CI/CD

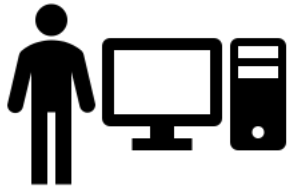
生産性向上チーム
宮田 淳平

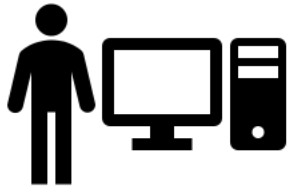
この講義の目的

- CI/CD の基本的なところを一通り知ってもらう
- キーワードを把握して今後の学習の取っ掛かりとしてもらう

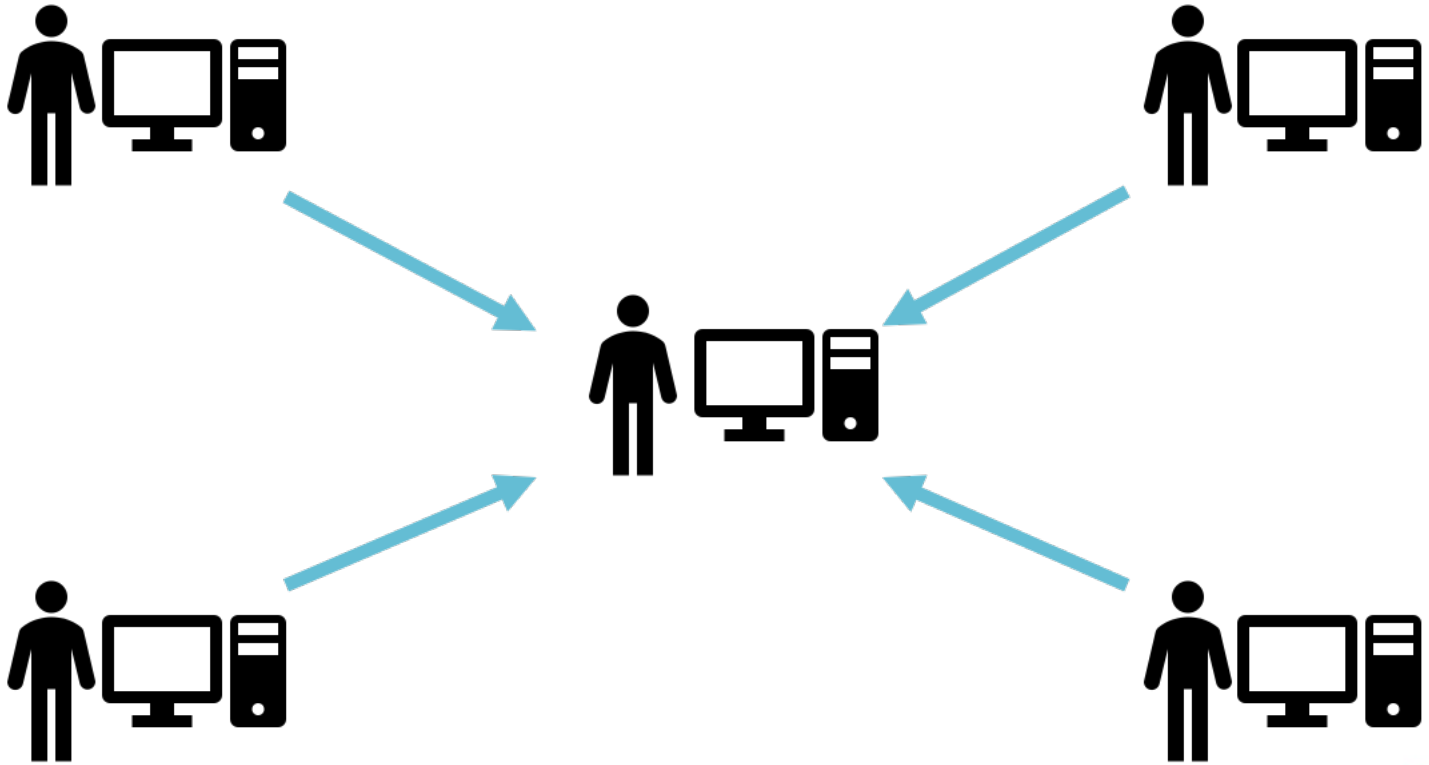
CI/CD がない開発の例

各自の環境で開発

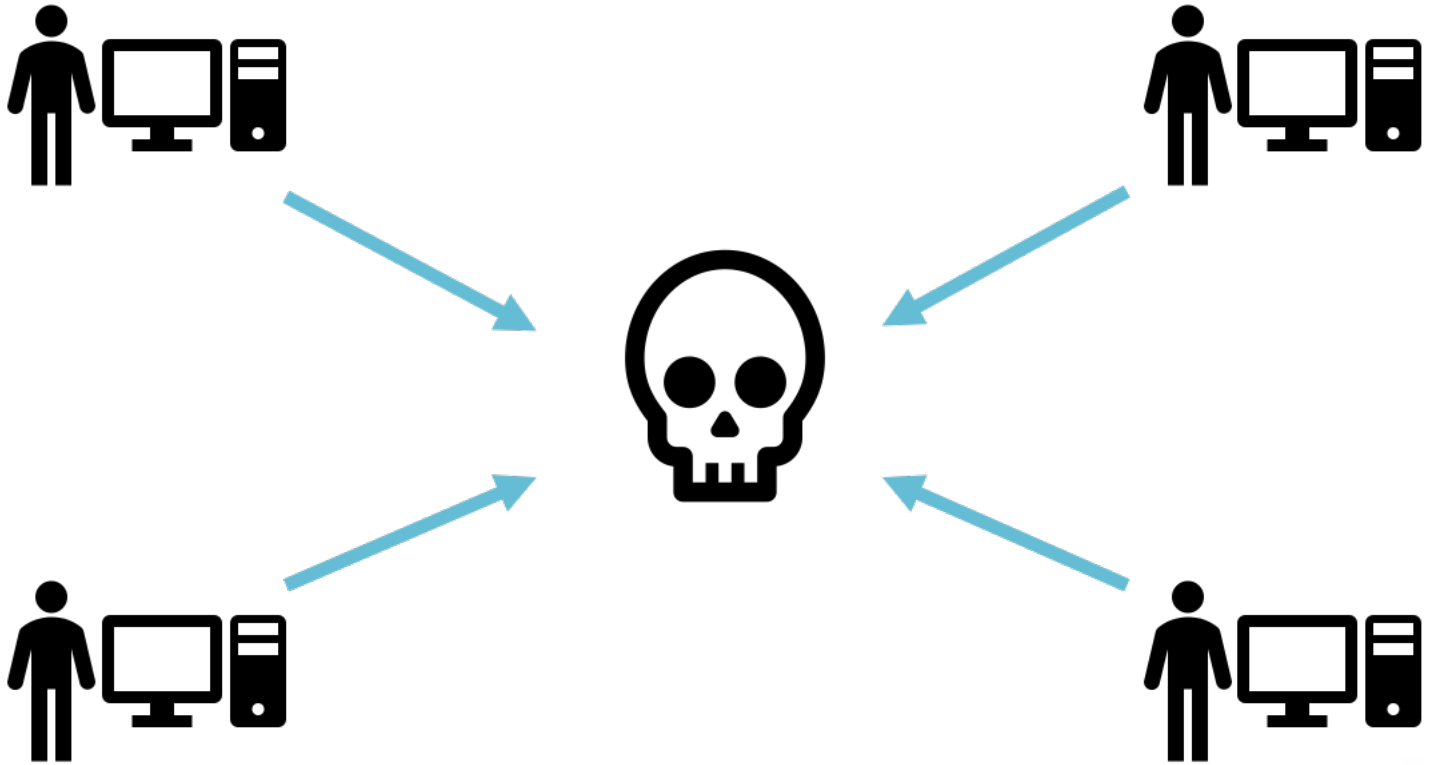




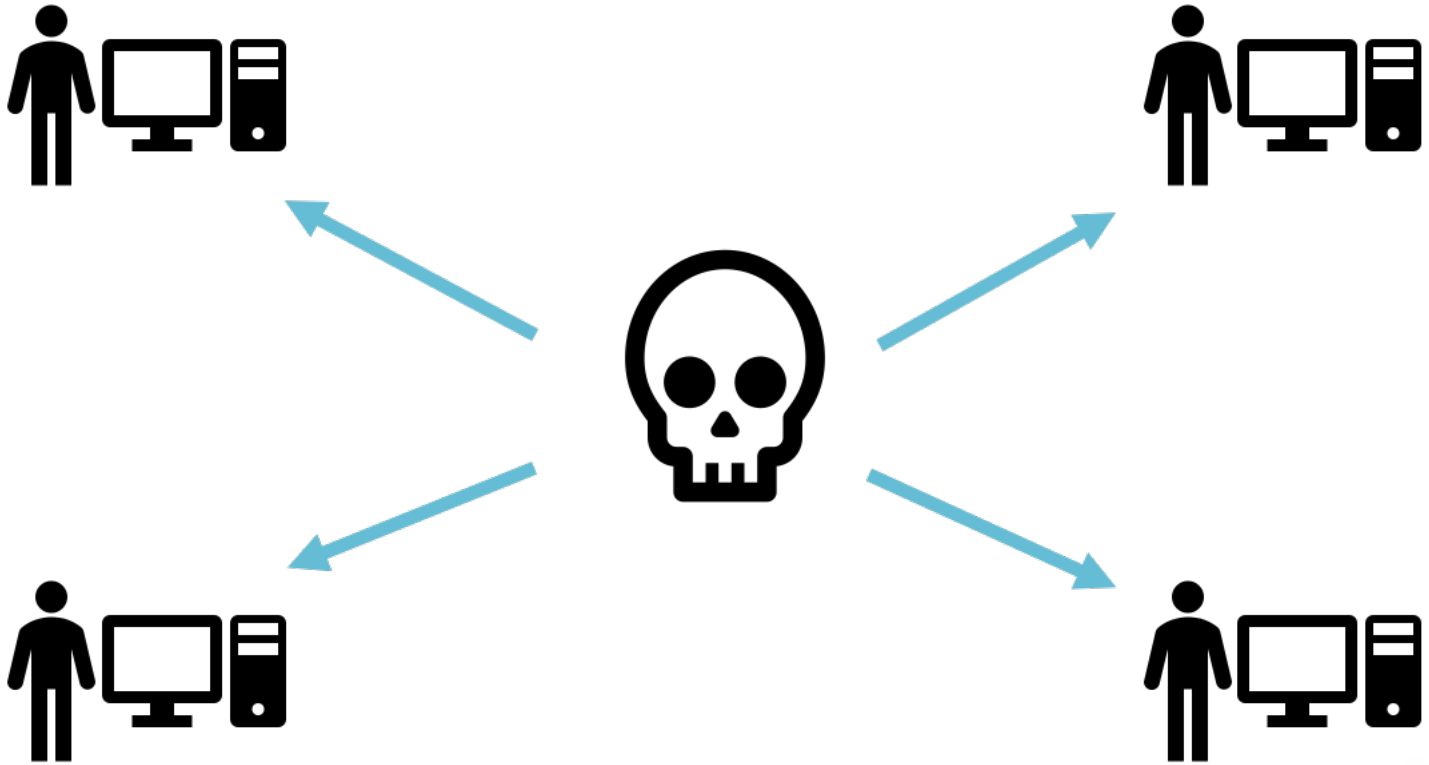
リリース前に各自の変更を結合して試験



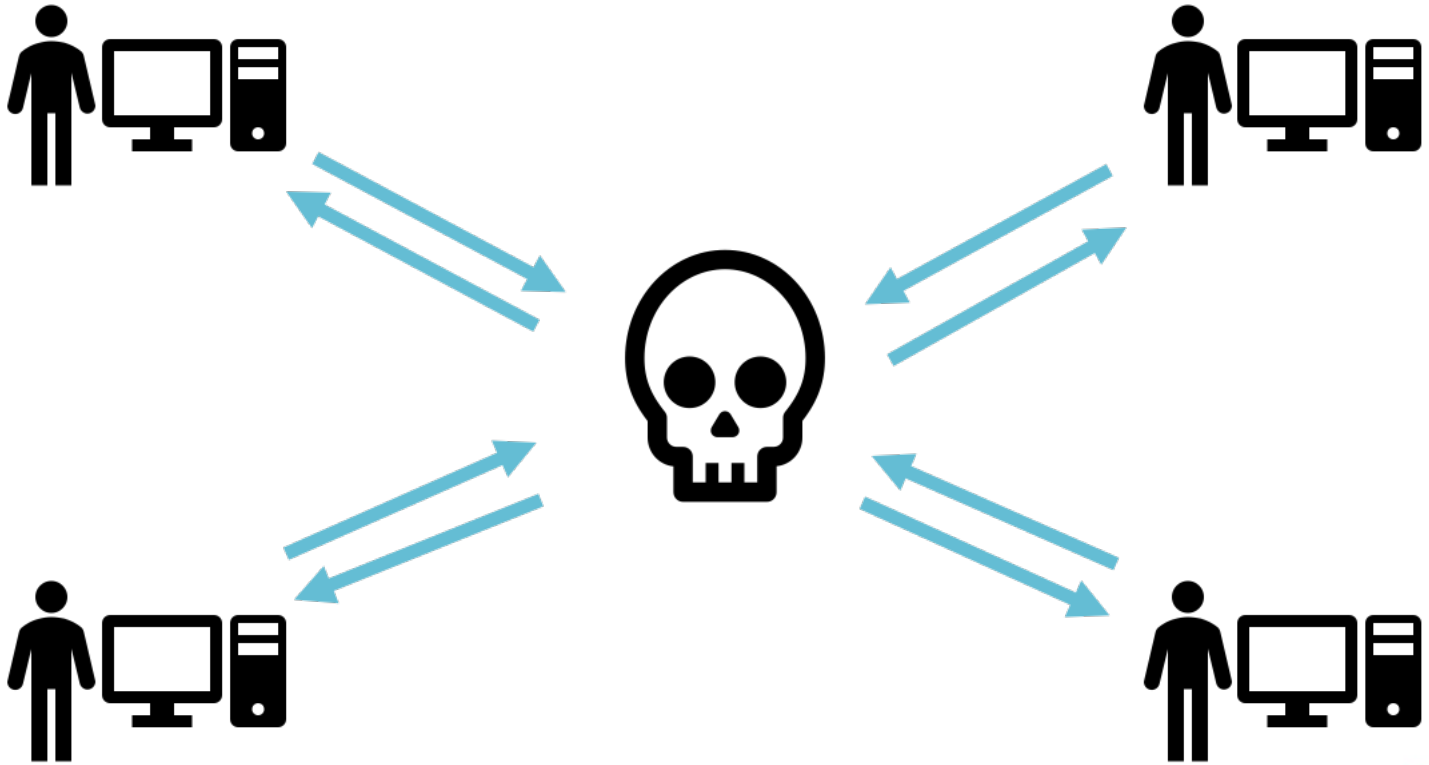
不具合だらけ



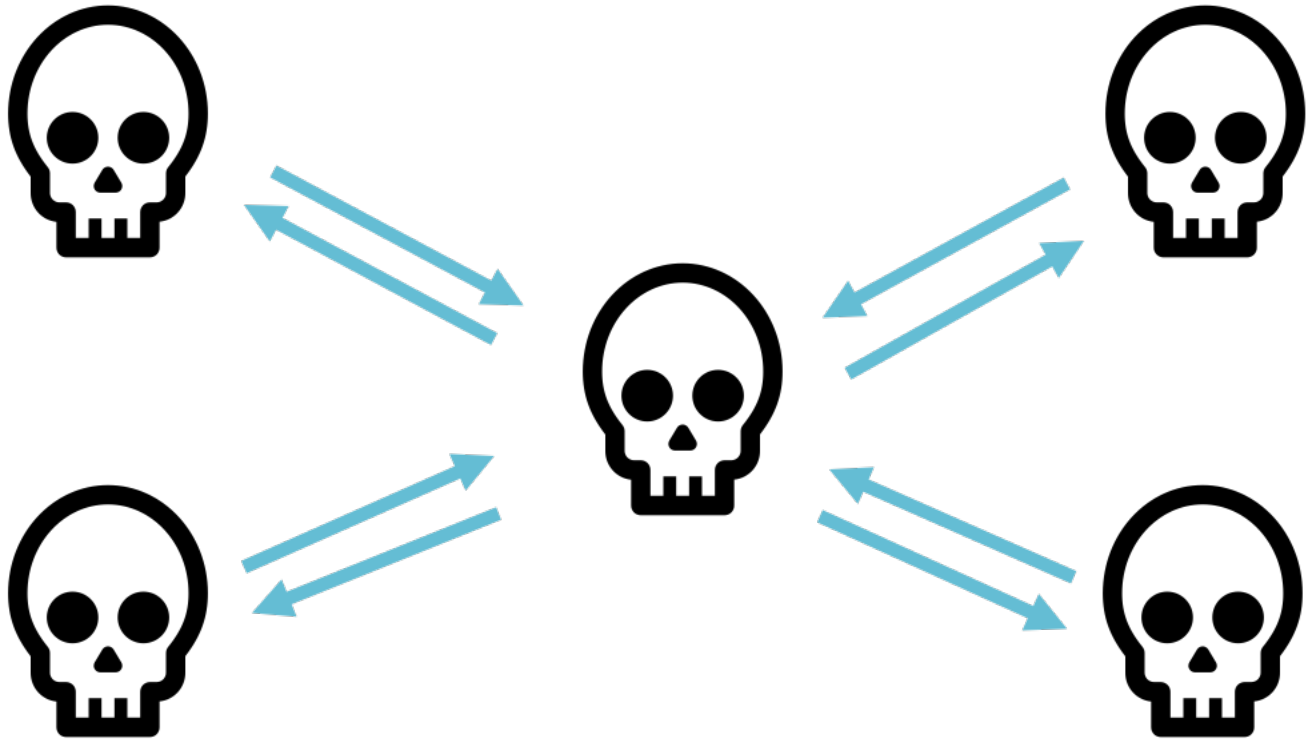
修正を依頼しても原因特定が難しい



修正しても新たな不具合を埋め込んで以下繰り返し



終わりが見えなくて辛い



問題

- 結合のタイミングでまとめて大きな差分が発生する
  - 壊れやすい、原因究明が困難
- 変更してから問題が見つかるまでのタイムラグが大きい
  - 学習が遅れるのでその間にも類似不具合が埋め込まれる可能性が高い
- リスク（不確実性）が高い
  - 結合後の対応がどれぐらいになるか予測しづらい

CI (Continuous Integration)

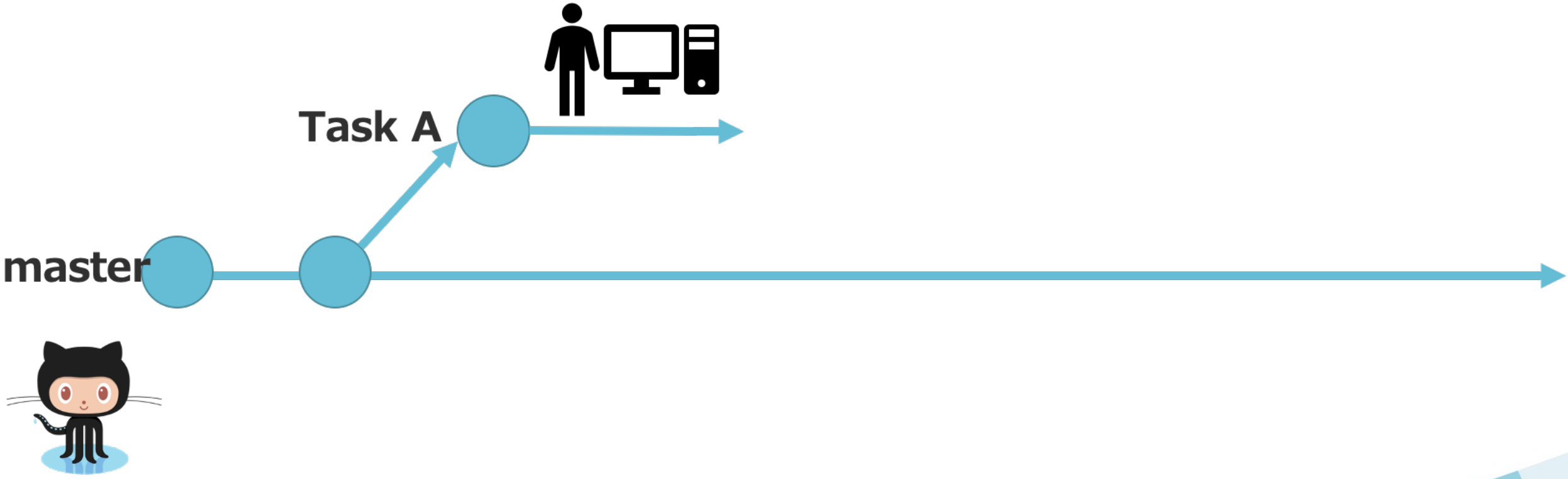
CI とは？

- 開発プラクティス
- 一日に何回もバージョン管理システムに変更をマージする
- 毎回テストなどを含む自動ビルドが実行される

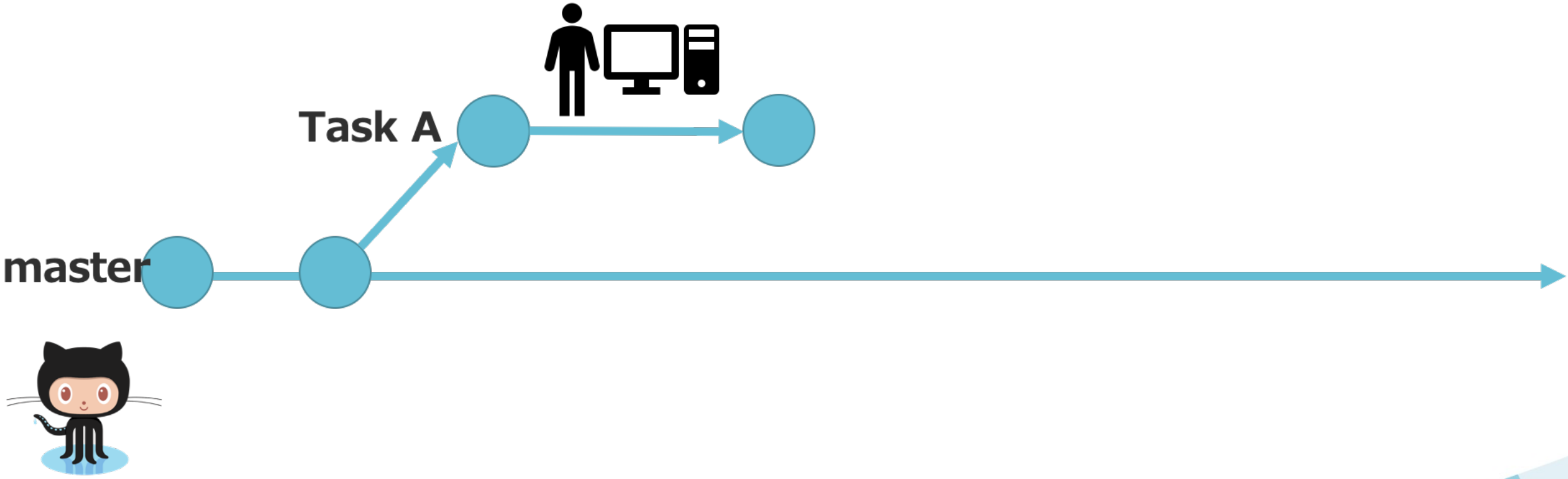
CI がある開発の例

開発者がメインラインからタスクブランチを作成する

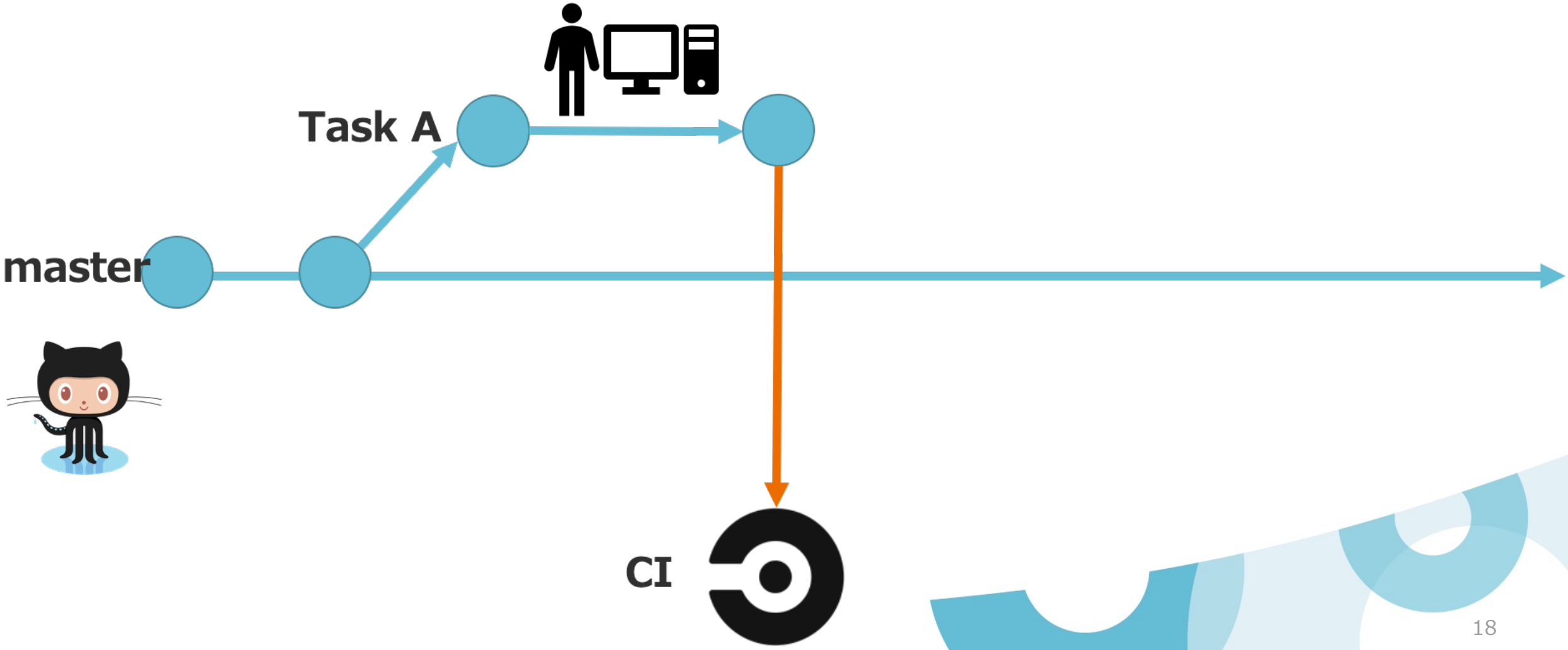
ローカルでコードを変更する



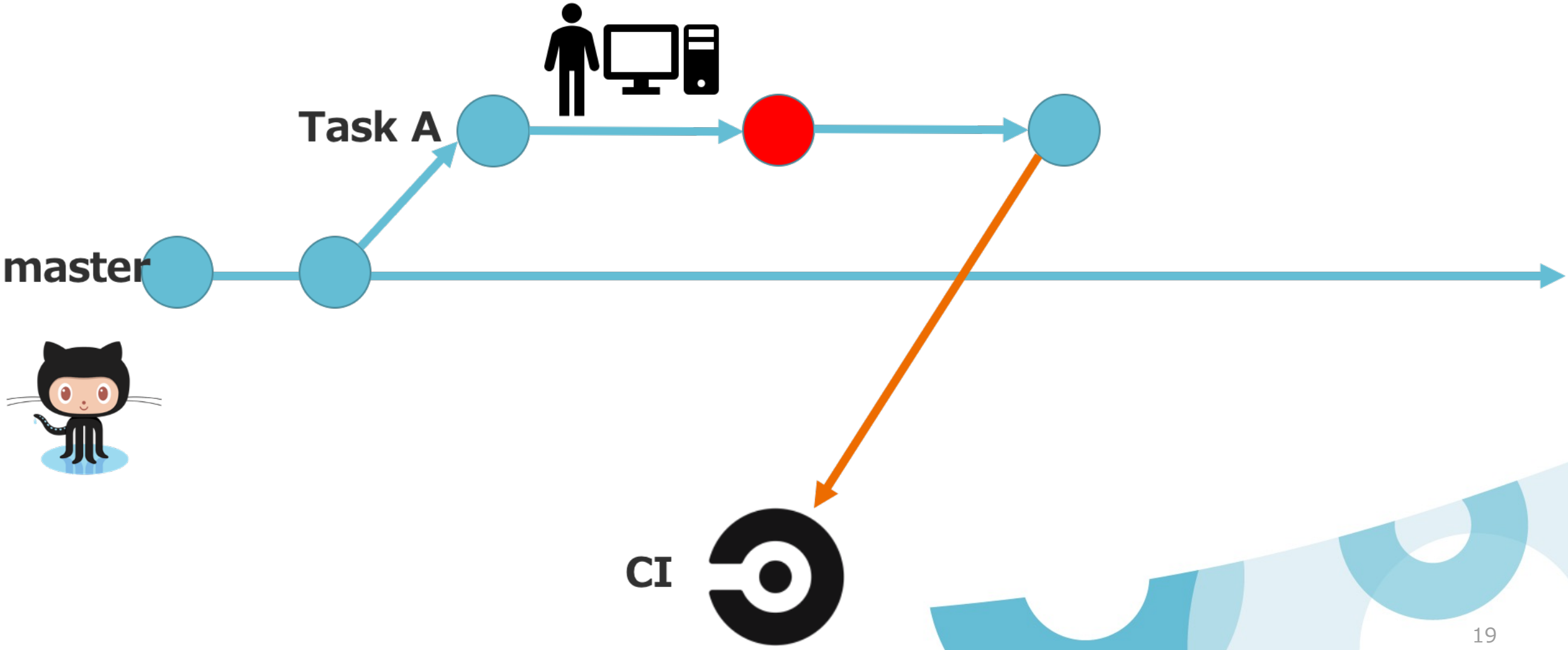
タスクブランチにプッシュして PR（プルリクエスト）を作成する



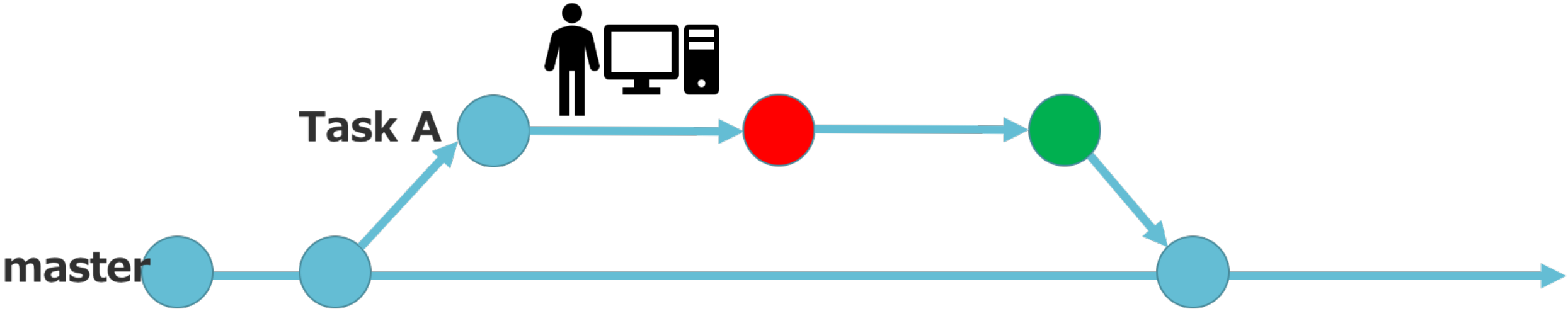
CI ツールがタスクブランチのコードでビルドを実行する



ビルドが失敗したら通るまで修正 & CI 再実行

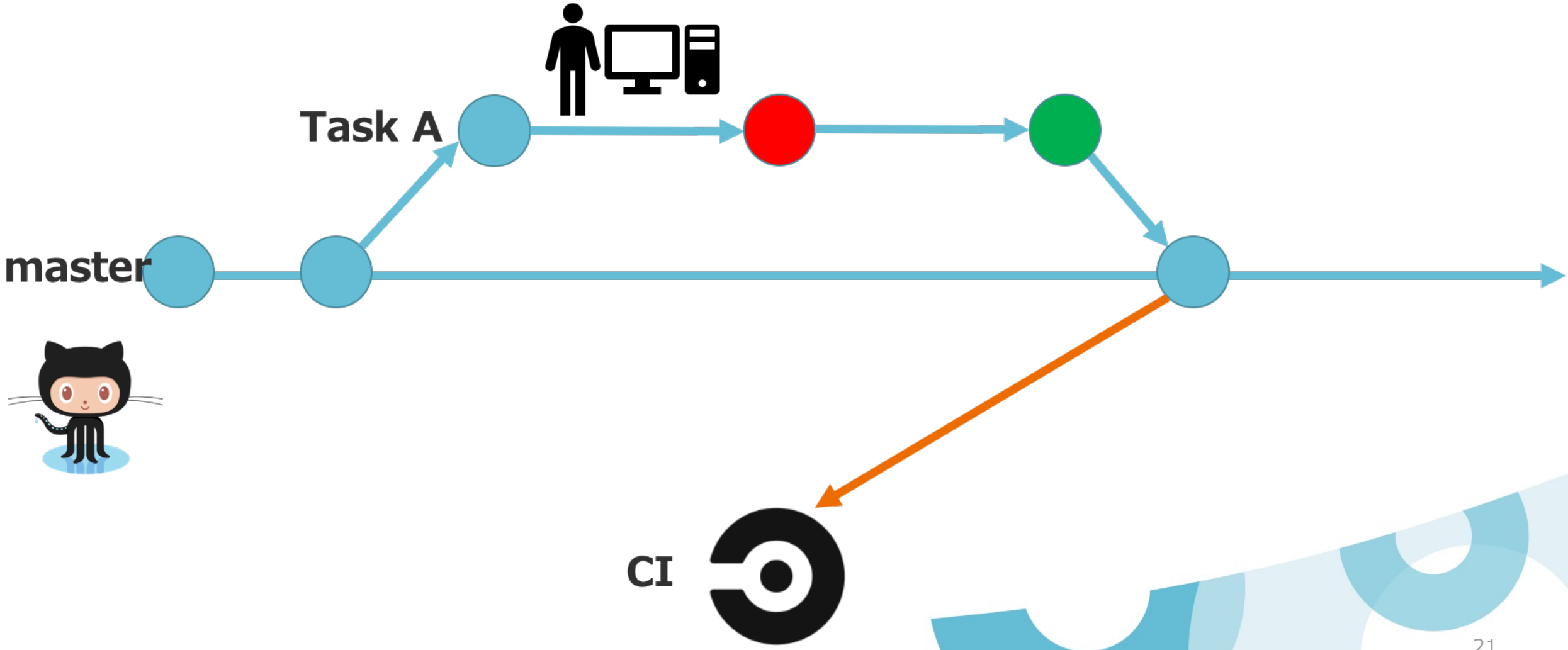


ビルドが成功したらメインラインにマージする

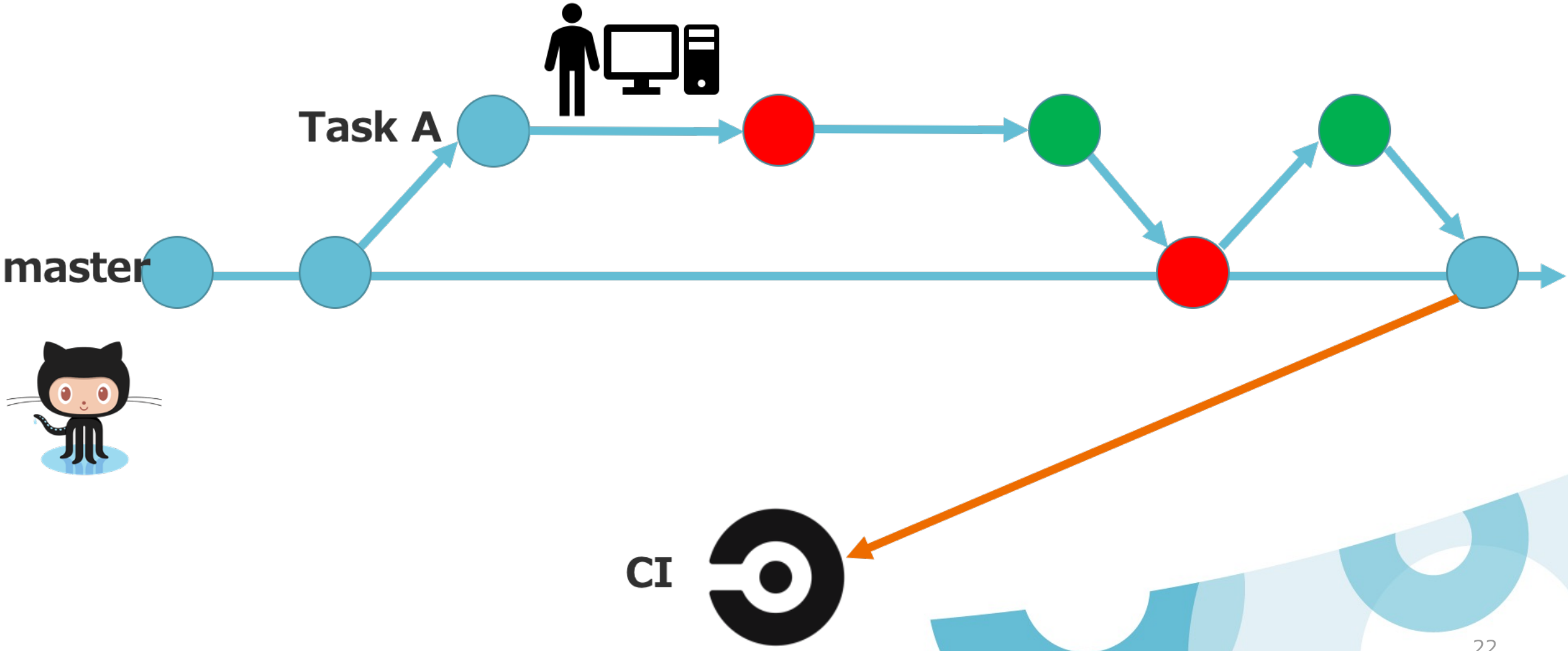


CI

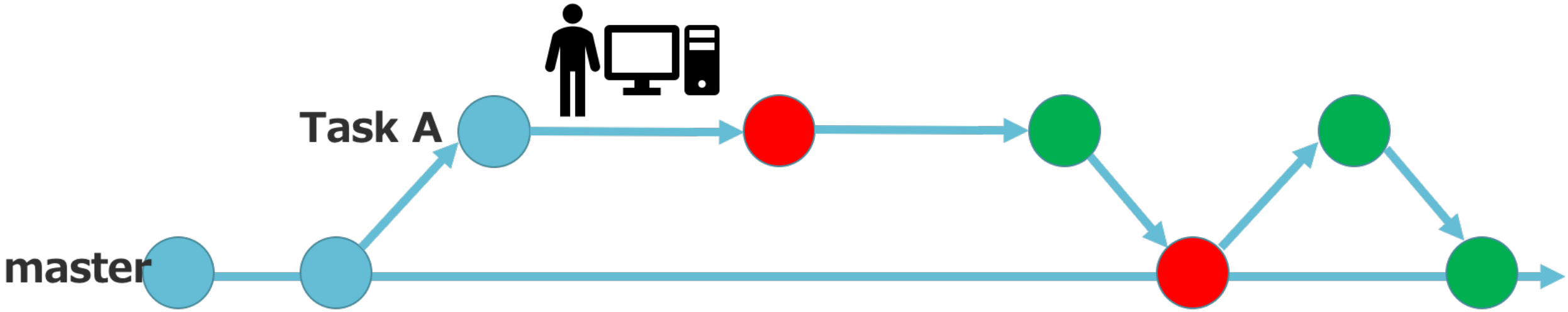
CI ツールがメインラインのコードでビルドを実行する



ビルドが失敗したら通るまで修正する



メインラインでビルドが成功したら完了



CI

CI の利点

- 高速なフィードバック
  - 問題の早期発見
  - 高速な学習
- 頻繁に変更がマージされるようになれば差分が大きくならない
  - 問題発見時の調査が簡単になる
  - リスク（不確実性）が減る
- Success/Fail の可視化によるコミュニケーション促進
- 毎回の変更が自動テストで保証される安心感

CD (Continuous Delivery)

CD とは？

- CI の発展型
  - コードの変更がトリガーとなって実行されることは同じ
- コード変更からリリースまでに必要な検証を行う
  - 常に信頼できるリリースができる状態を保つ
- デプロイパイプラインという形で自動化する
  - 部分的に手動作業が入ることもある

デプロイパイプラインの例

CD の利点

- リリースのコストやリスクを抑えられる
  - いつでもリリースできる
- コード変更からリリースまでのフローが可視化される
  - どこで問題が起きてるか、どこがボトルネックか、などがすぐにわかる

CI/CD 内で実行すること

静的解析

- 構文チェック
  - 構文エラーを防ぐ
- コードスタイルチェック
  - 可読性を高める、本質的でない議論を防ぐ
- コードパターンチェック
  - エラーが発生しやすいパターンを防ぐ

自動テスト

- 単体テスト
  - 小さい単位のコードが役割通り動作するかチェックする
- 結合テスト
  - 複数のコードを組み合わせた機能が正しく動作するかチェックする
- 受け入れテスト（E2E テスト）
  - ビジネス要求を満たしてるかチェックする
- 上記以外にもいろいろ
  - テストの種類ごとの呼び方や目的はチームによって異なるので、認識を揃えることが大事

非機能要件のテスト

- 性能検証、脆弱性検証など
- CI/CD にどのように組み込むかは時と場合による
  - 毎回実行するには長時間になりがち
  - 組み込めるなら組み込んだほうがいい

アーカイブ作成

- デプロイ・リリース時に使用するアーカイブ
- 結合テスト、E2E テスト、非機能要件のテストでも使用する
  - テストに通ったアーカイブをリリースする

デプロイ・リリース

- メインラインにマージされたときだけ実行されることが多い
  - タスクブランチでも動作確認用環境にデプロイとかはある
- ステージング環境
  - 本番環境によく似せたステージング環境でまずデプロイする
- 本番環境へのリリース戦略
  - 万一の問題発生に備えることが重要
    - 一部の環境から広げていく、ロールバック、無停止
    - カナリアリリース、ブルーグリーンデプロイ、フィーチャーフラグ など

その他いろいろ

- コード変更からリリースまでに必要なことはなんでも
- デプロイパイプライン作成後もどんどん変化していく

CI/CD ツール

Jenkins

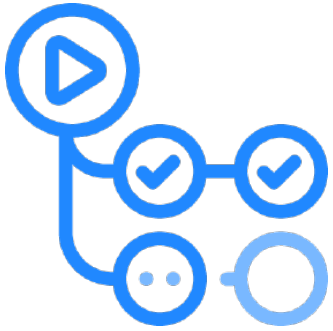
- OSS
- オンプレ構築できる
- コミュニティが大きいのでプラグインが豊富
- 柔軟でやろうと思えばなんでもできる

CircleCI

- クラウドサービス
  - オンプレ版の CircleCI Server もサイボウズで利用してます
- シンプル、導入しやすい
- 認証とか権限とか GitHub と連携してるので導入が楽

GitHub Actions

- GitHub が提供する CI/CD サービス
- パブリックリポジトリでは完全無料
- CI/CD に限らず GitHub の様々なイベントにフックできる



その他の CI/CD ツール

- AWS Code シリーズ
  - CodeBuild, CodeDeploy, CodePipeline, ...
  - AWS と権限周りを統合しやすい
- Kubernetes 用 CD ツール
  - Argo CD など
  - GitOps と呼ばれる手法がよく使われる
    - https://www.weave.works/technologies/gitops/

CI/CD 導入

新規開発への導入

- 最初から CI/CD を導入するのがおすすめ
- 後回しにすると自動化しづらい作りになってしまいがち

既存開発への途中からの導入

- レガシーな部分が原因で難易度が上がりがち
- 手を付けやすく効果の高そうなところから始めるのがおすすめ
  - ビジネス的に重要な部分の正常系テストとか
- いきなり長時間かかるジョブを構築するのはおすすめしない

CI/CD は一日にしてならず

- すべてを一気に導入する必要はない
  - コスパのよさそうなところから徐々に
- 決まった型はない
  - ソフトウェアやチームの性質による
- チームで認識を合わせることも大事
- プロダクトと同じで CI/CD も継続的に改善していく

自動化する時間がない？

- 自動化しないから時間がないのです

うまく回らないとき

- チームで振り返る
- 他のチームの運用を参考にする
- 詳しそうな人に相談する

アンチパターンとベストプラクティス

ローカルで長時間開発しすぎる

- 変更差分が大きくなる
  - 他の開発者の変更と衝突しやすい
  - 壊れやすい
  - 原因究明が困難
- 変更してから問題が見つかるまでのタイムラグが大きくなる
  - 問題に気づくのが遅れるほど対応コストは大きくなる

頻繁に変更をバージョン管理システムにコミットする

- 目安的には全員が 1 日 1 回以上
- コミットが大きくなりすぎないように意味単位で分割する
  - 問題発見やレビューが簡単になる
- タスクも粒度が小さくなるように分割したほうが不確実性が減る

ビルドの実行頻度が低い

- 一日に一回とかしか実行されないケース
- ビルド失敗時にどの変更が原因かわかりにくい
- 変更してから問題が見つかるまでのタイムラグが大きくなる
  - 問題に気づくのが遅れるほど(ry

すべてのコミットでビルドを実行する

- ビルドが失敗したときは直前のコミットが原因の可能性が高い
  - 調査しやすい
- フィードバックが早い
  - 対応コストが小さくなる

ビルドが失敗しても放置される

- 属人的になりがち
- 誰も対応しなくなると CI/CD の利点がすべて失われる

ビルドが失敗したらチームは最優先で復旧する

- ビルド失敗＝リリースできない問題が存在する
- 目安は 10 分以内
  - 直前の変更をリバートするのが一番楽
- ビルド失敗はチームメンバー全員が見てるところに通知する
  - 状態が可視化され、コミュニケーションが円滑になる

Jenkins

kintone-task-branches #241
ビルドに失敗しました
Build Branch: origin/KINTONE-3934
[コンソール出力]

変更履歴:

- [yohei-karikawa] KINTONE-3934 不要なdispatchEventを削除
- [yohei-karikawa] KINTONE-3934 gaia.app.calendar.Menuのメソッド名を一部変更、不要になった関数削除
- [yohei-karikawa] KINTONE-3934 CalendarとTableとで重複しているupdate処理を削除。
- [yohei-karikawa] KINTONE-3934 tooltipのclassNameをrevert
- [yohei-karikawa] KINTONE-3934 添付ファイル、リッチエディタ、ユーザフィールド実装
- [yohei-karikawa] KINTONE-3934 不正なタイプのフィールドをサーバ側で弾くようにした
- [yohei-karikawa] KINTONE-3934 listスタイルの微調整

あやしいひとたち:
@刈川 陽平

10/26, 15:46 ♥4 *いいね！* *返信*

刈川 陽平
くぅ
10/26, 16:10 *いいね！* *返信*

天野 祐介
@刈川 陽平 body.jsでgjslintエラーになるのも直してー！
10/26, 16:12 *いいね！* *返信*

刈川 陽平
@天野 祐介 あい！
10/26, 16:14 *いいね！* *返信*

https://blog.cybozu.io/entry/2386

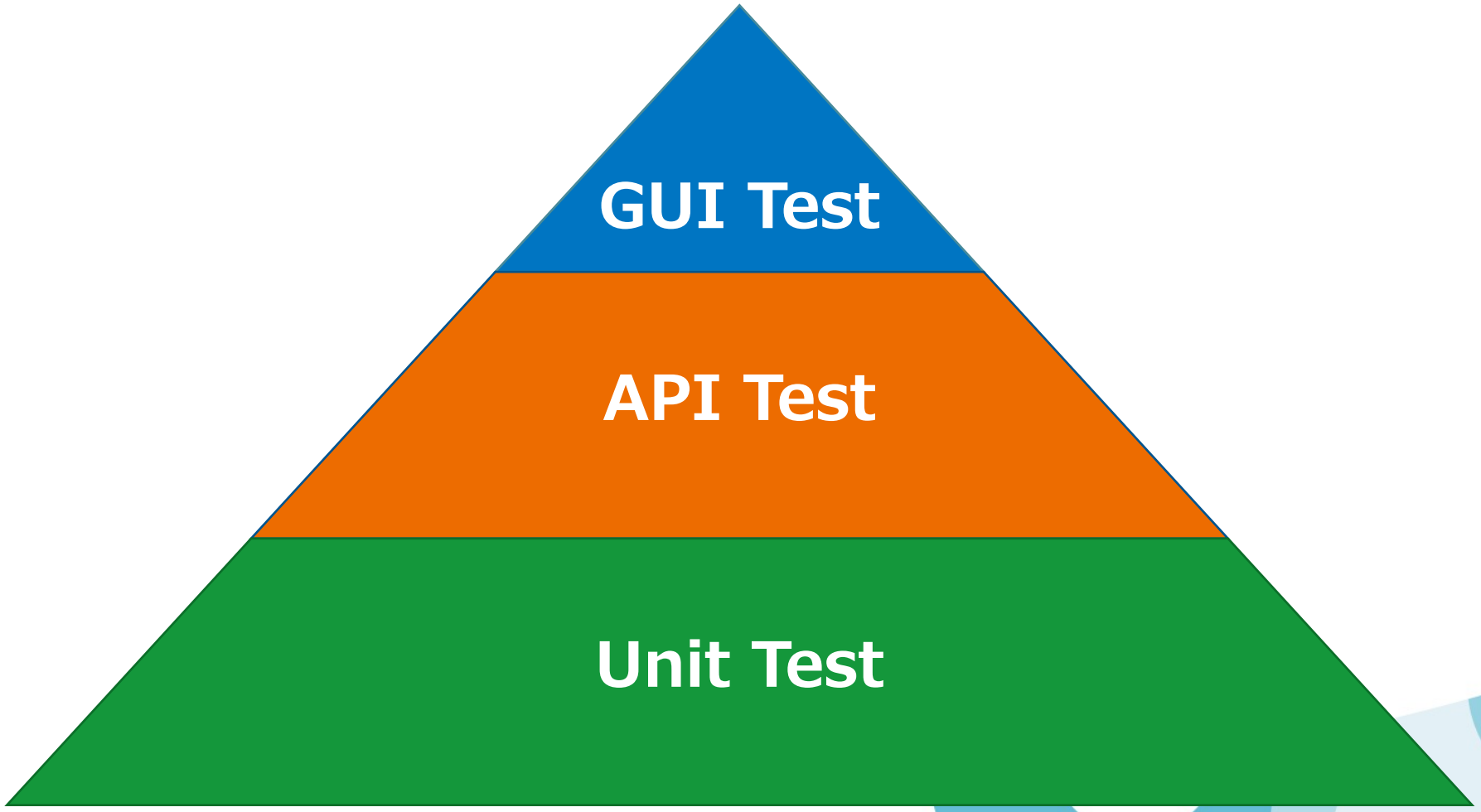
CI/CD のビルド時間が長すぎる

- 結合テストや受け入れテストを厚くしすぎるとなりがち
- 1 時間以上とかになってくると厳しい
  - 失敗時に再実行することとかを考えると辛い

CI/CD を高速に保つ

- 可能な限り並列実行する
- 自動テストの役割を継続的に見直す
  - テストピラミッドを意識する

テストピラミッド



テストピラミッドのポイント

- いろいろな粒度のテストを組み合わせる
- 粒度が大きくなるほど実行時間やメンテナンスコストが高くなる
- より小さい粒度のテストで防げるものは防ぐ

不安定なビルド

- 不具合ではないのにビルドが失敗する
  - CI/CD の信頼性が下がる
- 原因はいろいろ
  - 本番コードではないので書かれる手抜きスクリプト
  - E2E テストの微妙なタイミングのズレ
  - 不安定な環境
    - 構築手順が微妙に異なる、前のビルドのゴミが残ってる、など

ビルドを継続的に改善して品質を高める

- ビルドで実行されるタスクは製品コードレベルの品質を目指す
  - 特にメンテナンス性を高めることが大事
- ビルド結果を計測する
  - ビルドの失敗頻度やその原因を振り返れるようにしておくとあとから改善しやすい
- 防ぎづらいレアケースもあるので自動リトライも一つの手段
- 環境は仮想化して毎回クリーンにする
  - Docker コンテナ内でビルドするのが最近は一般的

まとめ

意識してほしいこと

- 高速なフィードバックループは不確実性を下げ、学びを最大化する
  - 顧客へ提供する価値の最大化につながる
  - 例えば Amazon では毎秒なにかしら本番環境にデプロイしてる
- ボトルネックを意識してバランス感覚を持って自動化する
- リリースしてようやく顧客に価値を提供できる
  - コードを変更して終わりではない
  - チーム全体でリリースやその後のフィードバックまで責任を持つ

参考文献

- 『継続的インテグレーション入門』
- 『継続的デリバリー』