

# いつやるか？ DevOps

---

**TIS株式会社**

テクノロジー&イノベーション本部

テクノロジー&エンジニアリングセンター



## 【前提】

ビジネスの価値創造のための**具体的な手段は、プロジェクトによって変わります**

(=プロジェクトによって使用すべきツール、対処すべき課題の優先度などが異なる)

## 【本コンテンツの目的】

- **サービスのデリバリ能力を高めるために必要な土台**が何かを知ること
- DevOpsの考え方を知り、**自分たちで活動できる状態**になること

## 【本コンテンツでは触れないこと】

- 具体的なツール
- DevOpsにおける実装方法

- 大きく2部構成
  - DevOpsについて
  - DevOpsを支える「3つの原則」について  
各原則ごとに以下を説明
    - どのような原則か
    - それに沿った考え方や活動にはどのようなものがあるか

DevOpsって、  
どのようなものか説明できますか？

- DevOpsに取り組もうとしたことはありますか？
- どのような取り組みをしましたか？

- CI/CDの導入だけでは**DevOpsは達成できない**
- DevOpsを理解し、これから何をしないといけないのかを一緒に学びましょう

## DevOpsについて

---

- なぜDevとOpsが対立するのか
- DevOpsに取り組む理由
- DevOpsの第一歩



- 本コンテンツでは、以下の意味で「Dev」と「Ops」という言葉を使用しています。
  - Dev :  
開発チーム。プログラミングだけでなく、構築するシステムやサービスの企画、設計などを担当する人々も含む。
  - Ops :  
運用チーム。日々運用を行っている人々だけでなく、インフラ担当者も含む。

## DevとOpsの一般的な組織目標の違い

---

Devの立場からOpsに対して・・・

新機能追加のために  
サーバを増やしたいから  
インフラの準備お願い

インフラのテストやデプロイ検証が  
必要だから2か月ほど待ってね

早くリリースしたいのに  
対応が遅すぎる...



Dev（開発チーム）



Ops（運用チーム）

Opsの立場からDevに対して・・・

お客さんから動かないって  
クレーム来てるよ

新しい機能を開発中だから  
ちょっと待ってよ

責められるのは俺たちなんだから、  
安定稼働できるようにしてよ...



Dev（開発チーム）



Ops（運用チーム）

## 開発チームと運用チームの典型的な方向性の違い

**Dev**  
(開発チーム)

ビジネスのために  
サービスに変化を加えリリースしたい

新しい機能をユーザに届けるために、素早く**変更**をしたい

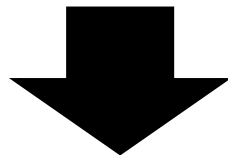


**Ops**  
(運用チーム)

ビジネスのために  
システムを安定稼働させたい

システムの安定を図るために、**変化を加えたくない**

**DevとOpsが組織的に分かれている**



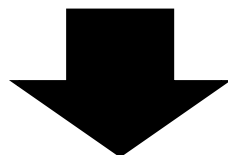
**組織のミッションが異なる**

**Dev**

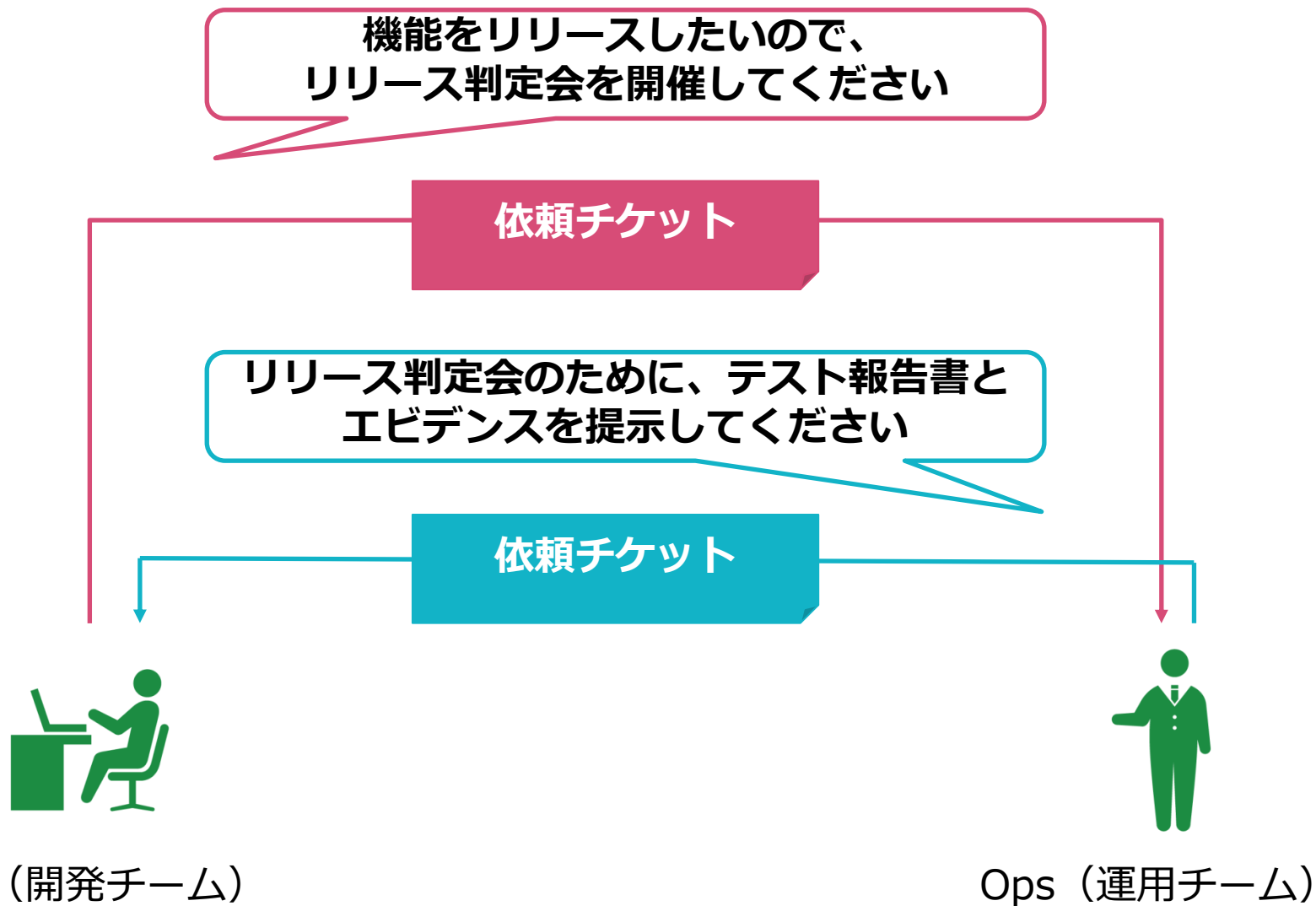
**Ops**

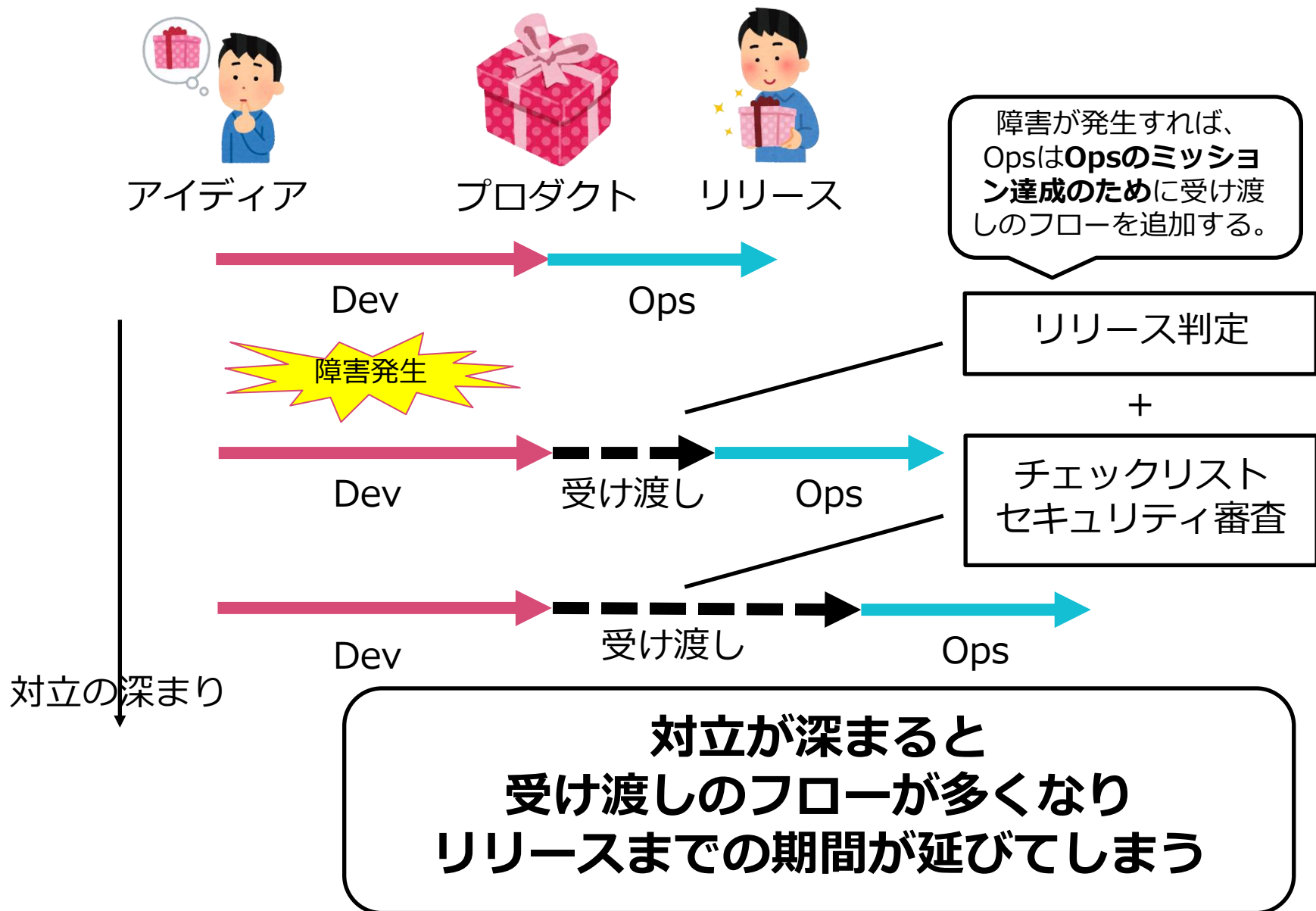
新しい機能を開発して  
ビジネスを成功に導く

安定したシステム稼働で  
ビジネスを成功に導く

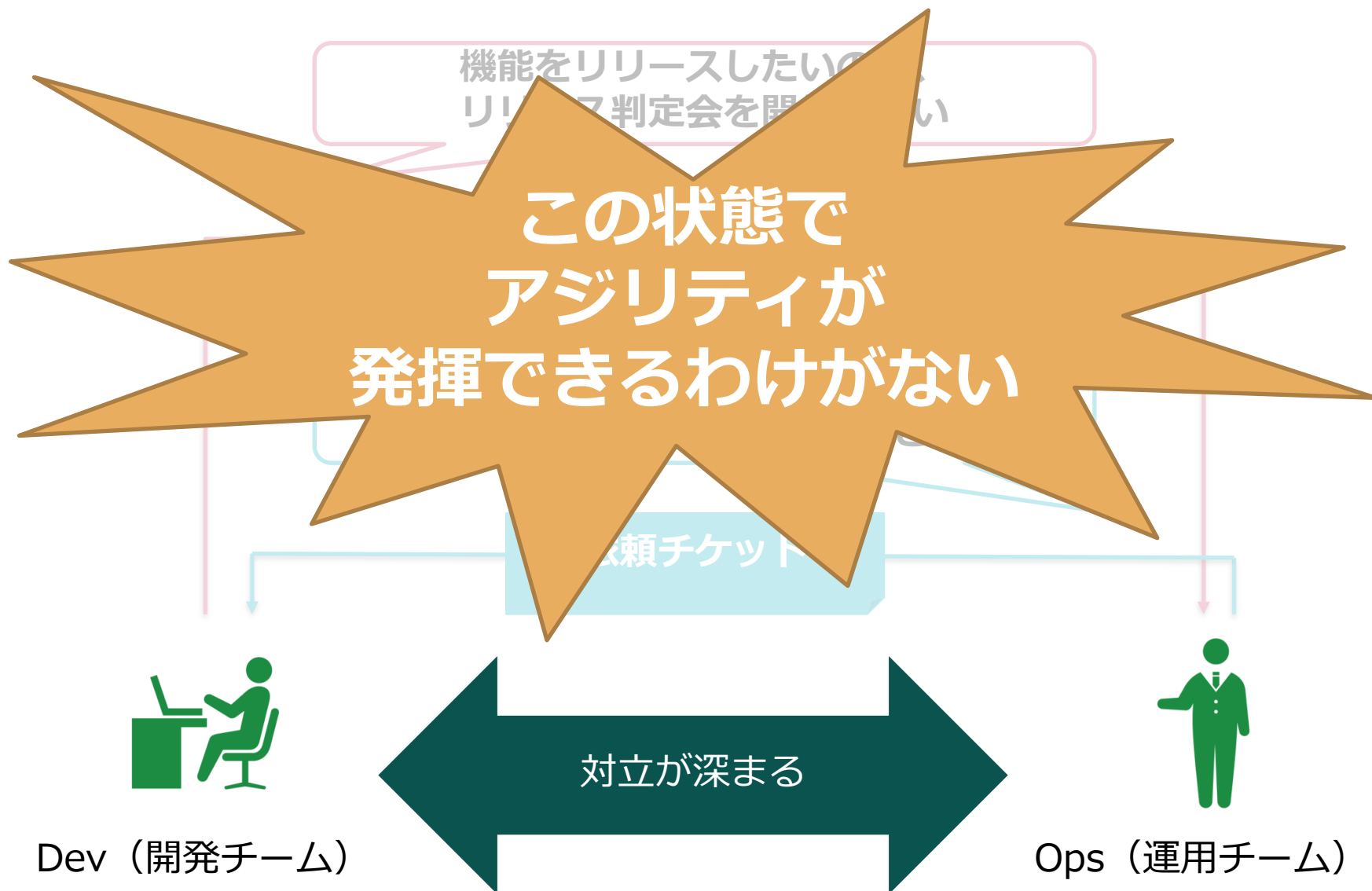


**DevとOpsの対立**





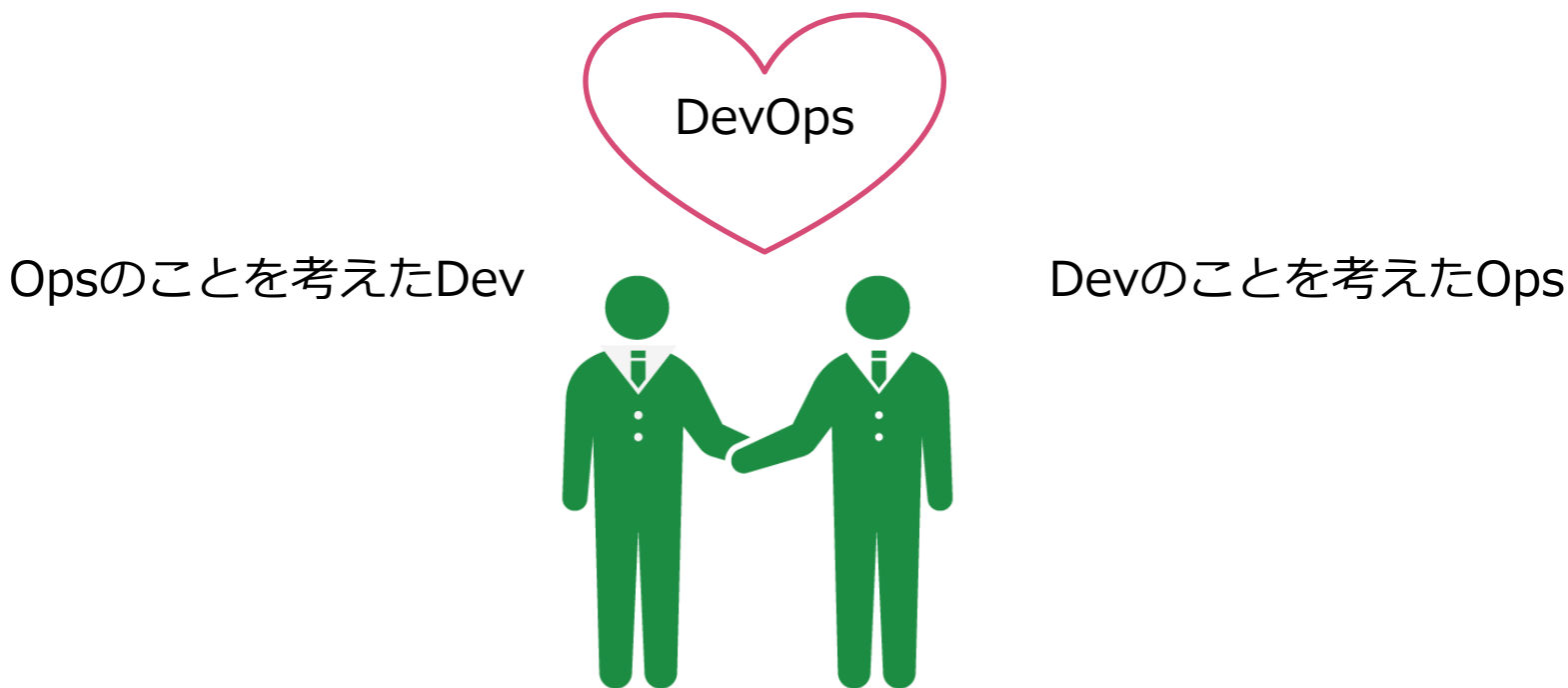




**高いサービス価値を  
素早く・継続的に顧客に届けたい**

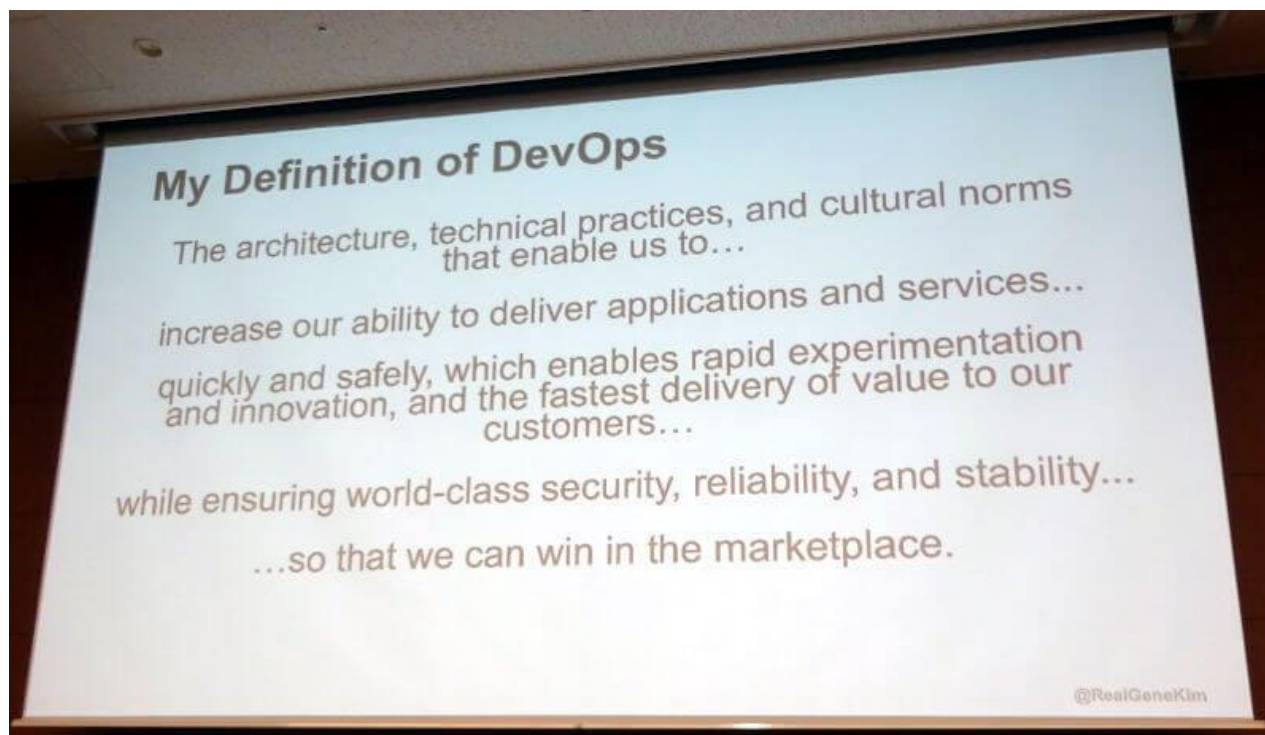
DevもOpsも「ビジネスを成功に導く」という目的は同じ。  
ただ、手段や考え方が異なっていた。

# DevとOpsが手を取り合う事で、 ビジネスを成功に導く



# DevOpsとは何か

---



「アーキテクチャーであり、技術的な実践であり、文化的な常識」であり、それが  
「アプリケーションとサービスを実装する能力を高める」ことによって  
「**新たな実験や革新を素早く安全に顧客に対して価値にすること**」でありながらも  
「**同時に世界トップレベルの安全性、信頼性、安定を兼ね備える**」ことによって

**「市場で勝ち抜く」ことを実現すること**

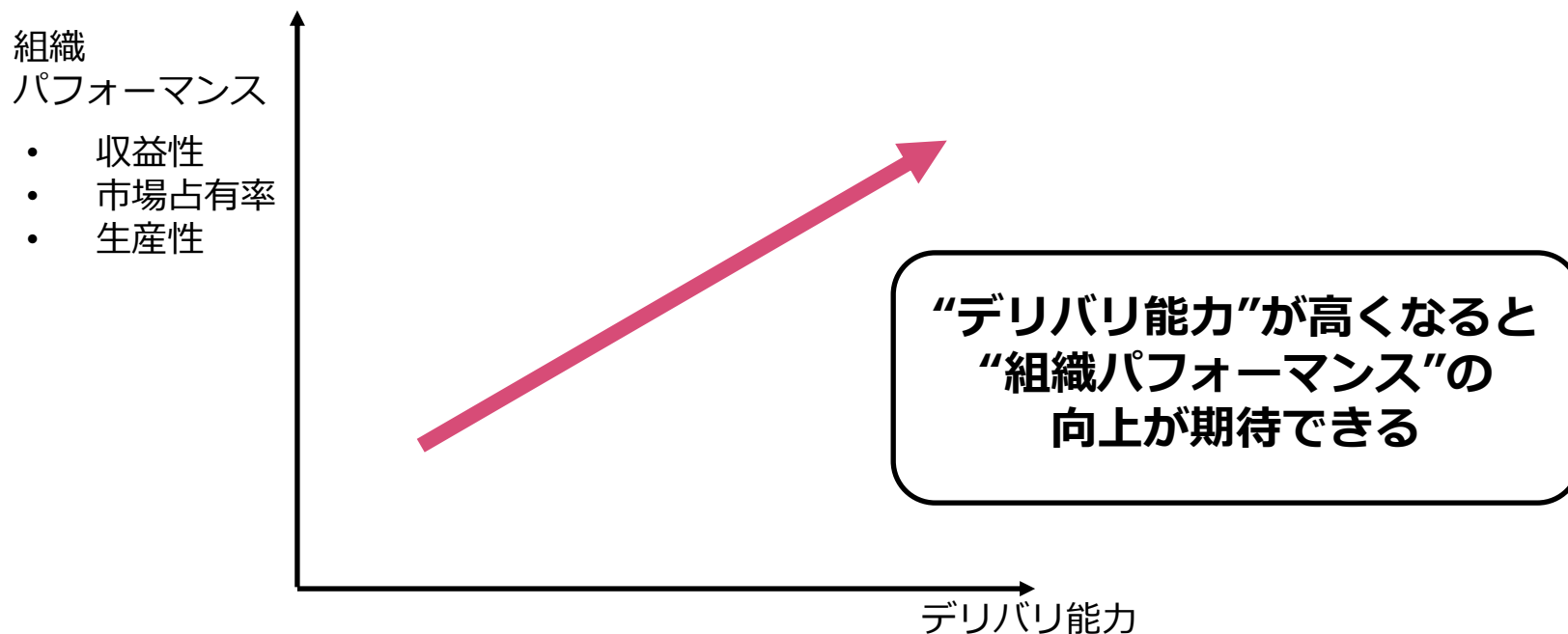
DevOpsDays Tokyo 2019開催。  
DevOpsの伝道者Gene Kimのキーノートセッションを紹介  
(<https://thinkit.co.jp/article/16052>)  
2020年8月23日15時の最新情報を取得

“組織パフォーマンス”と“ソフトウェアデリバリ能力”の間には  
明確な関係性がある

*Four key metrics*  
(<https://www.thoughtworks.com/radar/techniques/four-key-metrics>)  
2020年8月23日16時の最新情報を取得

“ハイパーフォーマーのこうした基準でのパフォーマンスの測定結果は  
ローパーフォーマーの（略）**2倍を超えている**”

*Nicole Forsgren Ph.D, Gene Kim, Jez Humble*  
「LeanとDevOpsの科学 [Accelerate] テクノロジーの戦略的活用が組織変革を加速する」  
インプレスブックス



# Four Key Metrics

*Four key metrics*  
(<https://www.thoughtworks.com/radar/techniques/four-key-metrics>)  
2020年8月23日16時の最新情報を取得

メトリクス	定義	Elite Performer	Low Performer
リードタイム	Commitされたコードが本番環境にデプロイされるまでの時間	1日～1週間	1か月～半年
デプロイ頻度	本番環境へのデプロイ頻度	1日複数回	月1回 ～半年に1回
MTTR	エンドユーザに影響する障害が発生してから回復するまでの時間	1時間以内	1週間 ～1か月
変更失敗率	リリースが失敗(あとで修正が必要になる等)する割合	15%以下	46～60%

6年間のデータ分析の結果、これらのメトリクスによって  
「組織のパフォーマンスを予測することができる」ことが示されている

## ELITE PERFORMERS

Comparing the elite group against the low performers, we find that elite performers have...

### デプロイ頻度



208

TIMES MORE

frequent code deployments

### リードタイム

106

TIMES FASTER

lead time from  
commit to deploy



### MTTR



2,604

TIMES FASTER

time to recover from incidents

### 変更失敗率

7

TIMES LOWER

change failure rate  
(changes are  $\frac{1}{7}$  as likely to fail)



Throughput Stability

State of Devops 2019

(<https://www.devops-research.com/research.html>)

2020年9月15日14時の最新情報を取得

DevOpsで、このような状態を目指していきたい



## DevOpsに向けた変化の第一歩

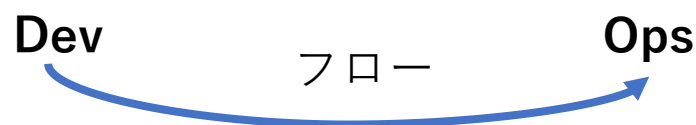
---

## DevOpsのすべての行動やパターンの基礎になっている原則

### 第1の道：フローの原則

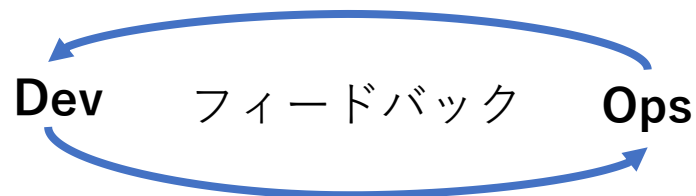
開発から顧客へ価値を届けるまでの  
フロー全体の高速化

(Business) (Customer)



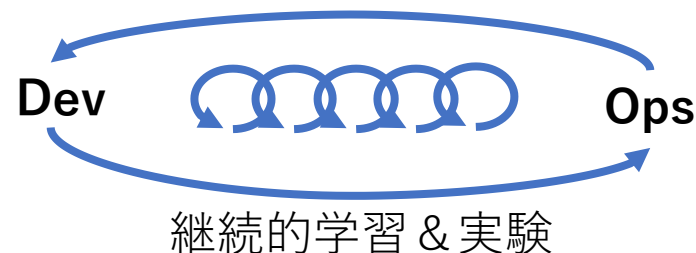
### 第2の道：フィードバックの原則

あらゆるステージにおける**素早く**  
**頻繁なフィードバック**の実現



### 第3の道：継続的学習&実験の原則

科学的な実験・リスクを取る  
ことを支持し、組織として学習する

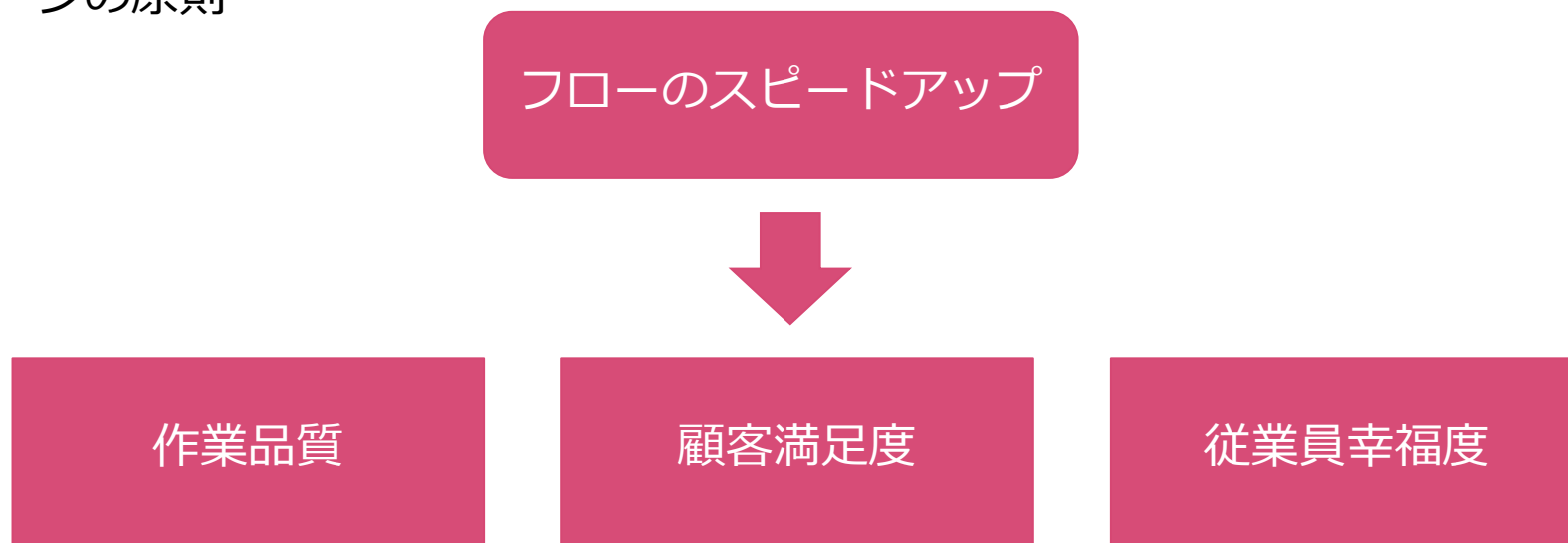


ジーン・キム、ジェズ・ハンブル、パトリック・ドボア、ジョン・ウィリス  
「The DevOps ハンドブック 理論・原則・実践のすべて」  
日経BP社  
をもとに作成

“私たちの目標は、**変更を本番環境にデプロイするまでに必要な時間を短縮し、サービスの信頼性と品質を高めること**である”

ジーン・キム、ジェズ・ハングル、パトリック・ドボア、ジョン・ウィリス

## リーンの原則

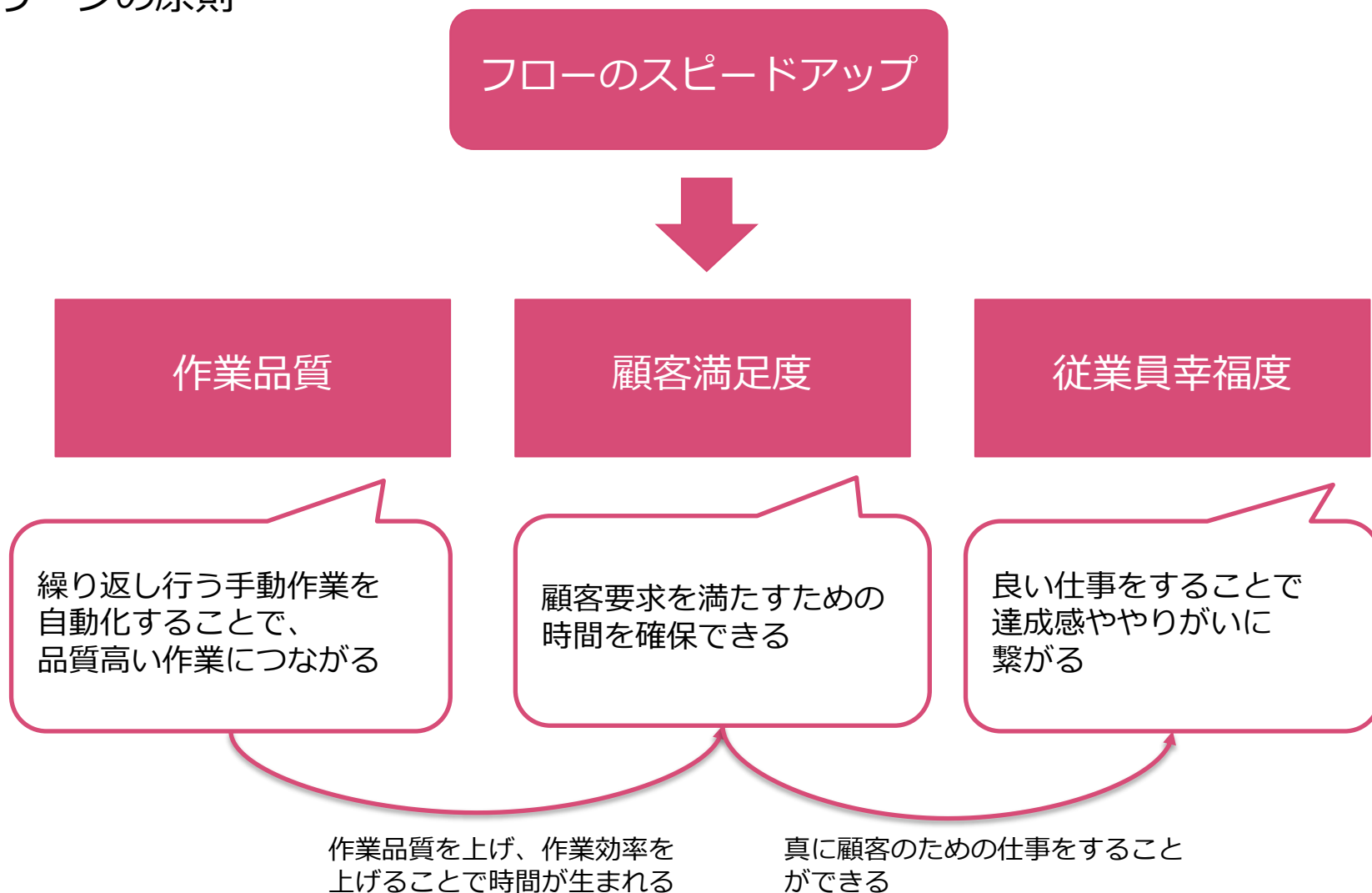


“リーンには2つの大きな原則がある。

- 1つは、**品質、顧客満足度、従業員幸福度を引き上げるためには、原材料を最終製品に変えるまでの製造リードタイムを短縮することがもっとも大切だ**という固い信念であり、
- もう1つはリードタイムを短縮するためには、仕事のバッチサイズを小さくすることが大切だという信念である。”

ジーン・キム、ジェズ・ハンプル、パトリック・ドボア、ジョン・ウィリス  
「The DevOps ハンドブック 理論・原則・実践のすべて」  
日経BP社

## リーンの原則



- ITバリューストリーム

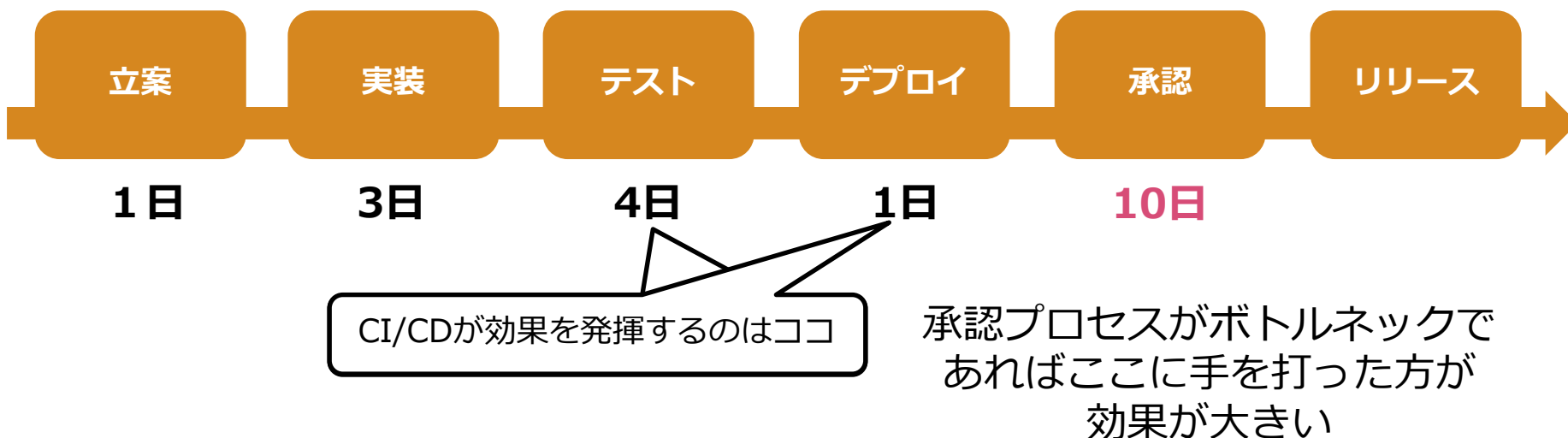
「ビジネス上の仮説を立案してから、顧客に価値を送り届ける技術サービスを生み出すまでの間に必要なプロセス」

ジーン・キム、ジェズ・ハンブル、パトリック・ドボア、ジョン・ウィリス  
「The DevOps ハンドブック 理論・原則・実践のすべて」  
日経BP社

**このプロセスを高速化することで  
顧客に素早く価値を届けたい**

## 目指すのは局所的な最適化ではなく**全体最適化**

- ボトルネックになっていない箇所を高速化しても効果薄
- ボトルネックが不明な状態でCI/CDを導入して、どこまでフロー全体が高速化されるのか

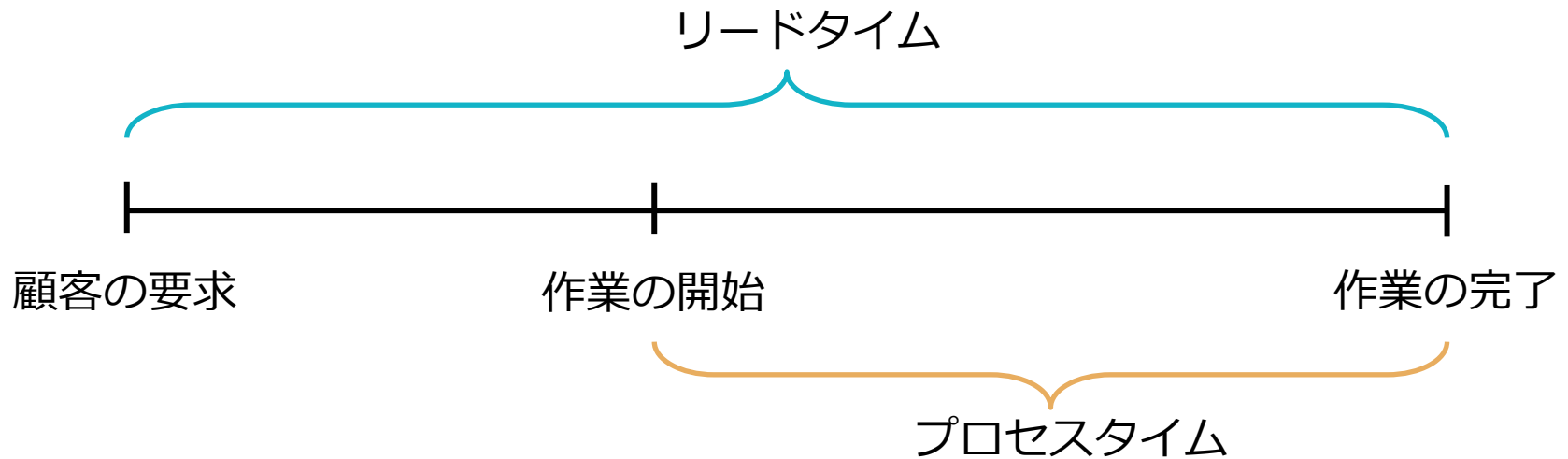


- リードタイム
- プロセスタイム
- 手戻り指標（完全正確率, %C/A）



- リードタイム (LT)
  - 「顧客が要求をしたときからその要求が満たされるまでの間」
- プロセスタイム (PT)
  - 「顧客の要求のための仕事を始めた時からその要求が満たされるまでの間」

顧客が要求をしたときから  
作業を始めるまでの時間ではない



プロセスタイムの短縮に  
意識が向くことが多い

プロセスタイム

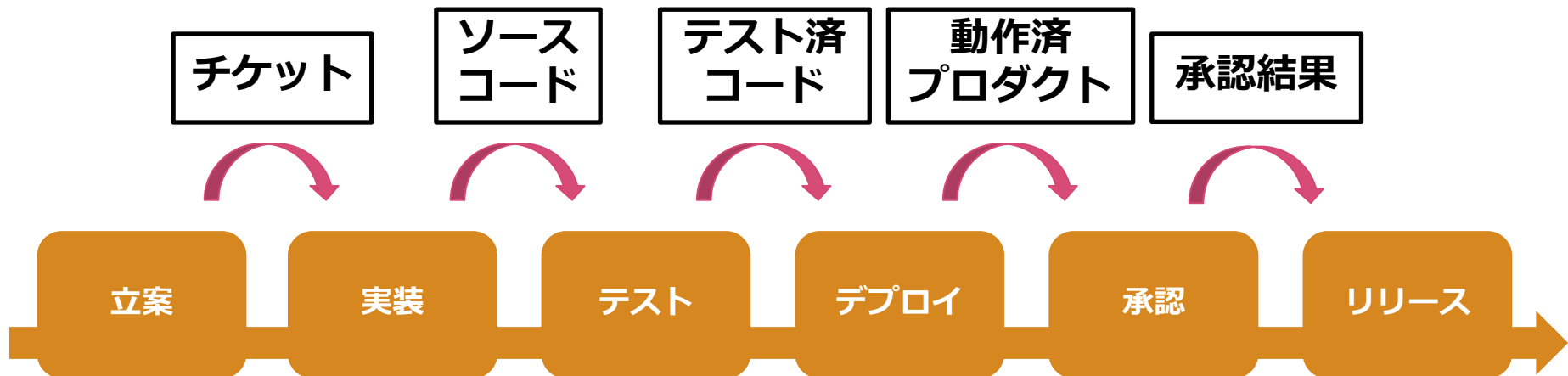


顧客が実際に体感する時間

リードタイム全体を総合的に改善する

- 前のプロセスで出力された成果物が、次のプロセスで**そのまま利用できる**割合

= 前プロセスで作成された成果物の**品質指標**

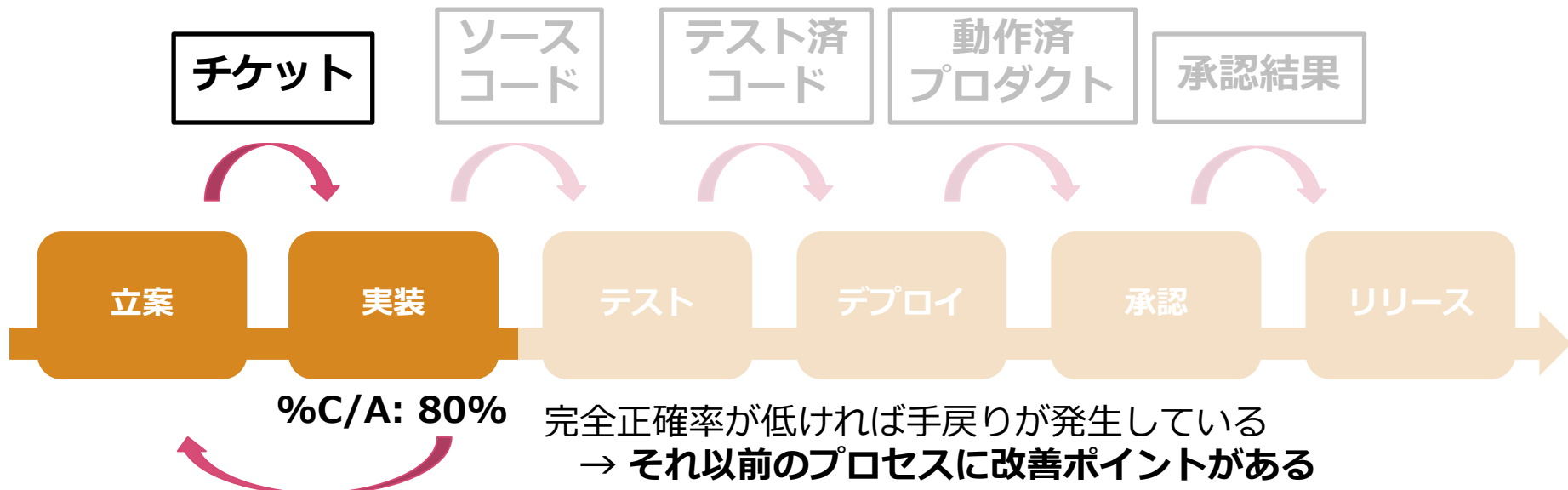


渡されたチケットが5つあるけど  
1つは具体性に欠けているので  
そのまま実装に移れないな…



→ 1つのチケットは次のプロセスでそのまま利用できる  
成果物ではない

= 完全正確率は**80%**



## 第1の道：フローの原則

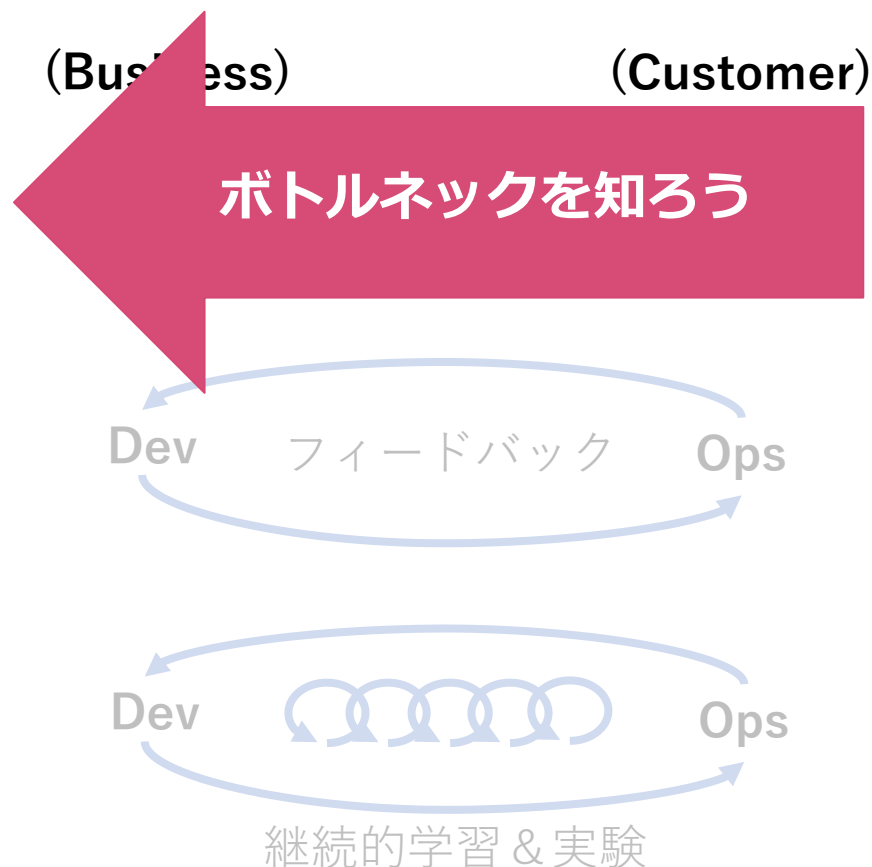
開発から顧客へ価値を届けるまでの  
フロー全体の高速化

## 第2の道：フィードバックの原則

あらゆるステージにおける素早く  
頻繁なフィードバックの実現

## 第3の道：継続的学習&実験の原則

科学的な実験・リスクを取る  
ことを支持し、組織として学習する



ジーン・キム、ジェズ・ハンプル、パトリック・ドボア、ジョン・ウィリス  
「The DevOps ハンドブック 理論・原則・実践のすべて」  
日経BP社  
をもとに作成

- **価値がどのようにして顧客に届けられるかの理解・見える化**
  - 誰が
  - どのような作業をして
  - どのようなステップを踏めば改善できるのか

見える化するためのプロセス：  
**バリューストリームマッピング**

## バリューストリームの可視化

---

「製品が流れるバリューストリームに沿って、  
**モノと情報の流れ**を見て理解することを助けるもの」

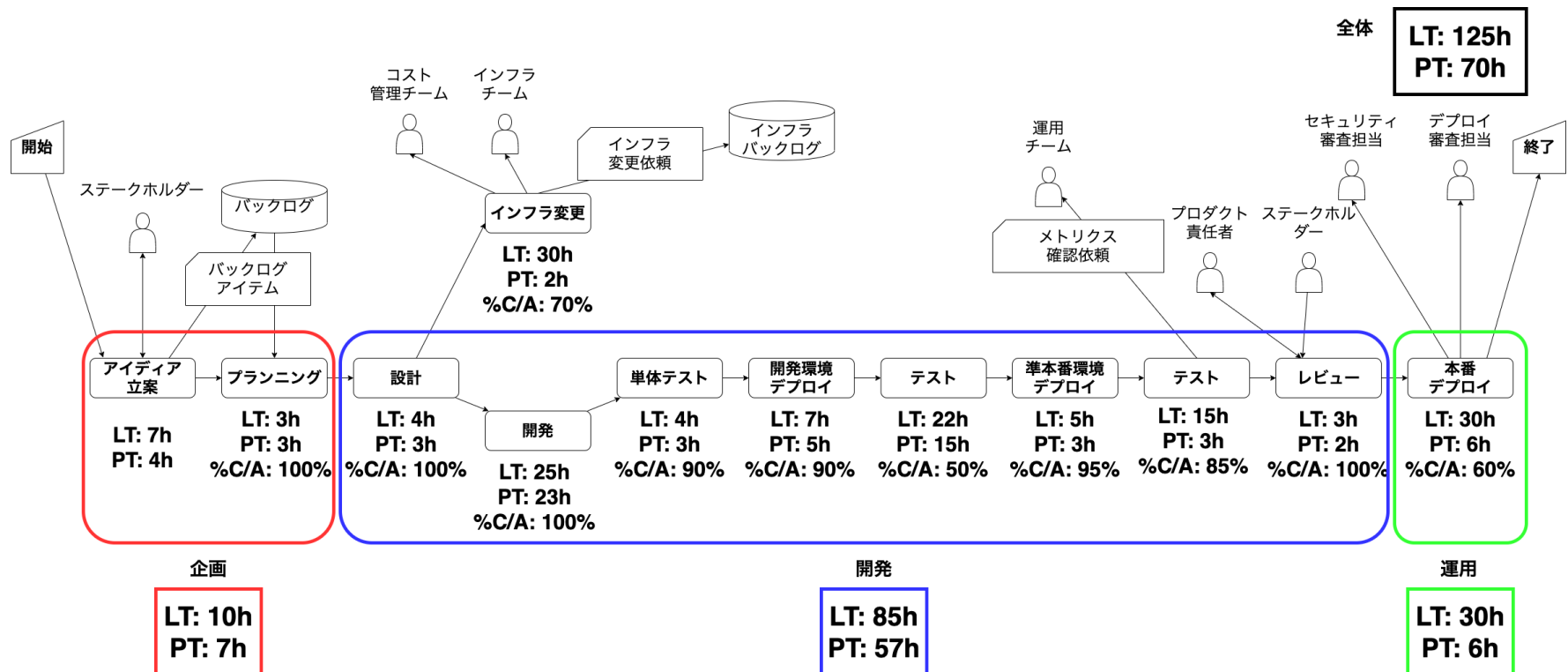
マイク・ローザー、ジョン・シュック  
「トヨタ生産方式にもとづく「モノ」と「情報」の流れ図で現場の見方を変えよう」  
日刊工業新聞社

- ・ 現在のプロセスのボトルネックの見える化
- ・ チーム内でプロセスの課題点を共有

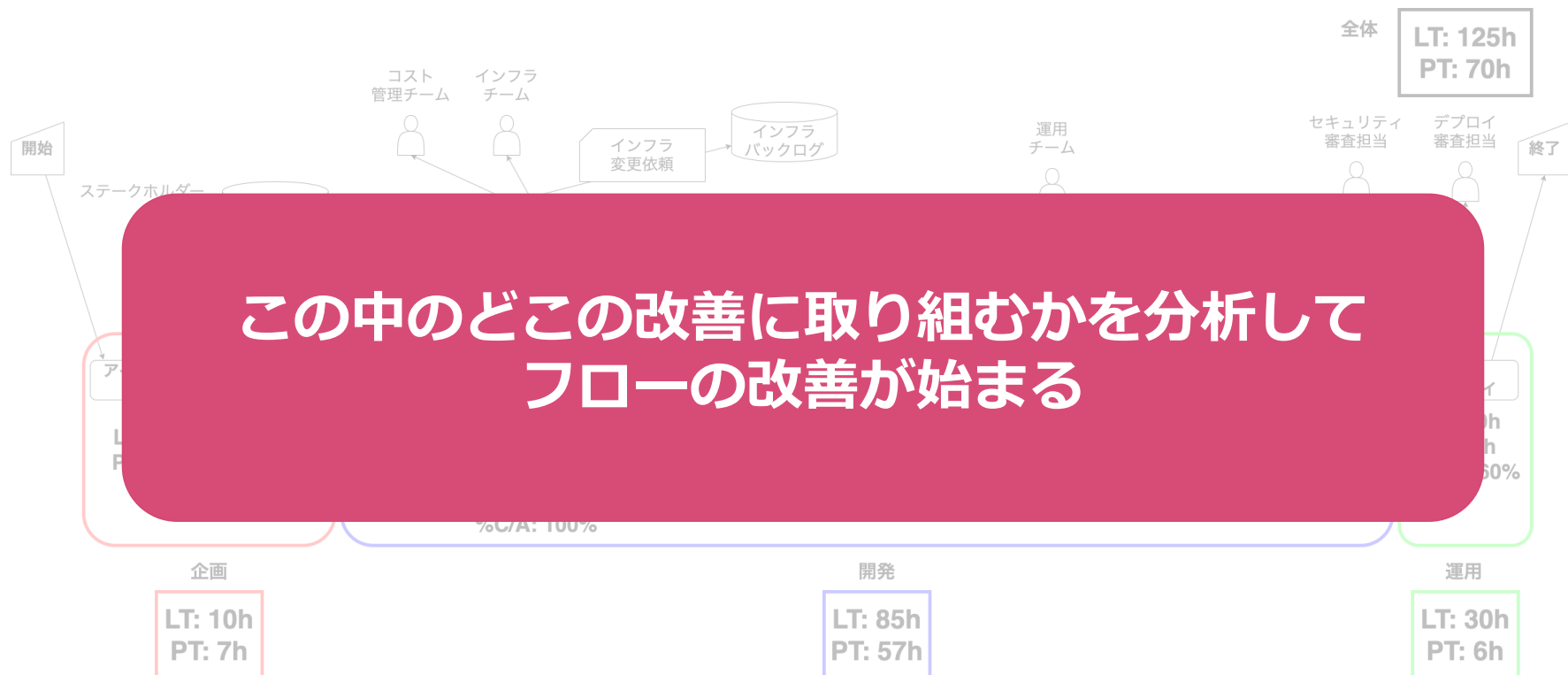
プロセス全体の改善へつなげる



# これがバリューストリームマップだ



# これがバリューストリームマップだ



- 顧客に価値を届けるバリューストリームを定義する
  - チームが実施しているプロセスが、誰にどのような価値を届けているのかを明確にする
- フローを5~15のプロセスに分解する
  - ステップが細かすぎると、ボトルネックが見えてこない
  - 「成果物の受け渡し」や「停滞すると影響がある」単位でプロセスを分割する
- 全プロセスの関係者を集める
  - プロセスの可視化と共有のため、可能な限り関係者を集める
    - できるだけ決定権を持つ上層部の人を集める

このプロセスって本当に必要なの？

〇〇のために必要だね。

もっと短い△△で代替できない？

それなら代替可能かも…



偉い人 1



偉い人 2

**改善のフローをはやく回せる**

- なぜDevとOpsが対立するのか
- DevOpsに取り組む理由
  - サービスの価値を、顧客に**素早く・継続的に**届けたい
  - **ビジネスを成功に導きたい**
- DevOpsの第一歩として、まずはボトルネックを理解する
  - **バリューチェーン**の全体像を理解するため  
**バリューチェーンマッピング**を実施する

## フローの原則における改善方法

---

“私たちの目標は、**変更を本番環境にデプロイするまでに必要な時間を短縮し、サービスの信頼性と品質を高めることである**”

ジーン・キム、ジェズ・ハングル、パトリック・ドボア、ジョン・ウィリス  
「*The DevOps* ハンドブック 理論・原則・実践のすべて」  
日経BP社

## 1. 基本的な考え方

1. 作業の可視化
2. WIP（仕掛かり）の制限
3. バッチサイズの縮小
4. 受け渡し数の削減
5. 絶えず制約条件を見つけ出して尊重する

## 2. 実践

1. オンデマンドの開発環境
2. 継続的デプロイメント
3. 継続的インテグレーション



ITの作業は  
目に見えにくい

作業の受け渡しは非常に簡単  
チケットをクリックするだけ

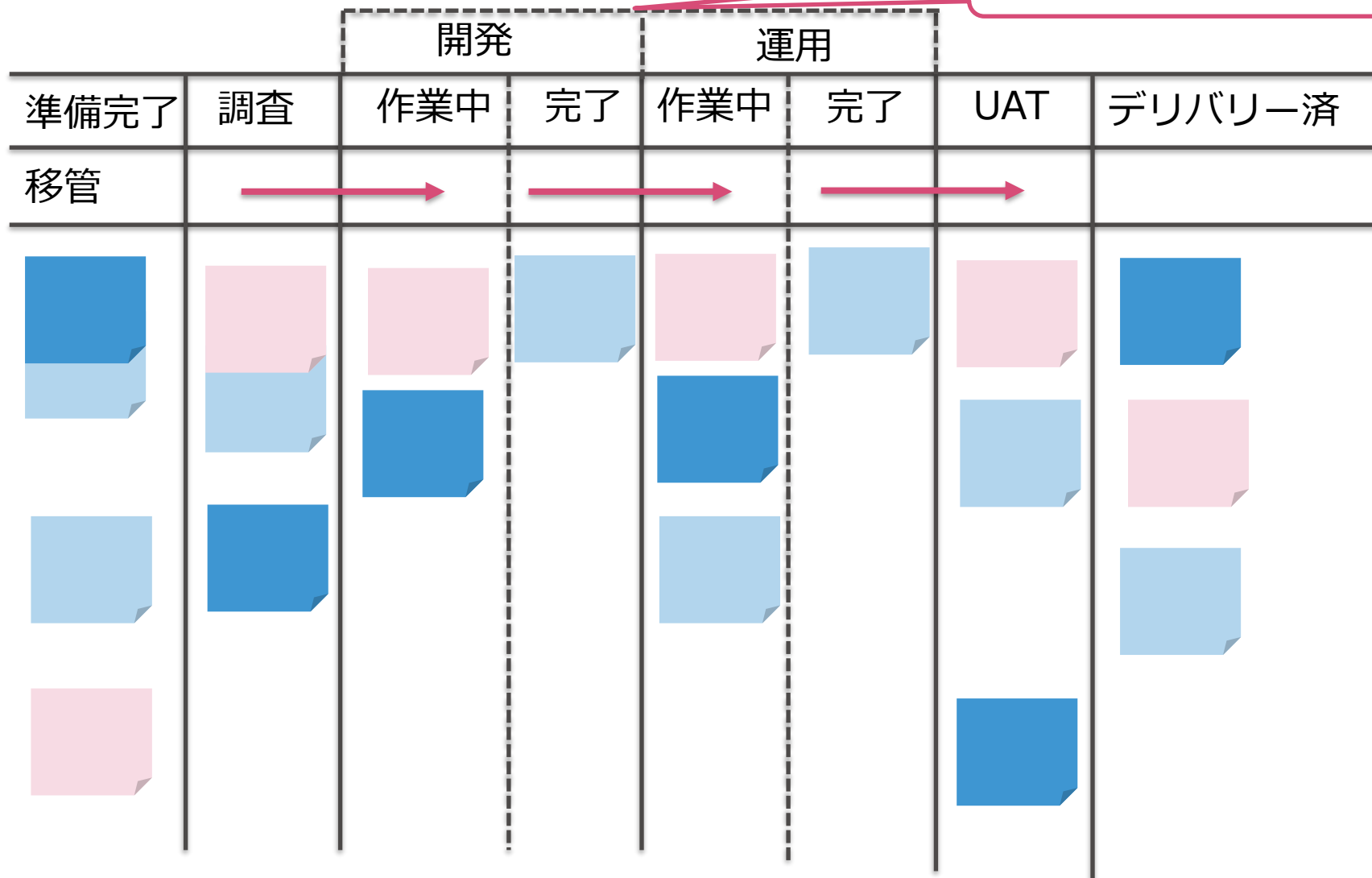


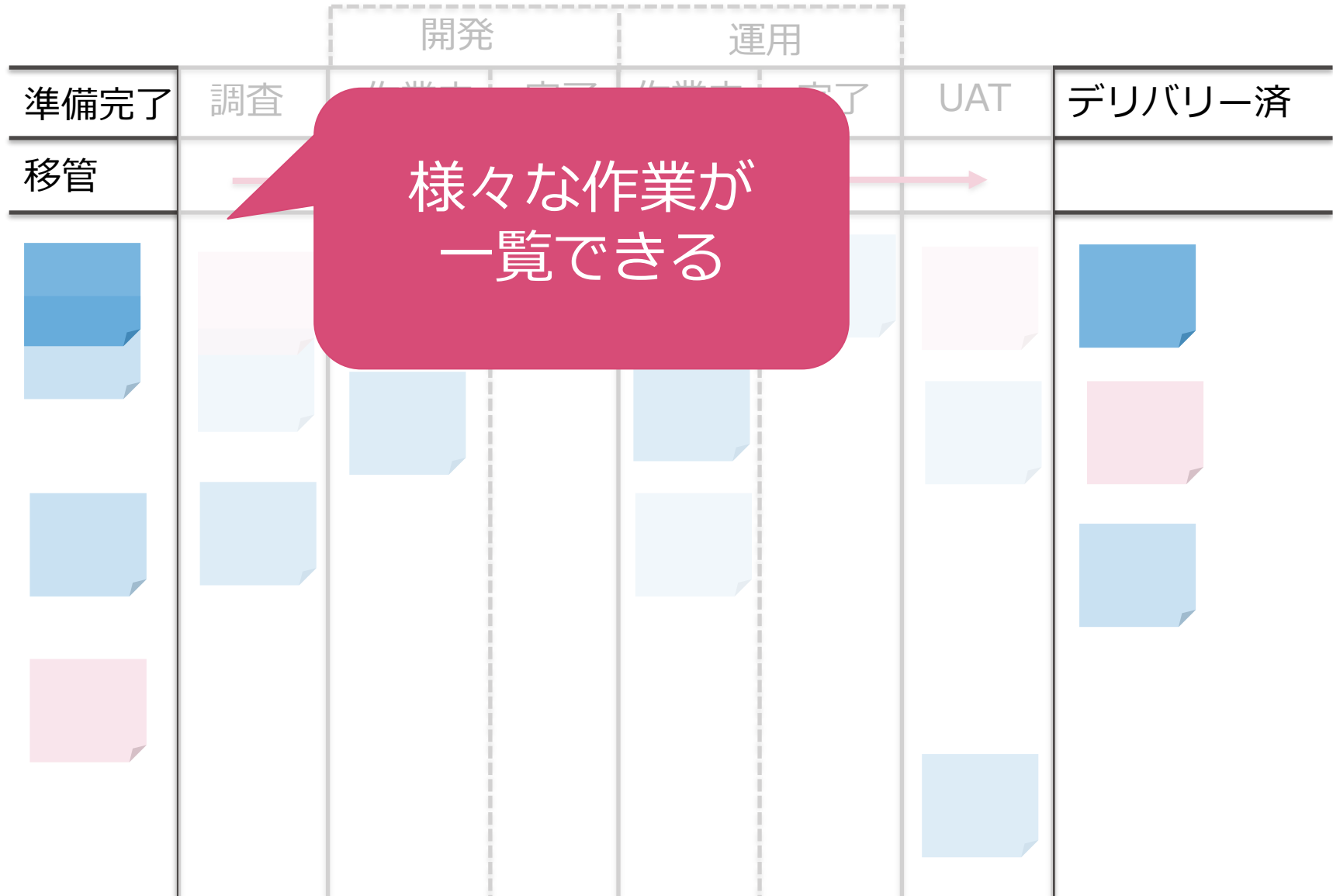
問題が潜んだまま次のチームに作業を流してしまう  
やり直しの多発、本番障害を引き起こす

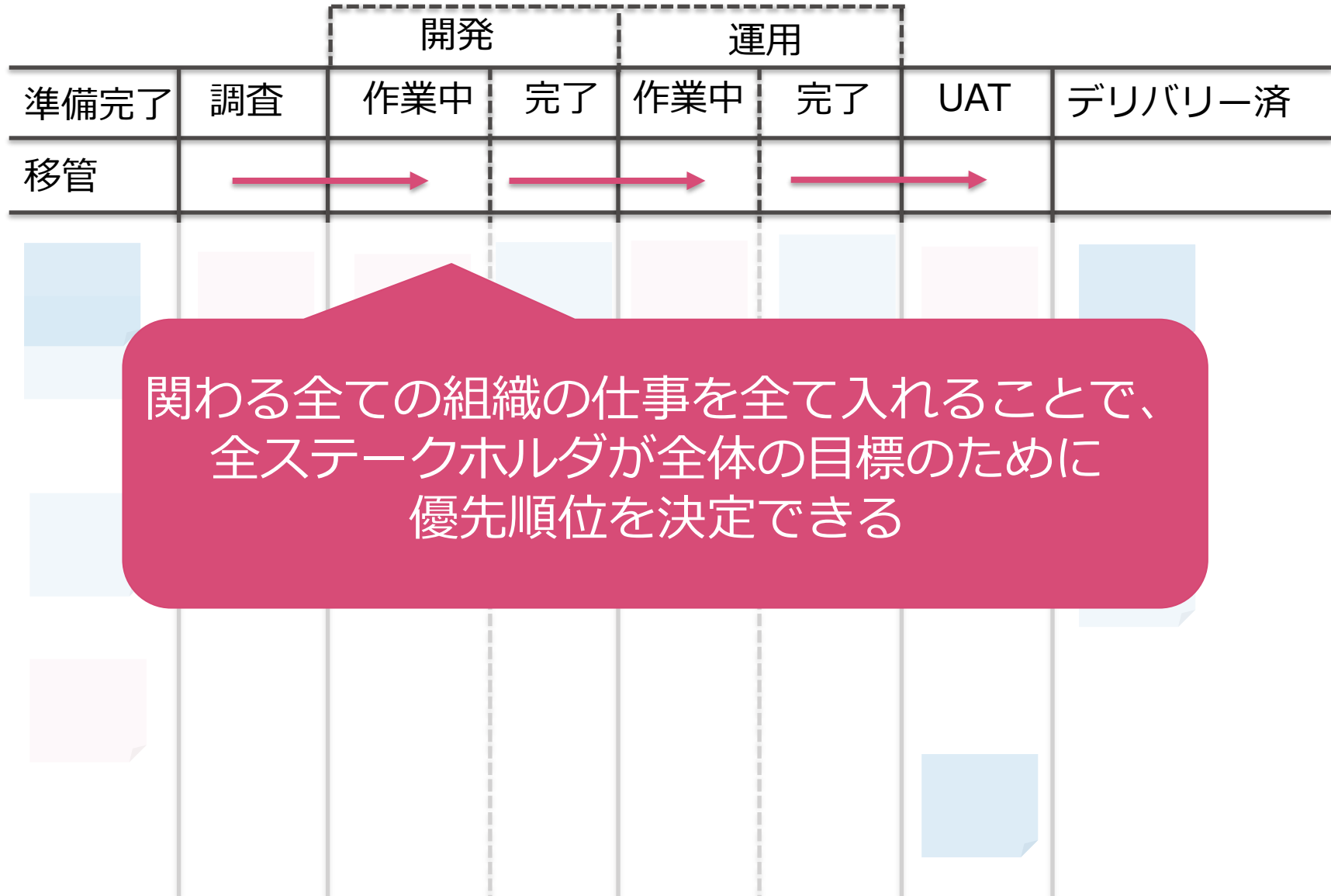
**作業状況や問題を少しでも把握しやすくするために  
できる限り作業の状況を可視化したい**

## 「カンバン」の導入

関係組織全ての作業を  
1つのカンバンで可視化する







## 1. 基本的な考え方

1. 作業の可視化
2. WIP（仕掛かり）の制限
3. バッチサイズの縮小
4. 受け渡し数の削減
5. 絶えず制約条件を見つけ出して尊重する

## 2. 実践

1. オンデマンドの開発環境
2. 継続的デプロイメント
3. 継続的インテグレーション

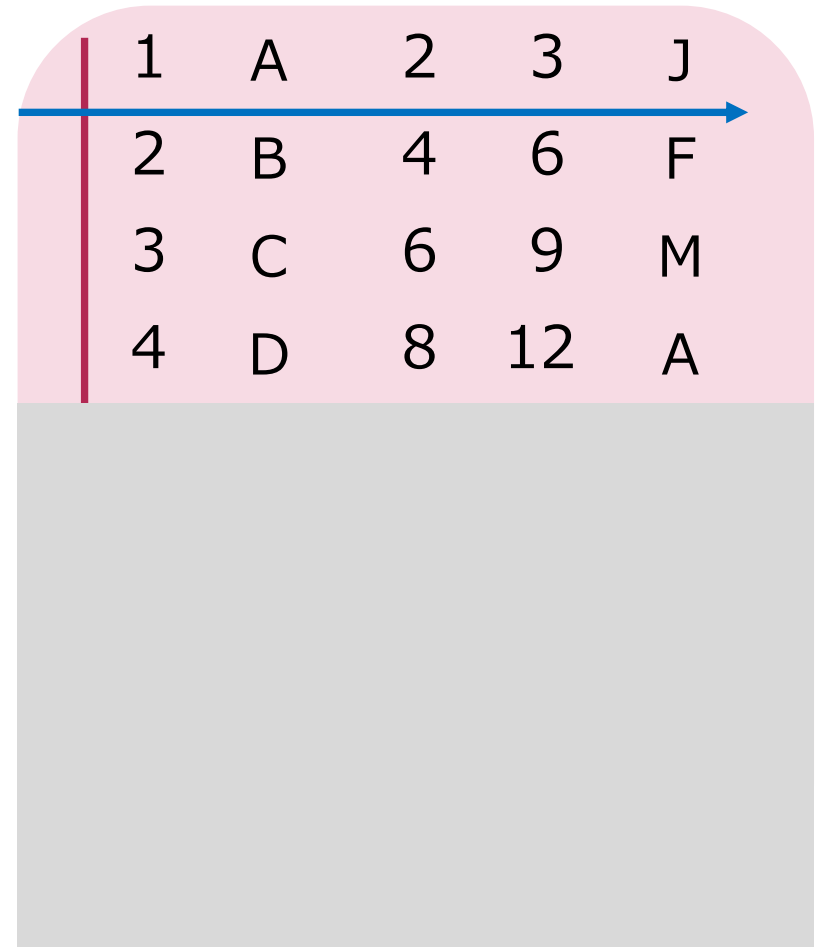
頭を切り替え（コンテキストスイッチ）することは**難しく、効率が悪い**

同時 プロジェクト数	プロジェクト1つに 使える時間の割合	コンテキストスイッチ ングの割合
1	100%	0%
2	40%	20%
3	20%	40%
4	10%	60%
5	5%	75%

*Jeff Sutherland, J.J. Sutherland*  
「SCRUM: The Art of Doing Twice the Work in Half the Time」参考

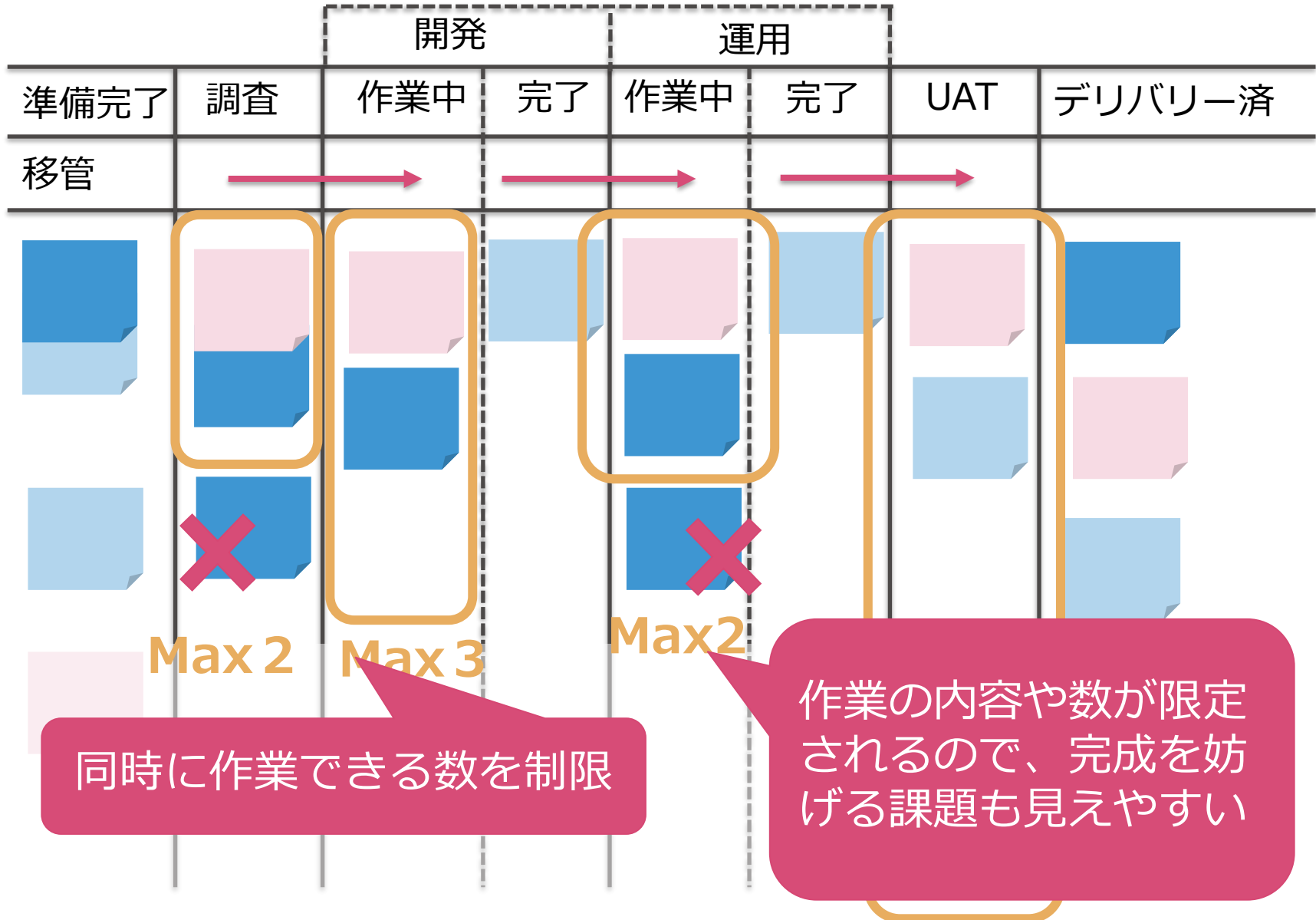
- 1列目は1から9
  - 2列目はAからI
  - 3列目は2から2刻み
  - 4列目は3から3刻み
  - 5列目は月の英語の最初の文字
- 
- 縦に書いて、この表を完成させる
    - コンテキストスイッチがない状態
  - 横に書いて、この表を完成させる
    - 列を移動するときにコンテキストスイッチが発生する状態

ぜひ、紙とえんぴつを用意して体感して見てください！



1	A	2	3	J
2	B	4	6	F
3	C	6	9	M
4	D	8	12	A

「Stop Trying to Multitask. You're Terrible at It」  
(<https://medium.com/@ccanipe/stop-trying-to-multitask-you-re-terrible-at-it-a61747e112e5>)  
2020年12月17日14時に最新情報を取得





## 1. 基本的な考え方

1. 作業の可視化
2. WIP（仕掛かり）の制限
3. バッチサイズの縮小
4. 受け渡し数の削減
5. 絶えず制約条件を見つけ出して尊重する

## 2. 実践

1. オンデマンドの開発環境
2. 継続的デプロイメント
3. 継続的インテグレーション

## 1. バッチサイズが大きいとどうなるか？

パンフレットの  
郵送10件

1. 1つのWIPが大きい
2. 内容の変更を受けやすい
3. リードタイムが長くなる  
手戻りの可能性増

## 2. バッチサイズを縮小するとどうなるか？

パンフ  
レットの  
郵送1件

×10回

1. 1つの**WIPが小さい**
2. 内容の変更を受けにくい、**影響は小さい**
3. リードタイムは**短く**なる  
エラー、間違いを**早く検知**し対処可能

これから例を用いて説明します

- 封筒ゲーム

- レターメールを10個作成してもらう
  - 各ステップの作業は、一つ10秒かかる

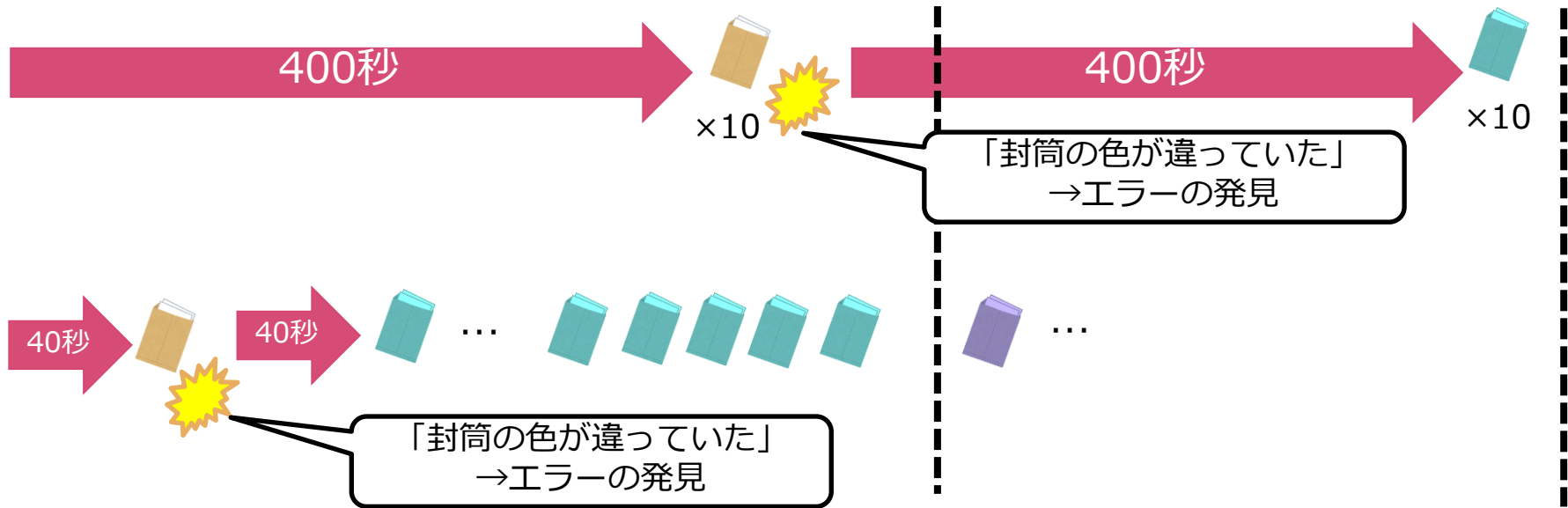


## グループA

- 各ステップ、10個の作業が終わってから次ステップに成果物を渡す

## グループB

- 各ステップ、1個の作業が終わったら次ステップに成果物を渡す
- これを10回繰り返す



- リードタイムが短くなる
  - 1つ目のメールを作成するのにかかる時間
    - バッチサイズが大きい方は400秒
    - バッチサイズが小さいほうは40秒
- WIPが減る
  - 作業が小さいため、仕掛りが発生しにくい。横やりも少ない。
- エラーの発見が早まり、手戻りが減る
  - 最初の製品でエラーが見つかったも、手戻りは1フロー分のみ

## 1. 基本的な考え方

1. 作業の可視化
2. WIP（仕掛かり）の制限
3. バッチサイズの縮小
4. 受け渡し数の削減
5. 絶えず制約条件を見つけ出して尊重する

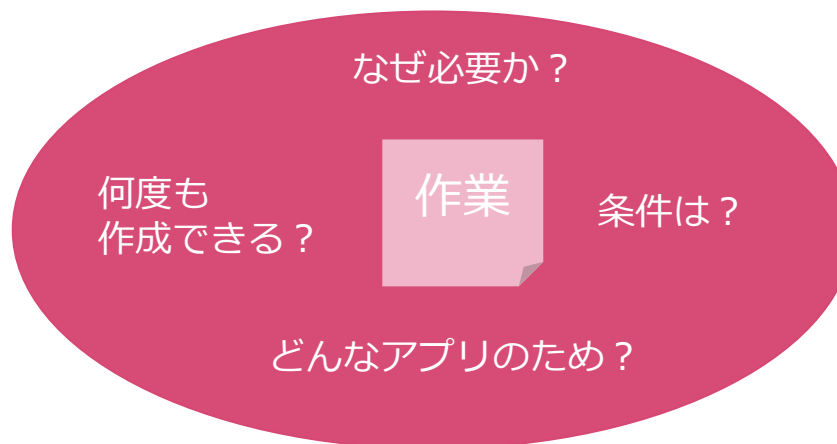
## 2. 実践

1. オンデマンドの開発環境
2. 継続的デプロイメント
3. 継続的インテグレーション

- 受け渡しをするたびに情報（コンテキストが抜け落ちる）
- 受け渡し先が忙しいとリードタイムが増大する

例：ユーザアカウントの作成を別チームに依頼する

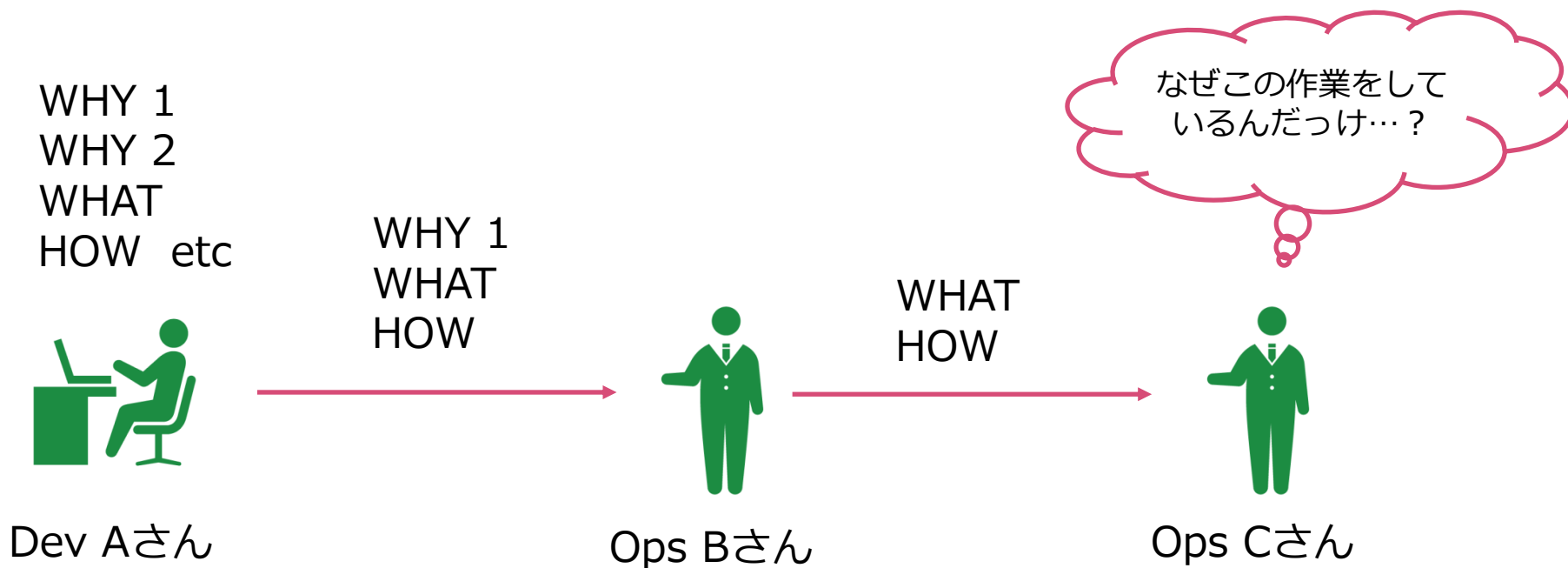
## コンテキスト



コンテキストを知るための会議やコミュニケーションが発生する。  
また、優先順位付けや確認作業も増える。

# 受け渡し数の削減をする理由：コンテキストが抜け落ちる

せっかく議論しても、  
受け渡しの回数を増やせば増やすほど  
情報は抜け落ちていく





# 受け渡し数の削減をする理由：忙しいとリードタイムが増大する

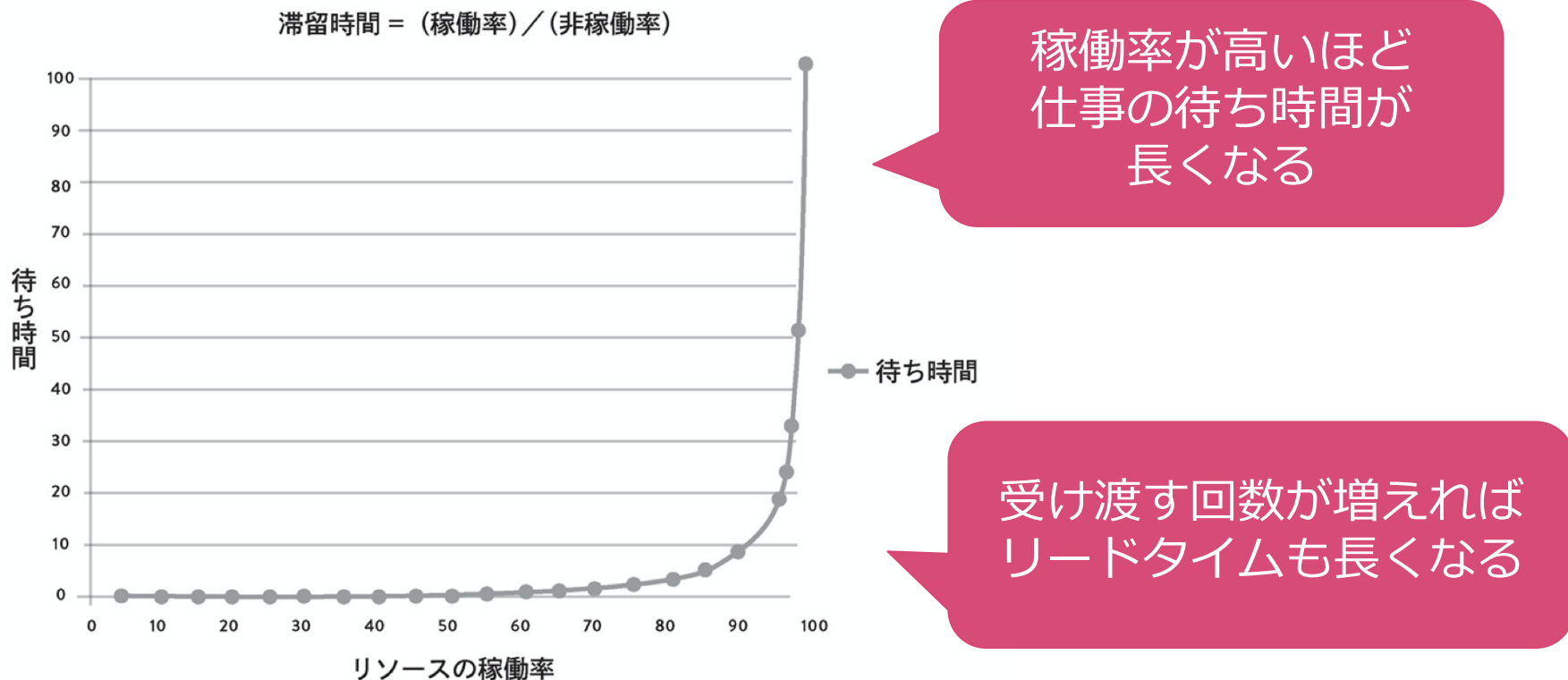


図 47 稼働率の関数として表したキューのサイズと待ち時間

※待ち時間の単位は「時間」ではないが、1単位を1時間として考えるとわかりやすい

ジーン・キム、ジェズ・ハンブル、パトリック・ドボア、ジョン・ウィリス  
「The DevOps ハンドブック 理論・原則・実践のすべて」  
日経BP社

作業の自動化

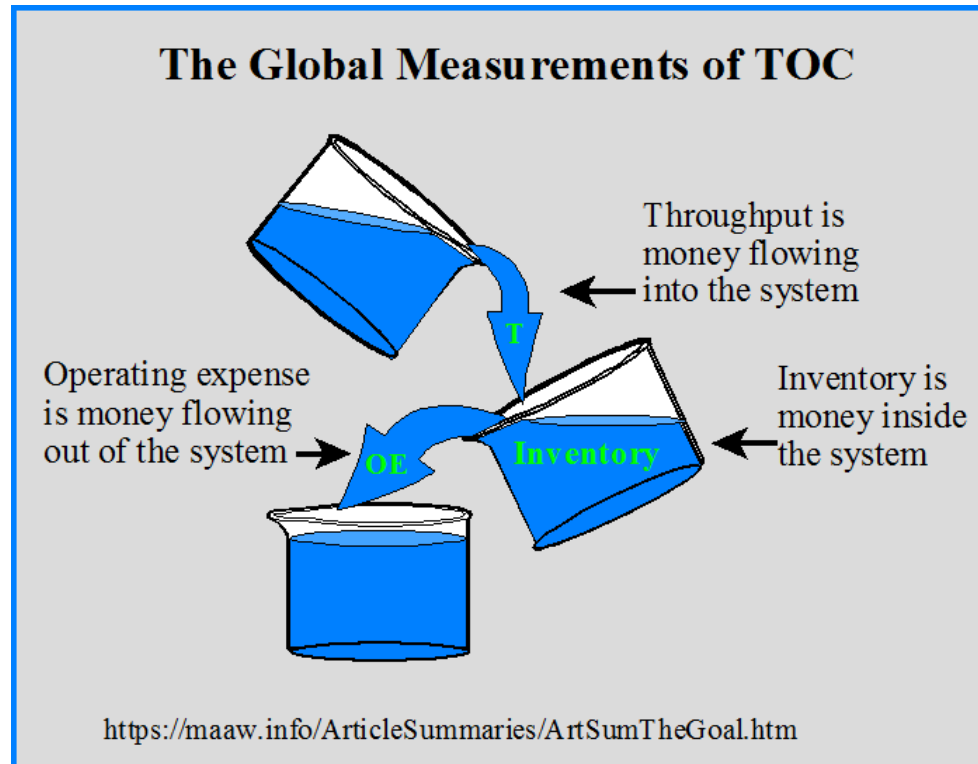
チームを再編し受け渡し回数を減らす

## 1. 基本的な考え方

1. 作業の可視化
2. WIP（仕掛かり）の制限
3. バッチサイズの縮小
4. 受け渡し数の削減
5. 絶えず制約条件を見つけ出して尊重する

## 2. 実践

1. オンデマンドの開発環境
2. 継続的デプロイメント
3. 継続的インテグレーション



- 例えば、2つ目のバケツが小さい（＝制約条件）なら、これを大きくすることを第一に考えるべき  
→「尊重する」とは「まず、制約条件から改善を図る」  
2つ目のバケツを大きくしたら、残り2つのうちの小さい方が新たな制約条件となる  
→「見つけ出して」とは「改善した後、制約条件は移動するから常に見つけ続ける」

「Global Measurements of The Theory of Constraints」  
(<https://maaw.info/TOCMeasurements.html>)  
2020年9月3日14時に最新情報を取得

一般的には、以下のようにボトルネックは変化していく

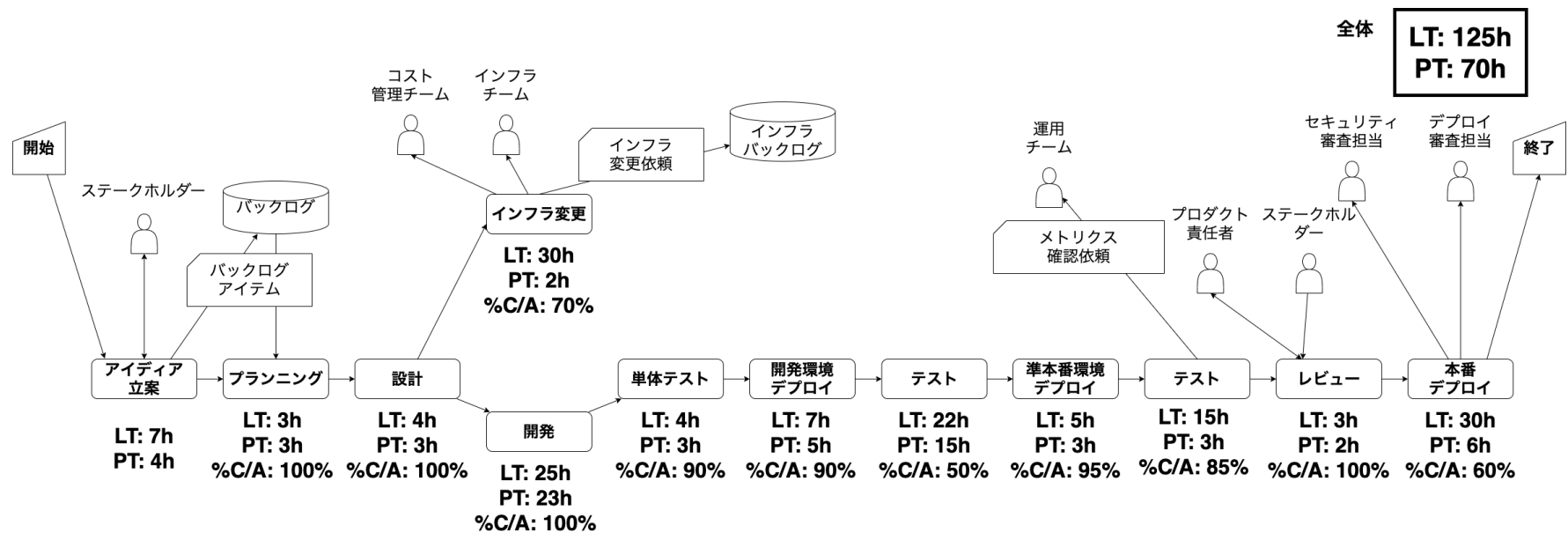
順番	ボトルネック	解決方法
1	環境の作成	オンデマンドな環境づくり
2	コードのデプロイ	デプロイの自動化
3	テストの準備と実行	テストの自動化、 デプロイ時に実行
4	過度に密結合なアーキテクチャ	アーキテクチャの疎結合化 (チームが安全に変更可能な状態を目指す)

## 技術面の改善

---

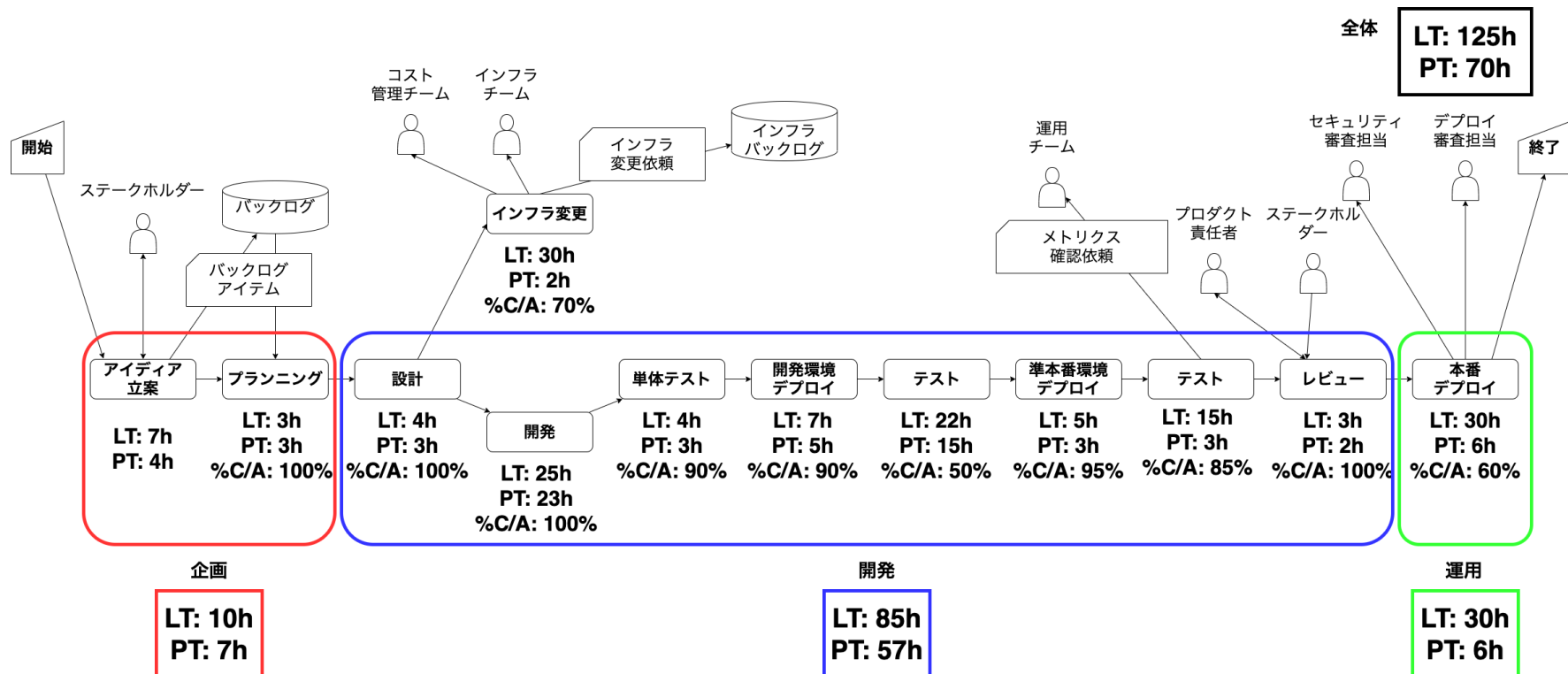
- オンデマンドの開発環境
- 継続的インテグレーション
- 継続的デプロイメント

# プロジェクトのバリューストリームマップ例

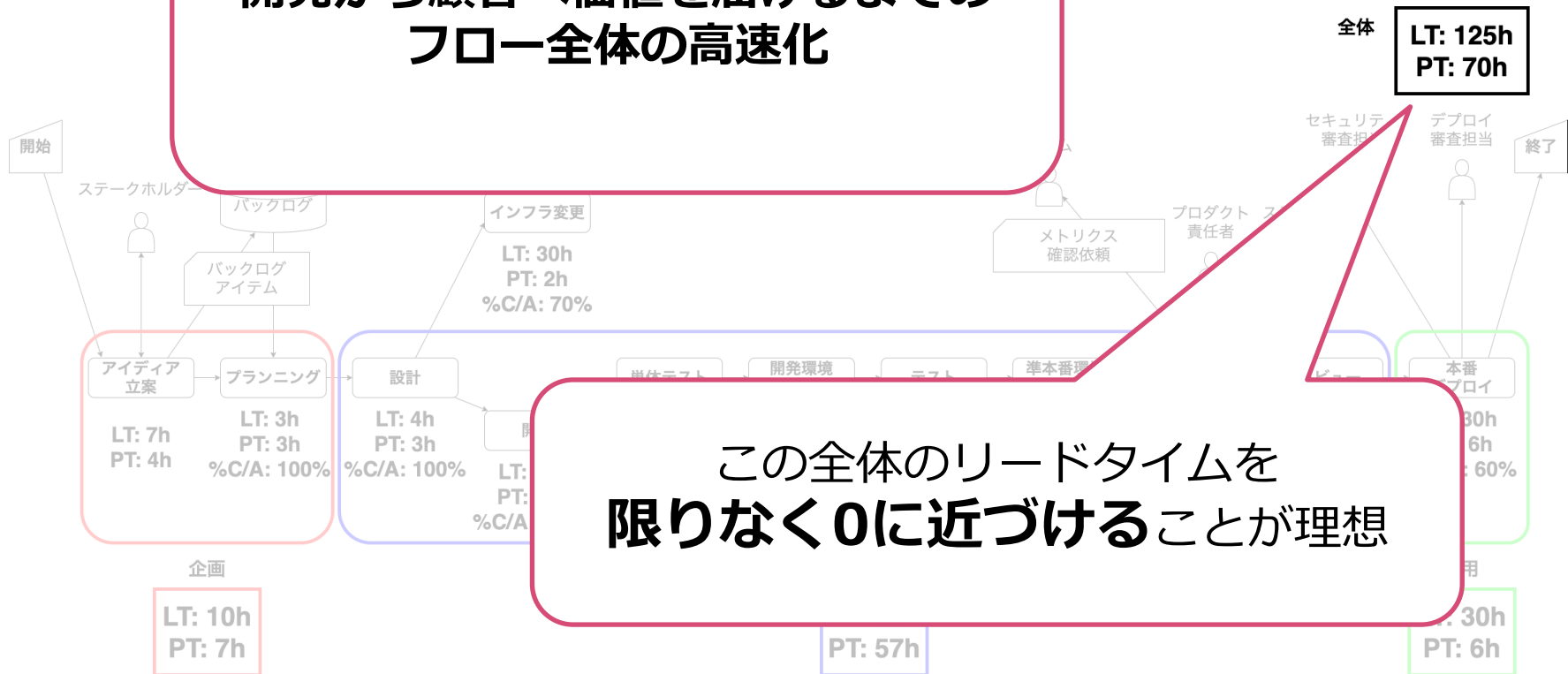




# プロジェクトのバリューストリームマップ例

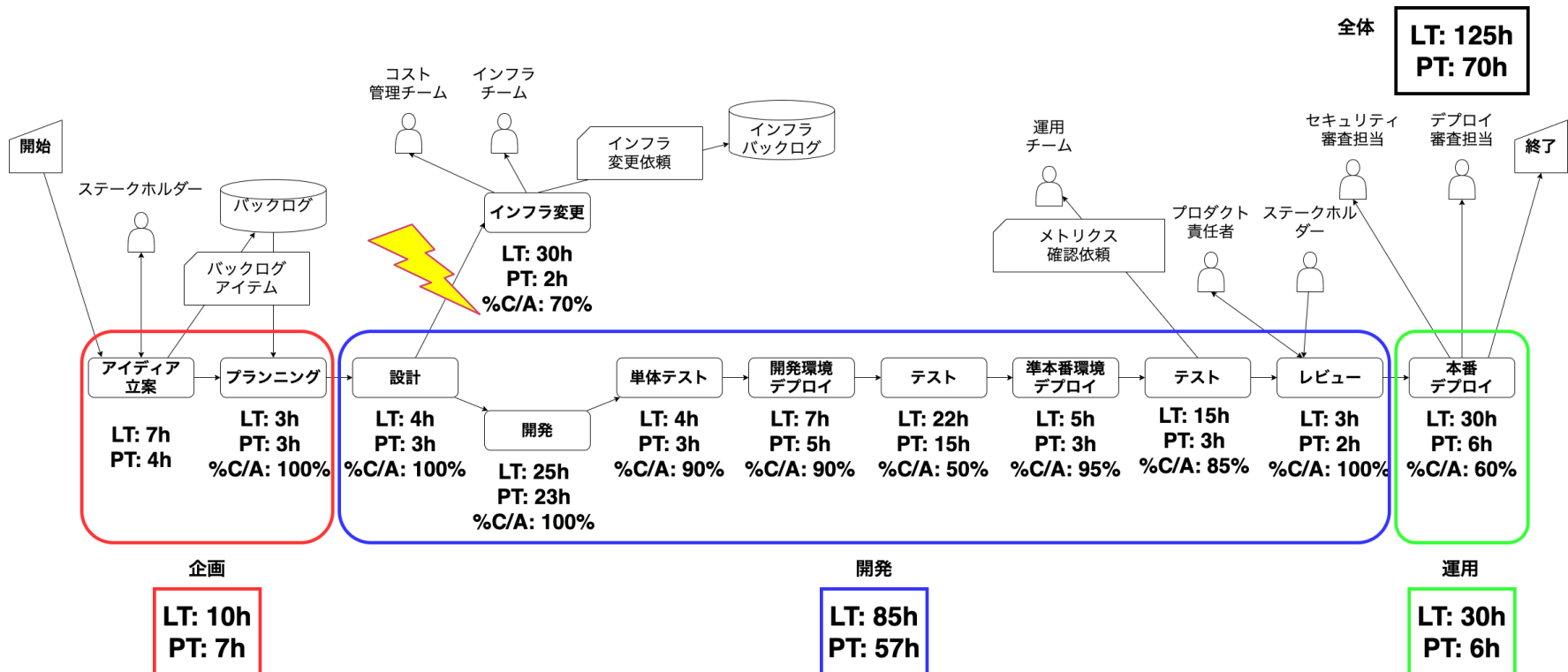


## 開発から顧客へ価値を届けるまでの フロー全体の高速化

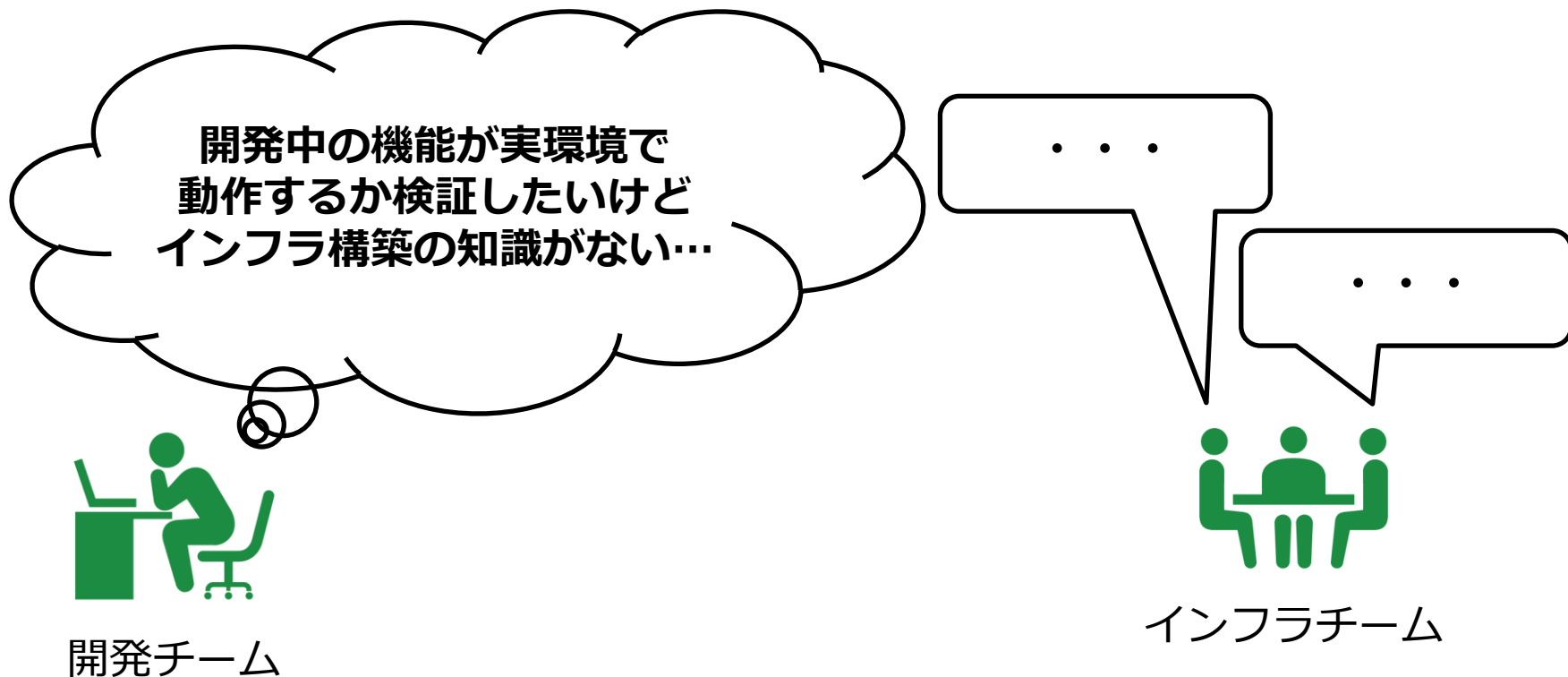


## オンデマンドの開発環境

---



- 別チームに環境構築を依頼したが遅れが発生
- 開発環境での動作確認や検証が遅延



- チーム構成の見直し
- オンデマンドな開発環境の構築

- サービスチームを作る
  - バリューストリームの関係者をチームに引き入れて  
開発サービス専任の**サービスチーム**を作る

**その他のチームに依存することなく**  
自分たちのサービスの開発・運用に責任を持てるチーム

→ チーム間の**受け渡しが少なくなり**  
**素早い意思決定**が期待できる

- 必要な時、必要な環境を自分たちで作る
  - オンデマンドかつセルフサービス

「セルフサービス」がポイント。  
そういう仕組みを作る。

→ サービスの動作確認や検証を  
早い段階で実施できるようになる



サービスチーム

機能が動作するか検証したいから  
テスト用の環境を作ろう！

障害が発生したから  
再現環境を作ろう！



こんな場面見たことないですか？

今まででも「手順書の作成と引き継ぎ」で取り組んできた。  
が、いざ環境を構築しようとして…

**環境構築手順通りに構築したのに  
本番環境と少し違う…**

**手作業で、この作業も必要なんだ**

**追加の作業もしたのに  
上手くアクセスができない…**

**実はこの作業にはコツがあってね**



Dev



Ops

環境構築手順通りに構築したのに

環境構築の度にこんな状況では、  
フローの高速化はできない



Dev



Ops

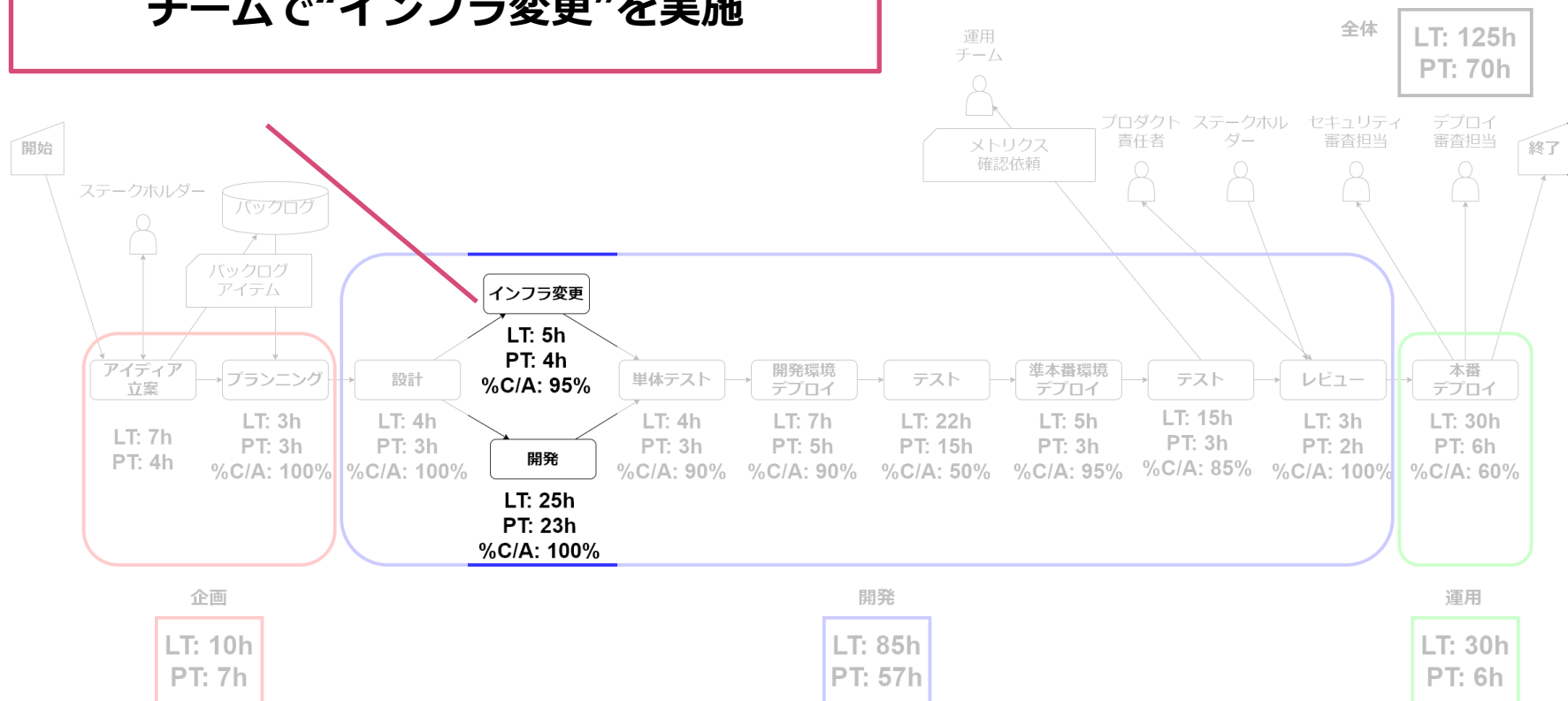
- 環境構築の自動化
  - ドキュメントやWikiによる環境構築手順の文書化ではなく環境を自動で作成できる**ビルドの仕組み**を作成

## 環境構築自動化の例

- 仮想環境イメージ
- IaC による構成管理
- 仮想イメージやコンテナ

- 全環境に統一性を強制できる
- 手作業によるエラーリスクの軽減
- 開発と運用の共通知識の拡大
- 環境を再構築できる

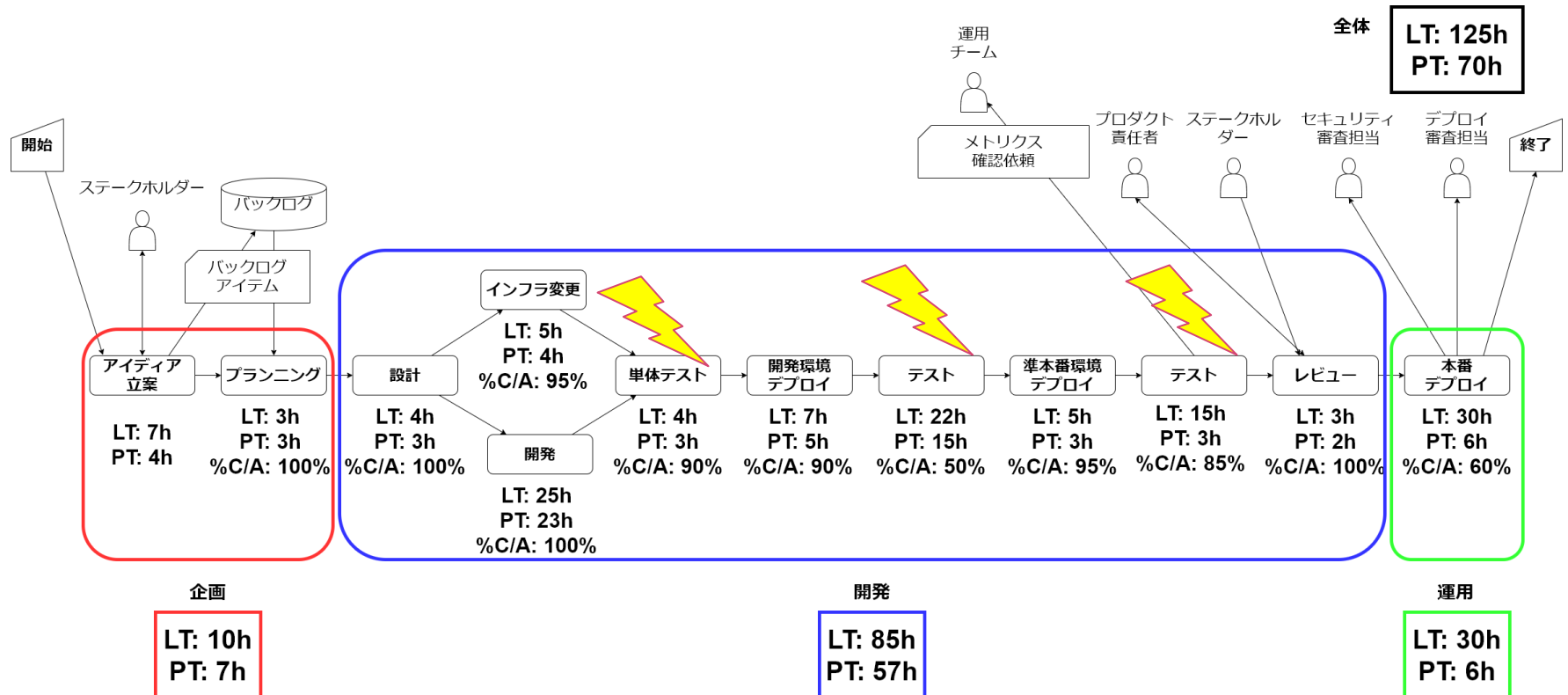
## インフラチームをチームに加えることで チームで“インフラ変更”を実施



## 継続的インテグレーション

---

# 次に改善したいポイント



- 手動テストに頼っており時間がかかっている
- 実環境にデプロイすると動かない



単体テストは通っていたのに、  
この機能が動かないな…  
どの変更が悪かったんだろう…



- 継続的インテグレーションの実施

“バージョン管理システムにコードがチェックインされるたびに、本番に近い環境で自動的にビルド、テストが行われるようにするもの”

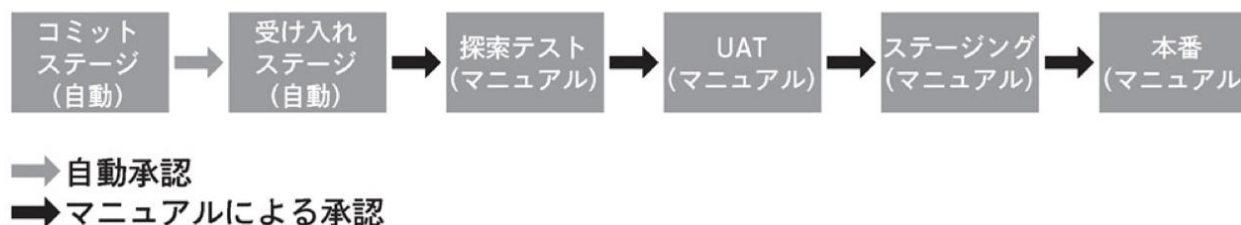


図 13 デプロイパイプライン

(出典: Humble and Farley, *Continuous Delivery*, 3)

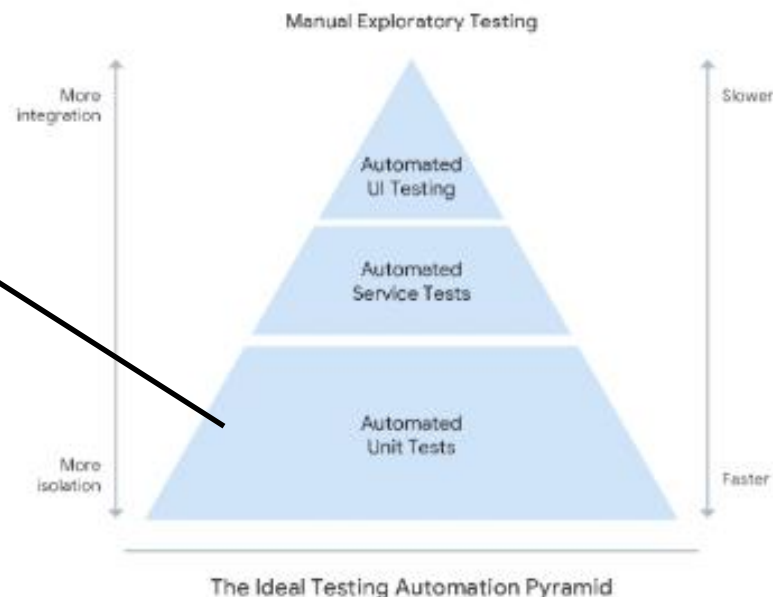
あらゆる工程を自動化しなくてはならない、ということはない。

ジーン・キム、ジェズ・ハンブル、パトリック・ドボア、ジョン・ウィリス  
「The DevOps ハンドブック 理論・原則・実践のすべて」  
日経BP社

デプロイパイプラインで、  
コードのチェックインからデプロイまでの  
一連の作業を迅速に行えるようにする

- パイプラインを、高速で信頼性が高い自動テストができるように設計する
  - テストの早い段階で誤りに気づき、素早く修正をしたい

ユニットテストで、  
後ステージで実施するテストの  
エラーを検知できるようにする



「DevOps 技術: テスト自動化 | Google Cloud」  
(<https://cloud.google.com/solutions/devops/devops-tech-test-automation?hl=ja>)  
2020年9月11日13時に最新情報を取得

**自動テスト落ちてるけど、いつも落ちてるし  
次の機能開発を優先しよう**

誰かが対応してくれるでしょ



**自動テスト落ちてるし  
テストコードも書かなくていいよね  
コードをチェックインしちゃおう**

後で直せばいいや



## デプロイパイプラインの陳腐化

誰かが対応してくれるでしょ

- 自動テストの陳腐化
- リリース前のテスト負荷の増大
- 本番アプリケーションで障害



- デプロイパイプラインの異常が発生したら、すぐに修正に取り掛かる

異常に気づいたらそれを皆に知らせ、このような行動を取ることを「アンドンの紐を引く」と呼びます。

- 正常になるまでコードのコミットを禁止する
- 問題が発見・解決が出来たら、リグレッションを防ぐためのテストを追加する

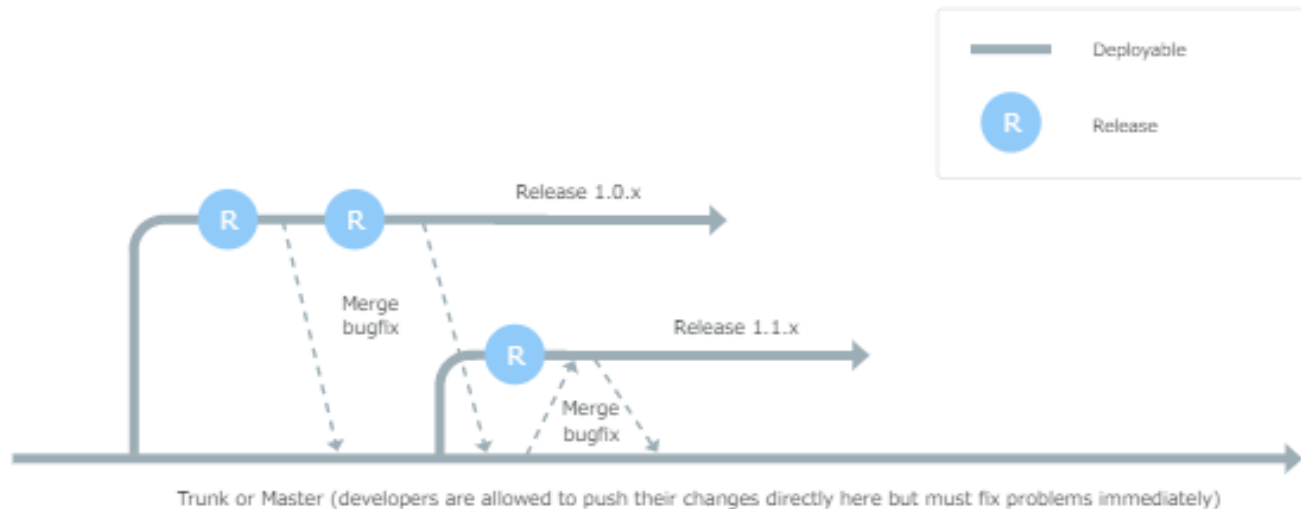
**変更が大きくなればなるほど  
問題箇所の特定と修正が大変になるので  
素早く修正することが重要**

- 本番にデプロイするコード状態を早い段階で作り上げる
  - Trunkとかけ離れたbranchで作業するほどリスクが上がる
  - Branchが多くなるほど指数関数的にマージが困難になる
  - マージが難しい作業とわかってしまうと将来のマージがさらに難しくなる

早い段階でコードを統合し、  
**デプロイ・リリースが可能な状態を保証する**

- トランクベースの開発

- 1日に少なくとも1回、コードをmasterにチェックインする

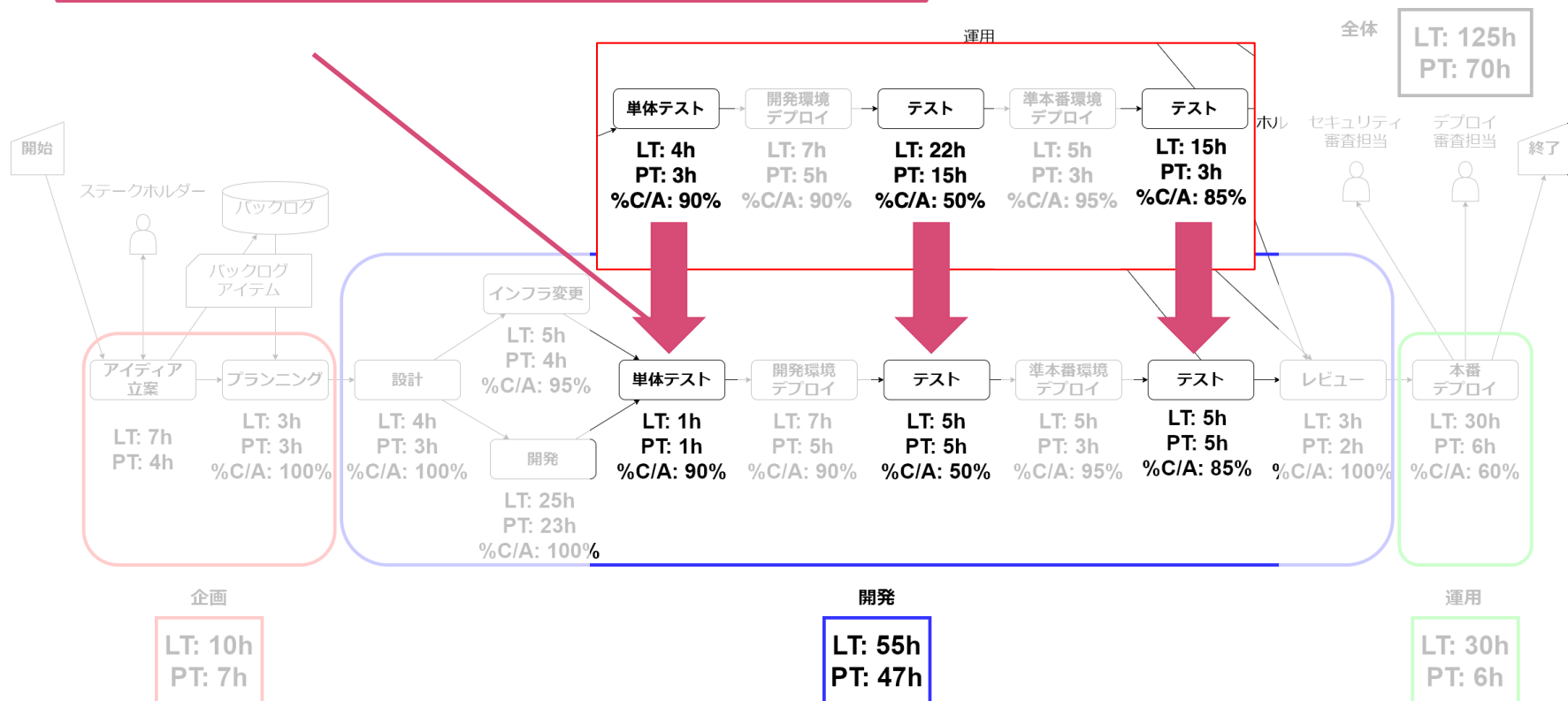


「DevOps 技術: トランクベース開発 | Google Cloud」  
(<https://cloud.google.com/solutions/devops/devops-tech-trunk-based-development?hl=ja>)  
2020年9月10日18時に最新情報を取得

**小規模なマージを繰り返して  
コードを常に最新の状態に保つ**



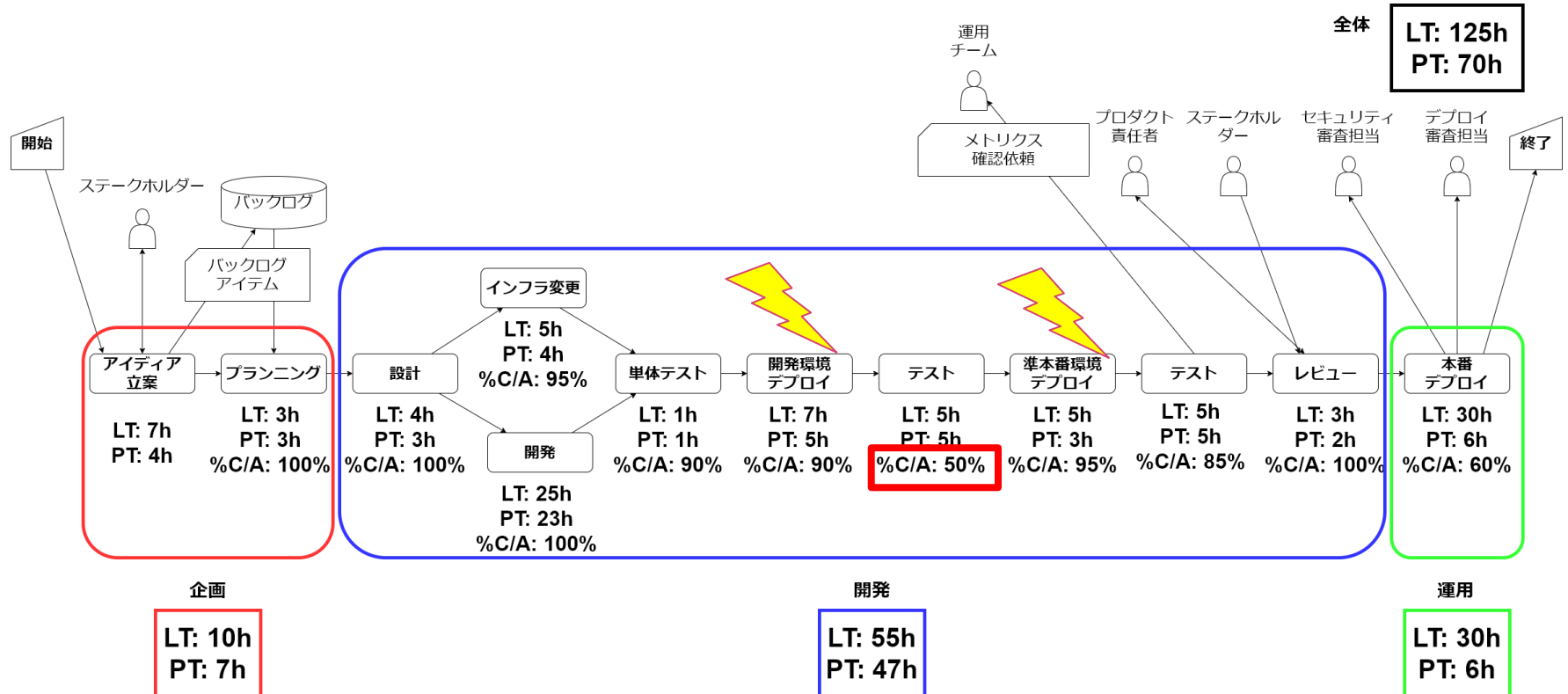
## テストを自動で実施することで テストのリードタイムを短縮



## 継続的デプロイメント

---

# 改善したいポイント



- デプロイ作業に時間がかかっている

開発環境に追加機能をデプロイするぞ  
手順が変わるから手順書作成しないと



開発チーム

あれ？  
既存の機能が動かないぞ？  
デプロイからやり直した…

- デプロイメントの自動化

- 簡素な手順で、各環境にシステムをデプロイできるようにする
  - すべての環境に**同じようにデプロイ**する

- デプロイとリリースの分離
- ローリスクなリリースパターンの選定

- デプロイとリリースとの目的の違い

	目的
デプロイ	指定された環境に 指定されたバージョンの <b>ソフトウェアをインストール</b> する
リリース	すべて、または一部の顧客に対して 機能を利用可能な状態にする

本番デプロイ ≠ リリースにすることで  
サービスチームは、**素早く頻繁なデプロイ**  
製品オーナーは、**リリース判断**  
に責務を持たせることができる



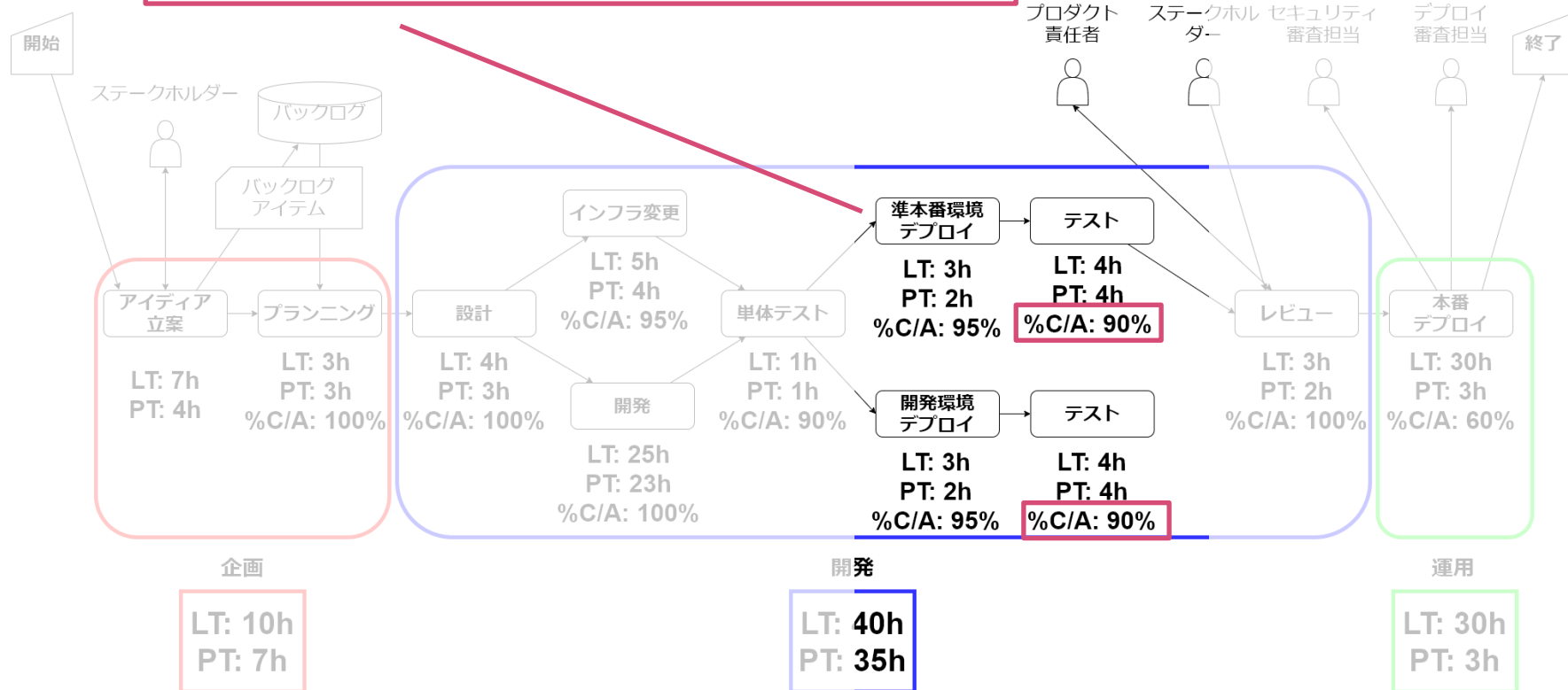
## 障害発生リスクを軽減したリリースパターンの例

- 環境ベース
  - ブルーグリーンデプロイメント
  - カナリアリリース
- アプリケーションベース
  - フィーチャートグル
  - ダークローンチ

デプロイを自動化することで、  
デプロイ失敗率も低下

全体

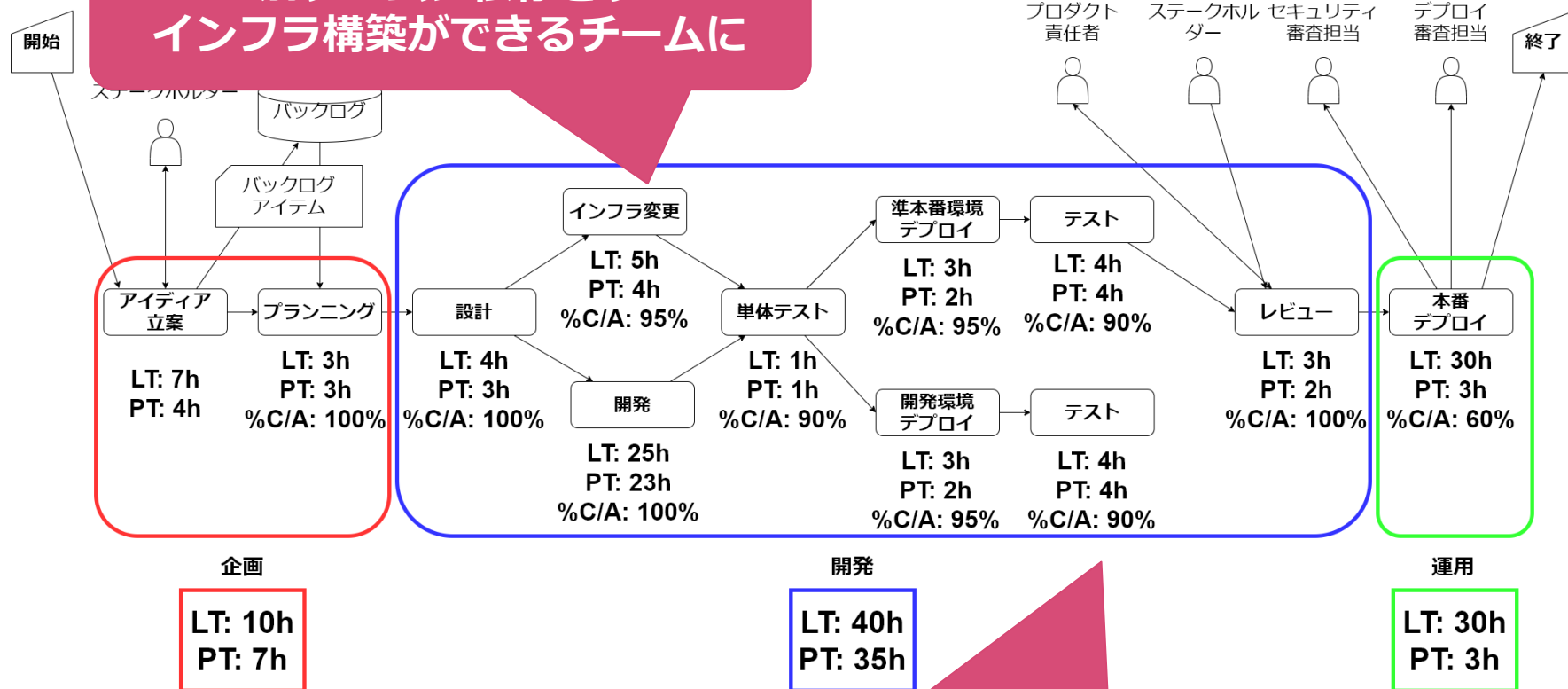
LT: 80h  
PT: 45h



全体のリードタイムが  
125h -> 80hに減った

全体  
LT: 80h  
PT: 45h

別チームに依存せず  
インフラ構築ができるチームに



デプロイ失敗による  
手戻り率も減った

## 1. 基本的な考え方

1. 作業の可視化
2. WIP（仕掛かり）の制限
3. バッチサイズの縮小
4. 受け渡し数の削減
5. 絶えず制約条件を見つけ出して尊重する

## 2. 技術面の改善

1. オンデマンドの開発環境
2. 継続的デプロイメント
3. 継続的インテグレーション

## フィードバックの原則

---

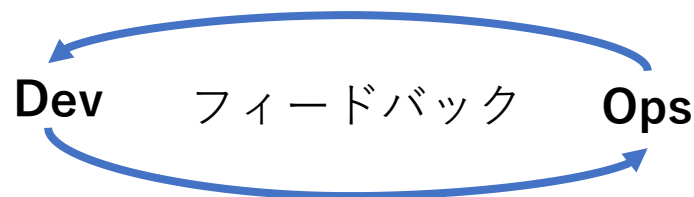
### 第1の道：フローの原則

開発から顧客へ価値を届けるまでの  
フロー全体の高速化



### 第2の道：フィードバックの原則

あらゆるステージにおける**素早く**  
**頻繁なフィードバック**の実現



### 第3の道：継続的学習&実験の原則

科学的な実験・リスクを取る  
ことを支持し、組織として学習する



ジーン・キム、ジェズ・ハンプル、パトリック・ドボア、ジョン・ウィリス  
「The DevOps ハンドブック 理論・原則・実践のすべて」

日経BP社  
をもとに作成

“私たちの目標は、  
今まで以上に安全で回復力のある作業システムを構築することである。”

ジーン・キム、ジェズ・ハンブル、パトリック・ドボア、ジョン・ウィリス  
「The DevOps ハンドブック 理論・原則・実践のすべて」  
日経BP社

問題を素早く発見し  
顧客からのフィードバックを素早く得るための  
サービスを目指す

- 発生と同時に問題を検知・対応できるようにする
- モニタリングシステムの構築
- 運用のことを考えたロギング・メトリクスを生成する
- 上流での品質担保



発生と同時に問題を検知・対応する

---

アプリケーションや関連システムがエラーを吐いている

自分の開発環境だし、**とりあえず**再起動しちゃおう



あなた  
(運用 or 開発)

理由はよくわからないけど  
**いつもやっていることだから**、〇〇で対応しよう

原因がわからぬまま対応してしまう

重大な障害が発生しなければ  
問題を検知できないことが多い

安定稼働状態

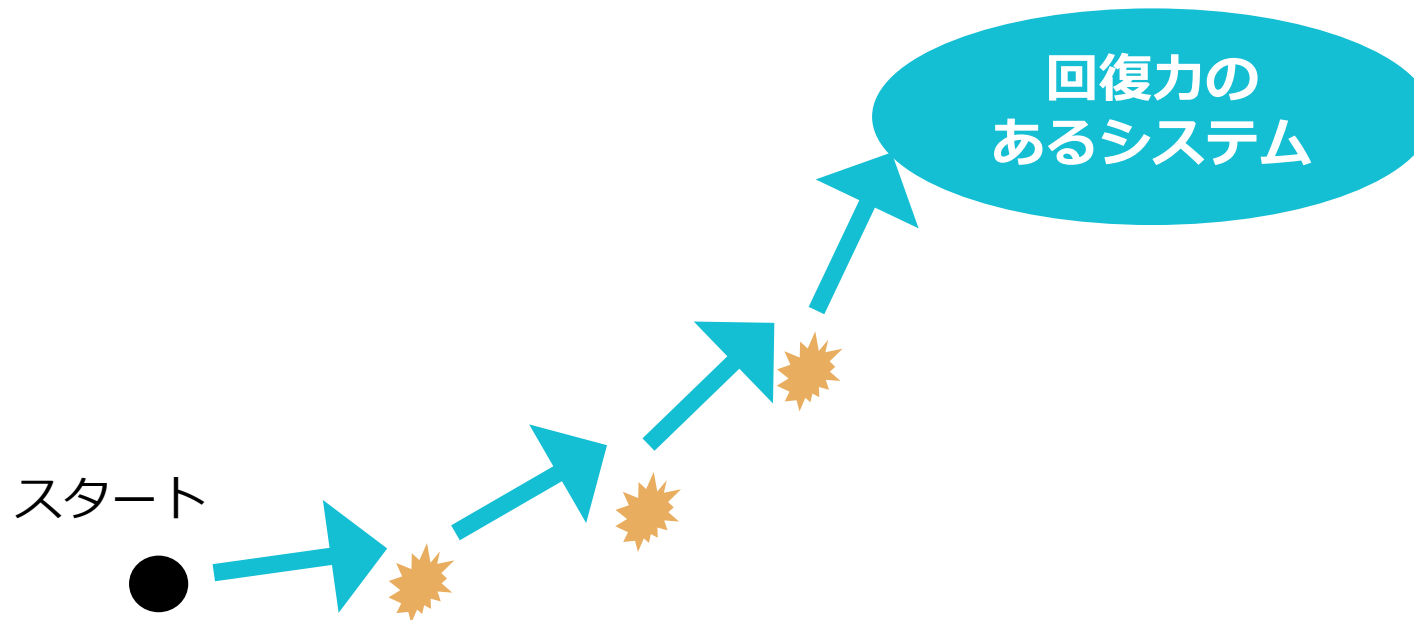


非稼働

大規模  
障害



修正しようにも  
問題が大きくなりすぎているので  
時間が必要だ



問題は小さいうちに見つけ、  
因果関係を把握したうえで  
自信をもって正確・確実に対応する

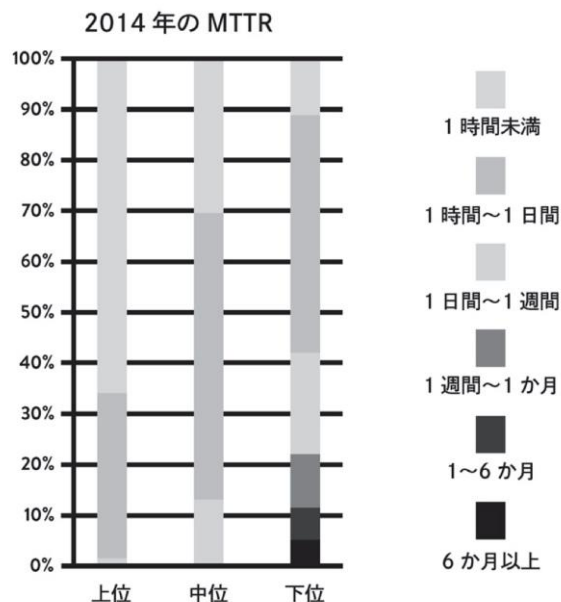
- 障害が発生した時点で、障害の原因を分析し  
「**正確で確実な対応**」を見つける
  - 日常的な実践が必要

DBにアクセスできなくなっていたのは  
この変更でコネクションを新しく作っているから  
コネクション数が足りなくなったんだ！



この機能でメモリリークが  
発生しているから  
サーバーのメモリが枯渇しているぞ

## MTTR（平均修復時間）の向上



上位企業のMTTRの中央値は分単位、  
下位企業の中央値は日単位

図 25 パフォーマンス上位、中位、下位の組織がインシデントを解決するためにかかる時間

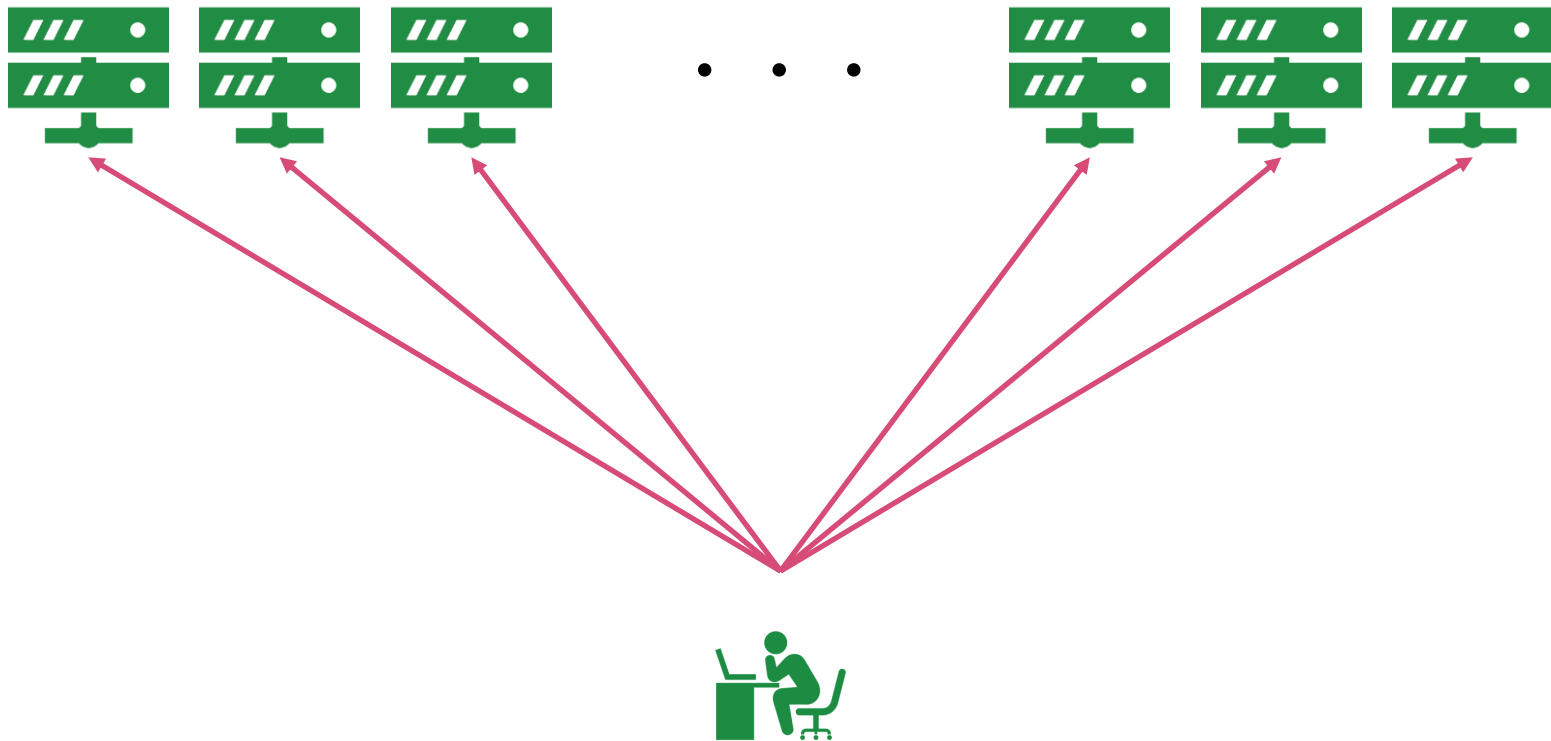
(出典：Puppet Labs, 2014 State of DevOps Report)

ジーン・キム、ジェズ・ハンブル、パトリック・ドボア、ジョン・ウィリス  
「The DevOps ハンドブック 理論・原則・実践のすべて」  
日経BP社

## モニタリングシステムの構築

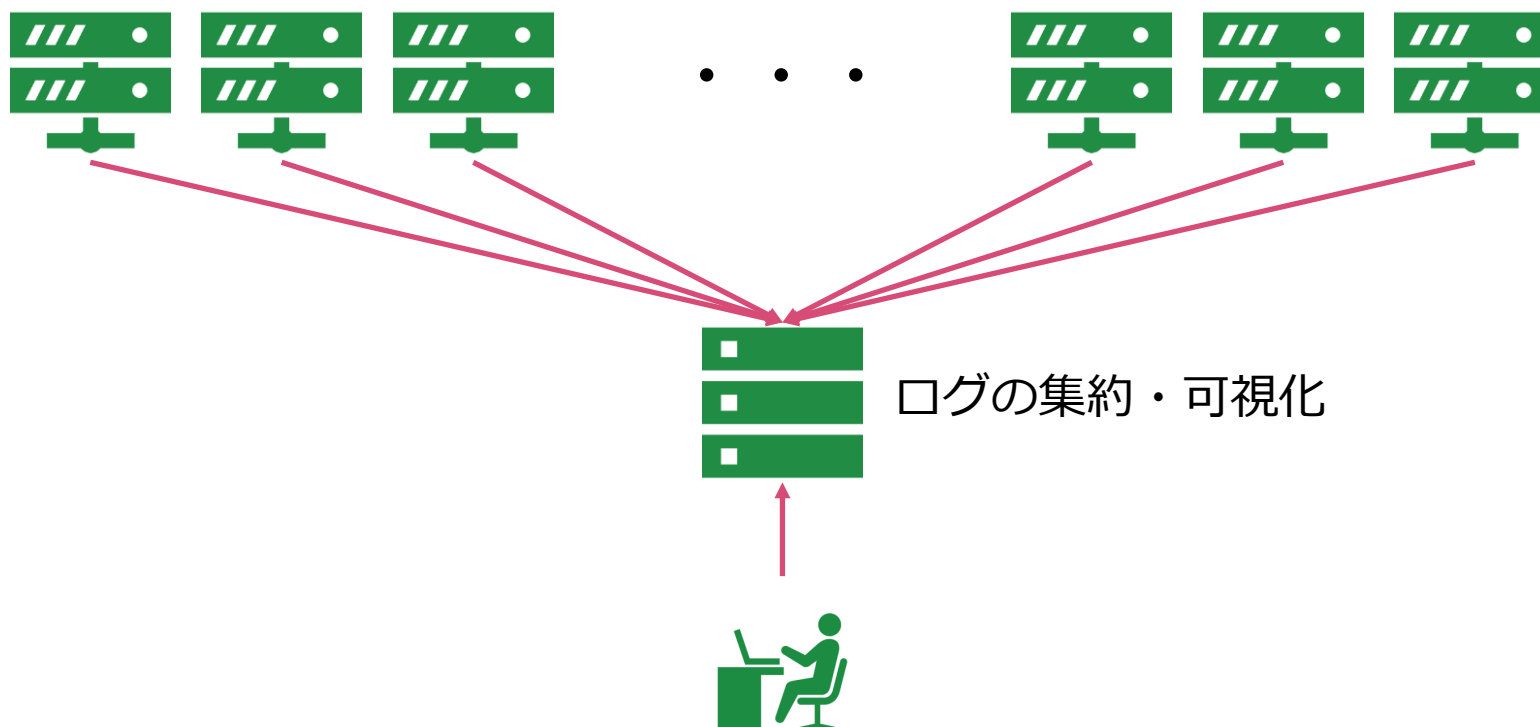
---

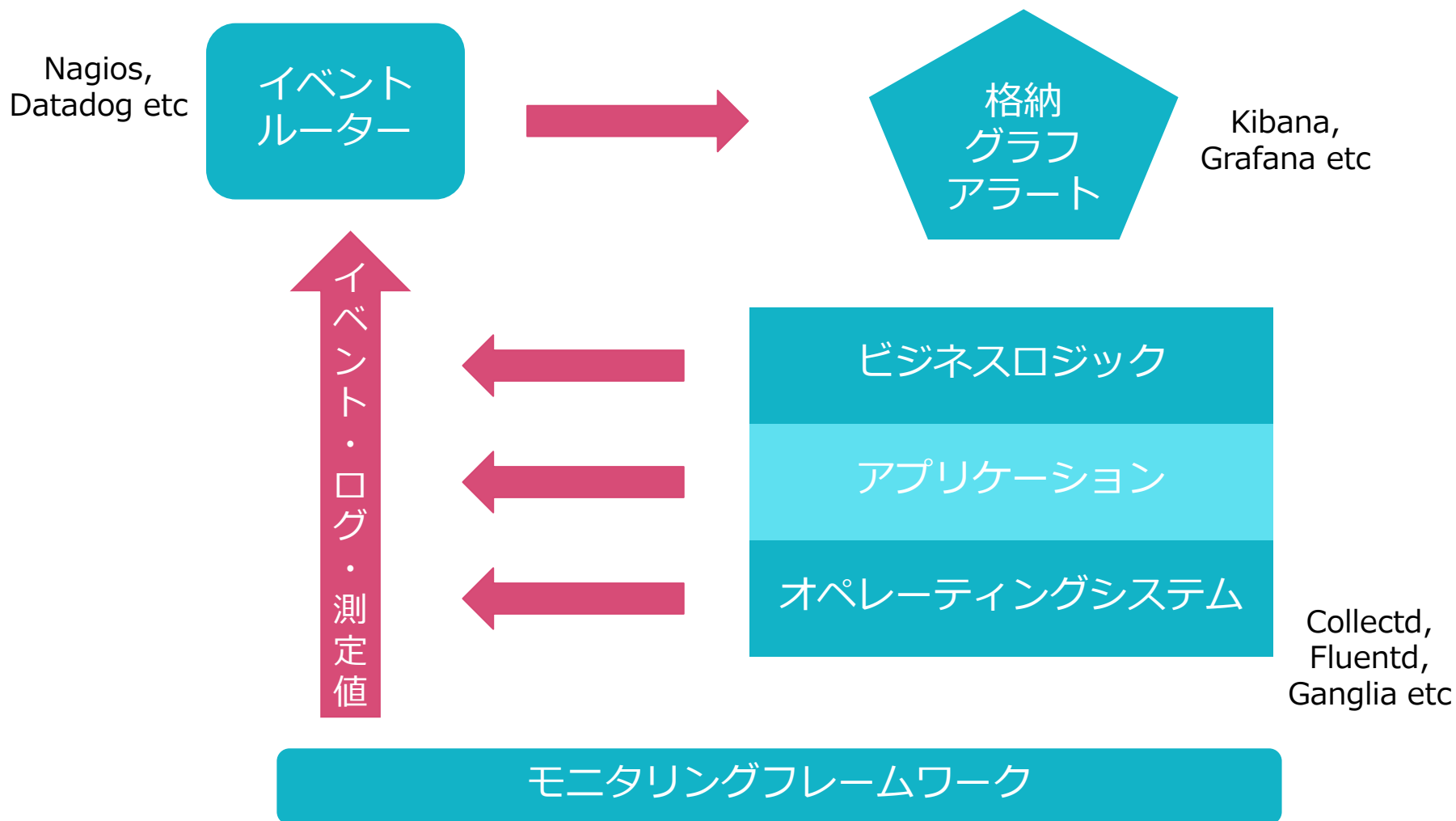
100 台あるサーバがローカルにログを出力していたとき、  
100 台のサーバにログインしてログを確認しますか？





100 台あるサーバのログを集約・メトリクスにして、  
可視化することで、全体の状況を把握する





- アプリケーションレベルやインフラレベルの指標だけでなく、**ビジネスレベル**の指標化を含める

DevOpsでは「**ビジネスレベル**」の指標もメトリクスとして加える。  
(DevOpsとは「市場で勝ち抜く」ことを実現すること、だから)



システムが「組織目標にどの程度貢献しているか」を可視化することができる

## アクセスログの例

```
:::1 - - [07/Sep/2020:10:56:11 +0900] "GET /login HTTP/1.1" 200 43
:::1 - - [07/Sep/2020:10:56:11 +0900] "GET /favicon.ico HTTP/1.1" 200 209
:::1 - - [07/Sep/2020:10:58:43 +0900] "GET /sign-up HTTP/1.1" 200 79
...
:::1 - - [07/Sep/2020:11:38:15 +0900] "POST /account/upgrade HTTP/1.1" 500 1204
```

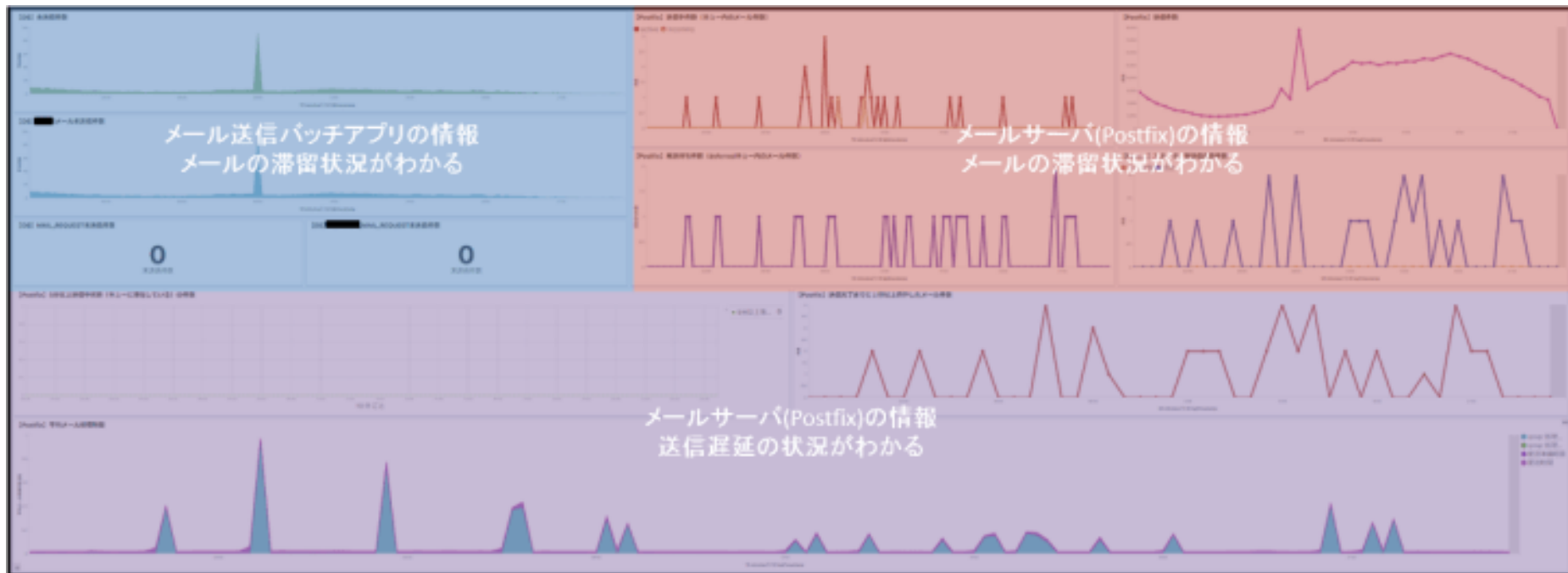
## 指標例

### ■ システム指標

- 500エラーはサービス全体で何件発生したか
- 10 秒以上応答を返せなかったリクエストの割合が x %

### ■ ビジネス指標

- 顧客がどれだけ新規に登録しているのか
- サブスクリプション契約できたユーザ数はどれだけいるのか



「サービス開発する人にまねしてほしい、ログの集約と活用事例」  
(<https://fintan.jp/?p=4685>)  
2020年9月14日11時に最新情報を取得

## 測定するデータがなければ意味をなさない

Dev

開発者が関心を持つ  
イベントのみをロギング

Ops

環境が落ちていないかしか  
モニタリングしない

まずいことが起きても、なぜ想定通り動かないのかわからない



### 問題の発見・修正は日常業務

システムの全体的な動作を理解できるだけのデータを生成するように  
アプリケーション・環境を設計しなければならない

## 運用のことを考えたロギング

---

**運用時に**出力したログを**どのように使うか**を  
考えて実装していますか？





例：xxx.csvが読み込めなかったというエラーが発生



どの機能でエラーが発生したか  
分らないと、運用時に困るのでは？



**ログのコンテキストに  
機能を特定できる情報を埋め込む**

例：xxx.csvが読み込めなかったというエラーが発生



何が発生したか以外に  
顧客影響をどうやって特定する？



**ログに顧客影響を特定できる情報を含める**

- 問い合わせ時に顧客を特定できるID
- 注文を特定できる識別子

- 障害が発生した場合に、原因を特定しやすいログ出力
  - コンテキストをしっかりとログに含めておくことが重要

以下を**素早く検知・発見**できるようなログ出力を心掛ける

## コンテキストの例

- エラーレベル
- アクセスされたサーバ
- アクセスされたAPI
- エラーが発生したリクエスト内容
- エラーとして返却したレスポンス
- アプリケーション内部で発生したエラー内容

レベル	内容
デバックレベル (DEBUG)	プログラムで発生しているあらゆる情報を知らせる。 トラブル時に一時的に有効にされる。
情報レベル (INFO)	ユーザ主体の動作やシステム固有の動作に関する情報を知らせる。
警告レベル (WARN)	障害発生の可能性を知らせる。 (DB呼出の時間が長いなど)
障害レベル (ERROR)	障害の詳細情報。 (APIコールエラー、内部エラー条件など)
致命的レベル (FATAL)	異常終了が必要な状態を知らせる。

- どのようなアラートは誰に連絡するか、という基準・ルールをあらかじめ決めておく

**ERRORレベル以上はシステムの可用性に関わるので即座に全員に報知しよう**

**WARNレベルはシステム継続可能だから業務時間内にのみ発報するようにしよう**



サービスチーム

**あらかじめ関係者で合意しておく**



システム責任者  
(顧客)

## データの可視化・公開

---



yyyy/mm のアクティブユーザの推移を教えてください

△△サーバでエラーが発生したので、  
〇〇のタイミングでのアクセス状況を確認してください



担当者がサーバからログをダウンロード



ダウンロードしたログを見やすいように加工して提供

アジリティの低下

MTTRの低下

サービスにとって**必要な知識・事実**を、  
サービスに関わる**全ての人がすぐに手に入れられる**環境が必要



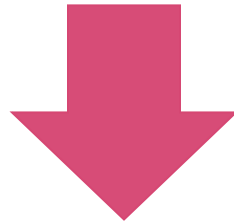
チーム全員が見える場所に  
データを掲示することで  
全員でデータを共有する

TIS『bit&innovation』の全体イメージ  
([https://www.tis.co.jp/news/2016/tis\\_news/20160915\\_1.html](https://www.tis.co.jp/news/2016/tis_news/20160915_1.html))  
2020年9月7日17時に最新情報を取得



- 全員が現実を同じように見れる
- 透明性を確保できる
  - チームがステークホルダーに対して何も隠していない
- 責任に正面から向き合える
  - 何が起きているかを認識できる

- デプロイパイプラインで全てのエラーを検知することが理想だが、捕捉できないエラーが残る可能性はある



**モニタリングで問題を素早く検知し、  
素早くシステムを正常な状態にする**

## 上流での品質担保

---

- 本番環境のオンコールを  
バリューチェーンに関わる全員で共有する

→ Devは問題が発生した際にはフィードバックを体感できるし、  
どのように利用者に使われているかを知ること、  
Opsや顧客に近づくことができる



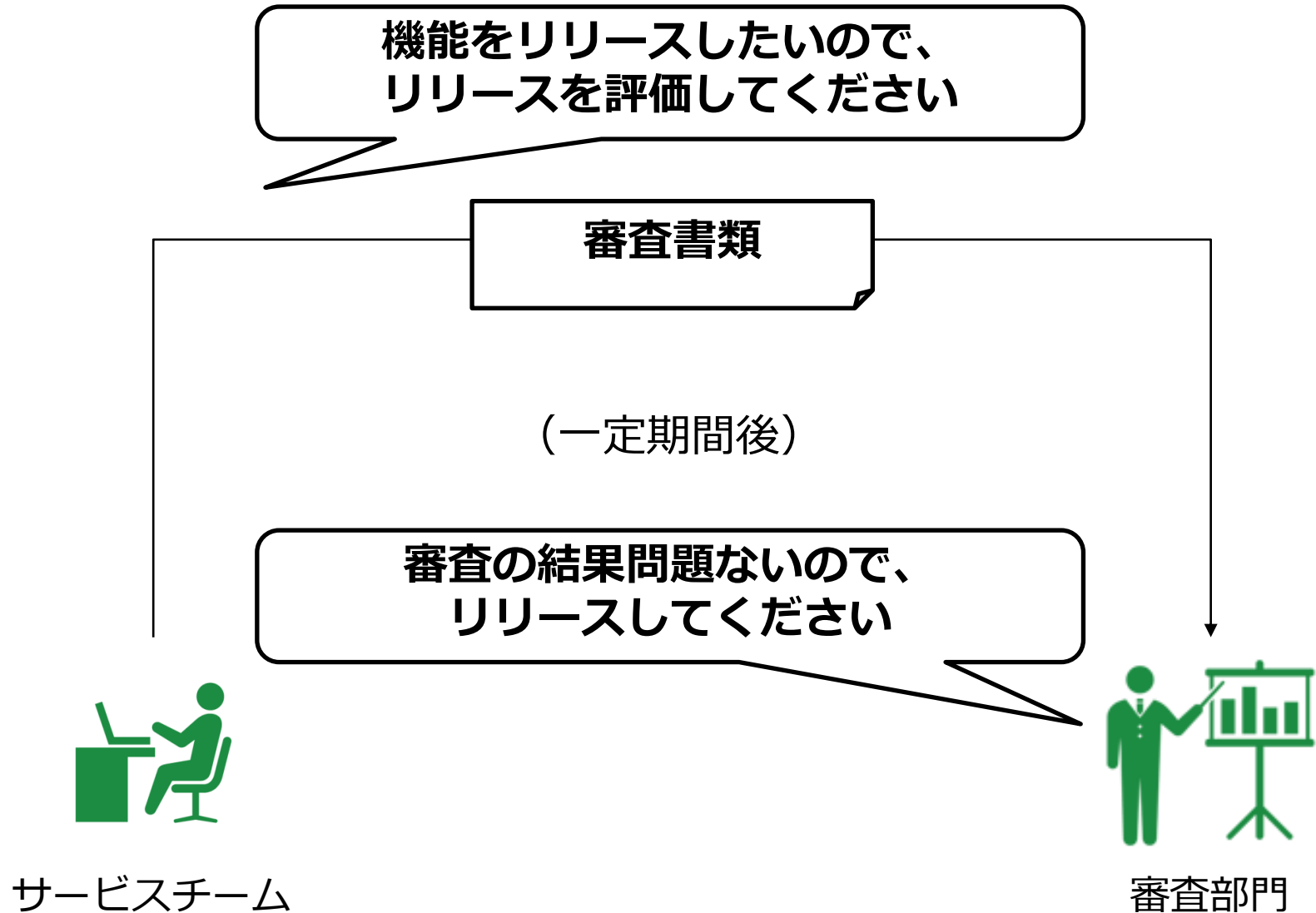
サービスチーム

問題が発生したから、ロールバックで対応しよう

顧客にこの機能が良く使われているのか



品質に対する責任感



### 問題発生

問題が発生したので

- 承認ドキュメントとして〇〇を追加します
- 承認のために、より偉い役職の承認を必要とします
- 評価の期間を延ばします



サービスチーム



審査部門

チーム外部の承認が変更のリリースに対して  
リスクを正しく評価できているのか？

- 承認ドキュメントとして〇〇を追加します
- 承認のために、より偉い役職の承認を必要とします

開発段階から  
リリース対象の変更に  
責任を持たないといけない



サービスチーム



審査部門

リリース対象の変更に責任を持つのはあくまでサービスチーム。  
サービスチームは品質の責務を審査部門に委ねない。

- 「問題に近い人が問題をよく知っている」
  - 本番環境にエラーを含む変更がリリースされる前に、  
上流でリスクを減らす

品質の確保はサービスチーム内での  
ピアレビューでも確認できる。

- 様々な場面でピアレビューを実施
  - 変更の品質向上だけでなく以下の効果を期待できる
    - 相互訓練
    - ピアラーニング
    - スキル向上



- 一般的に官僚主義的なプロセスは  
リードタイムを伸ばし、  
**組織パフォーマンスの低下要因**になっている

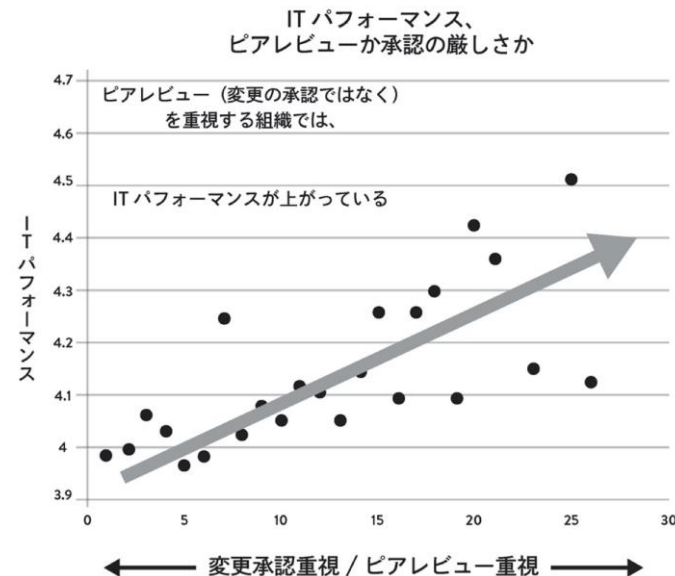


図 41 ピアレビューを重視する組織の方が、変更の承認を重視する組織よりも  
パフォーマンスが高い  
(出典：Puppet Labs, 2014 State of DevOps Report)

ジーン・キム、ジェズ・ハンブル、パトリック・ドボア、ジョン・ウィリス  
「The DevOps ハンドブック 理論・原則・実践のすべて」  
日経BP社

- 官僚的に実施しているようなプロセスはないでしょうか？
- どのようにしたら、取り除けるでしょうか？

- 発生と同時に問題を検知し、素早く対応する
- モニタリング基盤の構築
- 運用のことを考えたロギング
- 上流での品質担保

## 継続的学習 & 実験の原則

---

## 第1の道：フローの原則

開発から顧客へ価値を届けるまでの  
フロー全体の高速化



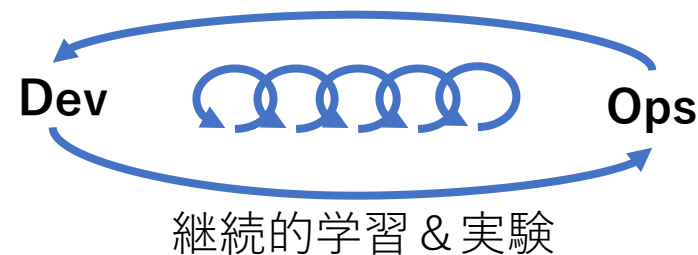
## 第2の道：フィードバックの原則

あらゆるステージにおける**素早く**  
**頻繁なフィードバックの実現**



## 第3の道：継続的学習 & 実験の原則

科学的な実験・リスクを取る  
ことを支持し、**組織として学習**する



ジーン・キム、ジェズ・ハンブル、パトリック・ドボア、ジョン・ウィリス  
「The DevOps ハンドブック 理論・原則・実践のすべて」

日経BP社

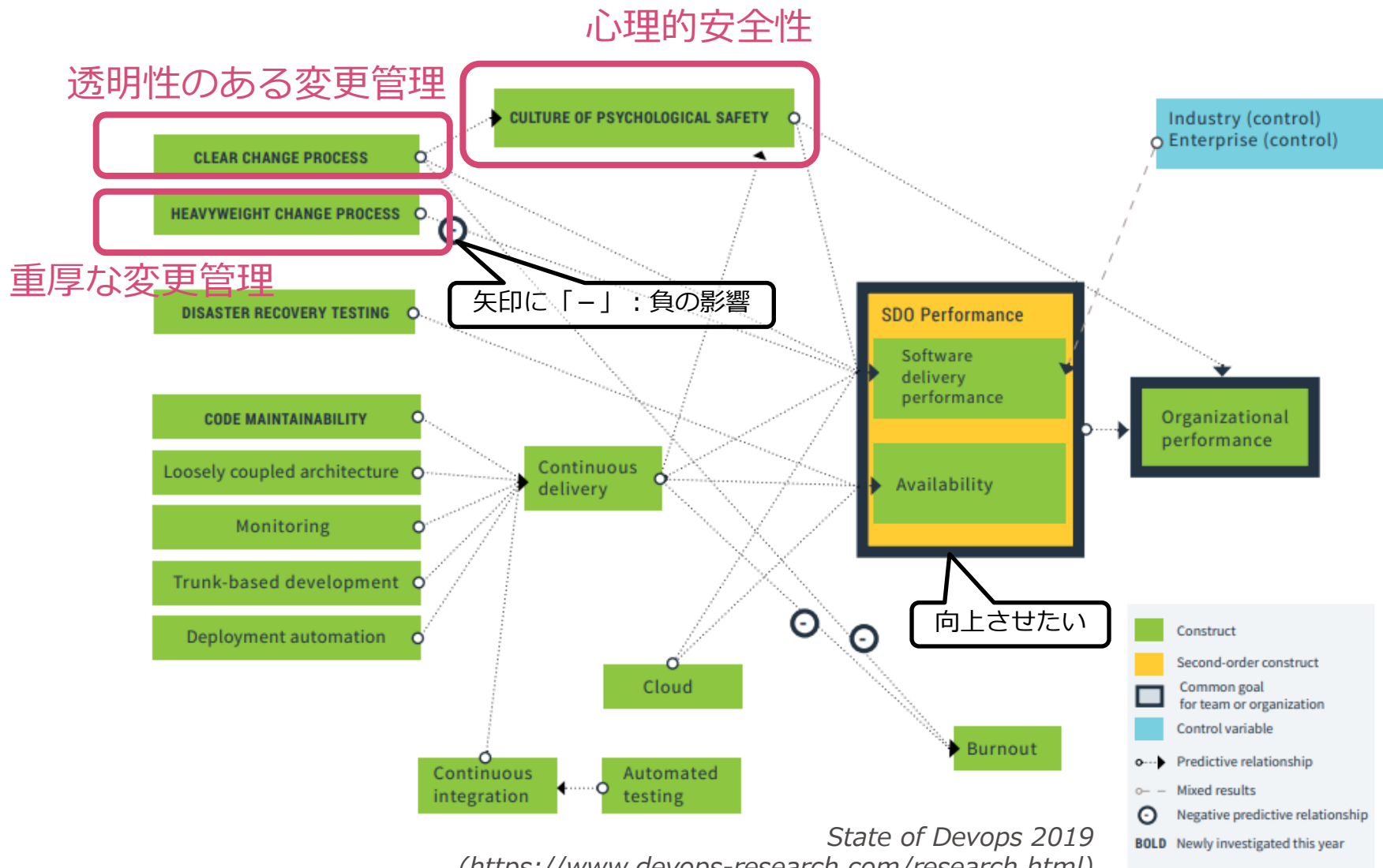
をもとに作成 ■ 149

“第3の道は、継続的な学習と実験の文化を創り出すことに着目する。  
これは、**個人がコンスタントに知識を生み出し、それをチームや組織の知識に  
転化していく**ための原則である。”

ジーン・キム、ジェズ・ハンプル、パトリック・ドボア、ジョン・ウィリス  
「The DevOps ハンドブック 理論・原則・実践のすべて」  
日経BP社

継続的学習 & 実験の原則では、  
**成功と失敗から学びを得る**ことで、  
**市場で勝ち抜く組織**へと成長していくことを目指す

## SOFTWARE DELIVERY & OPERATIONAL PERFORMANCE



State of Devops 2019  
(<https://www.devops-research.com/research.html>)  
2020年9月15日14時の最新情報を取得

- 問題を学習機会と捉える文化を創り出す

この変更には事実として  
こういう背景があった

この意思決定には  
こういう背景があった

問題を再発させないためには  
デプロイパイプラインに  
これを組み込めばよいのでは





- **問題を学習**と捉える文化を創り出す

問題から得られる**学びの総量**を増やす

問題の詳細を自ら話し出せるような  
「**安全な環境**」をつくりだし、  
常に「**正しいと思える**」対処策を考える



- 人を非難することなく  
「**障害の状況と障害につながった意思決定プロセス**」に  
着目した検証を実施する

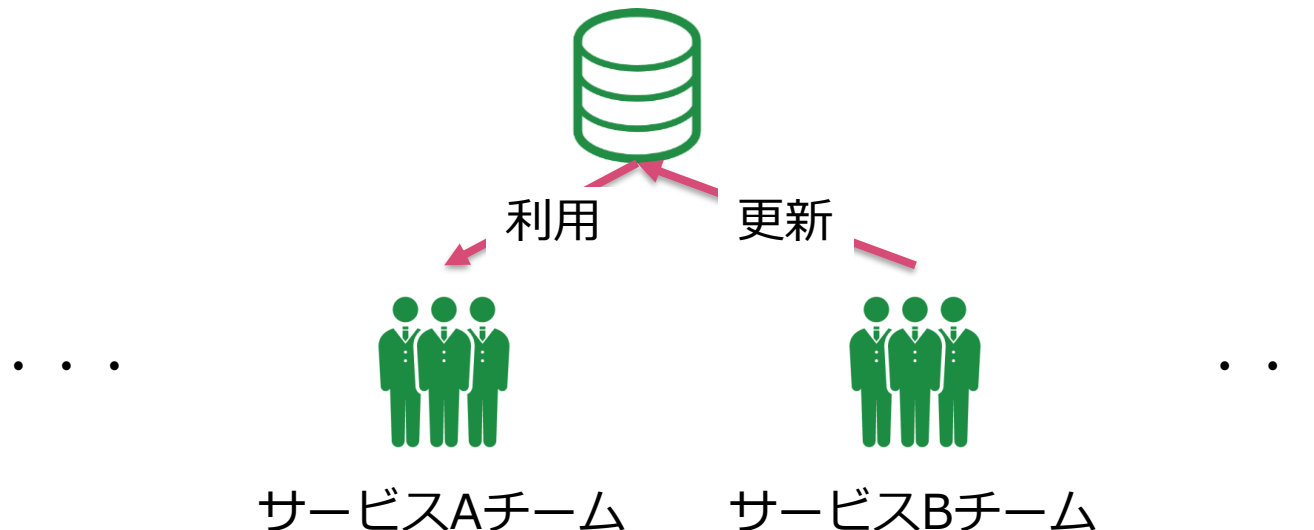
**失敗を非難するのではなく  
「その時になぜ正しいと思ったのか」を考えて  
再発防止に努める**

- 可能な限りポストモーテムを公開して  
**一部の学習と改善**をチームを超えた**学習と改善に転化**する

別のチームでこんな障害が起きたのか  
自分たちは大丈夫だろうか



- 再利用可能な成果物を全社で公開する
  - ドキュメントではなく、その他のプロジェクトでも利用可能な形でツール、プロセス、アーティファクトを共有する
  - 単一のレポジトリを作って管理をすることで、社内のどのような環境からも利用できるようになる



- 障害に対する訓練を実施する

### **例 : Chaos Engineering**

- 本番環境に意図的に障害を発生させ、  
チーム全体で素早く復旧できる状態を目指す

**学習機会を増やし、  
組織全体が成長することで市場の信頼を勝ち取る**

- DevOpsを実現するためには  
**1人1人がフィードバックを上げる**ことが重要
- 自分だけでは変えられないと諦めるのではない  
声を挙げ、集めて、**組織を変える文化**を  
**私たちが作っていく**ことが大切



- 障害を失敗ではなく学習と捉える組織文化
- 一部門の発見を全社的な進歩につなげる
- 高い回復力（レジリエンス）を持った組織になる
- 個人の声を集めて組織を変える

- DevOpsについて
  - DevとOpsの対立
  - DevOpsの定義
  - バリューストリームマッピング
- DevOpsを支える原則
  - 第一の道：フローの原則
  - 第二の道：フィードバックの原則
  - 第三の道：継続的学習&実験の原則



ITで、社会の願い叶えよう。



**TIS INTEC**  
Group