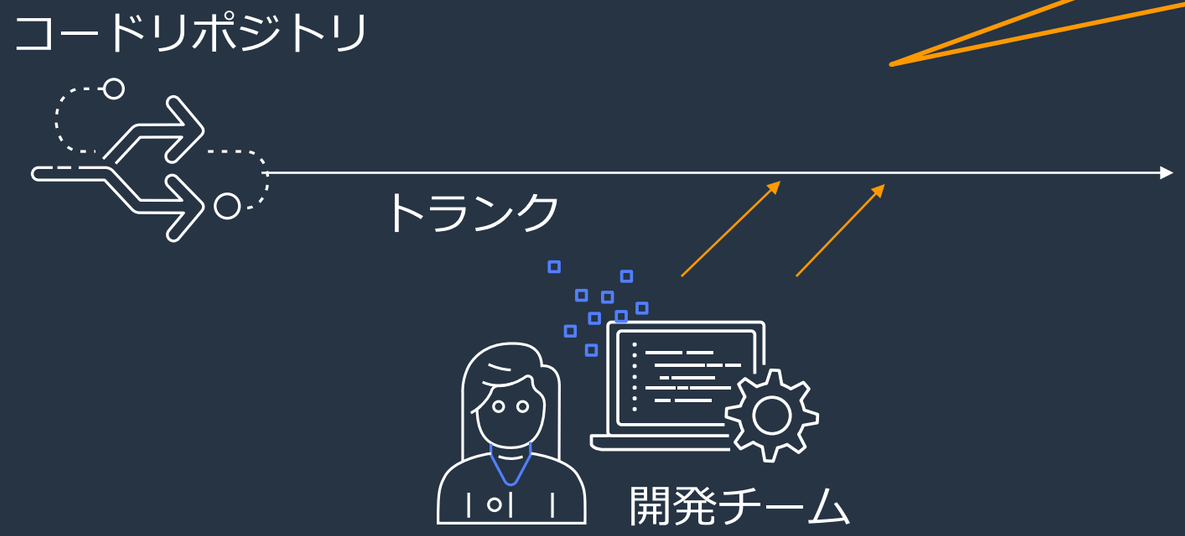
自動テストの整備
コードの整理、リファクタリング

トランクベース開発を実践

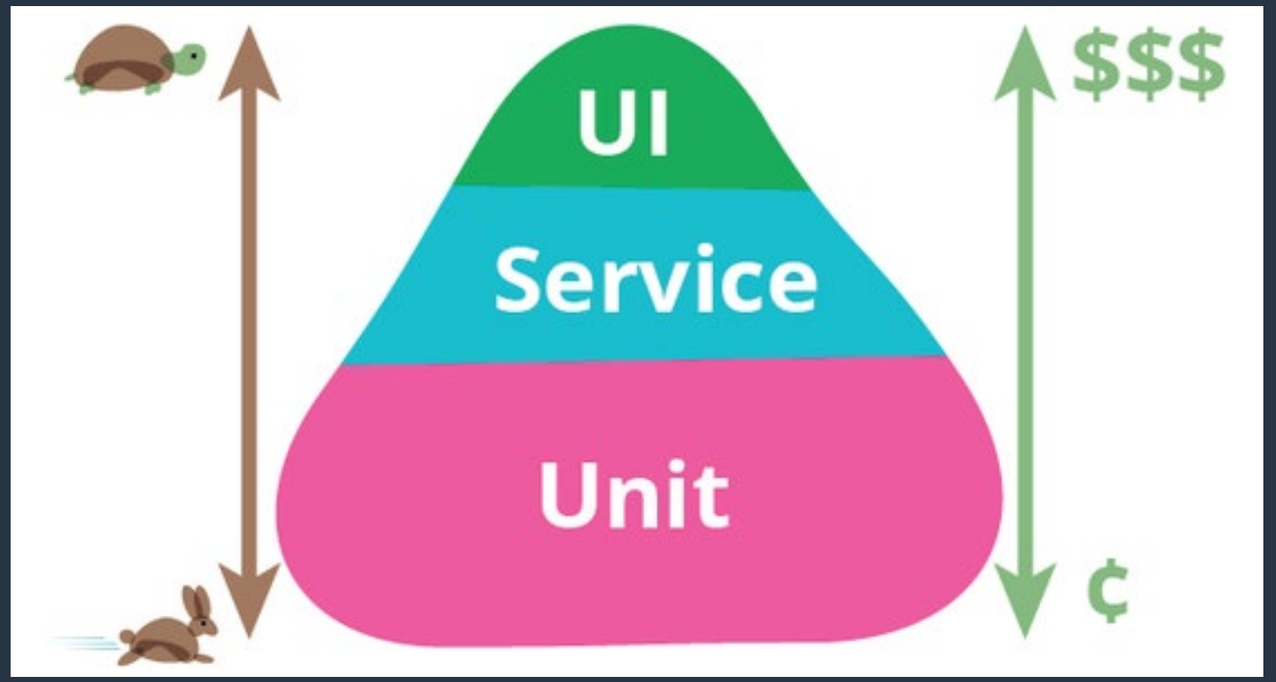
自動テストの整備

自動テストがないと始まらない
壊れたまま放置 -> 壊れたら開発を止めて直す文化

自動テスト、静的解析



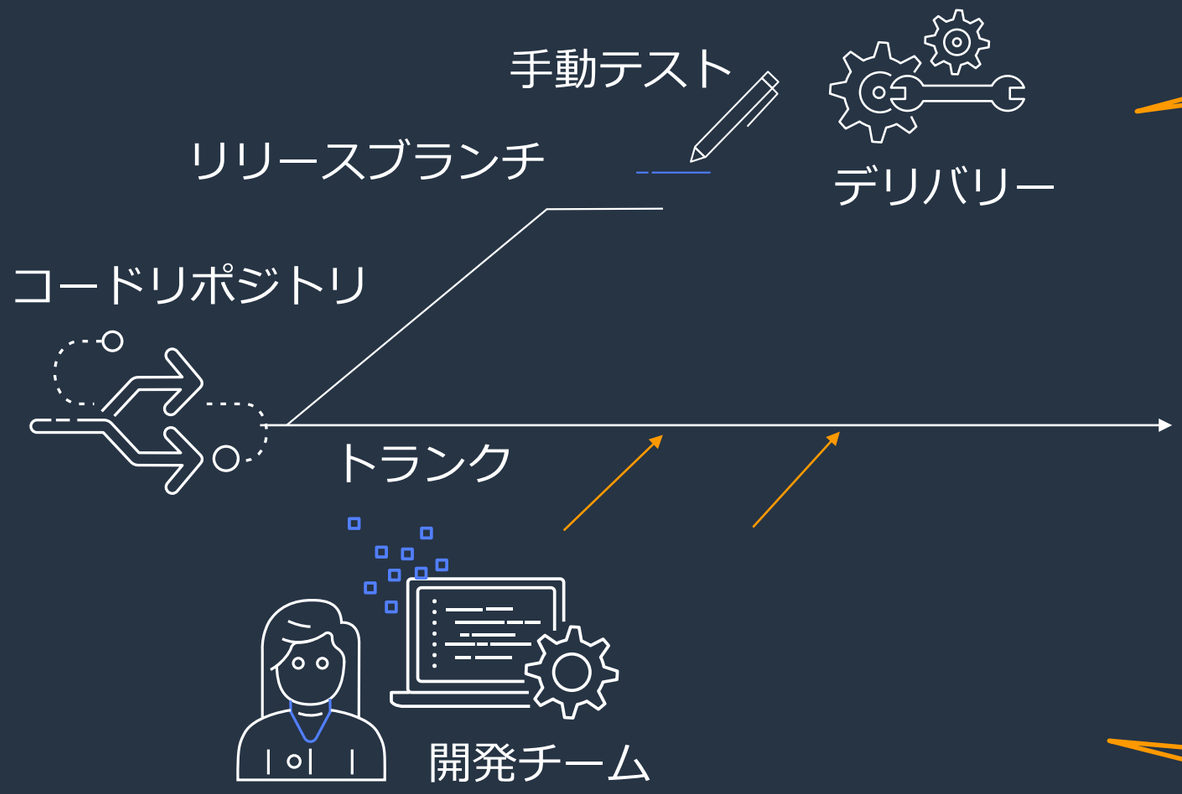
テストピラミッド



https://martinfowler.com/bliki/TestPyramid.html

リリースブランチを短命に

リリースブランチでやることを減らしていく



この部分の自動化を進める

テストを自動テストに移す
リリース後の監視、ロールバックなど
を自動化していく

自動テストの整備
コードの整理、リファクタリング

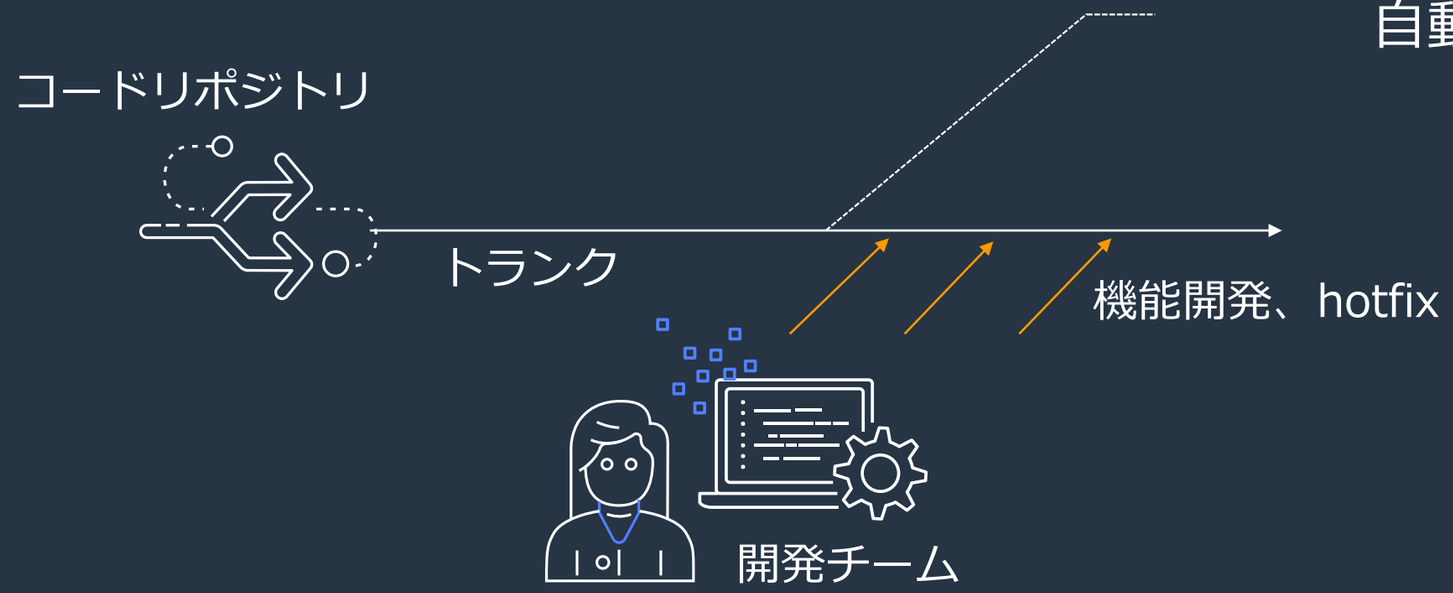
トランクベース開発を実践

リリースブランチの廃止

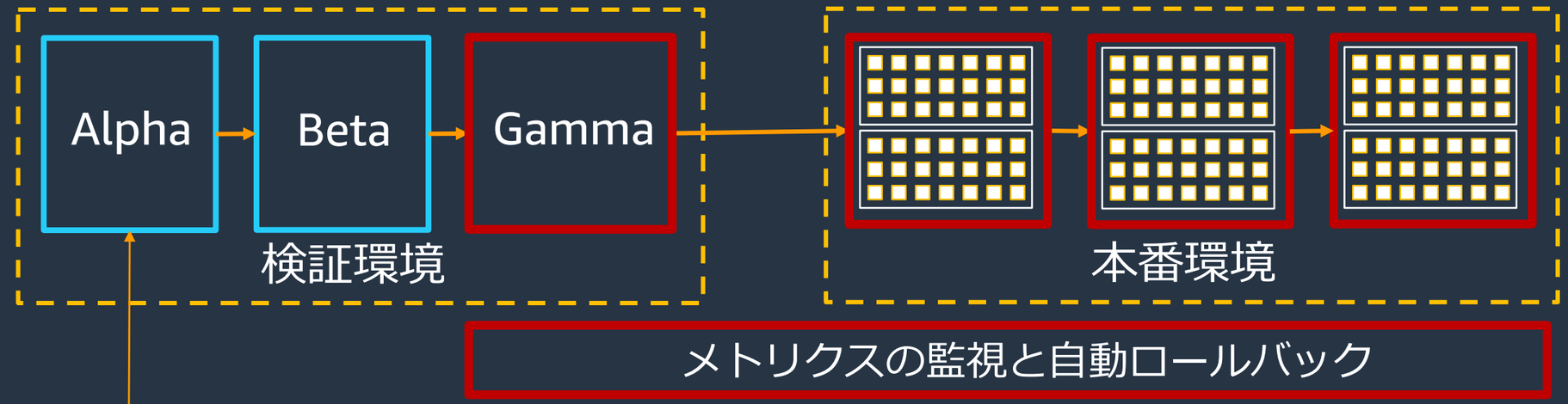
継続的デリバリーにより検証やデリバリーの自動化が進めば廃止できる

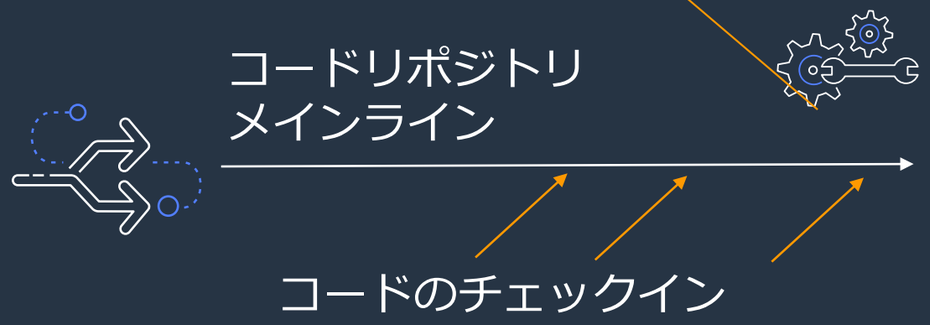
ブランチを継続的デリバリーに置き換え

hotfix もトランクで実施
自動化されたプロセスで継続的にデリバリー



Amazon / AWS のデプロイメントパイプライン





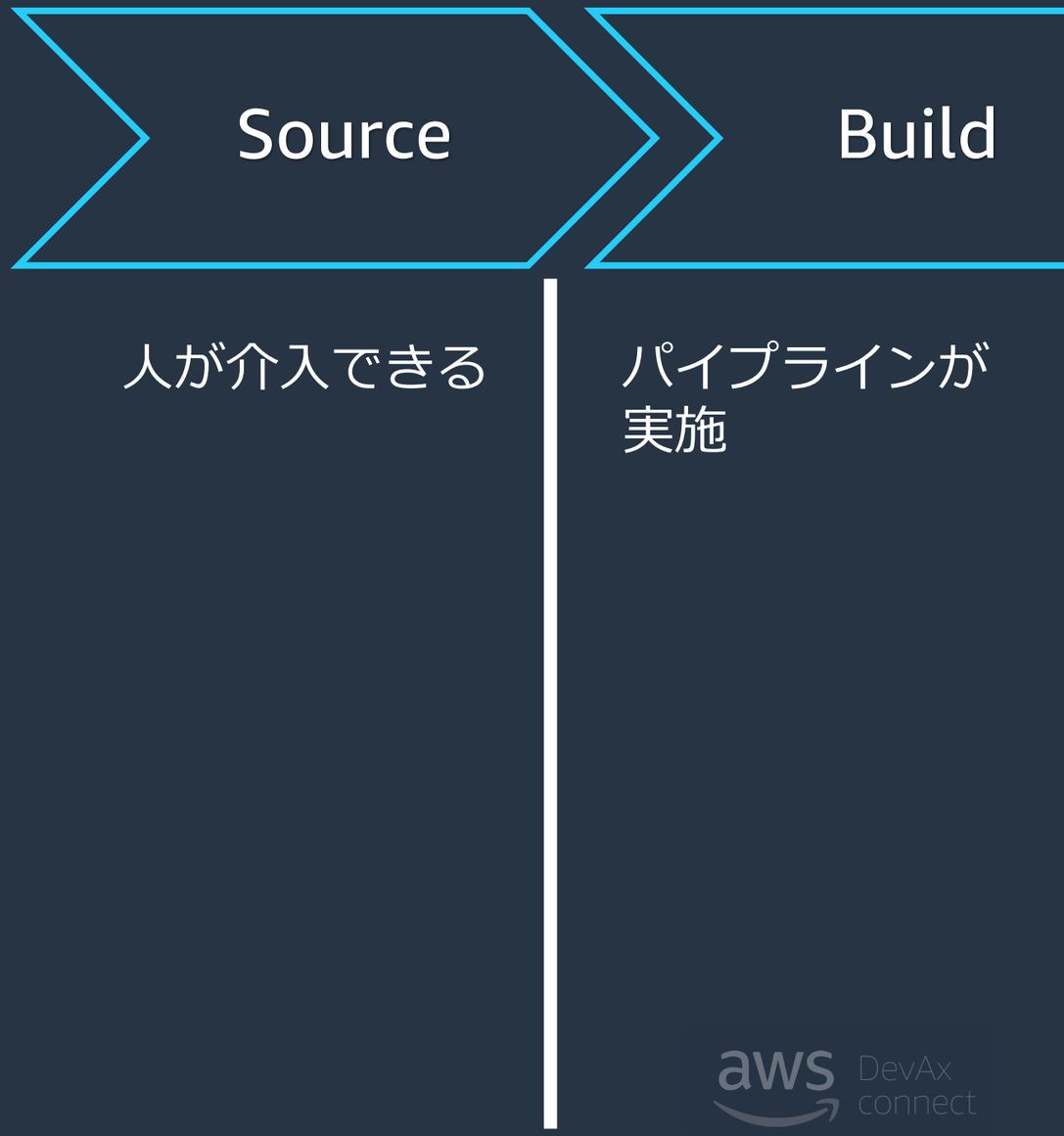
複数の検証環境
後半ほど結合度が高い「重い」自動テスト
メトリクス監視、アラートで自動ロールバック

本番環境へのデリバリー
複数環境に分割、Progressive にデリバリー
メトリクス監視、アラートで自動ロールバック

Amazon Builders' Library
https://aws.amazon.com/jp/builders-library/automating-safe-hands-off-deployments/

自動化が進んだ時のコードレビューの重要性

- マージされたら、本番デプロイまで人が関与することはない
- コードレビューは、手作業で行う最後のプロセス
- 本番にデプロイされる変更をレビューしたり承認したりできる最後のチャンス
  - テストが含まれているか
  - この変更は既存のシステムで監視可能か
  - 安全にローリングアップデート、ロールバックできるか
  - etc.



まとめ

- 適切な CI/CD の導入を目指す
  - トランクベース開発が CI/CD のキモ、前提になる
- トランクベース開発を導入するためのアクションを検討する
  - リリースブランチ、デプロイとリリースの分離などでブロッカーに対処
- ブランチ戦略を決定する
  - 長期ブランチを切る前に、本当に必要なのかよく検討する
  - なるべく長期ブランチを減らす、トランクベース開発を目指す

Thank you!

Masatoshi Hayashi
Specialist Solutions Architect, Containers
AWS Japan
@literalice

aws DevAx connect