



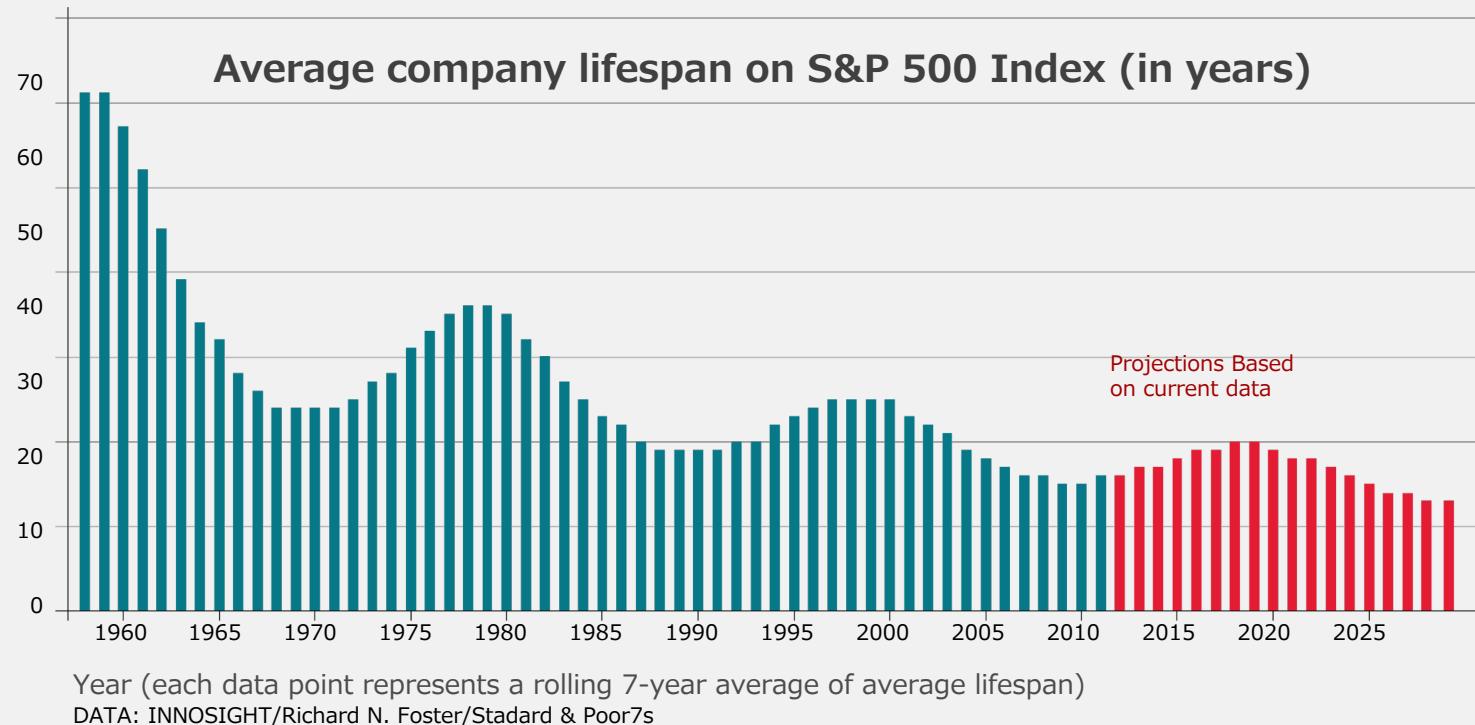
DevOps 1 Day Workshop

- shrink version -

2018/07/12

Yoshikazu YAMADA <yyamada@redhat.com>
Red Hat K.K. APAC DevOps Lead Senior Architect

S&P 500 における平均企業寿命の推移



2017 Disruptor 50 full coverage



Start Over 1-5 of 51

Meet the 2017 CNBC Disruptor 50 companies



Published 6:00 AM ET Tue, 16 May 2017 | Updated 2:14 PM ET Mon, 30 April 2018

CNBC

In the fifth annual Disruptor 50 list, CNBC features new companies from biotech to machine learning to transportation to retail — those innovations **are changing the world**. This year's thinking startups have identified unexploited niches in the market that have the potential to become billion-dollar businesses, they've used to fill them. Starting 31 at the top, that have already made it based on their business model. In the process, we're creating new ecosystems for their products and services. Creating corporate giants is no easy feat. But we ranked those venture capital-backed companies doing the best job. In aggregate, these 50 companies have raised nearly \$44 billion in venture capital at an implied Disruptor 50 market valuation of about \$239 billion, according to PitchBook data. Already it's hard to think of the world without them. Read more about the consumer and business trends that stand out in the 2017 list ranking and the **methodology** used to select this year's Disruptor companies.

1	Airbnb	It's a \$31 billion trip
2	Lyft	The car-ownership killer with a conscience
3	WeWork	Reworking the office
4	Grab	Uber-growth for an Asian ride-share rival
5	Uptake Technologies	Capturing Warren Buffett's billionaire energy
6	Houzz	The homiest e-catalog
7	Ginkgo Bioworks	Growing products in the lab
8	Palantir Technologies	Tracking the world's secrets

9	Cylance	Making cyberthreats idle
10	Udacity	Closing the skills gap
11	CrowdStrike	Going into the breach
12	23andMe	Bring your genome home
13	Progyny	Rocking the cradle
14	SpaceX	Humanity's interstellar escape plan
15	SurveyMonkey	Question everything
16	Ezetap	India's answer to Apple Pay
17	GreenSky	A credit to the mobile race
18	Myo	Measuring the muscle
19	Merck Serono	Getting real
20	Jaunt	VR that both Disney and Paul McCartney have experience in
21	IEX	The traders Michael Lewis made famous in a flash
22	GitHub	The biggest coding party in the world
23	Bloom Energy	Helping companies like Apple get off the grid
24	Drawbridge	An ad strategy Facebook and Google can't ignore
25	Jaunt	VR that both Disney and Paul McCartney have experience in
26	Coursera	Go to a top school, without going
27	MongoDB	The BIG idea in databases
28	Qualtrics	Surveying the corporate landscape
29	Domo	Complete cloud cover

30	Blippar	You, augmented
31	Pinterest	An image is worth \$1 billion
32	Illumio	A new segment in cybersecurity
33	Phononic	Quietly cool
34	Veniam	Constructing the global superhighway of data
35	Spotify	Not even Apple Music has slowed it
36	Dropbox	The file-sharing economy
37	Trulioo	Tracks twice as many people as Facebook: 4 billion, exactly
38	Svack	Who the IRS and DoD use against hackers
39	Location10	Signed, sealed, and strongly encrypted
40	Payoneer	Payments without borders
41	Skillz	A sport to surpass the NFL, with less injury risk
42	Blue Apron	What's for dinner
43	Robinhood	There is no brokerage fee low enough
44	Zocdoc	Real patient-centered health care
45	SoFi	\$18 billion in loans and counting
46	Foursquare	A success story turned inside out
47	Warby Parker	Still seeing things in new ways
48	Persado	A motivational speaker that's not human
49	Stripe	Visa is banking on this platform
50	Quid	The ultimate trendspotter

<https://www.cnbc.com/2017/05/16/the-2017-cnbc-disruptor-50-list-of-companies.html>

Digital Disruption の特徴

IT 駆動

- IT 技術を活用したビジネスモデル
- IT 中心の組織 (Value-Stream 全体に効果的に IT を活用可能なプロセスと組織・文化)

スピード

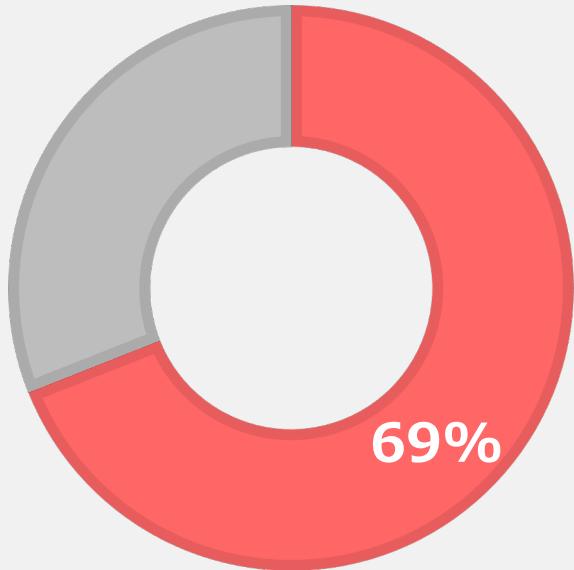
- IT 運営プロセスの効率化 と IT 活用コストの低下 による アイデア実現の高速化

顧客主導と顧客の進化

- 顧客が求める 価値・体験 主導
- 瞬時に拡散する顧客の評価主導 (マーケティング手法や顧客へのアプローチ方法の変化)
- 消費者は IT を駆使し入手した膨大な情報を活用し多様な選択肢から判断し即座に行動
- 消費者の要求は急速に 変化・多様化

業際的

- IT 技術を活用して既存産業に破壊的影响をもたらす新たなビジネスモデルを構築



日本の CEO の **69%** は 業界を超えた競争が激しくなる ことを見込んでいる。

(世界では56%)

どこの業界から競争相手が出現する可能性があると思うか？

1. テクノロジー (30%)
2. 小売り、卸販売 (25%)
3. 通信、エンターテイメント、メディア (20%)

参照：PWC:第18回世界CEO意識調査 日本分析版 (2015)





不確実性を考える。

ビジネスにおける不確実性

環境不確実性

- ・ マーケットの状況 (イノベーション、競合、顧客、etc.) はどう変化するのか？

目的不確実性

- ・ どんな プロダクト、サービス を作ればよいか？

方法不確実性

- ・ どのような 方法・手順 で作ればよいか？

通信不確実性 (コミュニケーション)

- ・ 理解の不確実性：人や事象の理解
- ・ 伝達の不確実性：コミュニケーションの不到達
- ・ 成果の不確実性：行動や成果物の期待とのギャップ



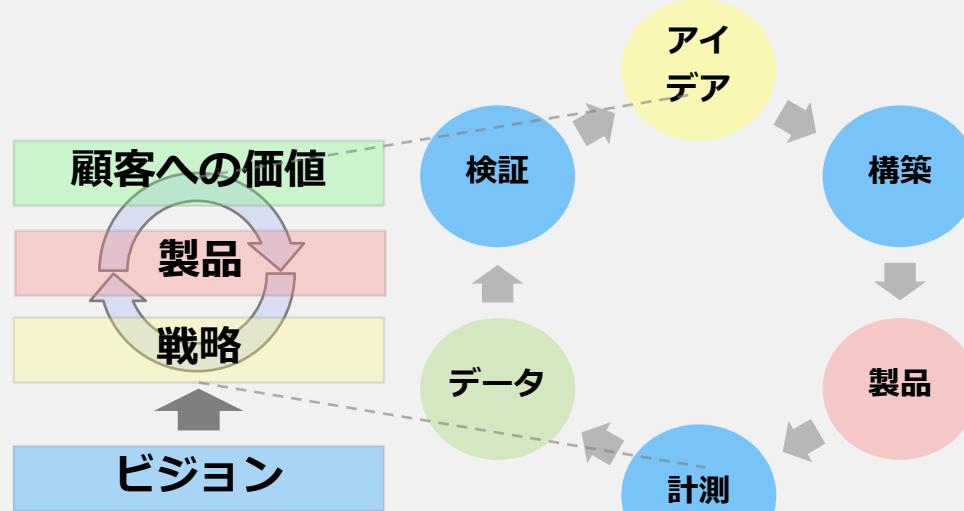
- ・ 情報の非対称性
- ・ 限定合理性



不確実性に対する 検証主導 アプローチ

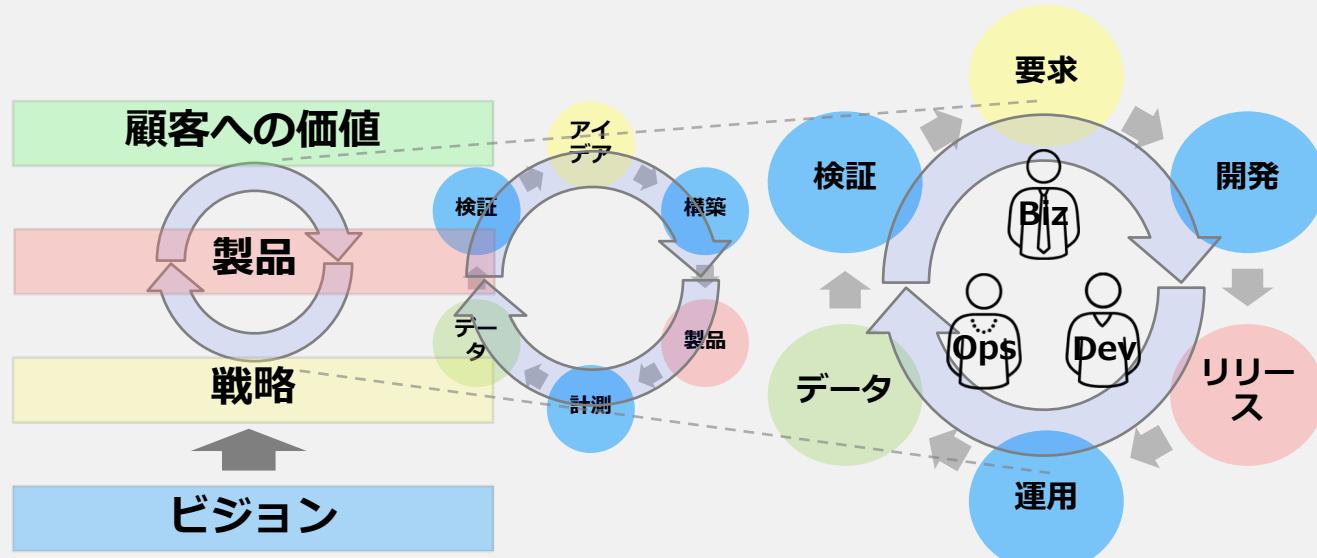
Lean Startup

- 検証と学び
- 戦略と製品の方向転換 (Pivot)
- 必要最小限のプロダクト (MVP)



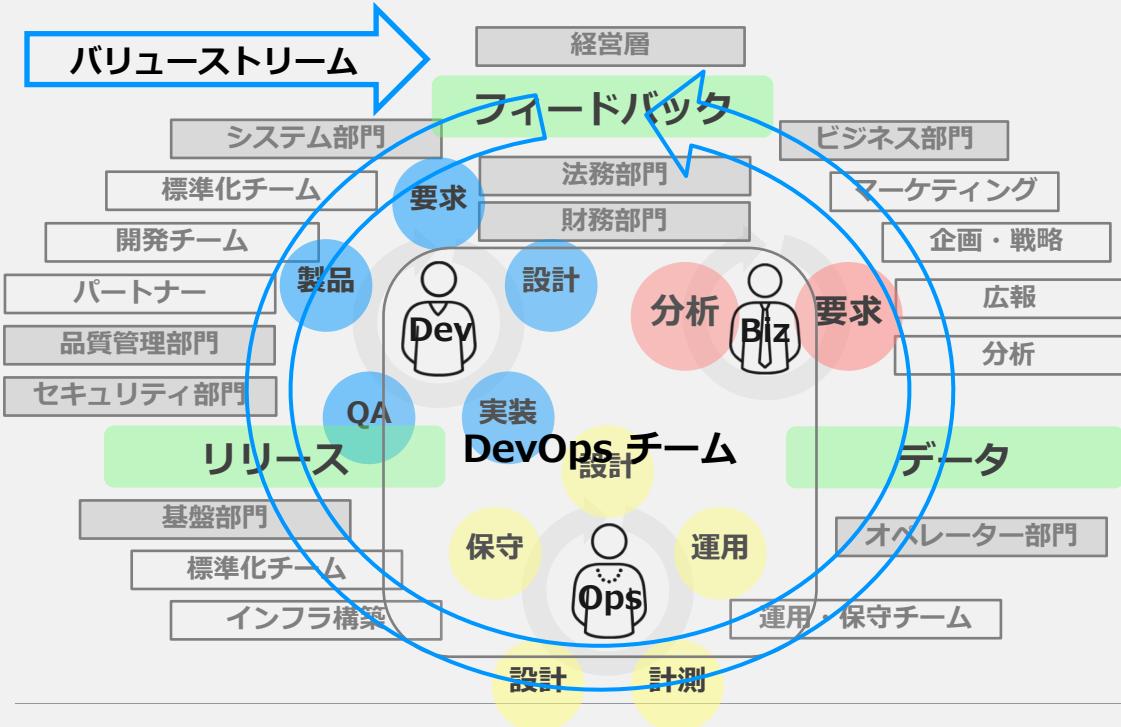
DevOps の定義

検証主導型製品開発のための プロセス、組織・文化、技術 を全社的（部門横断的）に確立



Value Stream

- ・ アイデアを実現しその価値を 提供・検証 できる状態にするまでの一連の流れ (Value Stream)
- ・ Value Stream の 効果性・効率性 を最大化するための プロセス、組織・文化、技術 の継続的改善



バリューストリームの最適化

- ・ プロダクトのリリースと継続的改善活動に関係する全社的(組織横断的)なフローの(バリューストリーム)整理と課題(ボトルネック等)の発見・解決

DevOps チームの構成と支援

- ・ バリューストリームの最適化において重要な改善要因となるプロセスを担当するメンバーから構成される DevOps チームを組織
- ・ DevOps チームを全社的(組織横断的)に支援する体制・プロセス・指針の確立

DevOps の起源

10 deploys per day
Dev & ops cooperation at Flickr

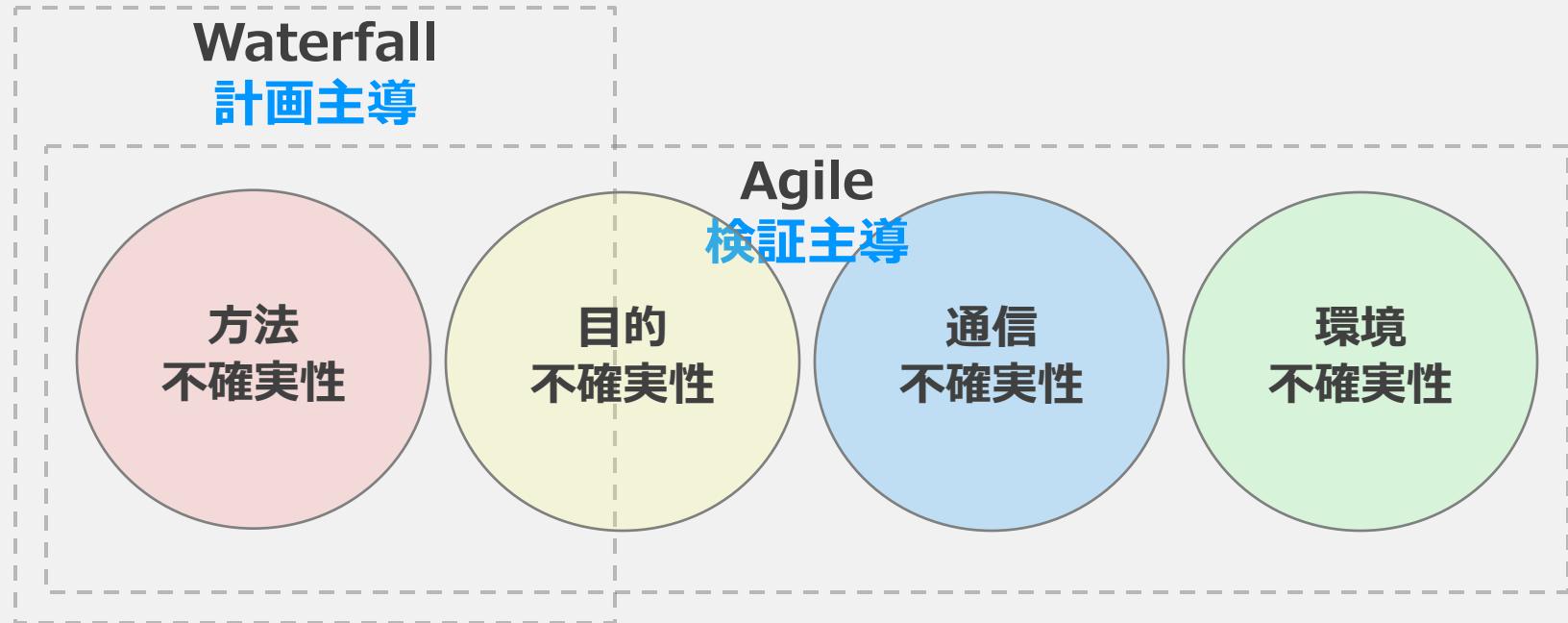
John Allspaw & Paul Hammond
Velocity 2009

10+ Deploys Per Day: Dev and Ops Cooperation at Flickr [1]

1. Automated Infrastructure
2. Shared version control
3. Feature flags
4. Shared metrics
5. IRC and IM robots

1. Respect
2. Trust
3. Healthy attitude about failure
4. Avoiding Blame

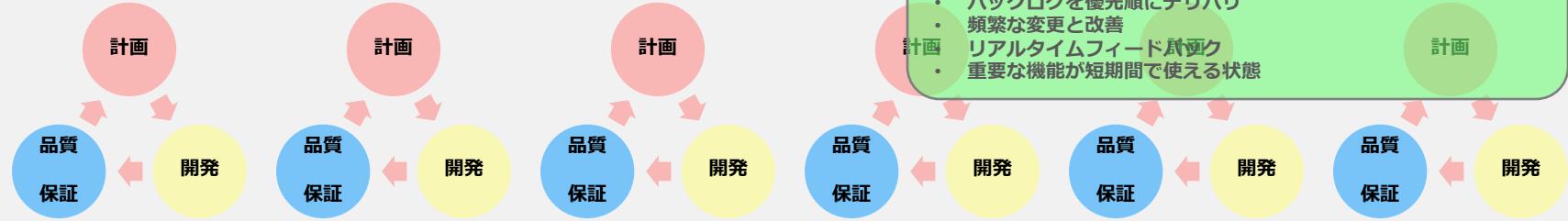
不確実性に関する Waterfall と Agile の関係



Agile と Waterfall の違い

アジャイル

- 反復的・継続的・漸進的・適応的なアプローチ



ウォーターフォール

- リソース、スコープ、スケジュールを固定するため、十分に事前計画を行う 計画的・予見的 アプローチ



アジャイルソフトウェア開発

歴史：アジャイルソフトウェア開発宣言

- 2001/02 に 17 名の著名なソフトウェアエンジニアが検討し文書化・公表 ([1], [2] 参照) したソフトウェア開発における原則（文化・価値観）

概要

- 特定の開発手法を指すのではなく「アジャイルソフトウェア開発宣言」をベースとした一群のソフトウェア開発手法を表す総称
※ Agile : 俊敏な、明敏な、頭の回転が早い、etc.
- 実際に動くソフトウェアを素早く提供しフィードバックと変更を実施（適応的、探索的）
- 短い反復単位（イテレーション）で 設計、実装、テスト、リリース を実施（反復的）
- イテレーションを繰り返すことによりソフトウェアの拡張と変更を実施（継続的、漸進的）

アジャイルソフトウェア開発手法

- スクラム
- XP（エクストリーム・プログラミング）、etc.

アジャイルソフトウェア開発宣言

私たちは、ソフトウェア開発の実践あるいは実践を手助けをする活動を通じて、よりよい開発方法を見つけだそうとしている。この活動を通して、私たちは以下の価値に至った。

プロセスやツールよりも**個人と対話を**、
包括的なドキュメントよりも**動くソフトウェアを**、
契約交渉よりも顧客との協調を、
計画に従うことよりも**変化への対応を**、

価値とする。すなわち、左記のことながらに価値があることを認めながらも、私たちは右記のことながらにより価値をおく。

Kent Beck	James Grenning	Robert C. Martin
Mike Beedle	Jim Highsmith	Steve Mellor
Arie van Bennekum	Andrew Hunt	Ken Schwaber
Alistair Cockburn	Ron Jeffries	Jeff Sutherland
Ward Cunningham	Jon Kern	Dave Thomas
Martin Fowler	Brian Marick	

© 2001, 上記の著者たち
この宣言は、この注意書きも含めた形で全文を含めることを条件に
自由にコピーしてよい。

アジャイルソフトウェア開発の特徴 - 1

動くソフトウェアを重視

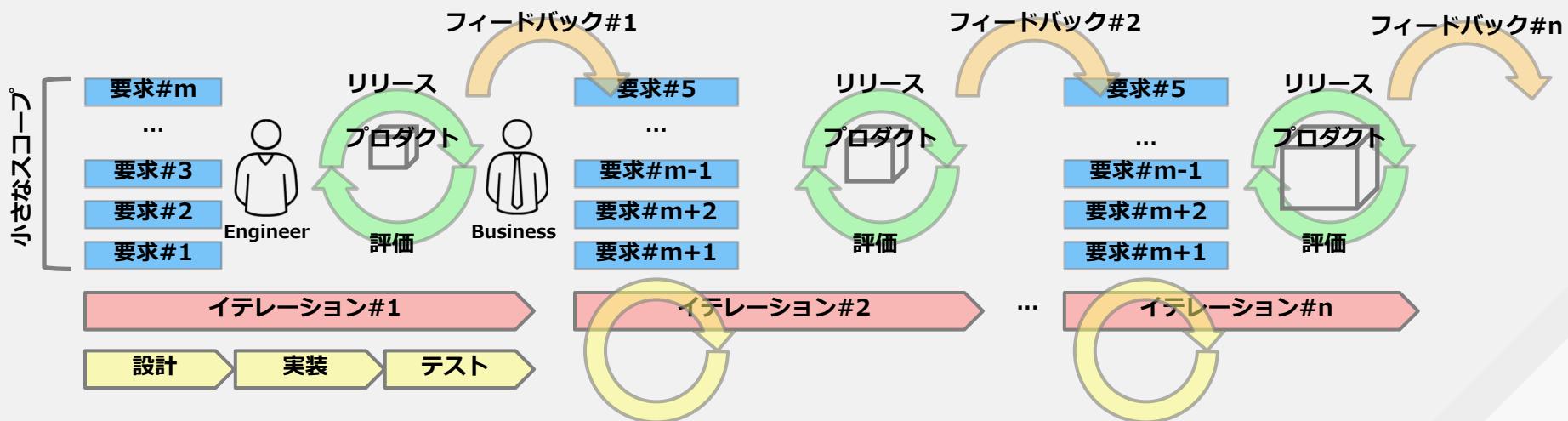
- ・動くソフトウェアを顧客に素早く提供、動くソフトウェアを対象にした評価とフィードバック

イテレーション

- ・定期で短い反復単位（イテレーション）で小さなスコープ（小さな機能セット）の設計、実装、テスト、リリースを行い、リリース可能なプロダクトを反復的・継続的に提供

変化への対応と改善

- ・イテレーション毎にプロダクトの改善・変更要求の取込を実施 ⇒ 漸進的・適応的・探索的な改善



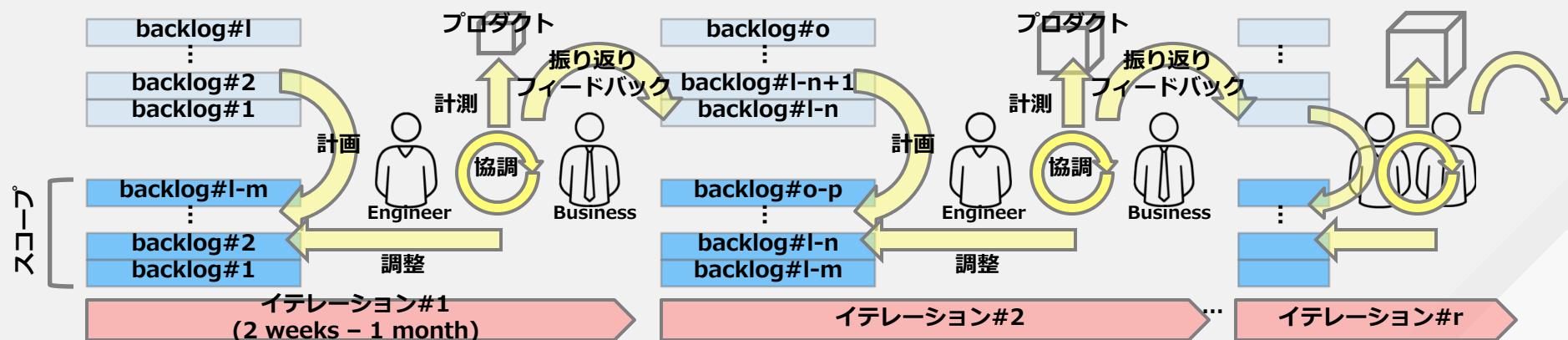
アジャイルソフトウェア開発の特徴 - 2

組織・文化

- ビジネス担当とエンジニアが日々同一の環境（近い距離）で（上下関係なく）協調し、チーム全員参加（Face to Face）によるプロダクト・課題・タスク管理を実施、変更の歓迎と変化によるプロダクトの価値向上

マネジメントモデル

- スコープ、ゴール：プロダクトに対する要求事項をバックログとして管理
- スケジュール：定期の固定されたイテレーション（2週間～1ヶ月）単位毎にバックログに対応
- リソース：固定されたリソース（通常10人未満のビジネス担当とエンジニアから構成されるチーム）
- 計画：イテレーション開始時に優先度に従いイテレーションで対応可能なバックログを見積
- 計測：動くソフトウェアを対象にチーム全員で実施
- 調整、フィードバック：イテレーション開始時の“振り返り”によるバックログの管理（追加/削除・優先度設定）



ソフトウェア開発における困難

不可視性

- ・ S/W の 実際の 外見・動作・得られる体験・効果 を**予め可視化することの困難さ**
- ・ S/W の 設計工程 における中間生成物(設計書)による**最終成果物をイメージすることの困難さ**

単一性

- ・ (SaaS、パッケージ製品 等を除く) S/W は**それぞれがただ一つの構造物**

変更性

- ・ (H/W 等に比べ) S/W は 環境・要求 の変化 に対して**容易に変更可能**であるべき
- ・ S/W をとりまく**環境(ビジネス、市場)の変化の早さ**
- ・ S/W に求められる**変化の圧力・頻度・対応速度** の高さ

複雑性

- ・ S/W の**構造と動作の複雑さ**
- ・ S/W の**開発工程の複雑さ**

不確実性(非予見的)

- ・ **目的(What)と手段(How)の不確実性**
- ・ S/W ライフサイクルにおける**変更要求を予測することの困難さ**
- ・ S/W ライフサイクルにおけるその**構造や統合の最適化を行うことの困難さ**
- ・ S/W の**開発計画(スコープ、スケジュール、リソース)を予測することの困難さ**

アジャイルのアプローチ

Why : ソフトウェア開発の困難

- ・ 不可視性
- ・ 単一性
- ・ 変更性
- ・ 複雑性
- ・ 不確実性 (非予見的)

How : 困難への対応

- ・ 予見性を前向きに否定
- ・ 実際に動くソフトウェアを素早く提供
- ・ 実際に動くソフトウェアで評価
- ・ 反復的・継続的・漸進的・適応的な開発手法
- ・ 短い反復単位で開発とリリース
- ・ 短い反復単位と小さなスコープと優先度
- ・ ビジネス担当者とエンジニアの協調
- ・ 反復的・継続的・漸進的・適応的な開発工程

ウォーターフォール・モデル（開発）

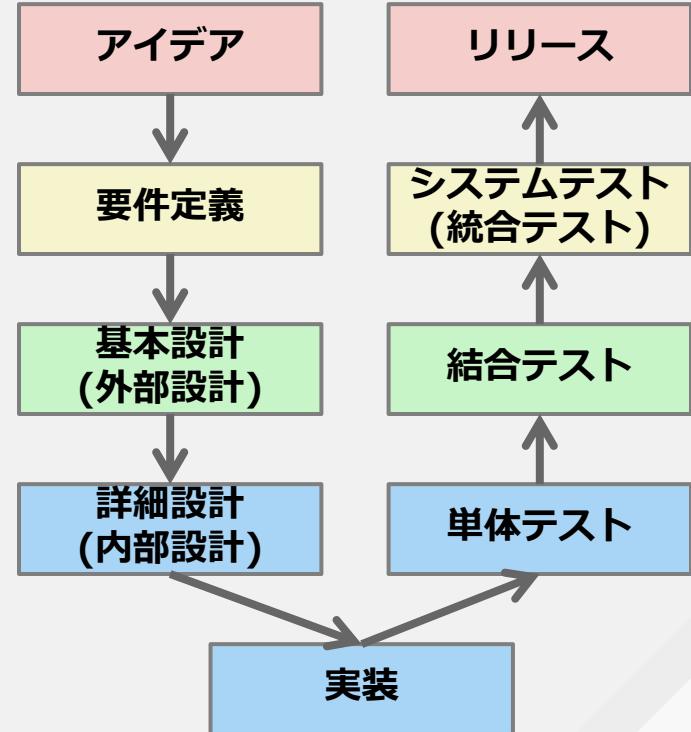
歴史

- 「ウォーターフォール・モデル」という用語は、文字通り「滝」を意味し、W.W.ロイスによって1970年に発表された論文「Managing the Development of Large Software Systems」の内容が元になったとされる。この論文において、「大規模ソフトウェア開発には、製品製造過程のようにいくつかの工程に分けたトップダウンアプローチが必要」と述べている。しかし論文には「ウォーターフォール・モデル」という記述は無く、また、前工程への後戻り（見直し）も提唱されており、元の論文の内容とは異なっている。
- 初めて「ウォーターフォール」という用語を用いたのはT.E.BellとT.A.Thayerによる1976年に発表された論文「Software Requirement」であり、B.W.Boehmが1981年に出版した本「Software Engineering Economics」においてウォーターフォールモデルのオリジナルはRoyceだと述べ、ウォーターフォール・モデルの起源がRoyceであるという誤解を広めた。

[ウォーターフォール・モデル (Jul. 30, 2016, 15:34 UTC). In Wikipedia: The Free Encyclopedia. Retrieved from <https://ja.wikipedia.org/wiki/ウォーターフォール・モデル>]

概要

- 予見的・計画的アプローチ
 - 計画工程においてスコープを予見的に定義
 - スコープからスケジュール・リソースを計画
- 各工程を完了（仕様書・ドキュメントの品質検査等）しぬの工程に進む（※ある工程は全行程が完了していることが前提で一回のみの実施）
- アイデア～リリースが長期（実際に動くS/Wは後工程で提供）
- 設計工程以降の“要求変更”は工程の後戻りとなる



ウォーターフォール・モデルの課題

ソフトウェア開発の困難への対応

- 予見性の前提に基づく計画性という非現実的なアプローチ
 - 見積りが不正確、進捗が不透明、期限遅れ、テストの短縮
 - チームの肥大化と生産性の低下

要求仕様抽出の効率性と“ムダ”な機能実装（機能の肥大化、投資の損失）

- ドキュメントからプロダクトを予想することは現実的に困難
- 変更が現実的に不可能 -> 要求仕様の詰め込み
- 実質的なフィードバックは開発の後工程

開発期間の長期化

- アイデアをリリースする段階でビジネス環境が変化
- 顧客価値（動く S/W）の長期化、開発中の仕掛品（在庫）

変化への対応性・俊敏性

- 設計フェーズ以降の変更は現実的に不可能

ビジネス担当者とエンジニアの分断

- ビジネス担当とエンジニアのゴールの不一致

統合と不具合の検知

- 後工程における大規模な統合、統合で具現化する不具合
- 不具合の作り込み～検知の時間の長期化

全ての IT 運営においてウォーターフォール・モデルが抱える問題ではなく web 上の IT ビジネスの開発プロセスとしてのウォーターフォール・モデルの課題であることに注意！！

ウォーターフォール・モデルの課題 - 統合と不具合の検知

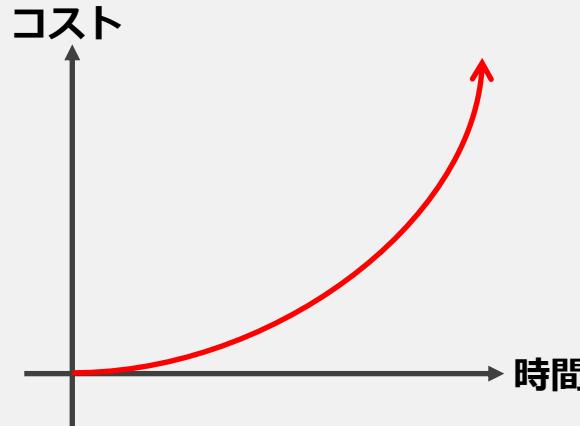
後工程での大規模な統合

- ・後工程でのシステム統合 (コンパイル、ビルド) と動作検証による**想定外の事象・不具合の頻出**

不具合の検知と修正

- ・大規模な統合に対する**不具合の検出・原因の特性・修正の効果性・効率性の低下**
- ・不具合の作り込み～検知の長期化による**不具合の原因の特定・修正の効果性・効率性の低下**
- ・**不具合の修正における手戻り**

不具合の修正に関する時間とコストの関係



アジャイルの開発手法

Scrum (スクラム)

- ・プロセスに関する手法
- ・アジャイル開発手法として最も普及

XP (eXtrem Programming)

- ・開発の仕方に関する手法 (プラクティス)
- ・概要
 - ・ケント・ベックらによって定式化され、提唱されているソフトウェア開発手法、1999年に書籍『XPエクストリーム・プログラミング入門—ソフトウェア開発の究極の手法』によって発表。XPは、軽量開発手法あるいはアジャイルソフトウェア開発手法と呼ばれる、同種の開発手法のなかで代表的
 - ・継続的インテグレーション (CI)、テスト駆動開発、ペアプログラミング、リファクタリング、YAGNI 原則が普及
- ・アジャイルを実践するにあたり CI (Continuous Integration) は必須

Scrum における“品質”の考え方

品質区分	製品品質	技術品質
品質の考え方	<p>プロダクトの価値</p> <ul style="list-style-type: none">・(機能)要求/要件の充足度・Market Value(市場価値)・製品/サービスの魅力や有用性	提供者が出荷時に担保すべき機能横断的な価値 <ul style="list-style-type: none">・(非機能)要求/要件の充足度・製品/サービスの安全性や信頼性
Scrum のルール	<ul style="list-style-type: none">・Acceptance Criteria・Product Backlog (User Story)	<ul style="list-style-type: none">・Definition of Done
例	<ul style="list-style-type: none">・食品のおいしさ	<ul style="list-style-type: none">・食品の安全性

用語の定義

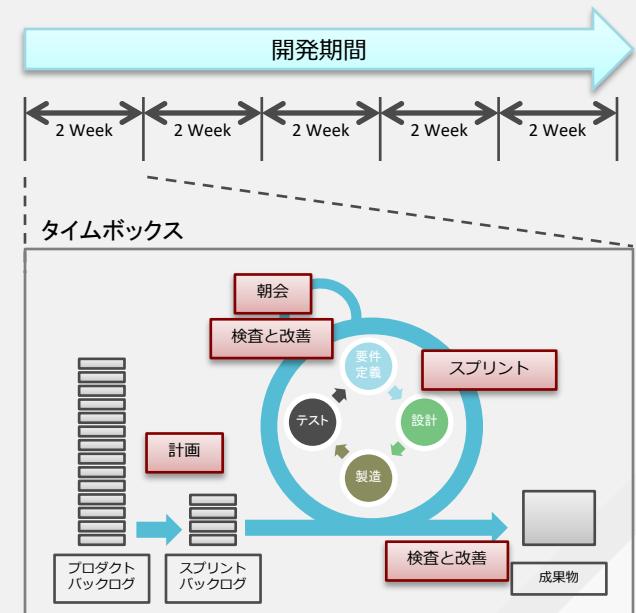
- ・ **Acceptance Criteria : 受け入れ条件**
 - ・個々の Backlog のアウトプットが満たさなければならない(機能要求関連の)条件※個々の Backlog で異なる。
- ・ **Definition of Done (DoD) : 完了の定義**
 - ・出荷可能なプロダクトするために Backlog 共通で満たさなければならない(非機能要求関連の)条件※全ての DoD を完了しなければ出荷可能とはならない。Sprint で対応できなかった条件は Undone として管理(以降の Sprint の負債)される。

スクラム概要

スクラムガイドによる定義

複雑で変化の激しい問題に対応するためのフレームワークであり、可能な限り価値の高いプロダクトを生産的かつ創造的に届けるための物である

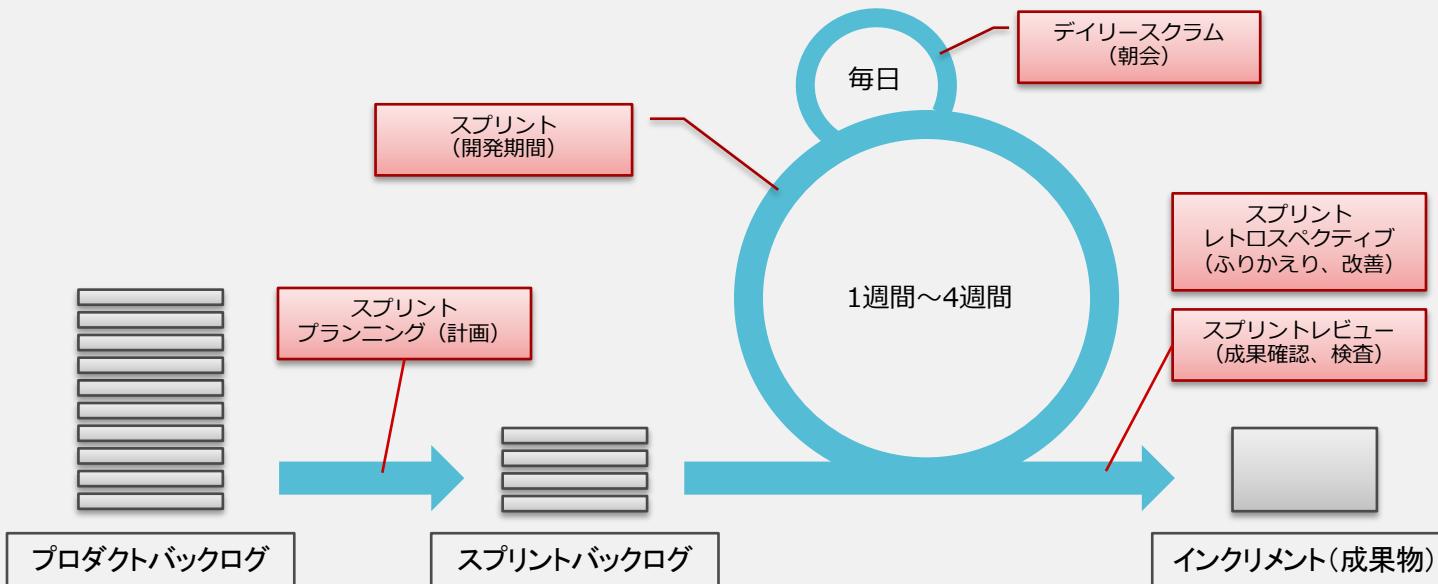
- 固定の短い時間（タイムボックス）に区切って作業を進める
- 要求の価値やリスク、必要性などを考慮した優先順位をつけ、その順番でプロダクトを作ることで価値を最大化する
- 現在の状況や問題点を常に明らかにするプロジェクトの透明性を重視する
- 進捗状況や作成されているプロダクトが正しいのか、仕事の進め方に問題が無いかどうかを定期的に検査し、継続的な改善を実施する
- スクラムチームと呼ばれるチーム単位で活動する
 - チームにはプロダクトオーナー、開発チーム、スクラムマスターという3つのロールがある
- スプリントで実施される開発手法については、定義はしていない



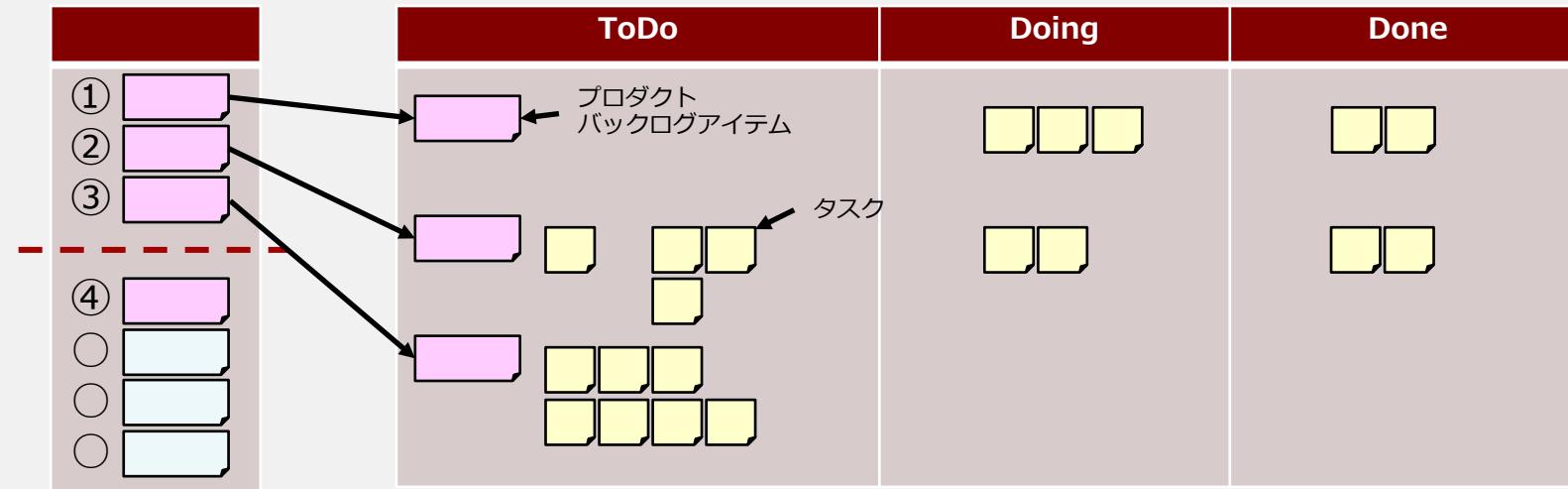
スクラムイベント

スクラムでは、繰り返されるプロセスを5つのイベントとして定義している。

スクラムイベントを通じてプロダクトの状態やチームの現状などの情報の透過性を重視し、スクラムチーム全員での共通理解を得ることが重要となる。



プロダクトバックログ&スプリントバックログ



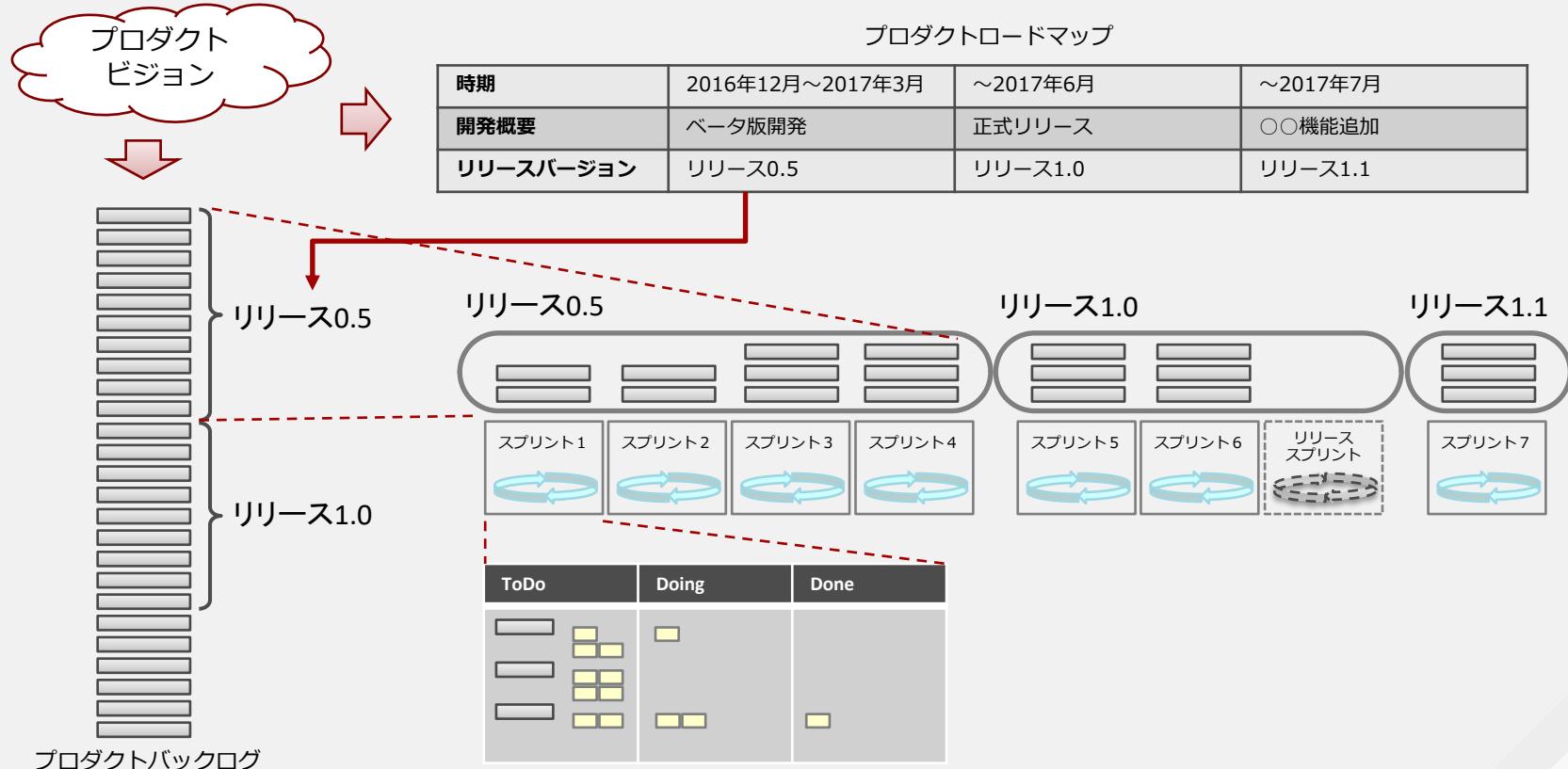
プロダクトバックログ

プロダクトで実現したいことを優先順位を付けて一覧化したもの。
機能の追加や修正、ユーザーの要望などが含まれる。
解決すべき技術課題などを含める場合もある。
プロダクトオーナーがその内容と優先順位に責任を持つ

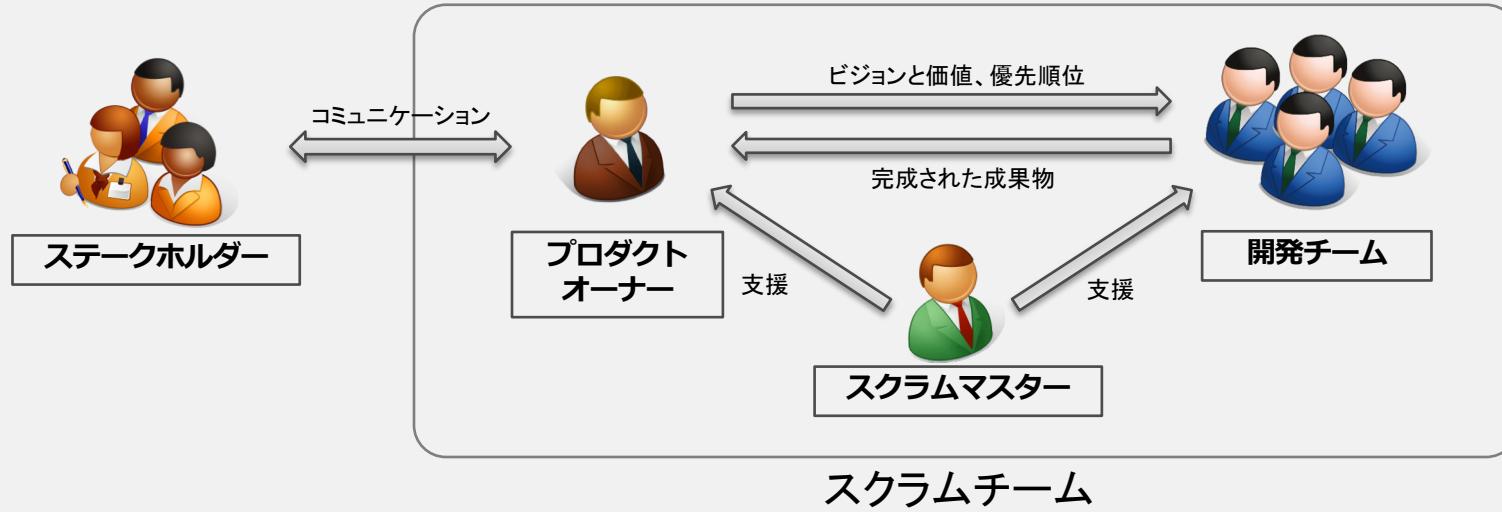
スprintバックログ

スprint期間内で行うと判断したプロダクトバックログのアイテムと、それを実現するためのタスクを俯瞰できるように表したもの
開発チームがその内容に責任を持つ

スクラム開発におけるリリース



スクラムチーム



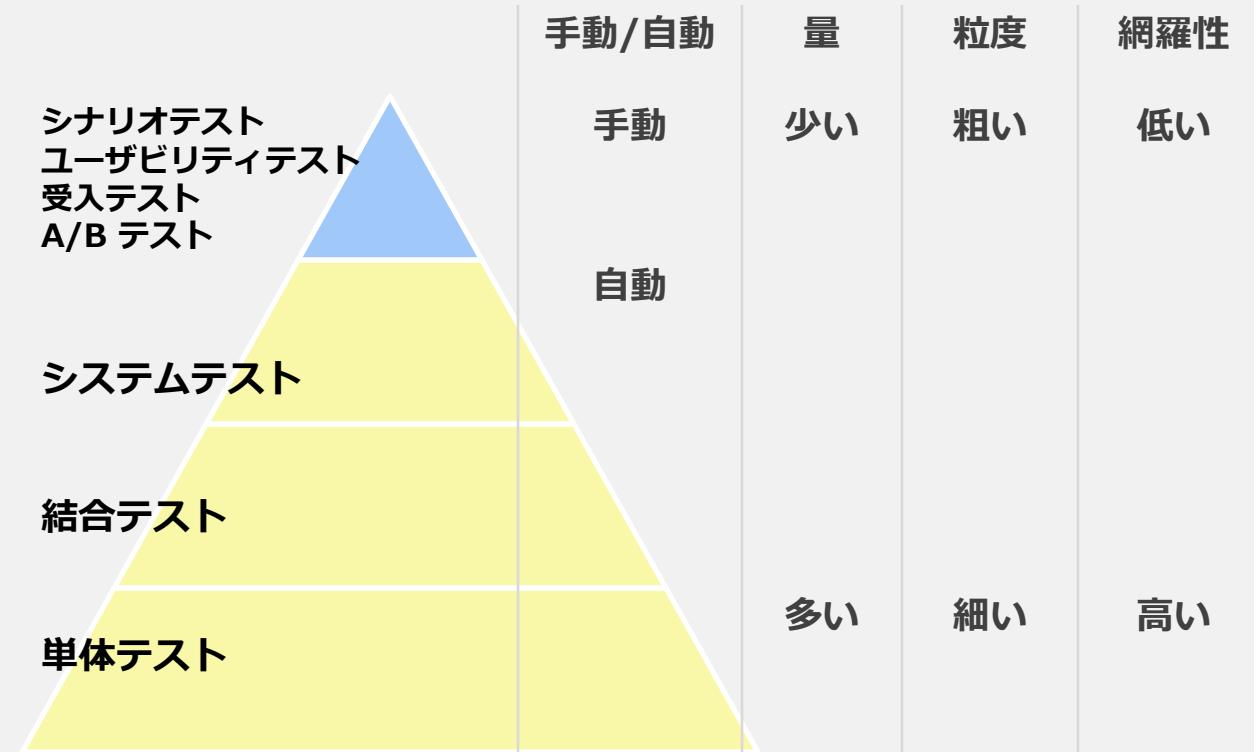
自己組織化するチーム

価値あるプロダクトをプロダクトを作る前提として、スクラムチームには「**自己組織化**」されていることが求められる。

スクラムガイドによる定義では「**自己組織化チームは、作業を成し遂げるための最善の策を、チーム外からの指示ではなく、自分たちで選択する**」とされている

スクラムチームのメンバー全員がチームの理想とする姿を考え、その理想に向かって**能動的に学習と成長を続けている状態**がスクラムにおける自己組織化

テスト自動化のピラミッド



ソフトウェアテストにおける観点とテストの体系

品質の観点		動作・構造 の観点		テスト体系	テスト技法
品質特性	外部品質 (⇒ 要求)	全体の動作		シナリオテスト ユーザビリティテスト 受入テスト A/B テスト システムテスト	動的テスト <ul style="list-style-type: none">• ブラックボックステスト<ul style="list-style-type: none">• 同値クラステスト• 境界値テスト• デシジョンテーブルテスト• ペア構成テスト• 状態遷移テスト• ドメイン分析テスト• ユースケーステスト• ホワイトボックステスト<ul style="list-style-type: none">• 制御フローテスト• データフローテスト• 探索的テスト
	内部品質 (⇒ 要件)	部分の動作	動的構造	結合テスト 単体テスト	
		静的構造		静的テスト • ソースコード静的解析	

補足

再帰テスト？

テストが自動化されたプロダクトではテスト分類の観点の再帰テストは不要



THANK YOU



plus.google.com/+RedHat



facebook.com/redhatinc



linkedin.com/company/red-hat



twitter.com/RedHatNews



youtube.com/user/RedHatVideos

DevOps と 技術・ツール

DevOps が 技術・ツール に求めるもの

DevOps と 技術・ツール

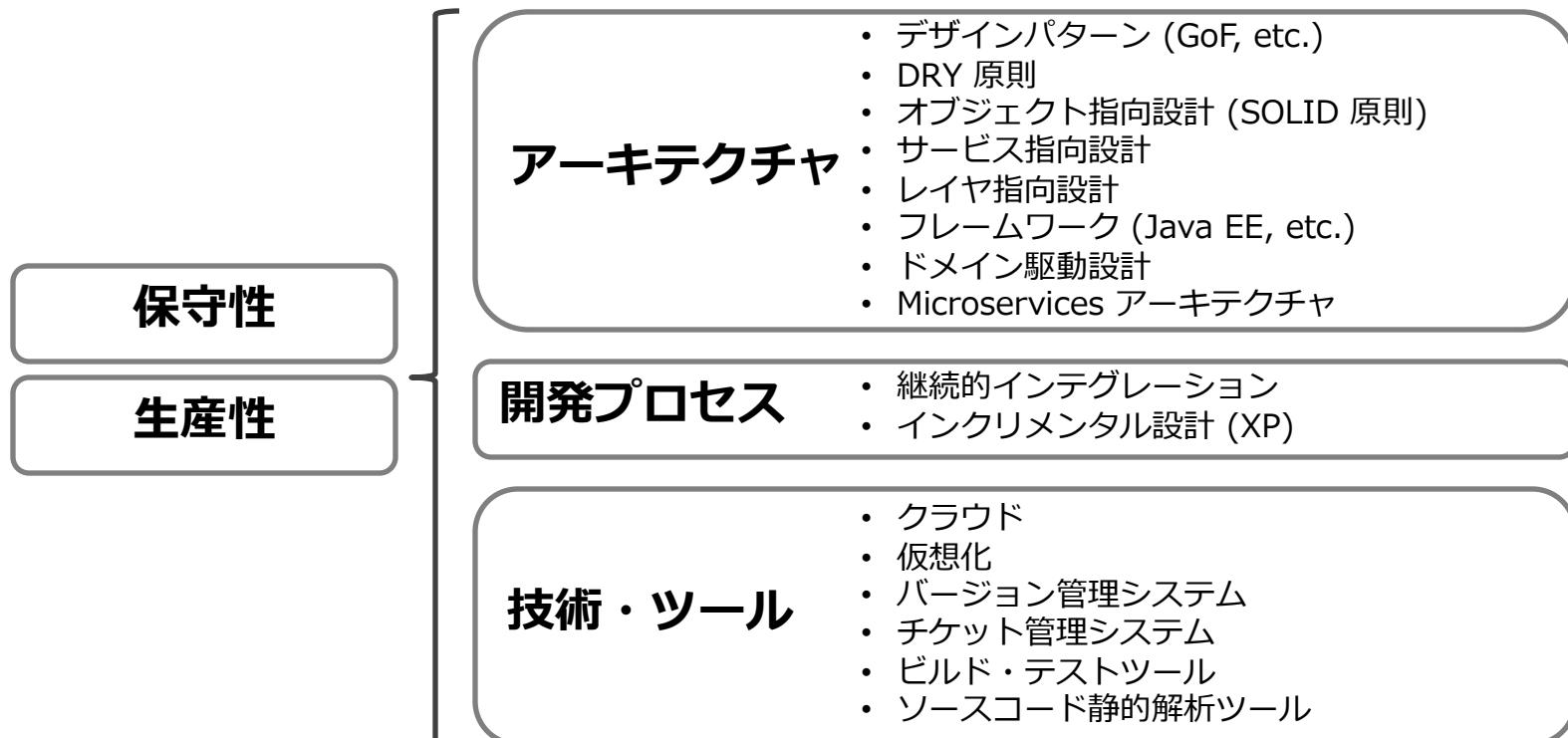
適切な 品質、コスト を伴った

- 繙続的変更 (継続的改善) の実現
- 変更スピード (リードタイム短縮) の向上
- チーム・プロセス への適合

継続的変更・変更スピードに関連する品質特性

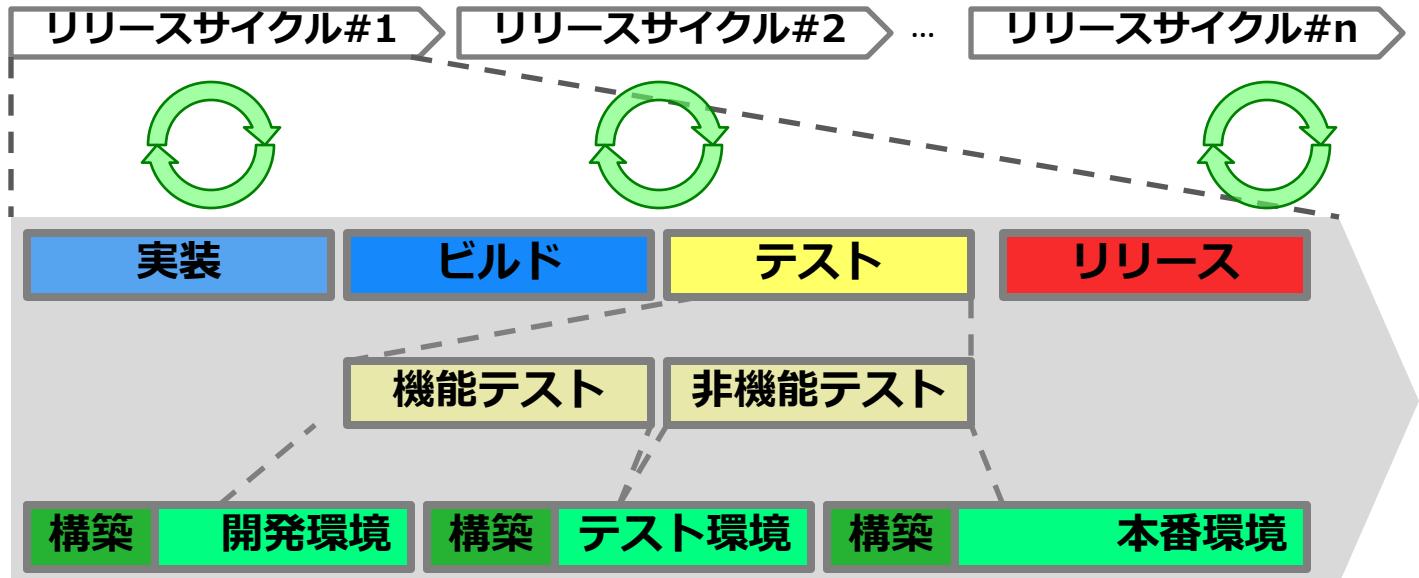
保守性	モジュール性	変更の影響範囲が局所化されている度合いやモジュール間の依存性の度合
	一貫性	記法・用語・概念が一貫していること
	再利用性	ソフトウェアコンポーネントが他のシステムを構築する際に利用できる度合
	解析性	障害・変更箇所の識別のし易さや変更の影響範囲の特定のし易さの度合
生産性	変更容易性	欠陥や品質の低下なく変更が効果的・効率的に行える度合
	テスト容易性	テストポリシー・テスト評価・テスト実装の効果性・効率性の度合
	可読性	ソースコードを読む際の、その目的や処理の流れの理解のし易さの度合
	簡潔性	実行されない・冗長性・複雑性の少なさの度合

品質特性を担保するためのアプローチ



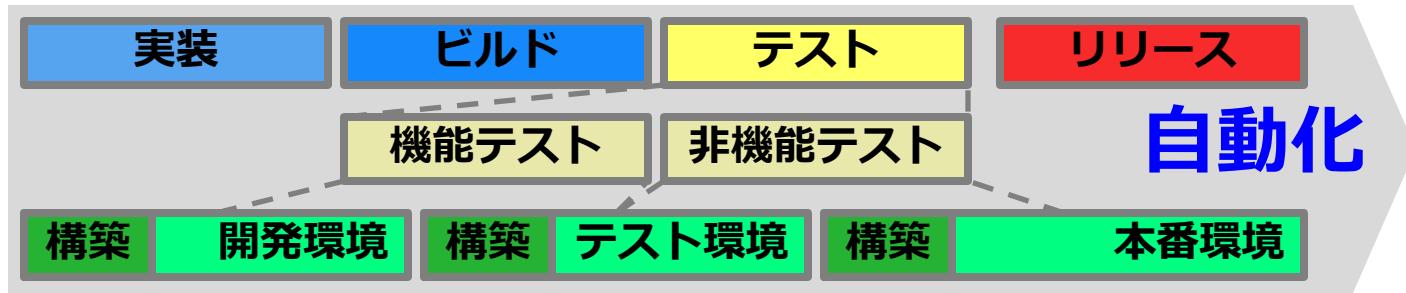
システムインテグレーション

DevOps と 技術・ツール



CI/CD パイプライン

DevOps と 技術・ツール



繰返し行われる手順の**自動化**

- ・ ビルド・構築 の 保守性・生産性 の向上
- ・ テスト・リリース の 保守性・生産性 の向上



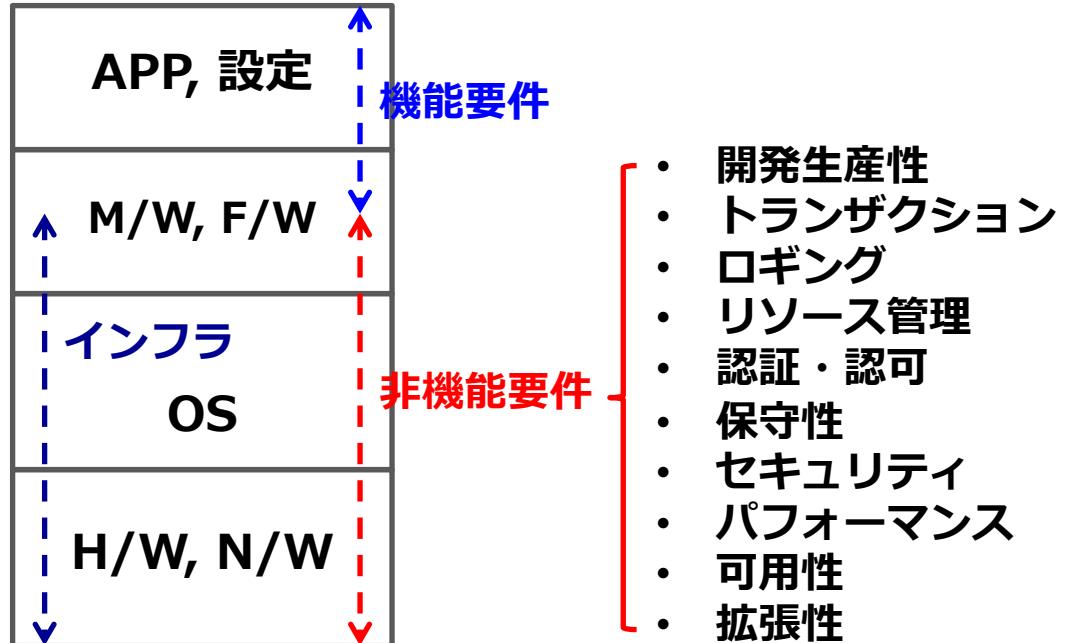
継続的変更の効率と変更スピードの向上

自動化とインフラレイヤの特性

DevOps と 技術・ツール

インフラレイヤの特性

- ・ 構築のコード化：難
- ・ 非機能要件のコード化：難
- ・ 環境依存性：高
- ・ 移植性：低
- ・ テスト容易性・効率性：低
- ・ 構成管理の容易性：低



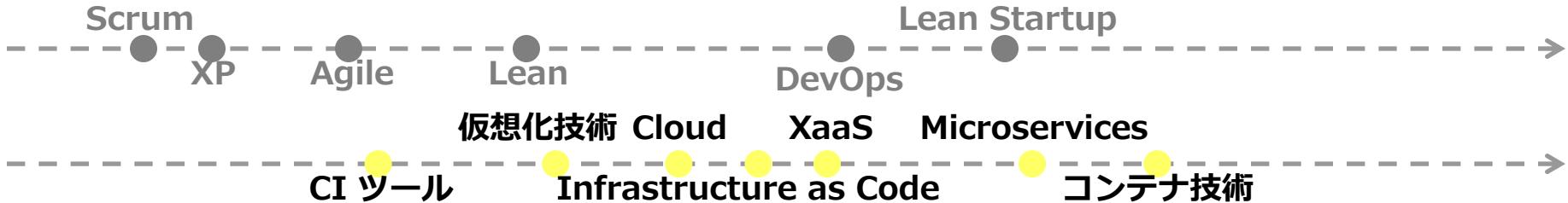
自動化における技術的課題

DevOps と 技術・ツール

- ・ インフラ構築の自動化
- ・ インフラ要件のコード化
- ・ インフラに依存するテストの自動化
- ・ インフラの構成管理の効率化

DevOps 周辺の歴史

DevOps と 技術・ツール



コンテナ

- Docker
- Kubernetes
- OpenShift

インフラの自動化

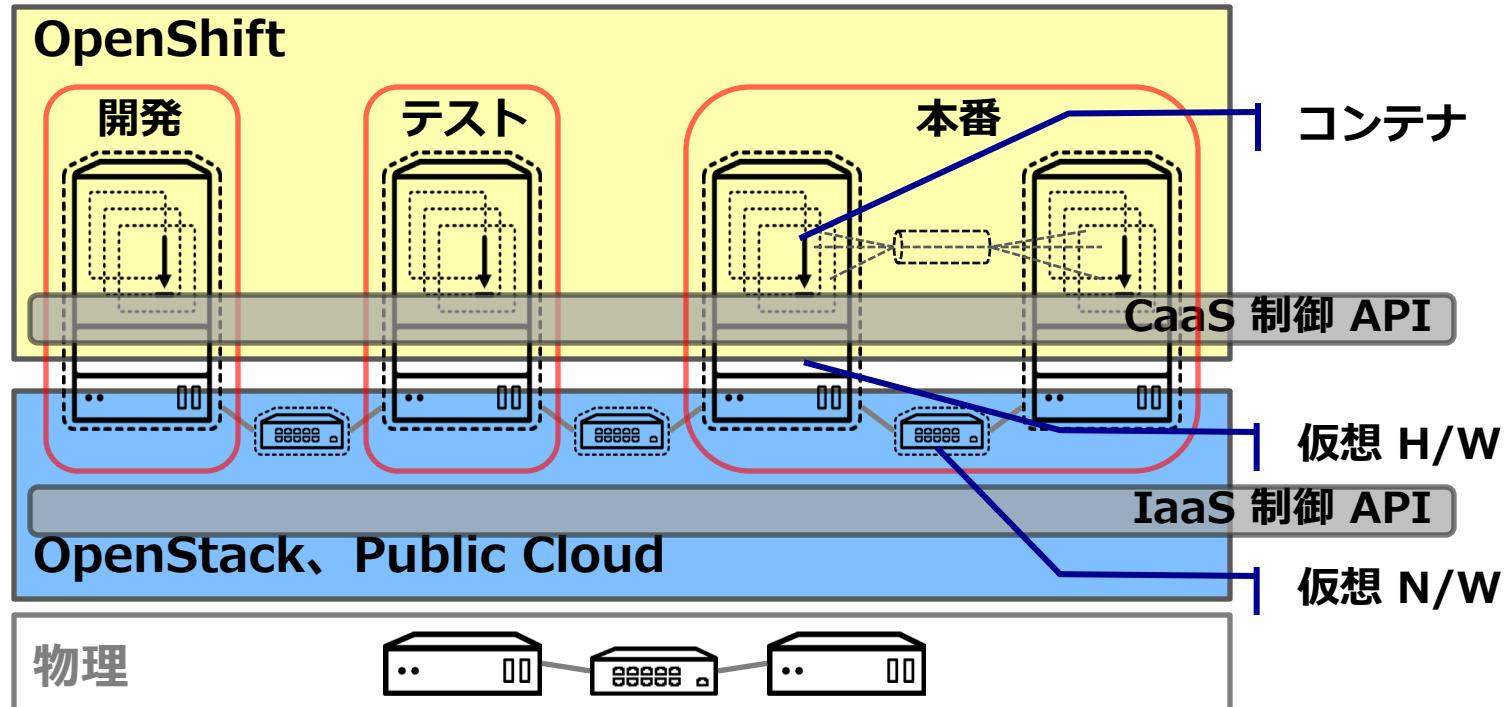
- Ansible
- OpenStack

その他

- Git, Jenkins, Serverspec

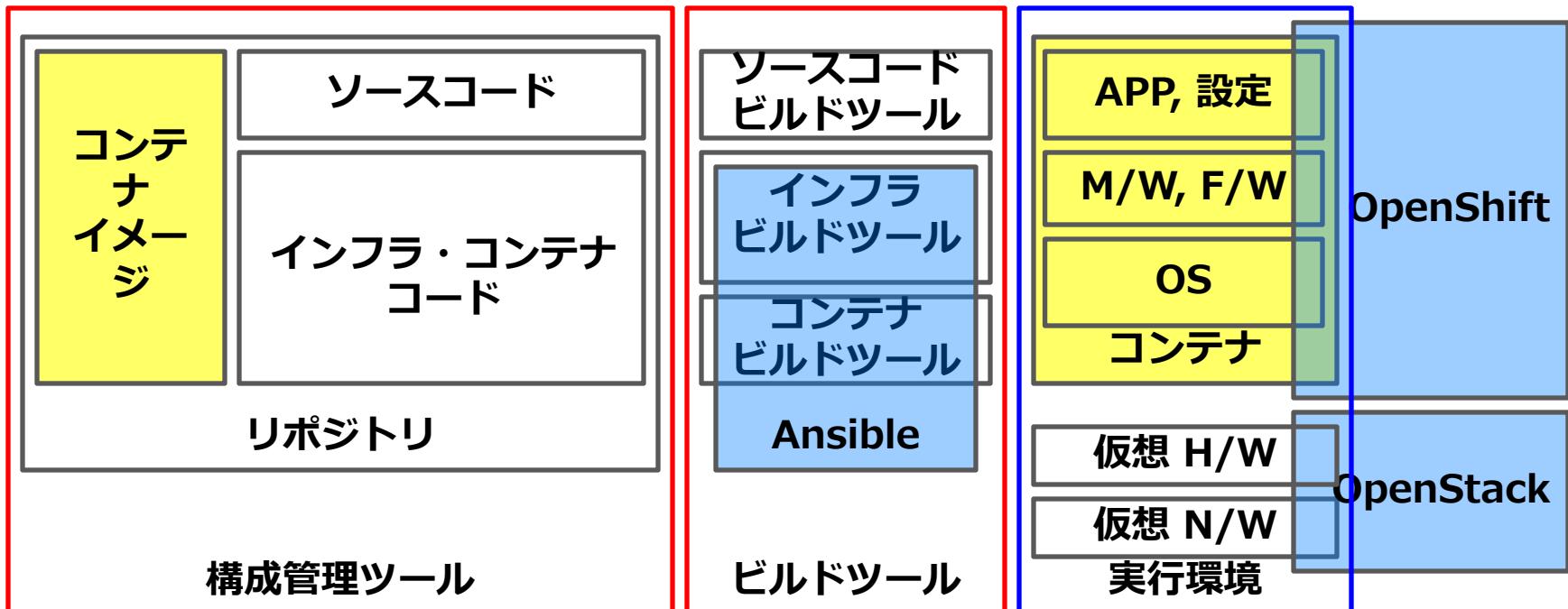
DevOps なインフラ

DevOps と 技術・ツール

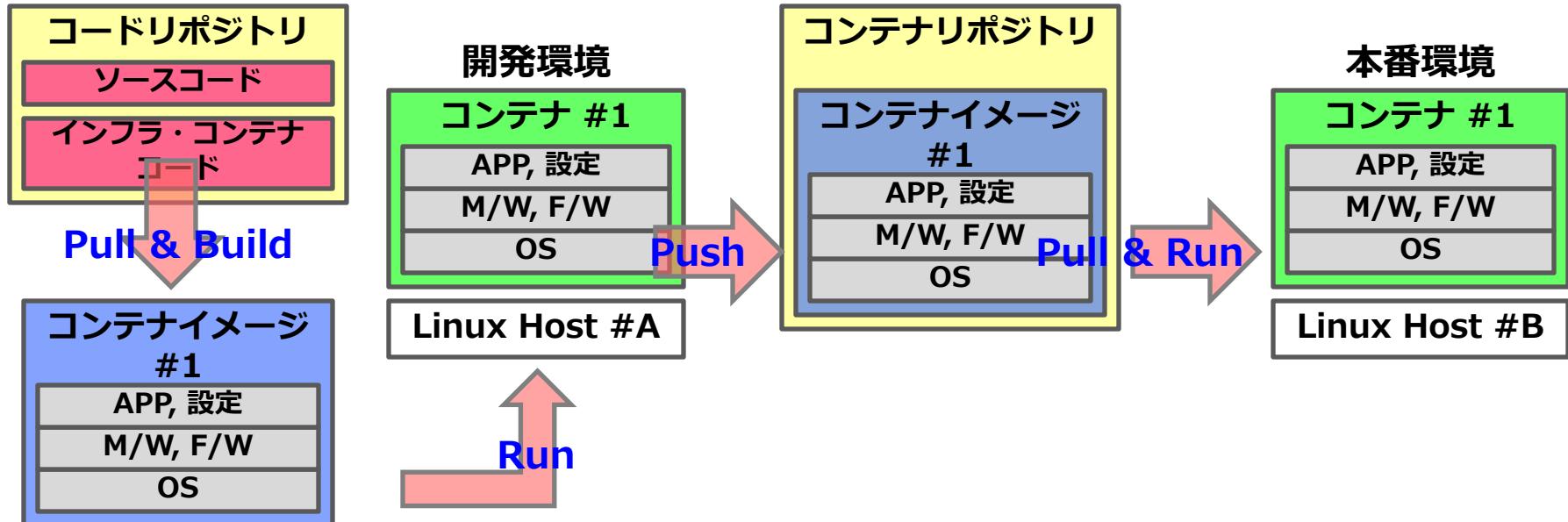


DevOps な 構成管理・ビルド

DevOps と 技術・ツール



コンテナ技術の特徴



- システムをコンテナイイメージとしてパッケージング
- コンテナイイメージはコンテナリポジトリ上で 名前:タグ名 で管理可能
- ホストの異なる任意のコンテナ環境でコンテナイイメージを実行可能

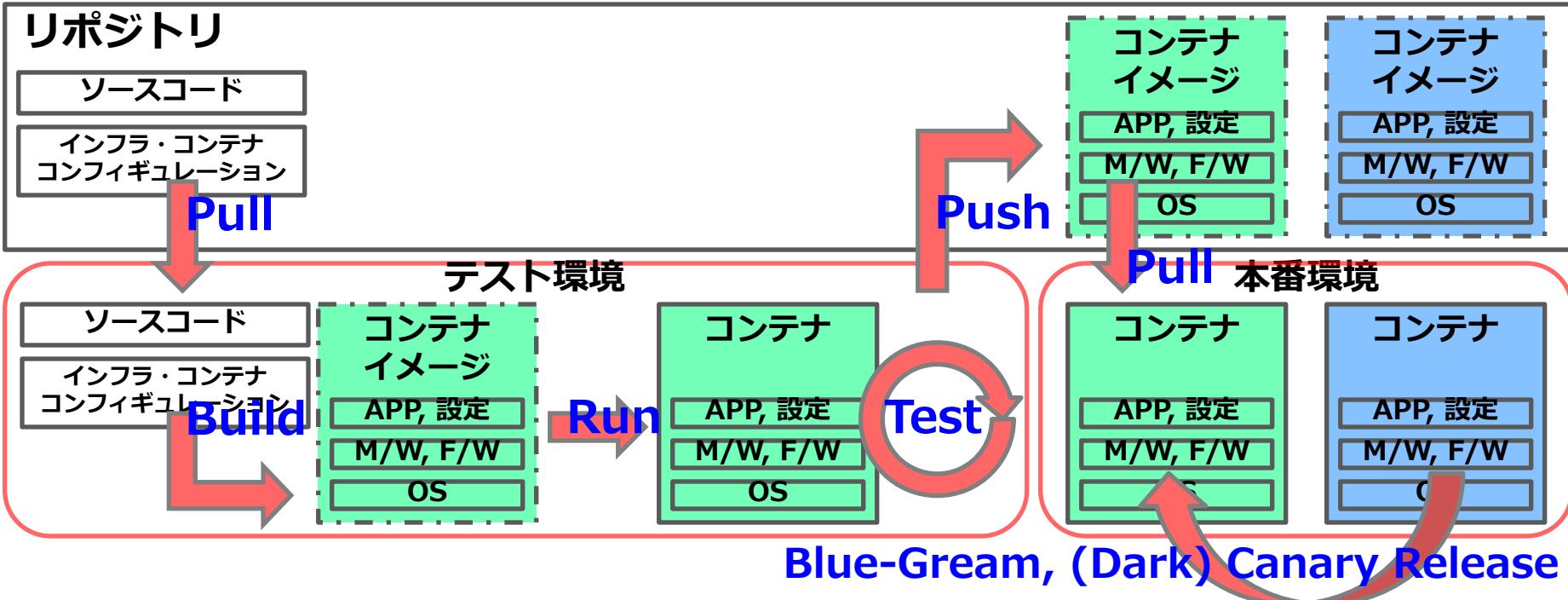
DevOps なインフラの特徴

DevOps と 技術・ツール

- ・ インフラ構築 の コード化・自動化
- ・ インフラ要件 の コード化
 - ・ 制御 API を使用した 非機能要件 の コード化
- ・ インフラ要件 の テストの自動化
- ・ システム の 高い移植性 による 保証性の高いテスト
 - ・ コンテナイメージの push&pull via コンテナリポジトリ
- ・ インフラ の 構成管理 の 効率化

DevOps な CI/CD

DevOps と 技術・ツール



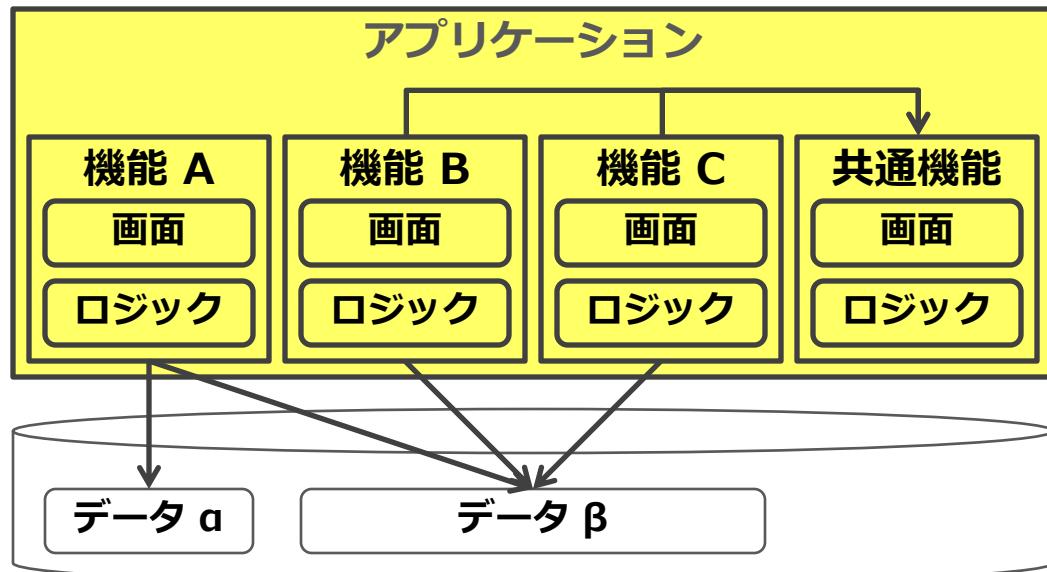
システムの“変更のしやすさ”とアーキテクチャ

DevOps と アーキテクチャ



DevOps と アーキテクチャ

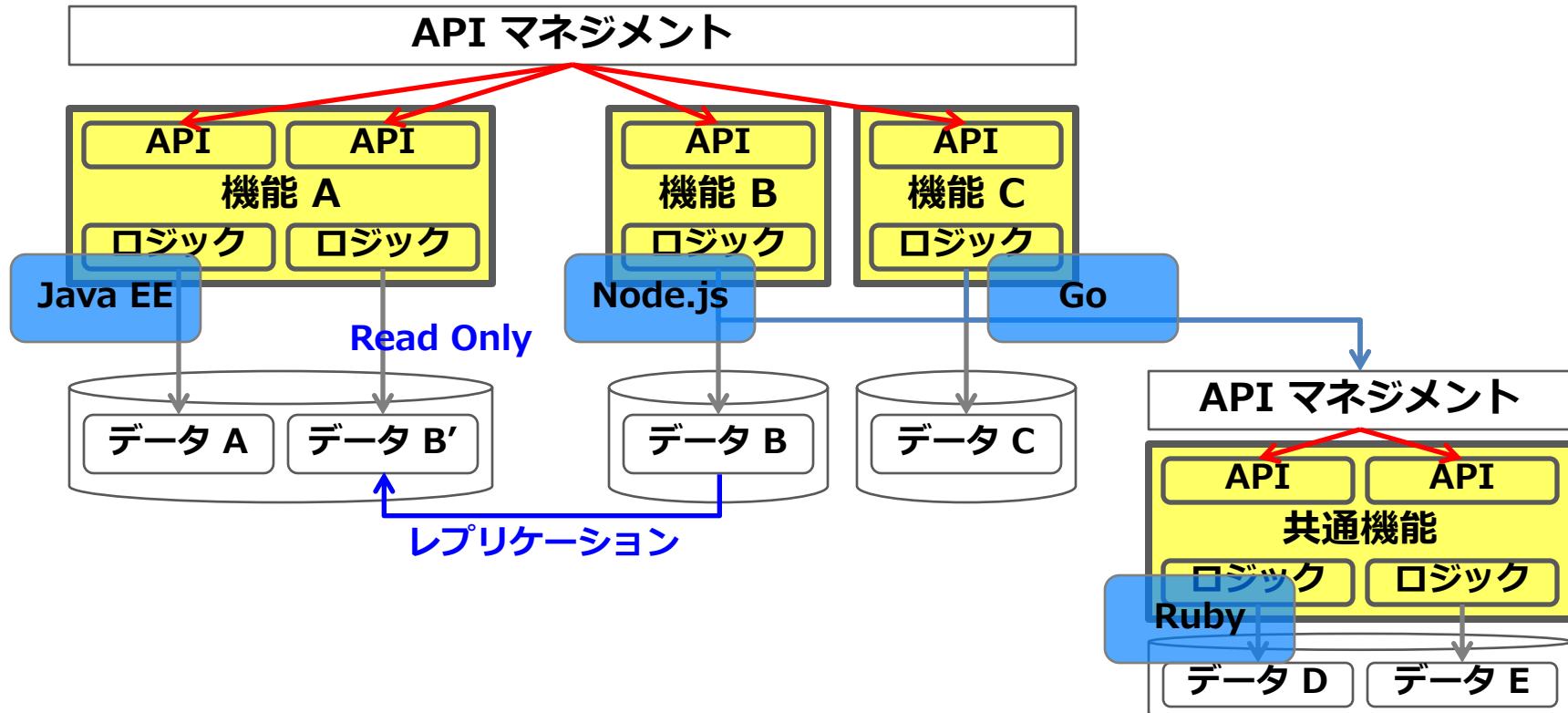
Monolithic アーキテクチャ



問題点

- 機能間の依存性
- 巨大な影響範囲
- 異なる機能間で以下を共有
 - リリースサイクル
 - インフラ

Microservices アーキテクチャ



Microservices アーキテクチャの利点

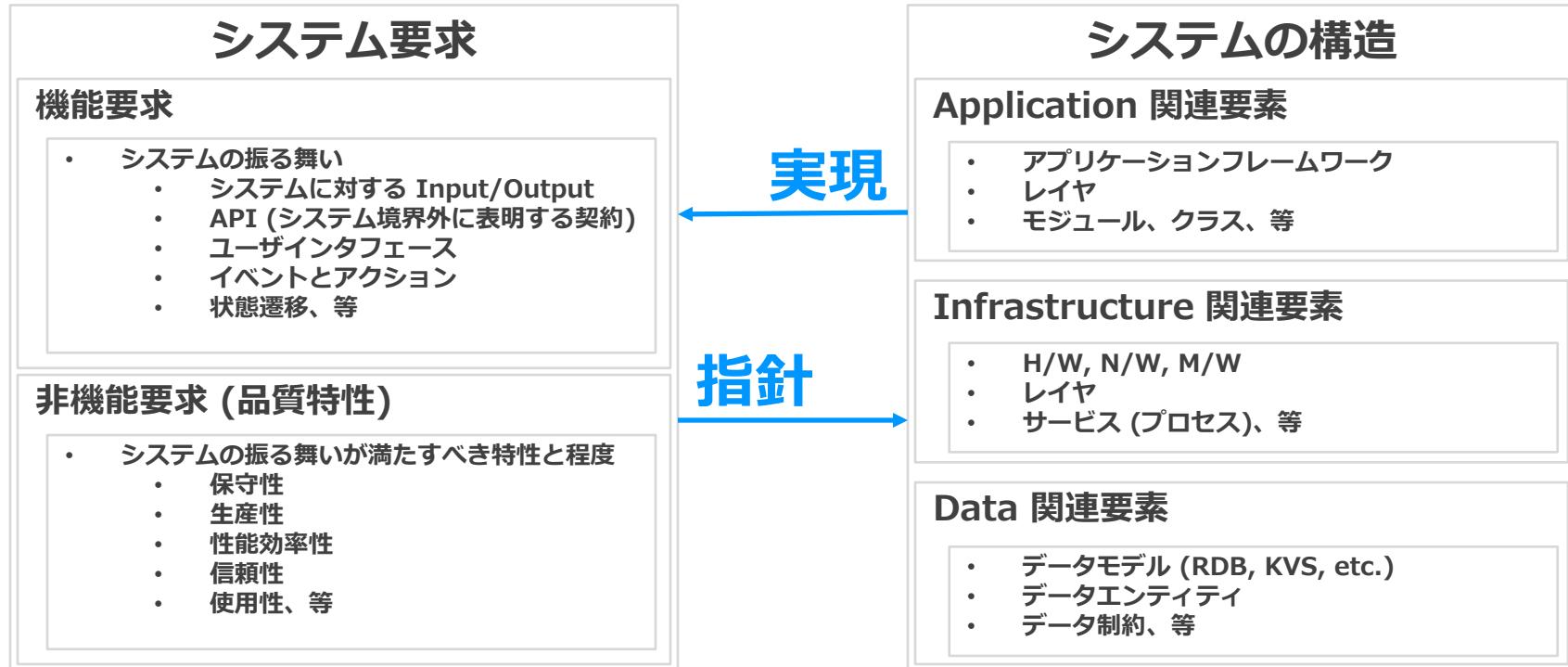
DevOps と アーキテクチャ

- 疎結合
- 小さな影響範囲 = 変更容易性
- 小さな関心事 = 変更スピード
- 個別のリリースサイクル
- 個別のインフラ = 個別に柔軟に非機能要件に対応
- チームにフィット (コンウェイの法則 [15])
- 技術異質性

アーキテクチャ戦略

アーキテクチャの定義

アーキテクチャ：システム要求を実現するためのシステムの構造



システム要求の変更とアーキテクチャ

システム要求の変更

- ・ システムの“振る舞い”に対する要求の変更
- ・ システムの“品質特性”に対する要求の変更
- ・ ⇒ システムの構造の変更：アーキテクチャの変更

システムの構造（アーキテクチャ）とシステムの振る舞い

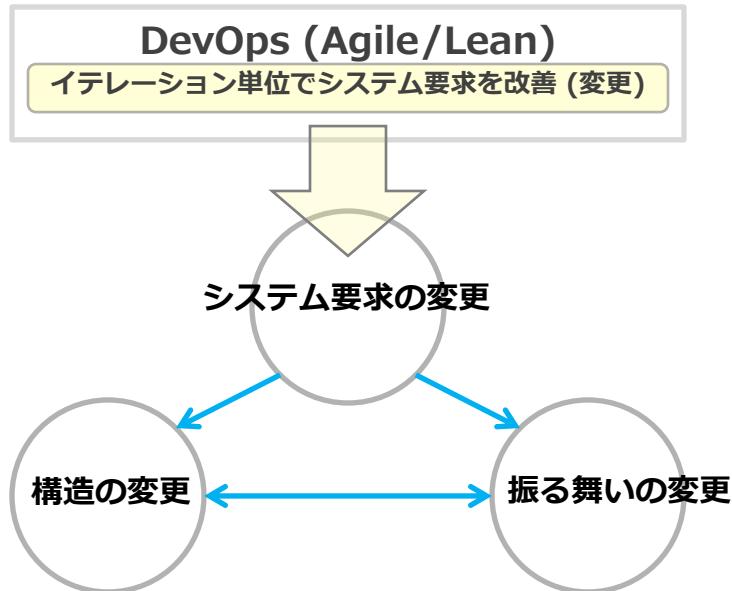
- ・ 振る舞い：機能実装
- ・ “振る舞い”は“システムの構造”上に配置

アーキテクチャの変更

- ・ IT運営に多大なインパクト
- ・ ⇒ アーキテクチャ戦略はIT運営に多大な影響

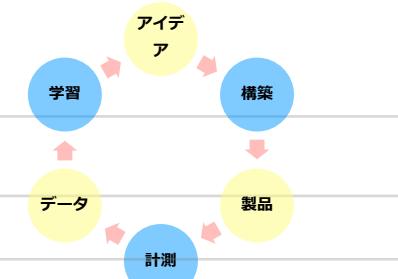
DevOps (Agile/Lean) におけるシステム要求

- ・ システム要求の変化に前向き
- ・ MVP (Minimum Viable Product) とフィードバック
- ・ イテレーション/スプリント毎に探索的・拡張性に変更
- ・ ⇒ アーキテクチャに影響を与えるレベルのシステム要求の変化を前提とした戦略・プロセスが必要



DevOps における品質特性の考え方

DevOps : リードタイム短縮 と 継続的改善 (変更)



Agile/Lean

イテレーション単位でシステム
要求を改善 (変更)

システムの 振る舞い
に関する変更

テスト (自動化) 戦略
Agile/Lean の実践を支えるプラクティス

CI/CD の実践

保守性と生産性
アーキテクチャに求められる特性
テスト容易性

アーキテクチャ

保守性

生産性

テスト容易性

DevOps の実践において重要なソフトウェア品質特性

Infrastructure As Code

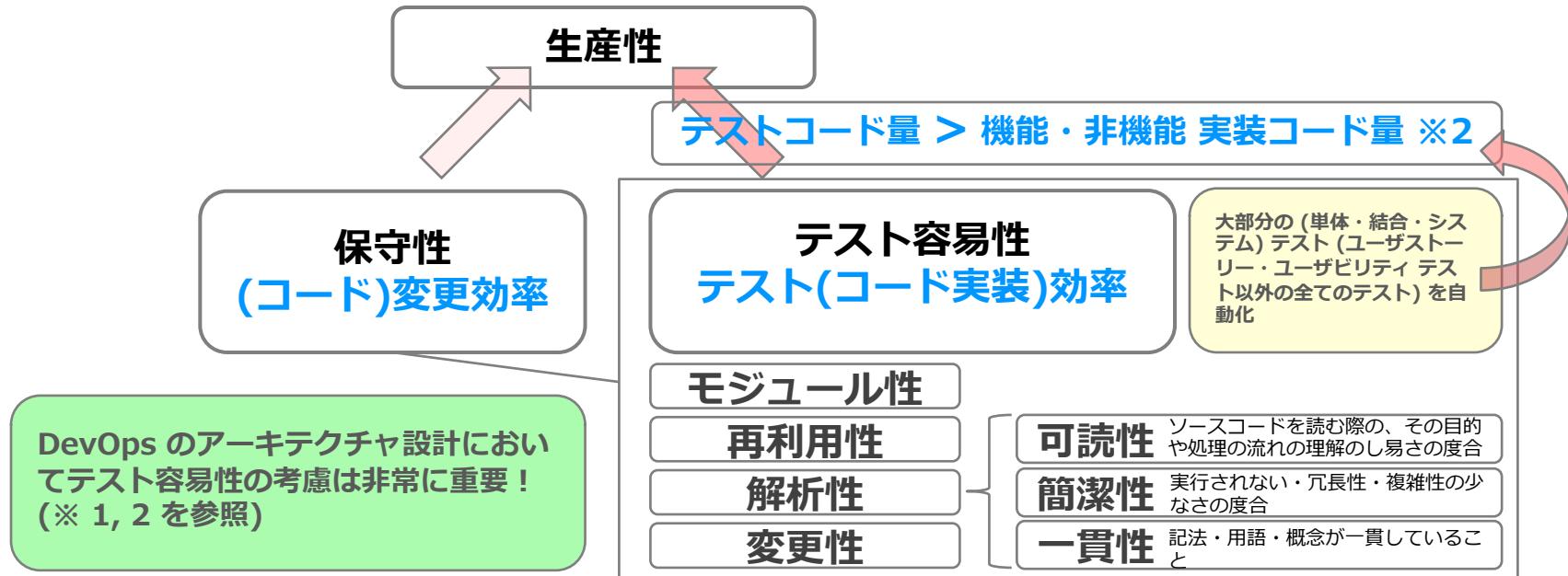
Cloud/Container

インフラ関連要件の効果性・効率性向上

DevOps におけるテスト容易性の重要性

生産性：変更・改善効率 \Rightarrow 変更・改善量/コスト(投入資源)

コード変更・改善効率 \doteq 機能・非機能コード実装効率 + テスト(コード実装)効率 $\times 1$



保守性・テスト容易性 低下の例

アーキテクチャ全体

- ・ **結合度：高** ※ 詳細は後述
- ・ **凝集度：低** ※ 詳細は後述

Application アーキテクチャ

- ・ 非レイヤ (責務によるコンポーネントのレイヤ化) 構成 ※ レイヤの例：UI, API, Application, Domain 等
- ・ 複数のメインロジックを混在させた コンポーネント・モジュール 構成
- ・ 画面実装とビジネスロジック実装の密結合
- ・ 技術依存の API を使用したサービス連携
- ・ サービス連携用データ (API I/O) の肥大 ※ 内部実装を隠蔽しない API
- ・ 複雑な 静的依存性・動的依存性 **構造はその上部構造空間に副作用を与えるべきでない！**
- ・ 過度な標準化・汎用化による複雑性・コードの肥大化
- ・ ステートフルアーキテクチャ
- ・ ソフトウェア静的解析 (可読性・一貫性・簡潔性) の不足、等

Infrastructure アーキテクチャ

- ・ 軽量でない構築プロセス
- ・ 軽量でないサービスプロセス (起動速度、リソース使用量)
- ・ 複雑な関連サービス構成、等

Data アーキテクチャ

- ・ 不適切なデータモデルの選択
- ・ データ中のビジネスロジック、等

保守性のための原則

保守性：モジュール・サービスの適切な分割点

凝集性

- ・ 関連する “振る舞い” が集まっている度合
- ・ ⇒ 要求 (振る舞い) の変更の影響範囲を局所化 (変更の効果性・効率性)

結合度

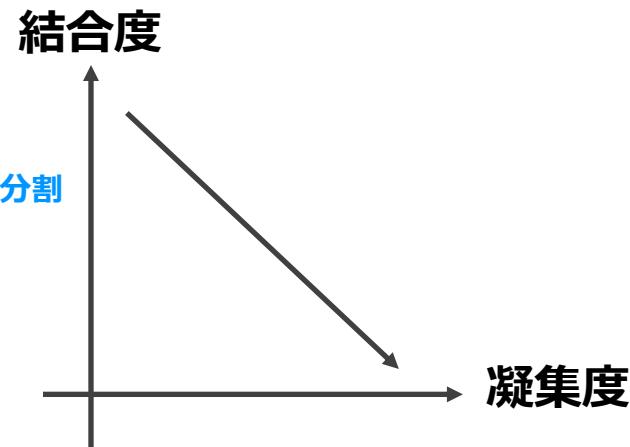
- ・ モジュール間の関連性・依存性の度合
- ・ ⇒ 結合度の低いモジュール間の変更の影響範囲を隔離 (変更の効果性・効率性)

凝集性と結合度の関係

- ・ 凝集性の高いモジュール間の結合度は低い (右図参照)

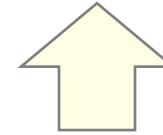
ドメイン駆動設計

- ・ ドメインとコンテキストの概念により モジュール・サービス を適切に分割



疎結合と高凝集性へのアプローチ

疎結合 と 高凝集性



設計思想・技法・開発手法

ドメイン駆動設計 (DDD)

プロセス・TODO に関する言及
及
オブジェクト指向設計・XP が
ベース

- コマンドクエリ責務分離 (CQRS)
- イベント駆動アーキテクチャ
- パイプ & フィルタ
- イベントソーシング

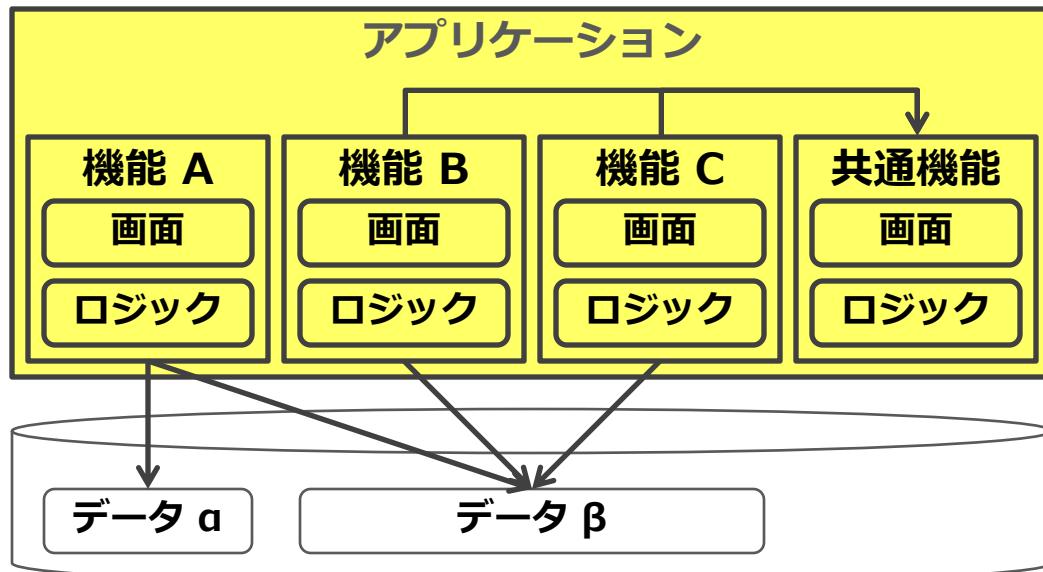
- The Twelve-Factor App
- オブジェクト指向設計
 - SOLID 原則
- XP (エクストリーム・プログラミング)
- etc.

アーキテクチャ設計思想

Microservices アーキテクチャ

- リアクティブアーキテクチャ
- ヘキサゴナルアーキテクチャ
- レイヤアーキテクチャ
- N 層アーキテクチャ
- SOA
- etc.

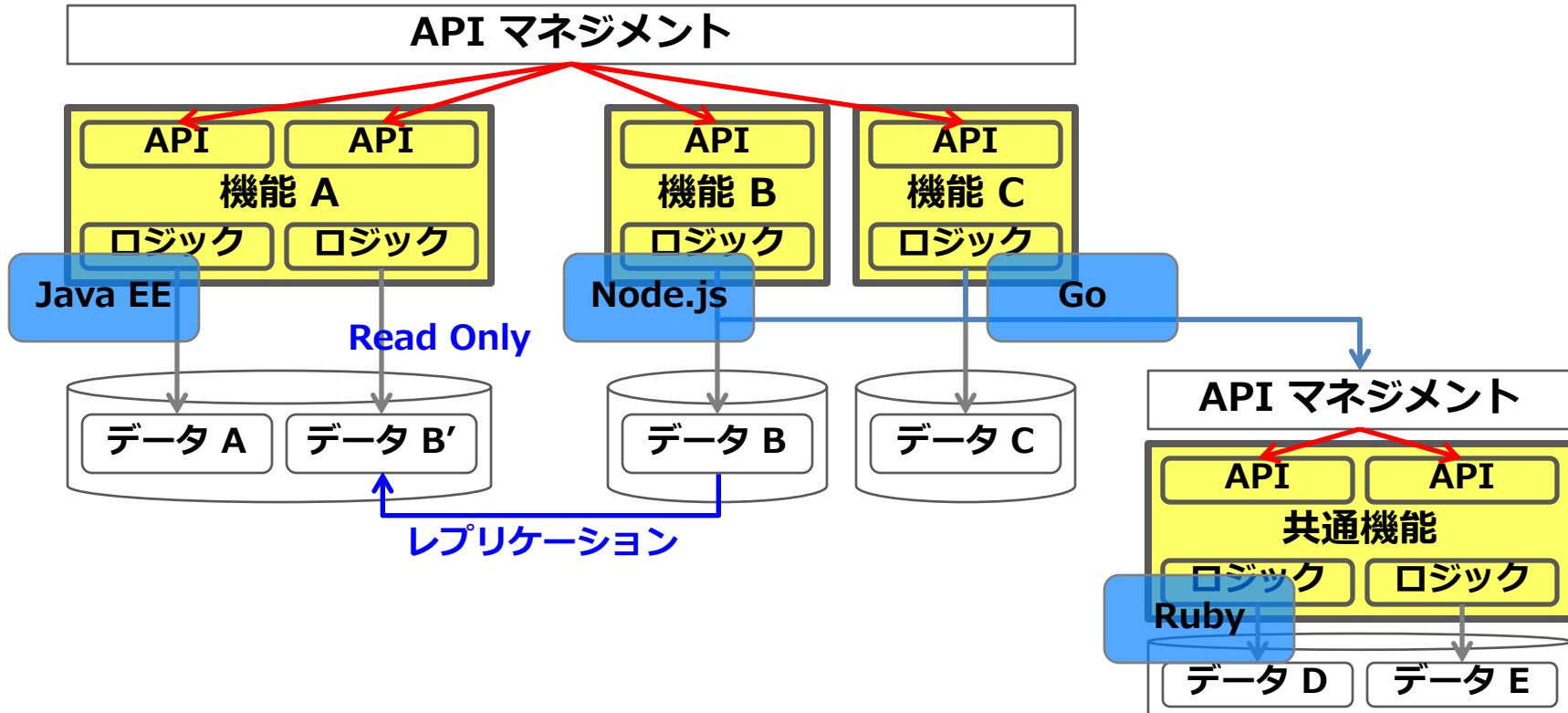
Monolithic アーキテクチャ



問題点

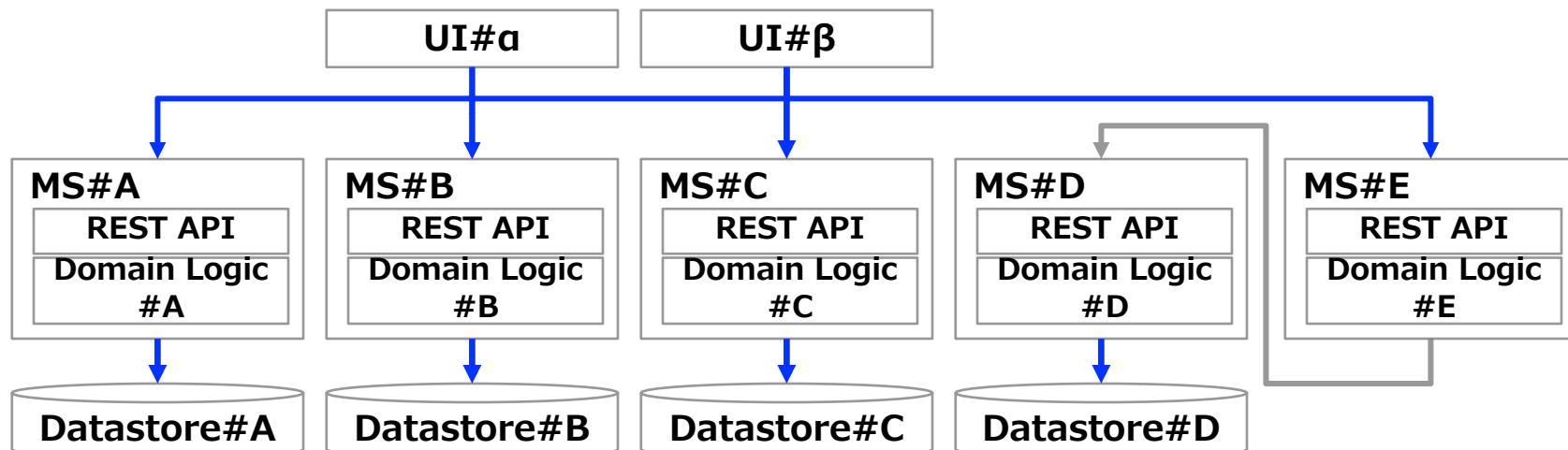
- 機能間の依存性
- 巨大な影響範囲
- 異なる機能間で以下を共有
 - リリースサイクル
 - インフラ

Microservices アーキテクチャ



Microservices Architecture の利点

- 自律性
- 疎結合
- 小さな影響範囲
- 回復性
- (小さな) チームにフィット
- 再利用性
- 技術異質性
- 試験性(テスト容易性), etc.



Microservices Architecture の 難しさ

設計

- 適度なサービスの分割点



課題・注意点

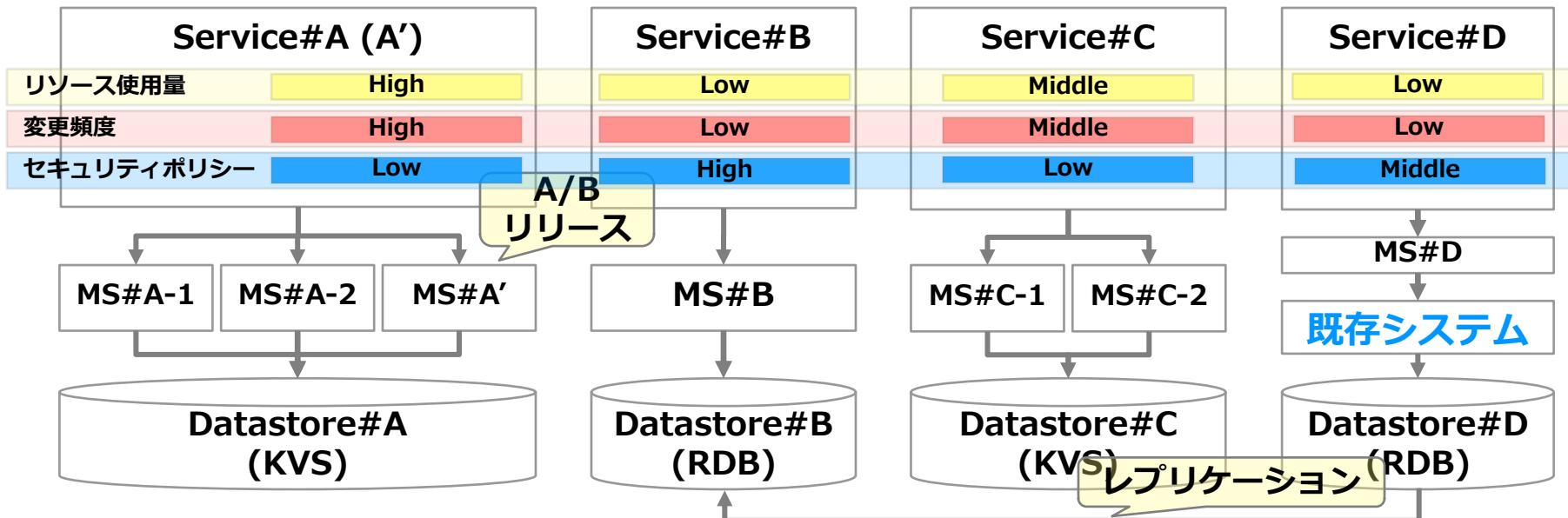
- トランザクション
- パフォーマンスト
- レーザビリティ
- 障害の分離
- 分割による複雑性の増大, etc.

DDD (ドメイン駆動設計) は “実世界の 言語・知識 をシステムに適用する” 手法であつて分割手法ではない。

Microservices Architecture の始め方

以下の観点でサービスを分割

- リソース使用量 (アクセス頻度)
- リリースサイクル (変更頻度)
- データ構造
- セキュリティポリシーの違い
- リリースポリシー (A/B リリースの有無 等)
- 品質特性 (信頼性 等) レベルの違い
- 既存システムとの接続有無



Infrastructure As Code の実践

Infrastructure As Code の 成熟度



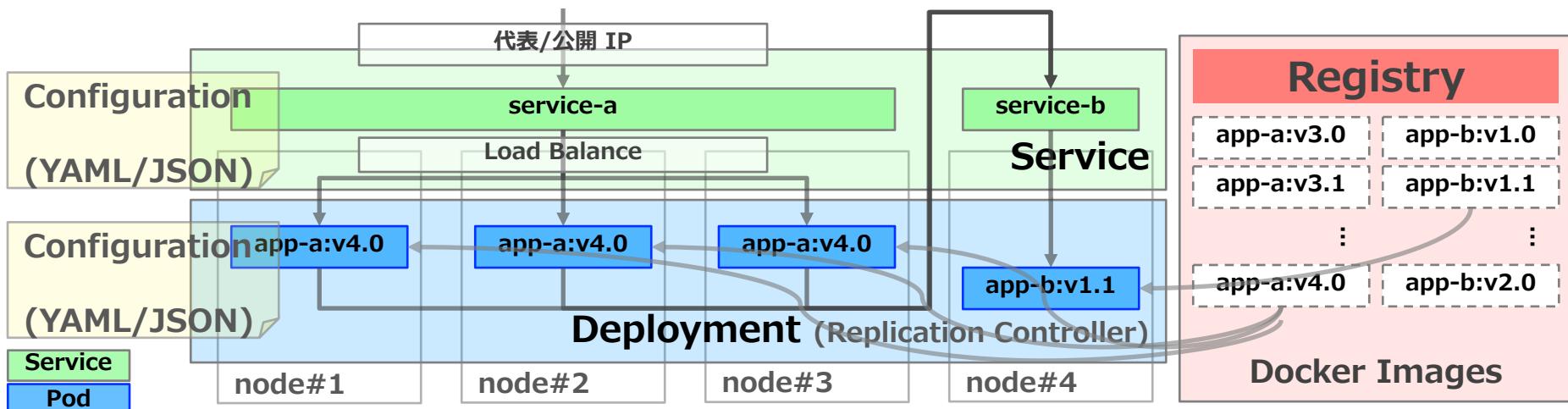
level 1.0 - 構築

Service

- ・ (複数) コンテナの 代表/公開 IP
- ・ ロードバランス
- ・ システムを構成する個々のサービスだけでなくシステム全体の構成をコード化
- ・ (非属人的) コード化範囲の拡大 -> テスト・運用 効率向上

Deployment (Replication Controller)

- ・ Pod (の構成) に関する以下の設定
 - ・ 稼働数, コンテナイメージ



level 2.0 - 非機能要求

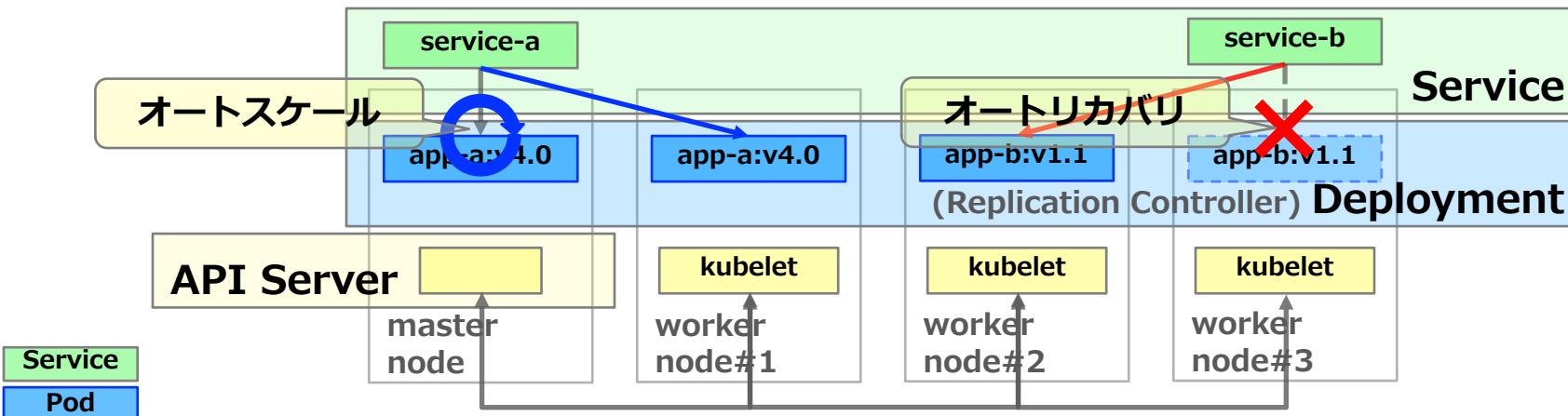
Service, Deployment, Scheduler

- ・オートスケール
- ・オートリカバリ

API Server

- ・コンテナクラスタ制御 REST API
- ・管理 GUI/CLI も本 API を使用

- ・**非機能要求(保守性、可用性、性能)の(非属人的)コード化**
- ・-> **運用効率・変更性の向上**



level 3.0 - リリース

Route

- Service に対するアクセスのルーティング
 - ルーティングの割合
 - A/B リリース
- リリースの(非属人的)コード化、効果・効率の向上
- > (無停止・A/B リリース等) ビジネスフィードバック機会の増加

Weight

Route

bluegreen-example

http[s]://bluegreen-example.redhat.com/

Registry

app-a:v1.0

app-a:v1.1

:

app-a:v2.0

Service

Weight (0:100)

service-a-green

service-a-blue

Deployment

app:v1.0

app:v2.0

Docker Images

テスト戦略

アジャイルソフトウェア開発におけるテストの考え方

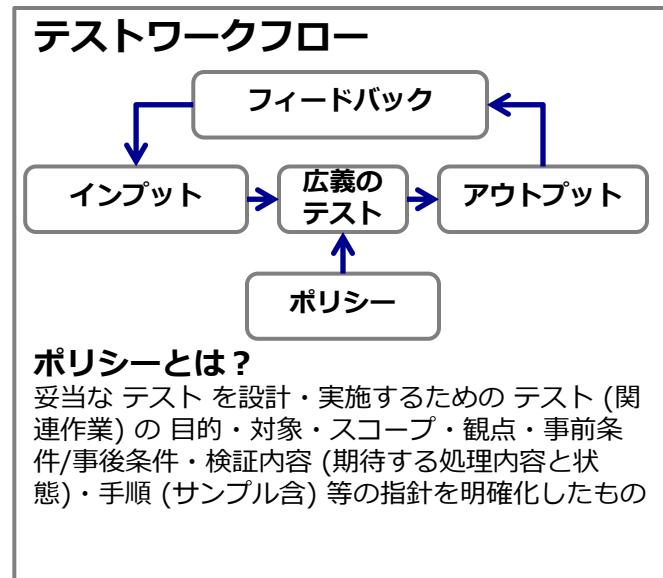
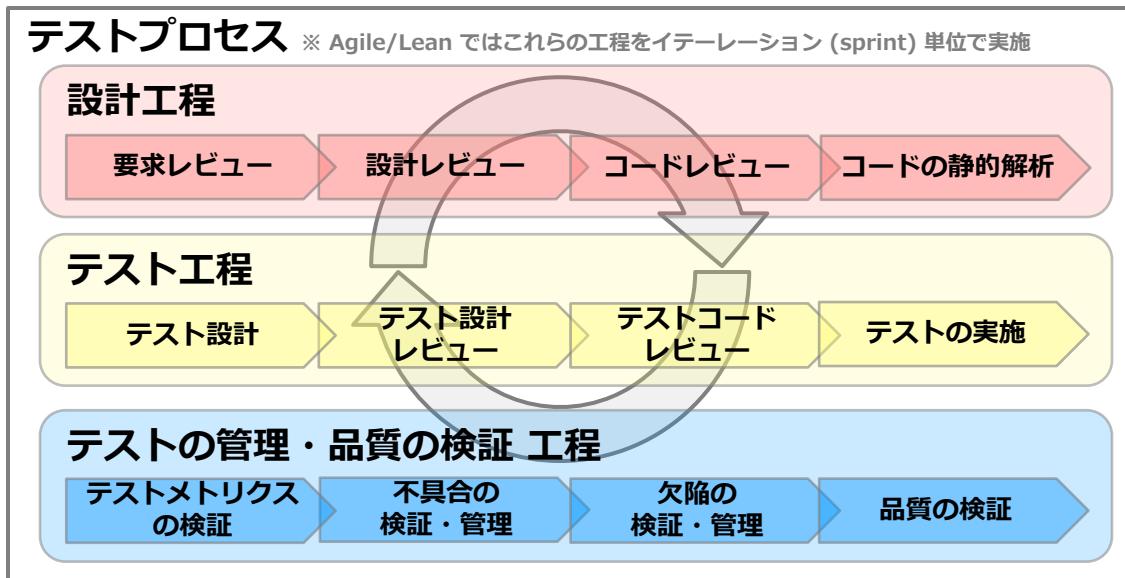
- ・ 手動で行うべきテスト：使用性・UX に関するテスト
- ・ 自動で行うべきテスト：手動で行うべきテスト以外の全てのテスト
- ・ 自動テストはワンストップ実行かつ高速で実行

テスト(戦略)の原則

品質特性	概要
網羅性	<p>プロダクト(ソフトウェア)がリリースの段階で内在しうる欠陥 [1] を以下の観点で網羅的に検知可能な テスト体系(分類)および作業フローが確立されていること</p> <ul style="list-style-type: none">工程(設計、実装、テスト等)、テスト特性(動的/静的、ブラックボックス/ホワイトボックス)システムのビルディングブロック(クラス、モジュール、プロセス、サーバ等)要求仕様(機能要求、非機能要求、SLA等)と品質特性アジャイルテストの4象限(自動/手動、ツール、テスト)
明確性	テストの目的・対象・スコープ・観点・事前条件/事後条件・検証方法(期待する処理と状態)・手順および設計手順(テスト設計を行うにあたってのインプットとアウトプット)が明確であること
検証性	テスト(それ自体)の網羅性および妥当性が定量的(テストメトリクス)または定性的(レビュー・ポリシー)に検証可能な手段・作業フローおよび検証手順が確立されていること
再現性・保守性	<ul style="list-style-type: none">テストを実行する(属人的でない)手段・手順が確立されていること上述の品質特性を満たした上で、全てのテストが常に正常実行可能であること
管理性・追跡性・関連性	<ul style="list-style-type: none">テストの実行結果・進捗状況・テストメトリクス等のテストプロセスを管理するための情報が一元管理・可視化されていることテストの実行結果とテスト実行時の状態(システムを構成する要素およびテストデータのバージョン管理)が追跡可能のこと個々のテストの概要が明瞭となるテストの命名規約・文書規約・ソースコードコメント規約・管理プロセスが確立されていること不具合の検知と欠陥の対応(コードの修正)の経緯・対応内容がバージョン管理・チケット管理等のツールにより追跡可能のことシステムの修正箇所に関連するテスト、不具合・欠陥の箇所が把握可能のこと
効率性	<p>上述の品質特性に加え以下が実現されていること</p> <ul style="list-style-type: none">不具合・欠陥のフィードバックが高速であることテストプロセス(大部分のテスト)が自動化されていることテストの実行および結果の検証(品質判定)が繰り返し実行可能で高速であることテストが再利用可能であること(再利用可能なテストの範囲が明確であること)継続的インテグレーション(CI) / 継続的デリバリー(CD)が確立されていること

テスト戦略の定義

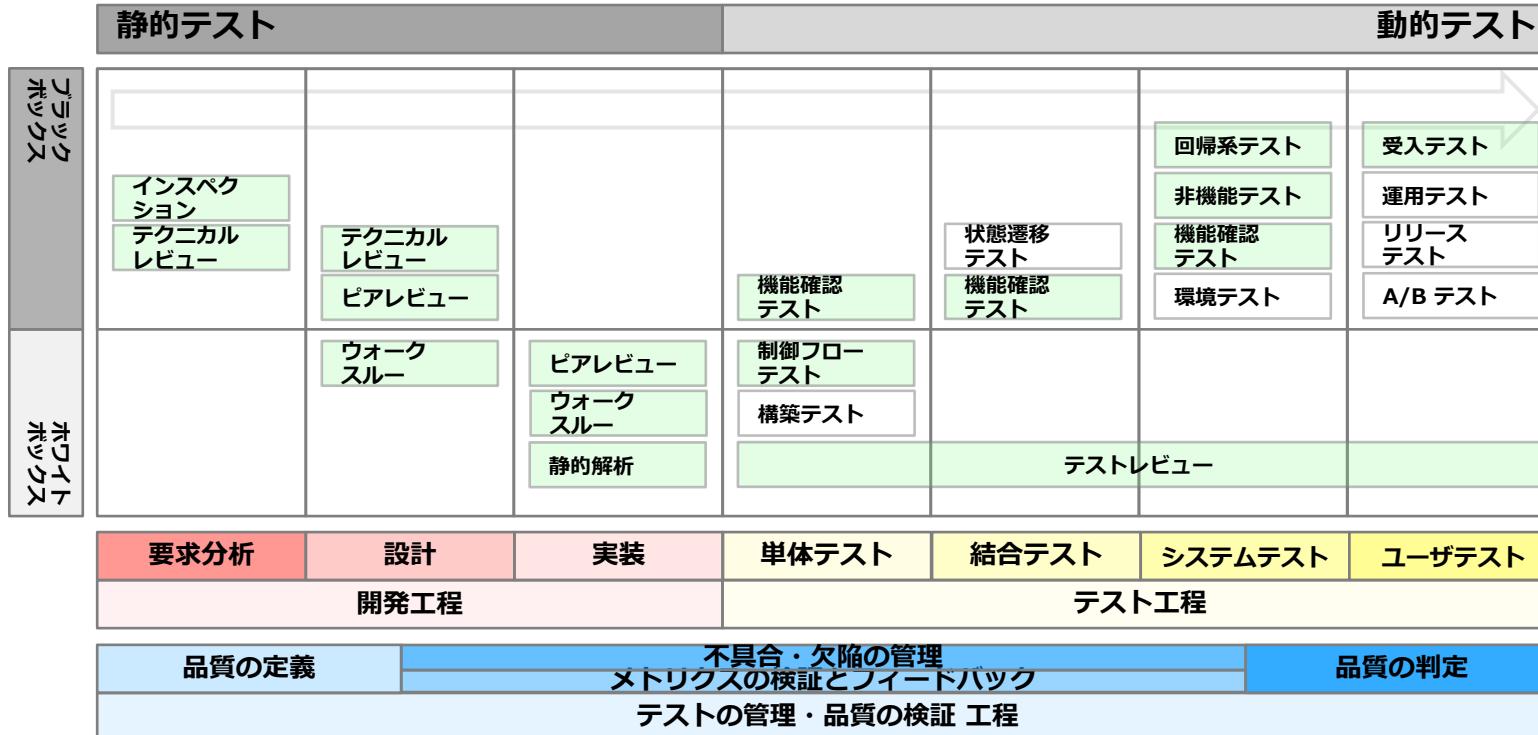
テスト戦略 とは **テスト（戦略）の原則を考慮し、テストプロセス全体のテスト工程 ([※1]) の構成（テスト体系）とフロー、各テストのワークフロー（インプット/アウトプット、ポリシー）** を 設計・定義 すること



[※1] レビュー → テスト (テスト = 品質の妥当性検証のための活動)

テスト体系 - 工程, テスト特性による分類

工程の観点でテストプロセス全体のテストの概要構成を定義する。下図中の は必須となるテスト工程



静的テスト

- ソフトウェアを動作させることなく行うテスト
- ドキュメントやソースコードが対象

動的テスト

- ソフトウェアを動作させて行うテスト

ブラックボックス

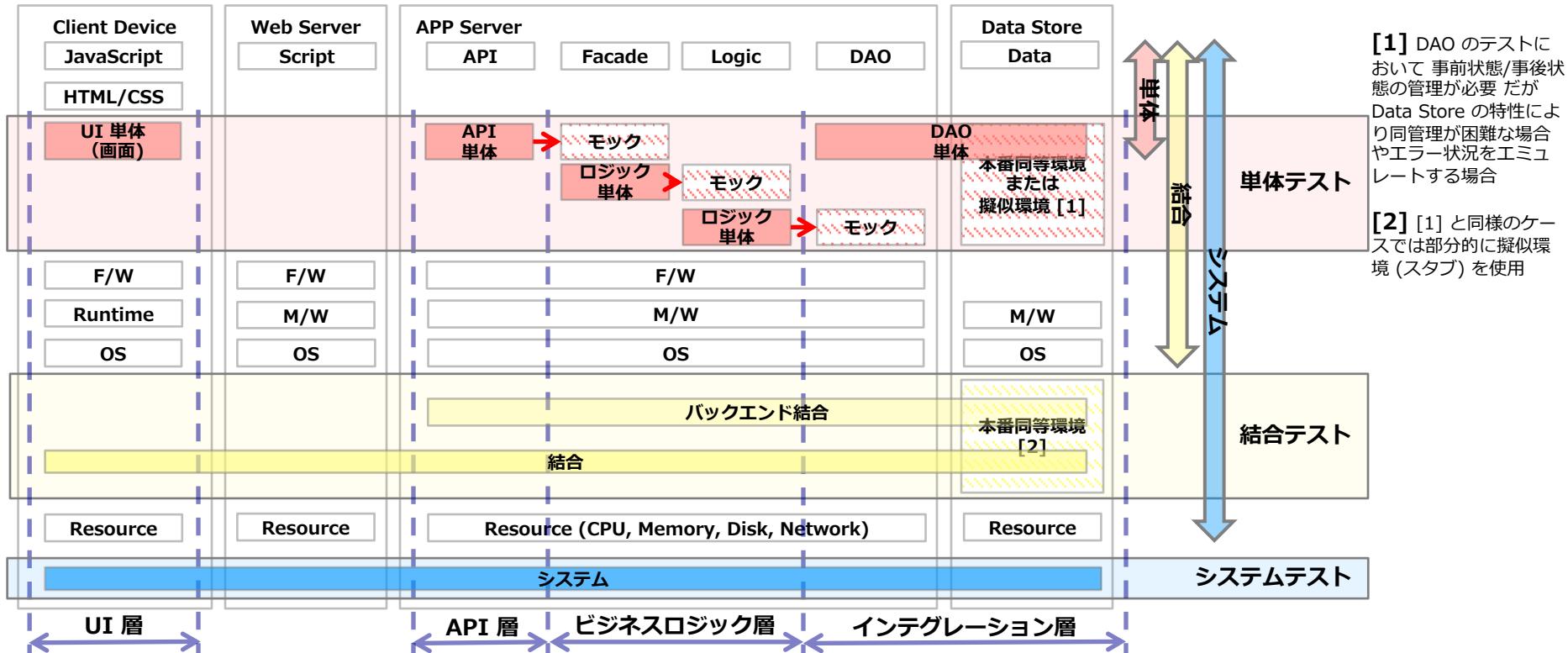
- ソフトウェア内部の論理構造は参照せず、入力/出力に注目して、ソフトウェアの妥当性を確認するテスト

ホワイトボックス

- ソフトウェア内部の論理構造に着目して、制御フロー・データ処理等の妥当性を確認するテスト

テスト体系 - システムのビルディングブロックによる分類

システムのビルディングブロック (+ テストの再利用性) の観点で 単体・結合・システム テストを細分化し定義する。



コンテナ基盤の確立

Kubernetes/OpenShift の関係

Kubernetes :

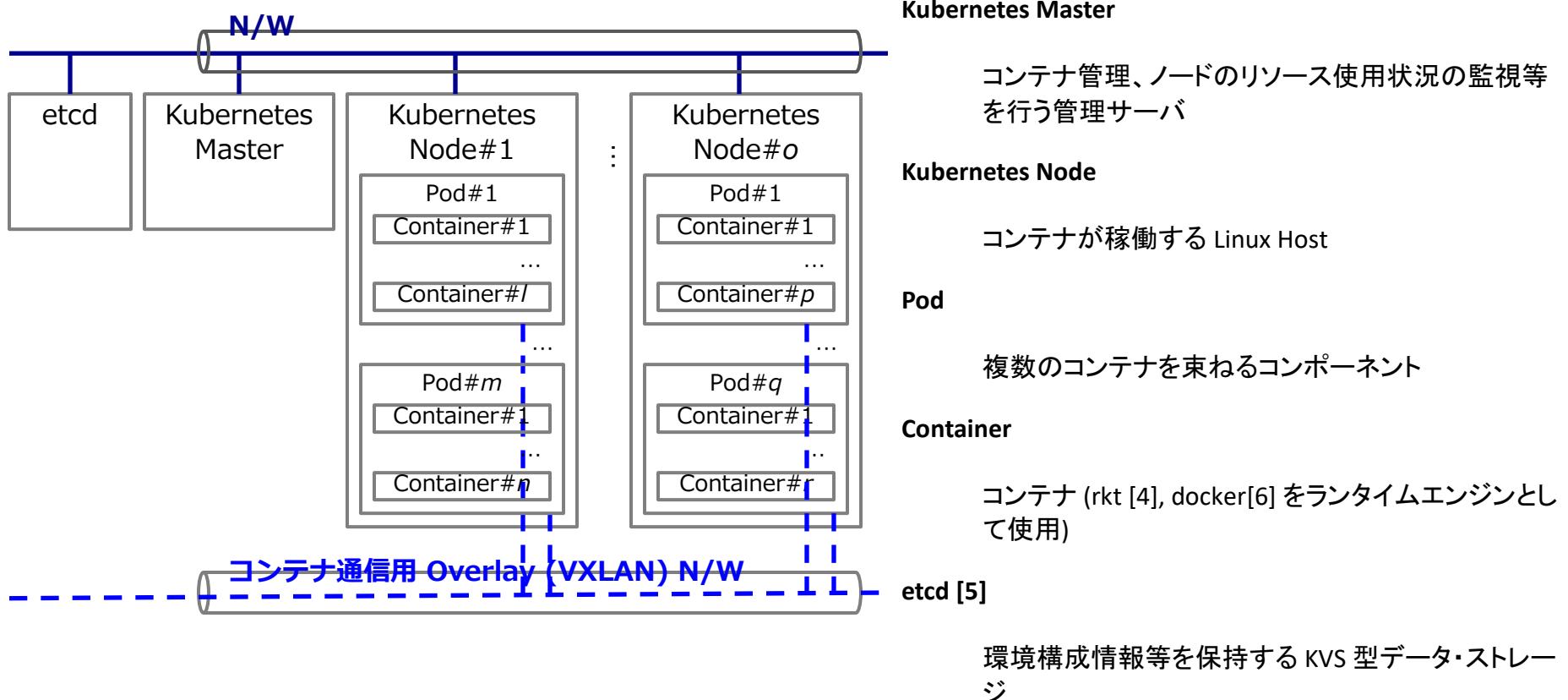
Google がスタートさせたコンテナオーケストレーションのための OSS、同社のコンテナベースのサービス運用において 10 年以上の実績をもつ Borg と Omega がベースとなっている。※ Borg/Omega は Google Search, Google Mail 等のサービスの管理に使用させている。Kubernetes プロジェクトは Google, Red Hat, Microsoft, IBM 等のベンダーによるアップデートが頻繁に行われている。

OpenShift :

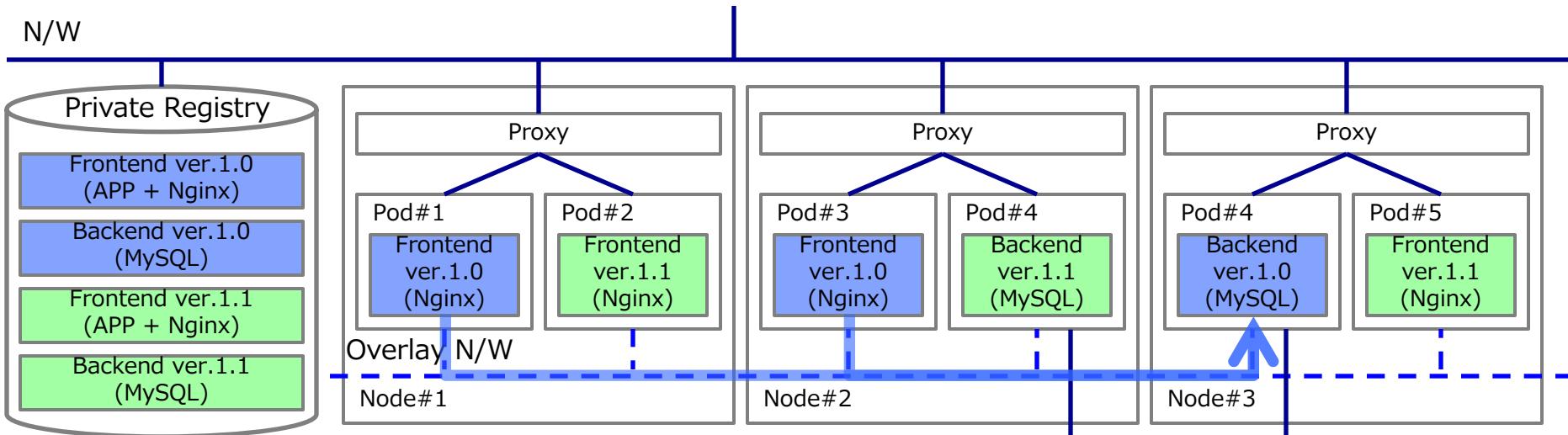
Kubernetes ベースに認証・認可、コンテナのビルド、Jenkins コンテナによる CI/CD フロー、管理 GUI/CUI 等の機能を付加したオーケストレーションツール。

※ OpenShift Origin : Red Hat Container Application Platform のオープンソースアップストリーム

Kubernetes アーキテクチャ概要

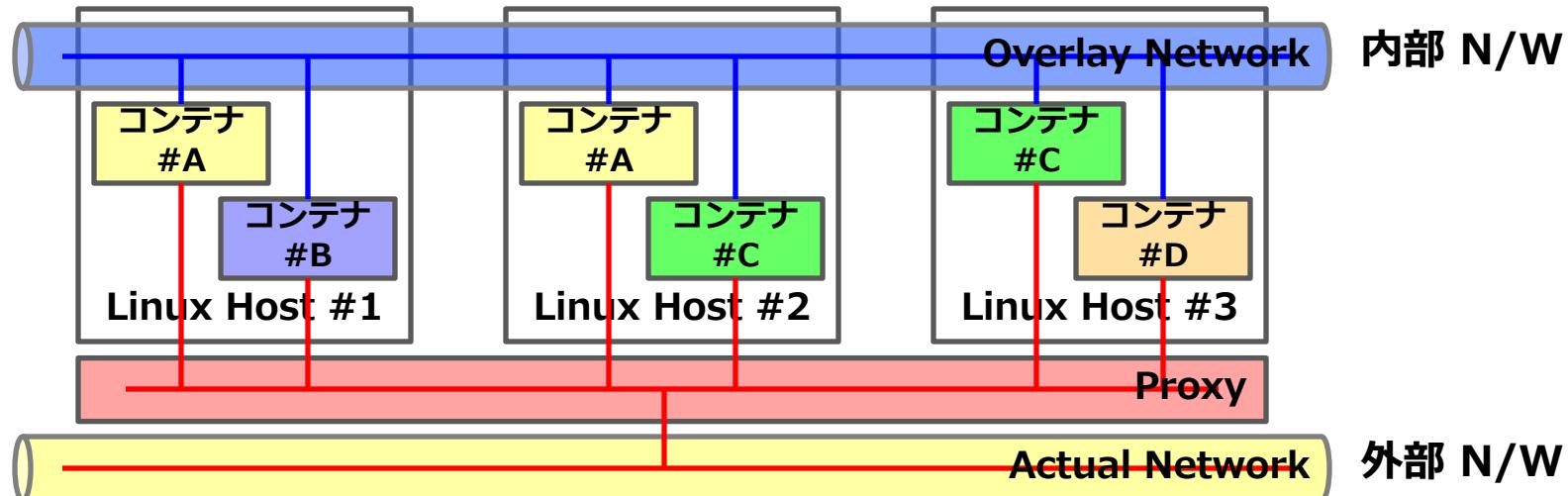


コンテナ環境における一般的な web システムの構成



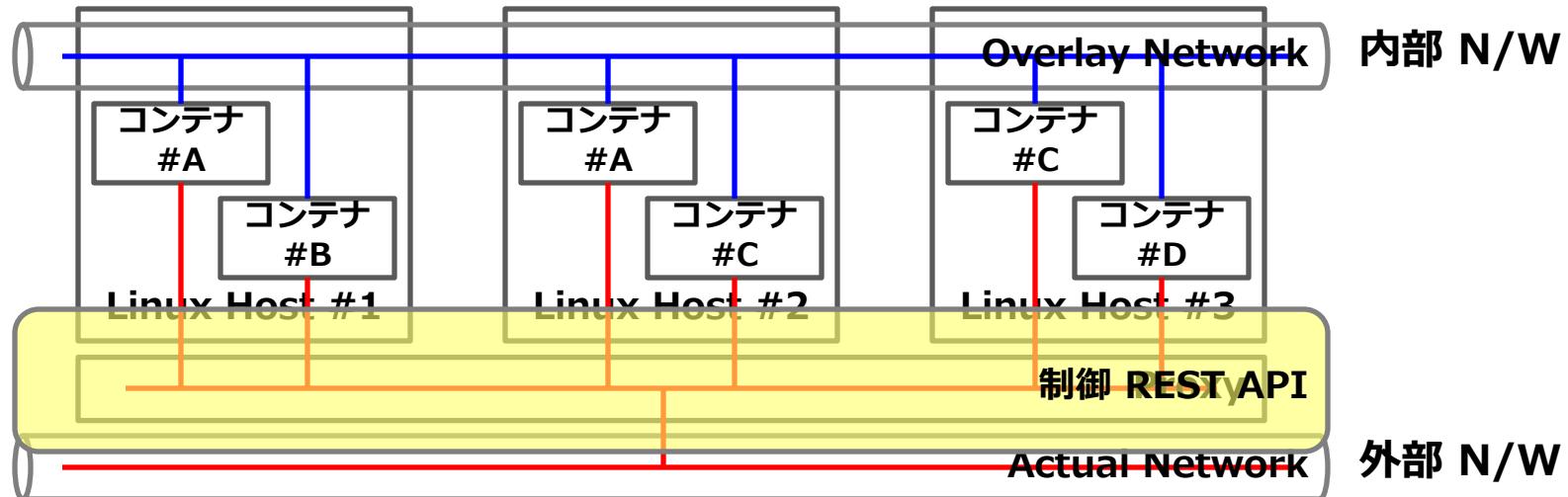
クライアントからのアクセスは Frontend (APP + Nginx) を Service として定義することにより、Kubernetes/OCP により透過的に適切な Pod への転送および Proxy によるロードバランスが行われる。Pod 間の通信はコンテナ間通信用 Overlay N/W を使用して行われる。

コンテナオーケストレーション – 基本機能



- コンテナのブートストラップ
- コンテナ連携用内部ネットワークの制御
- 外部ネットワークからコンテナへのアクセスの ルーティング, ロードバランシング

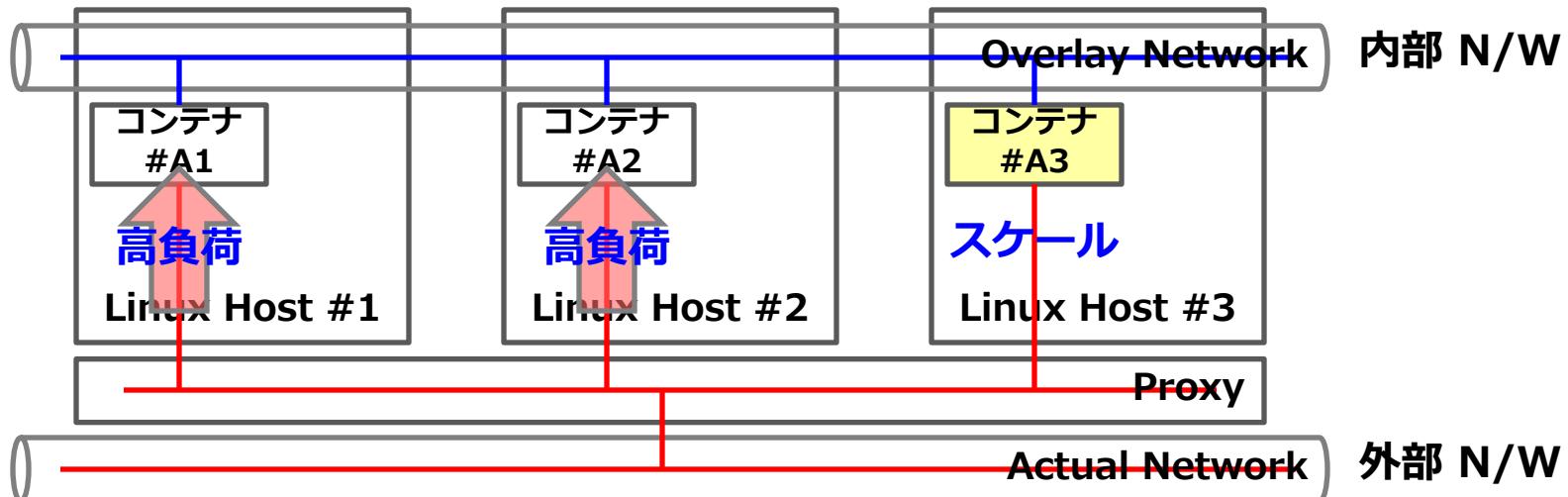
コンテナオーケストレーション - 制御 API



- コンテナの情報の取得
- コンテナのブートストラッピング, etc.

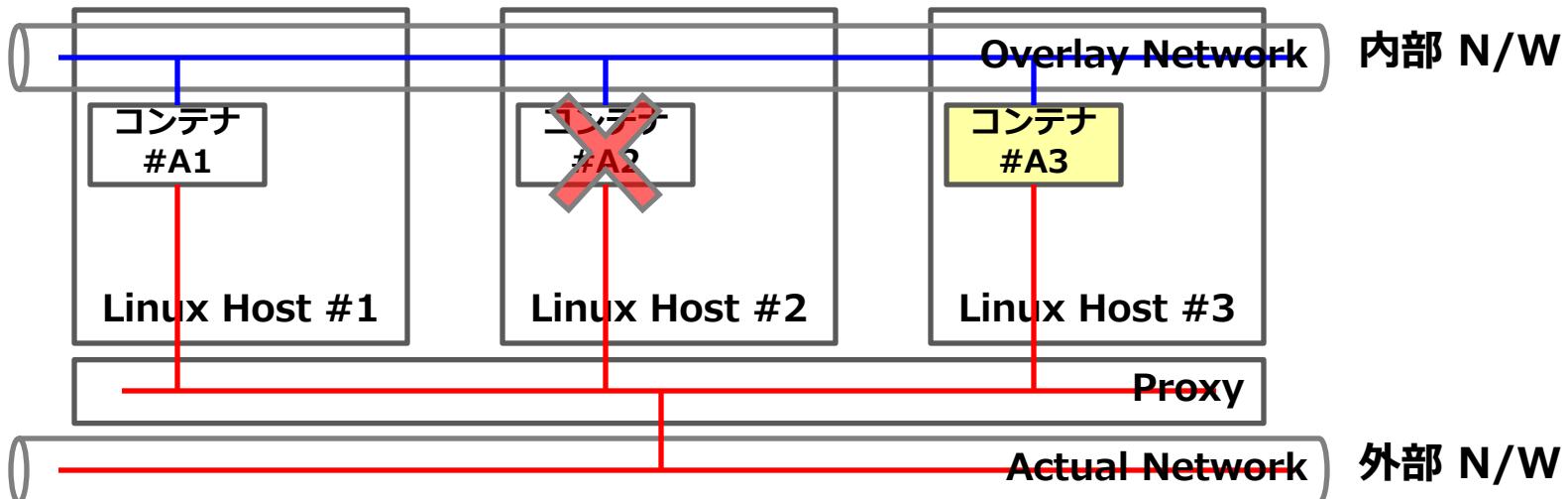
Kubernetes : <http://kubernetes.io/docs/admin/kube-proxy/>

コンテナオーケストレーション – オートスケール



- 負荷の閾値（任意）に従いスケール
- 新たなコンテナへのロードバランス
- 新たなコンテナを起動するホストのルール

コンテナオーケストレーション - セルフヒーリング



- コンテナ障害発生の検知と新たなコンテナの起動
- 新たなコンテナへのロードバランス
- コンテナの最小稼働数に関するルール

CI/CD の実践

自動化戦略

自動化戦略

自動化の目的

- (繰り返し実施される) 手順・プロセス の 効果性・効率性・再利用性 の向上
 - 検証・変更可能で妥当な方法・手順 を 最小限のコスト で 高速 に実施
- 手順・プロセス からの属人性の排除

自動化の対象範囲

- 繙続的デリバリにおける以下
 - Application/Infrastructure のビルド
 - 自動化可能な全てのテスト (単体テスト、結合テスト、システムテスト、画面 I/O テスト)
 - リリース
- メトリクスの収集と可視化
- 以下の運用タスク
 - 運用作業
 - 死活監視 (プロセス、ポート、ネットワーク)
 - リソース監視 (CPU, Memory, Disk, Network I/O)
 - ログ監視
 - 以下の保守作業
 - アップグレード
 - バックアップ/リカバリ

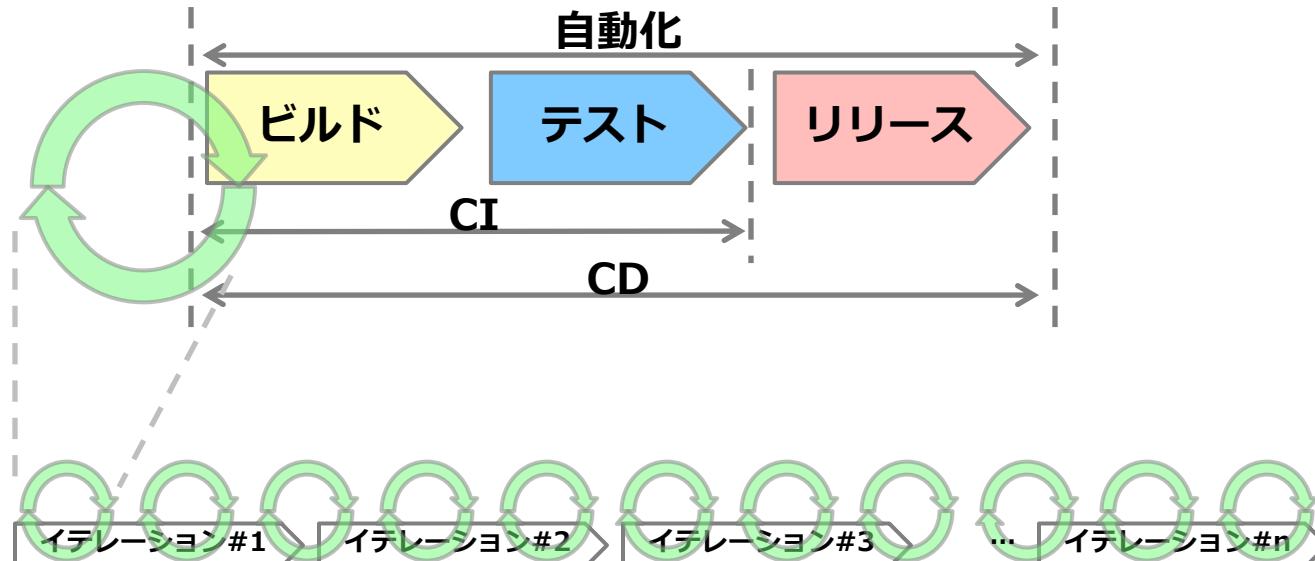
CI/CD とは？

CI (Continuous Integration) : 繰続的インテグレーション

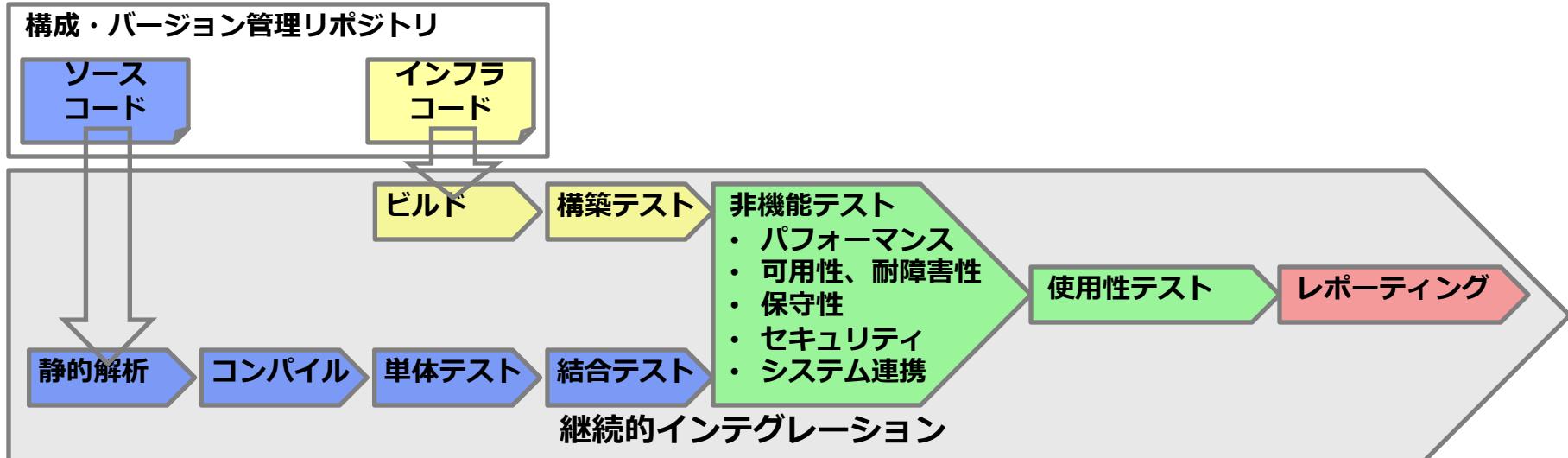
- ・コンパイル、ビルド、テスト（の繰り返し実行される）手順を自動化
- ・不具合をリアルタイムフィードバック
- ・リリース可能な品質のプロダクトを毎日統合

CD (Continuous Delivery) : 繰続的デリバリー

- ・自動化の範囲をリリースまで拡大



継続的インテグレーション概要



1. 小さな変更毎に ビルド&テスト を行うことによる不具合発生時の解析性向上 (影響範囲、経過時間)
2. 一貫性・可読性・簡潔性の低いコードの効率的な検知
3. テストメトリクスの継続的チェック
4. テストの効率化・高速化
5. 1.~4. による品質の効率的な維持、保守性・生産性の向上

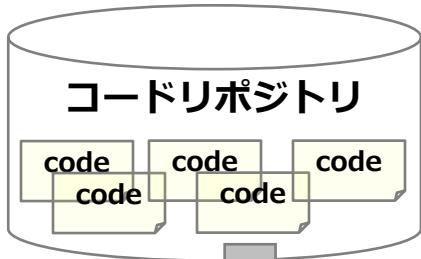
CI の概要

コードリポジトリ

- ・ コードのバージョン管理
- ・ ツール : GitHub, BitBucket, Gerrit, Gitlab, etc.

CI サーバ

- ・ ビルド、テスト、レポート 手順の自動化管理ツール
- ・ Jenkins, TravisCI, CircleCI, drone.io, etc.



CI サーバ

pull

ビルド

- ・ コードのコンパイルとアーカイブ
- ・ ツール :
 - Java : Gradle, Maven, etc.

デプロイ

- ・ アーカイブのアプリケーションサーバへのデプロイ
- ・ ツール
 - Ansible, Chef, etc.

テスト (ツール)

- ・ 静的解析 : Sonar, etc.
- ・ 単体 : Junit, xUnit, Serverspec, etc.
- ・ 結合、システム : Selenium, Jmeter, Gatling, etc.

ビルド

コンパイル

アーカイブ

デプロイ

テスト

静的解析

単体

結合

システム

レポート

CI/CD の原則

プロダクトの変更（改善）における以下のプロセスを自動化（CI/CD パイプライン）し、“production-ready”（本番環境適用可能）な品質のプロダクトを毎日リリースする。

- ・ アプリケーションコードの変更
- ・ インフラコードの変更
- ・ データ関連の変更
- ・ プロダクトのビルド
- ・ プロダクトのテスト

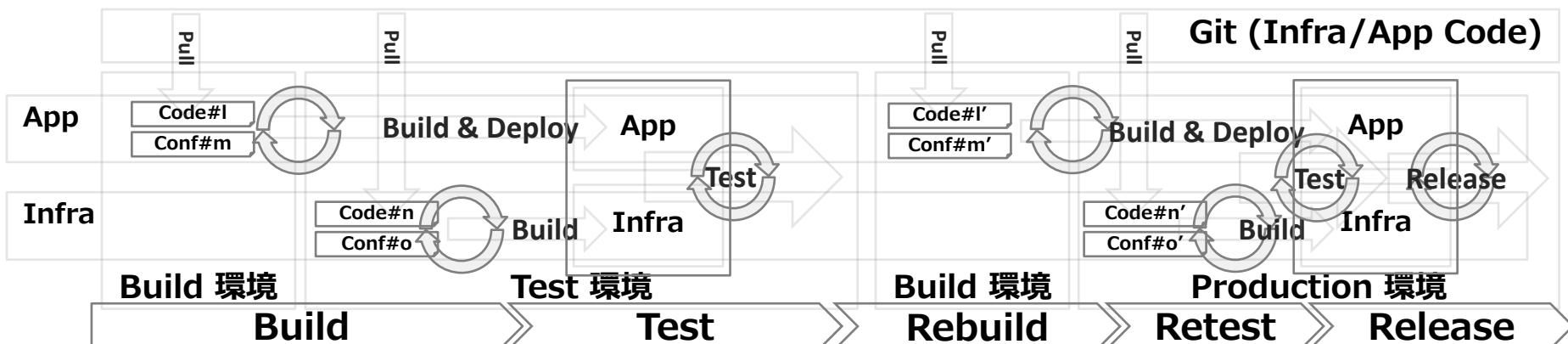
非コンテナ環境の CI/CD

課題と効果

1. Test/Production 環境の差異の低減
 - ・(テスト環境における) テストの高信頼性
 - ・Rebuild/Retest (ムダとミス) の削減
 - ・障害解析の効果性・効率性の向上
2. Test の高速化
 - ・早いフィードバック

目的

- ・テスト原則の充足度向上
- ・変更に“前向き”



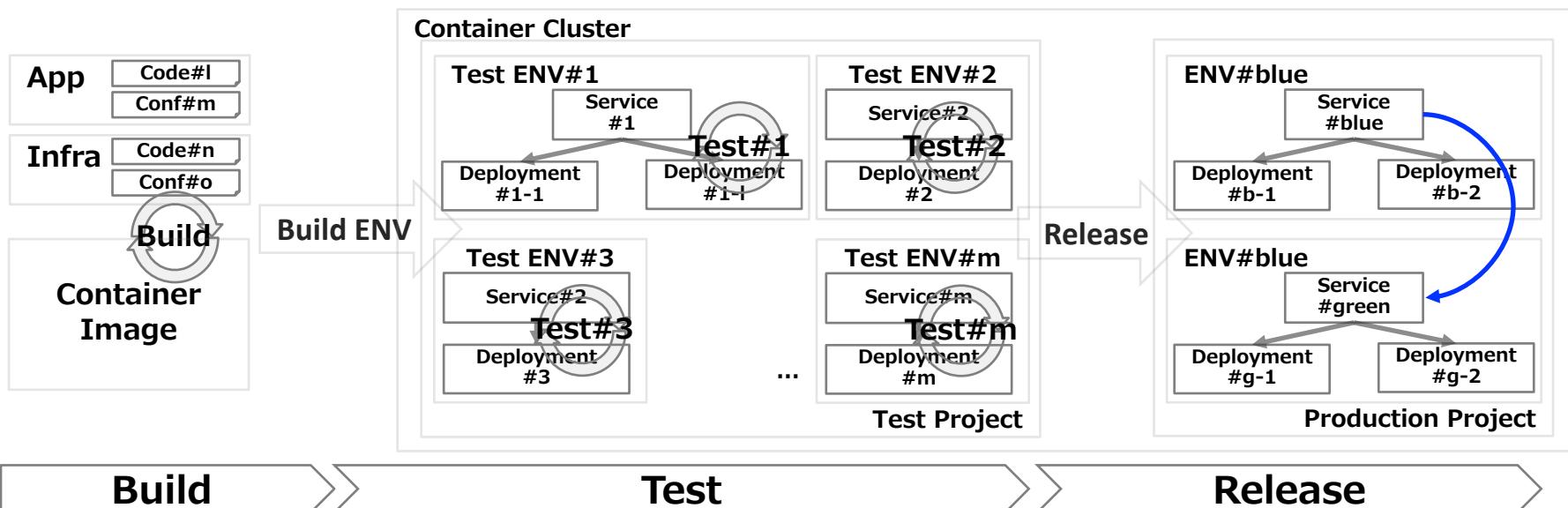
コンテナ環境の CI/CD

Test

- 複数のシステム構成を構築して並列化
- 同一クラスタ内で Test/Production 環境を分離 (Project)
- サンドボックステスト環境

Release

- テスト済のシステム構成の自動構築
- テスト済のシステム構成へ自動リリース



コンテナ環境の CI/CD

Test

- ・複数のシステム構成を構築して並列化
- ・同一クラスタ内で Test/Production 環境を分離 (Project)
- ・サンドボックステスト環境

Relese

- ・テスト済のシステム構成の自動構築
- ・テスト済のシステム構成へ自動リリース

- ・**テスト・リリース の 効果性・効率性 の向上 -> テスト・変更に “前向き”**
- ・-> リードタイム短縮・継続的改善 が加速
- ・-> チームに ビジネス革新 のための キャパティ が生まれる。



THANK YOU



plus.google.com/+RedHat



facebook.com/redhatinc



linkedin.com/company/red-hat



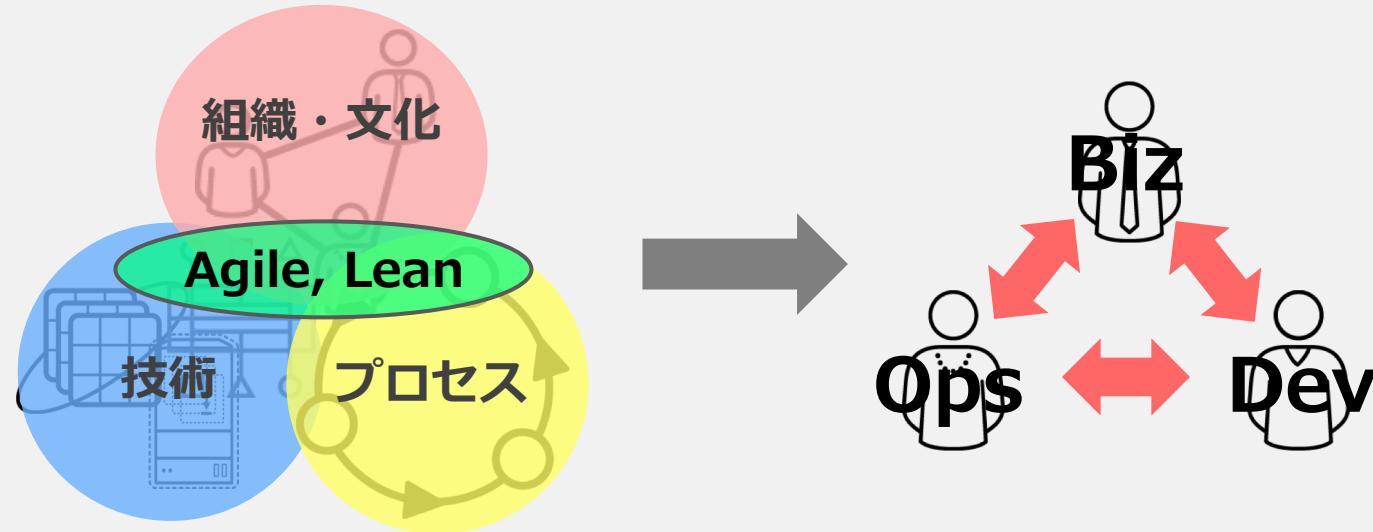
twitter.com/RedHatNews



youtube.com/user/RedHatVideos

DevOps とは？

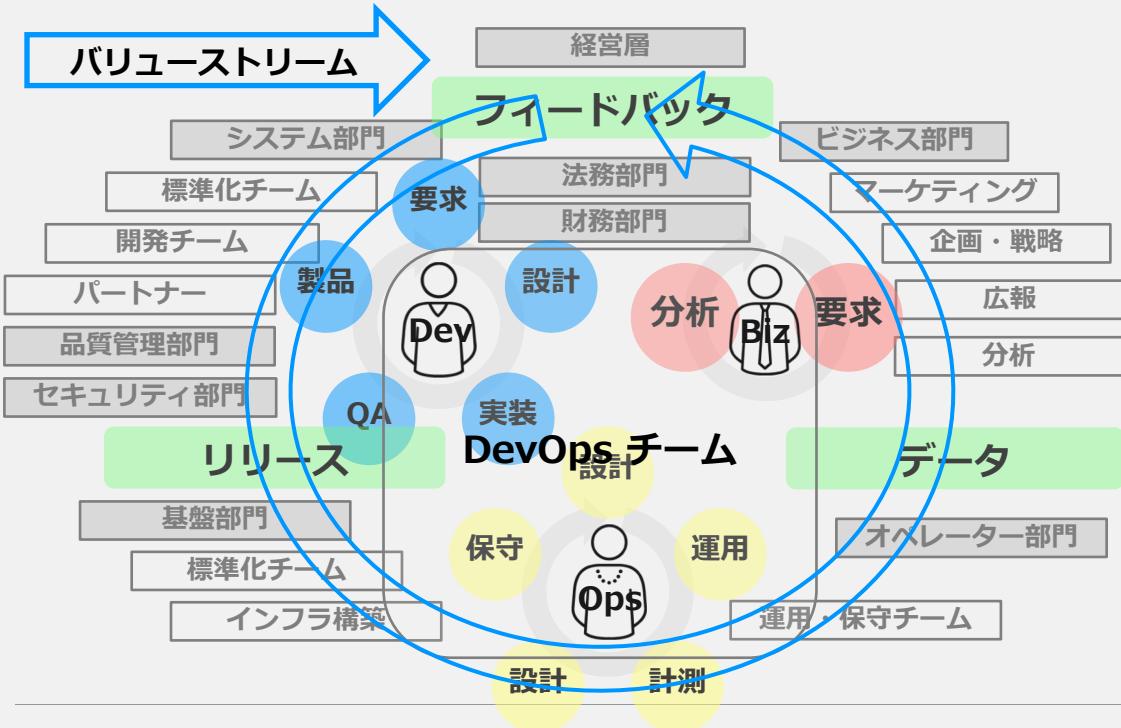
DTX で必要な IT ビジネス運営サイクルを実現するための プロセス、組織・文化、技術 に関するプラクティス



Agile, Lean 起源とする プロセス、組織・文化、技術 関するプラクティスの適用範囲を拡張
⇒ DevOps

DevOps に必要なプロセス とは？

- ・ アイデアを実現しその価値を 提供・検証 できる状態にするまでの一連の流れ (Value Stream)
- ・ Value Stream の 効果性・効率性 を最大化するための プロセス、組織・文化、技術 の継続的改善



バリューストリームの最適化

- ・ プロダクトのリリースと継続的改善活動に関係する全社的(組織横断的)なフローの(バリューストリーム)整理と課題(ボトルネック等)の発見・解決

DevOps チームの構成と支援

- ・ バリューストリームの最適化において重要な改善要因となるプロセスを担当するメンバーから構成される DevOps チームを組織
- ・ DevOps チームを全社的(組織横断的)に支援する体制・プロセス・指針の確立

メトリクスの体系

計測対象	メトリクスの分類		検証対象/改善対象
ユーザ			魅力品質
プロダクト	テスト	静的	魅力品質 構造の品質 (内部品質)
		動的	動作の品質 (外部品質)
	システム		運用の品質
プロジェクト	プロセス		活動の品質
ビジネス			ビジネスの品質

品質間にはトレードオフの関係が発生する
可能性があることを意識

用語の定義

- 要求：プロダクトに求められていること
- 要件：要求を実現するためにしなければならないこと

Scrum における“品質”の考え方

品質区分	製品品質	技術品質
品質の考え方	<p>プロダクトの価値</p> <ul style="list-style-type: none">・(機能) 要求の充足度・Market Value (市場価値)・製品/サービスの魅力や有用性	<p>プロダクト提供者が製品出荷時に担保すべき横断的な価値</p> <ul style="list-style-type: none">・(非機能) 要求の充足度・製品/サービスの安全性や信頼性
Scrum の ルール	<ul style="list-style-type: none">・Acceptance Criteria・Product Backlog	<ul style="list-style-type: none">・Definition of Done
例	<ul style="list-style-type: none">・食品のおいしさ	<ul style="list-style-type: none">・食品の安全性

用語の定義

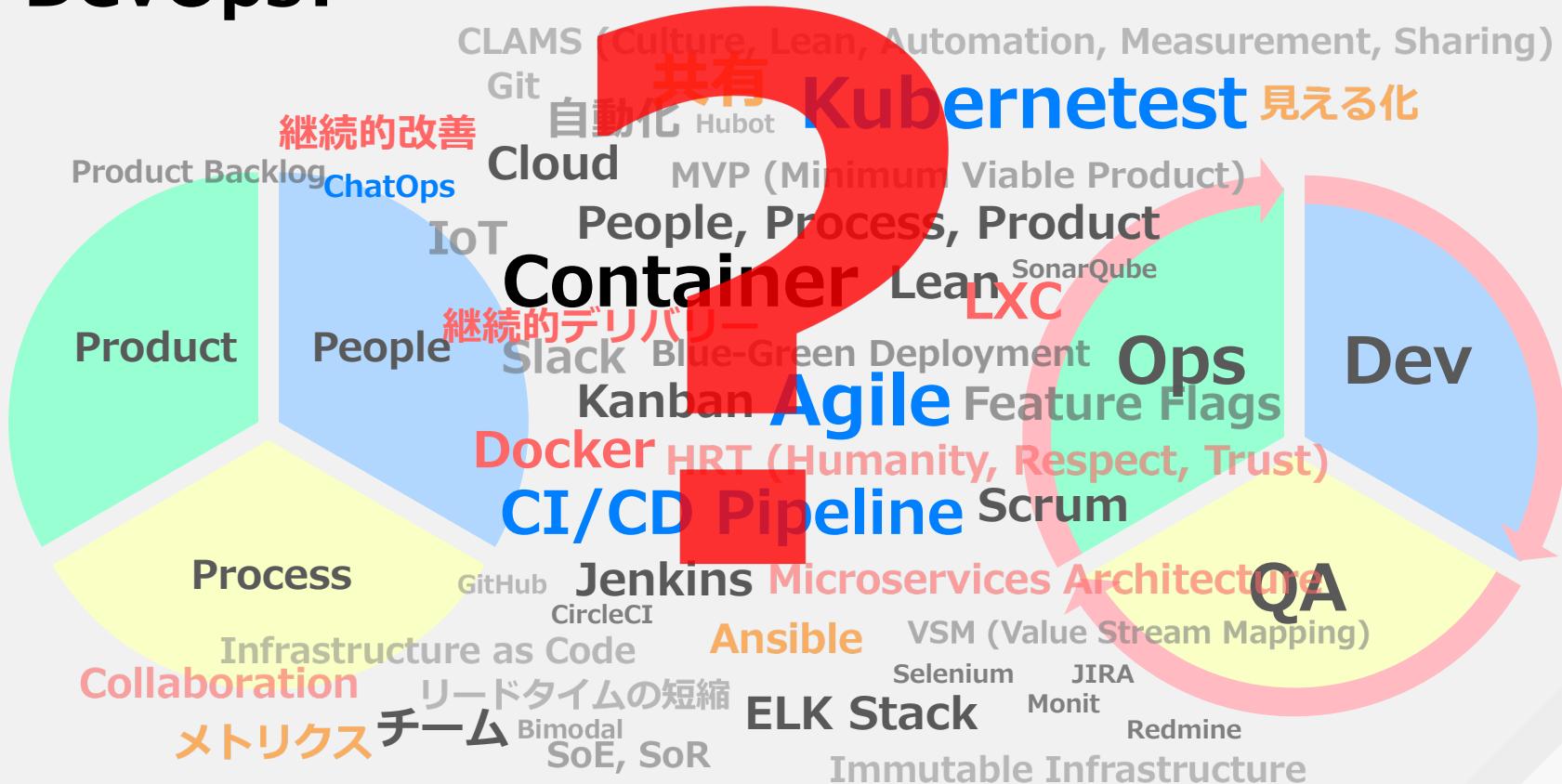
- ・ **Acceptance Criteria : 受け入れ条件**
 - ・個々の Backlog が満たさなければならない (機能要求関連の) 条件 ※ 個々の Backlog で異なる。
- ・ **Definition of Done (DoD) : 完了の定義**
 - ・出荷可能なプロダクトするために Backlog 共通で満たさなければならない (非機能要求関連の) 条件 ※ 全ての DoD を完了しなければ出荷可能とはならない。Sprint で対応できなかった条件は Undone として管理 (以降の Sprint の負債) される。

ソフトウェア品質特性

品質特性	副特性	概要	品質特性	副特性	概要
機能適合性		実装された機能が要求を満たしている度合	使用性		効果的、効率的に利用できる度合
	完全性	利用目的に対してユーザの使用上必要十分な機能が提供されている度合		理解性	使い方に関する理解し易さの度合
	正確性	必要な精度で正しい結果が得られる度合		習得性	使い方の学習のし易さの度合
	適切性	機能がその目的・処理を円滑に遂行する度合		運用性	運用・保守のし易さの度合
性能効率性		実行時性能や資源効率が要求を満たし妥当である度合		エラー防止性	誤操作の起きにくさの度合
	時間効率性	応答時間、処理時間等の処理能力の度合		操作性	操作のし易さ・一貫性の度合
	資源効率性	実行時の資源の量、種類 (CPU, Memory, Disk, Network 等のリソース)		デザイン性	ユーザインターフェースデザイン・操作の一貫性・親しみ易さ・満足感の度合
信頼性		システムが必要時に正しく継続動作できる度合		アクセシビリティ	様々なスキルや特徴をもったユーザが利用できる度合
	成熟性	通常時に故障を回避できる度合	セキュリティ		不正アクセスの防止、データの保護の度合
	可用性	必要時に継続して稼働できる度合		機密保持性	許可されたユーザのみがアクセスできるようデータを保護する度合
	障害許容性	障害時に運用できる度合		インテグリティ	アクセス権無しでのプログラムやデータへの変更を防止する度合
	回復性	障害時にシステムの復旧 (データを含む回復、再構築) できる度合		否認防止性	イベント・アクションの発生を証明する度合
互換性		他のシステムと正しく相互作用できる度合		責任追跡性	アクションが一位的に追跡できる度合
	共存性	共通の環境で稼働する他のシステムと資源を適切に共有できる度合		真正性	ユーザ、プロセス、システム、データ等が本物 (同一性) であることを証明できる度合
	相互運用性	他のシステムと連携できる度合		セキュリティ標準適合性	セキュリティに関する法律、規約、規格を遵守していること
保守性		効果的、効率的に維持や変更ができる度合	移植性		効率的、効果的に別環境に移植できる度合
	モジュール性	変更の影響範囲が局所化されるよう、コンポーネント間が独立した構成となっている度合		環境順応性	別の環境に効率的、効率的に対応できる度合
	再利用性	コンポーネントが新たなシステム構築において利用できる度合		設置性	副作用を与えずインストール/アンインストールする効果性、効率性の度合
	解析性	変更部分や障害原因の特定のための診断や修正箇所の識別・影響の評価の効果性、効率性の度合		置換性	同一の目的、環境下で他のコンポーネントに入れ替えられる度合
	変更性	欠陥や品質の低下なく変更が効果的、効率的に行える度合			
	試験性 (テスト容易性)	テストの設計・実施の効果性・効率性の度合			

導入編

DevOps?



DevOps?

- ・ 目的は何なのか？
- ・ どんな知識が必要なのか？
- ・ 具体的に何を実践すれば良いのか？
- ・ どのように始めれば良いのか？

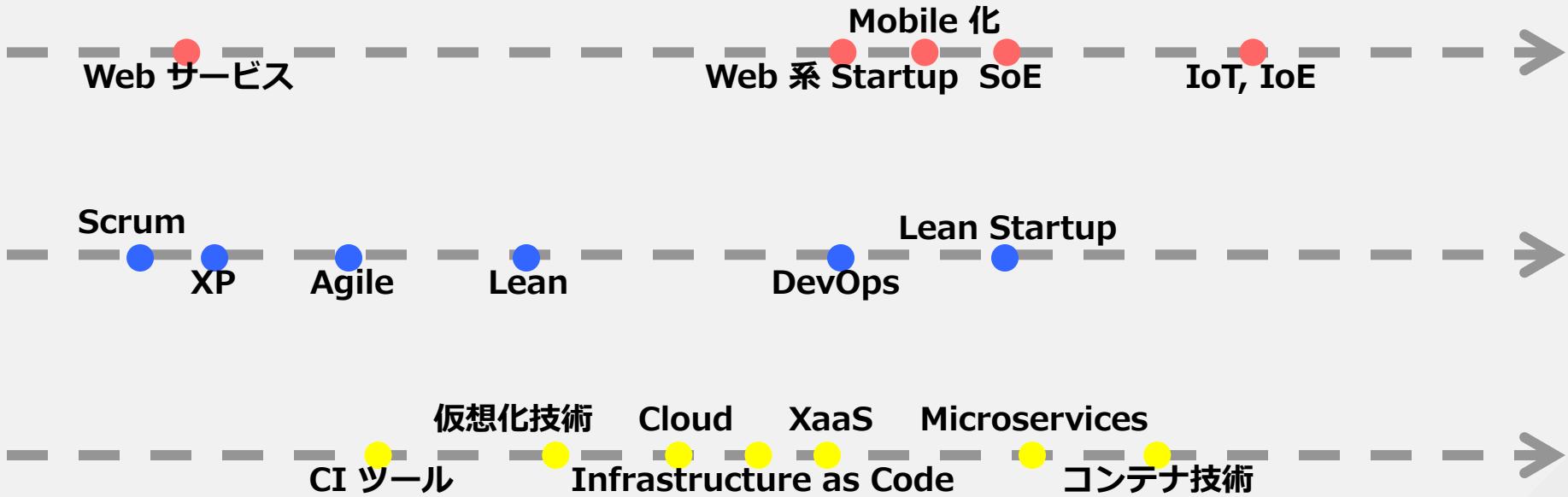
Agenda – 導入編

- DevOps 周辺の歴史
- DevOps と ビジネス
- DevOps と IT 運営プロセス
- DevOps と 技術・ツール
- DevOps の 目的 と 主要成功要因
- Red Hat DevOps Discovery Workshop による DevOps の 実践モデル
- Workshop 実施に向けたディスカッション

DevOps 周辺の歴史

DevOps 周辺の歴史

History around DevOps



DevOps 周辺の歴史

History around DevOps



DevOps と ビジネス

IT ビジネス と IT 運営プロセス の歴史

相関的視点で見る IT ビジネス と IT 運営プロセス の歴史



Agile

- ・ 顧客満足を最優先
- ・ 価値のあるソフトウェアを早く継続的に提供
- ・ 要求の変更を歓迎
- ・ 短い時間間隔でリリース
- ・ 持続可能な開発を促進
- ・ 定期的な振り返りと最適な調整

Lean

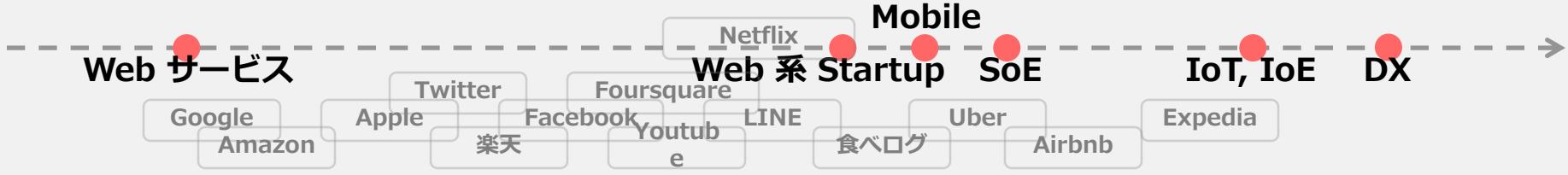
- ・ 顧客重視
- ・ 知識の創造
- ・ 早く提供 (リードタイムの短縮)
- ・ 全体最適化
- ・ フィードバック
- ・ イテレーション

※ アジャイルソフトウェア開発の 12 の原則 [8] の一部抜粋

※ リーンソフトウェア開発から抜粋 [4], [14]

IT ビジネス の歴史

IT ビジネス の変化の特徴



ビジネスの IT (web) への移行 → web のプラットフォーム化 → ビジネスの IT (web) 化

- ・消費行動 の web 化
- ・消費者データのデジタル化
- ・マーケティング手法の変化
- ・関連ビジネス の IT 化

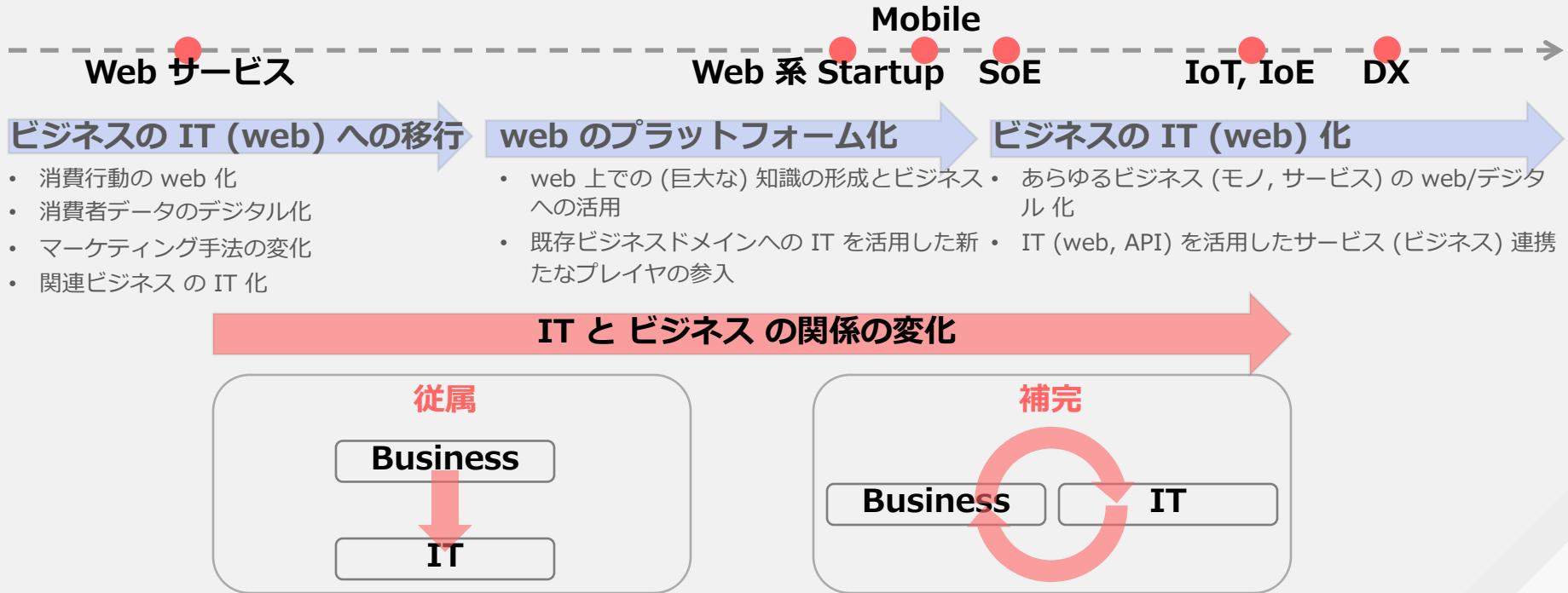
- ・web 上での (巨大な) 知識の形成とビジネスへの活用
- ・既存ビジネスドメインへの IT を活用した新たなプレイヤーの参入

ビジネスの IT (web) 化

- ・あらゆるビジネス (モノ, サービス) の web/デジタル 化
- ・IT (web, API) を活用したサービス (ビジネス) 連携

IT ビジネス の歴史

IT と ビジネス 関係の変化



IT ビジネス の特徴



- IT が ビジネスの 価値・成功 に直結する 領域の拡大・多様化
- IT ビジネス の 競争の激化 と 変化の高速化
- IT = 競争力・顧客に届ける価値



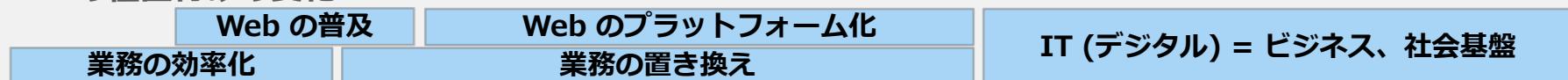
ビジネスにおける IT の特徴

- 俊敏性 (スピード) : アイデアの実現や変化への対応にはスピードが求められる。
- 非予見的 : アイデアや変化を完全に予測することは困難である。
- 重要性・複雑性 : ビジネスにおける IT の 重要性・複雑性 (関連領域の質的・量的) は増大する。

IT 潮流 – DevOps 登場の時代背景

IT = 社会基盤、ビジネスの中心

IT の位置付けの変化



技術的誘因



デジタルディスラプション (※1) の誘因



※1：最新の IT 技術を駆使し業界を超えて既存産業（商品、サービス、仕組み）にもたらされる創造的破壊・破壊的イノベーション（※3）

※2：研究・開発、調達、製造、マーケティング、販売、デリバリー、etc.

※3：2013 年に米国のアナリスト ジェイムズ・マキヴェイ 氏が同名の書籍を発表

DevOps 登場に至る IT ビジネスの特徴

IT 駆動

- ・ IT 技術を活用したビジネスモデル、IT 中心の組織
- ・ 研究・開発・顧客分析、調達、製造、マーケティング、販売、デリバリ に効果的に IT を活用

具現化・進化・変化 のスピード

- ・ IT 運営プロセスの効率化 と IT 活用コストの低下 によるアイデア実現の高速化
- ・ IT 運営プロセス と IT 技術 の 効果的・効率的 な活用による 進化・変化 の高速化

顧客主導, 顧客の進化

- ・ 顧客が求める 価値・体験 主導
- ・ 瞬時に拡散する顧客の評価主導、マーケティング 手法・プロセス や 顧客へのアプローチ方法の変化
- ・ 消費者は IT を駆使し入手した膨大な情報を活用し多様な選択肢から判断し即座に行動
- ・ 消費者の要求は 商品・サービス・仕組み の進化に合わせ急速に進化・多様化

業界を超えたビジネスモデル

- ・ IT 技術を活用して既存産業に新たなビジネスモデルを構築
- ・ 既存のコアビジネスに隣接するビジネスドメインへの進出
- ・ 他社との連携による複合サービス

DevOps 登場に至る IT 運営の考え方

- ・ 多様化し急速に変化する環境において完璧な プロダクト・サービス を予見することは困難という前提
- ・ 仮説/検証 型の 漸進的・探索的・適応的 アプローチが最も 効率的・効果的
- ・ 顧客からのフィードバックによる継続的改善
- ・ 効果的・効率的なリリースとフィードバックループ
- ・ 最新 IT 技術の積極的な活用

DevOps を駆動する IT 運営戦略

ビジネスドメイン・ビジネスモデル の 繼続的再定義

- 多様かつ高速に変化するビジネス環境における自らの提供する価値とビジネス戦略（ビジネスドメイン・ビジネスモデル）の継続的な分析と再定義

顧客分析 と 繼続的改善

- 顧客データの収集と顧客の求める 価値・経験 の分析
- 漸進的・仮説/検証 型アプローチによる 繼続的改善・ビジネス戦略へのフィードバック

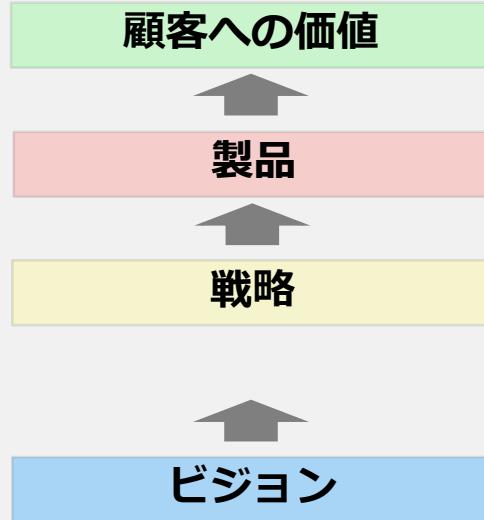
リードタイム短縮と継続的改善のための IT 運営プロセス - DevOps

- アイデアの早急な構築（リリース）
- 仮説/検証 および フィードバック プロセスの確立とプロセスサイクルの高速化・効率化
- 顧客分析に従った 漸進的・仮説/検証 型アプローチによる継続的改善

DevOps 実践のための組織改革

- 組織横断的（Biz, Dev, Ops）なチーム体制の確立、ビジョンの共有、文化改革
- コアビジネスドメイン・コアビジネスモデル に関する IT の内製化
- IT 活用・内製化 のため IT スキル
- バリューチェーン全体での戦略的な IT の活用

IT ビジネスにおける 戦略・製品 と 価値



顧客への価値

- ユーザが 客観的・主観的に認識できる 効果・体験

製品

- 戦略を具現化し顧客に価値を提供するための成果物
- 機能 (ユーザストーリー)、UX (ユーザーエクスペリエンス)、カスタマージャーニー 等

戦略

- ビジョンを具体化し持続可能な事業を構築するための体系的で検証可能な 仮説・計画
- ビジネス目標 (生産性の向上 等)、ビジネスモデル、製品ロードマップ、競合・提携、予想顧客、等

ビジョン

- ビジネスの目的と方向性

顧客分析と継続的改善

分析・検証による学び (仮説/検証型アプローチ)

- 顧客分析および仮説 (アイデア) の 定量・定性 的検証とフィードバック

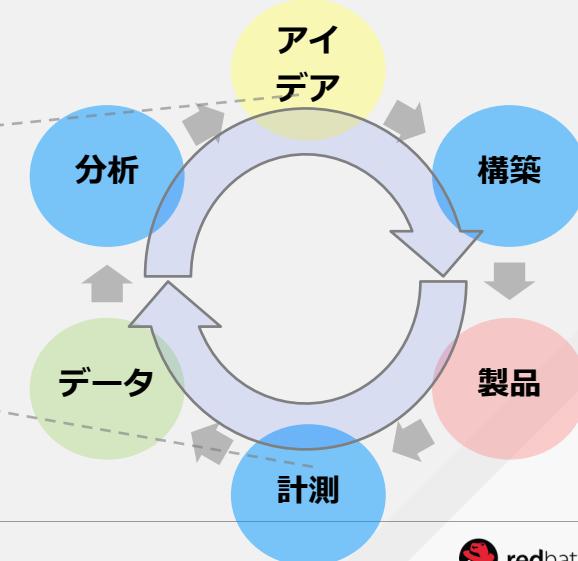
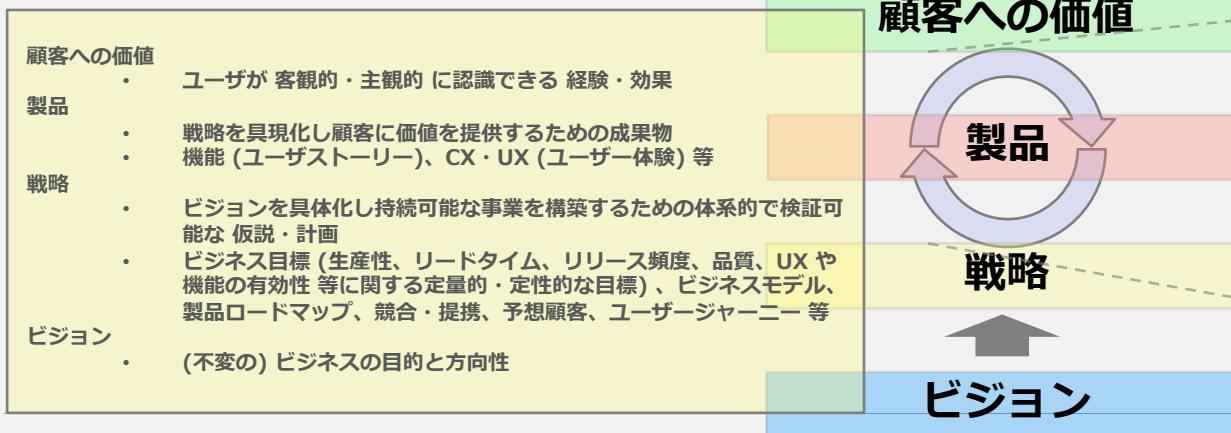
フィードバックループによる継続的改善 (漸進/探索/適応的アプローチ)

MVP (minimum Viable Product)、小さなバッチサイズ

- ユーザの価値に直結する必要最小限の機能 (アイデア) のリリース

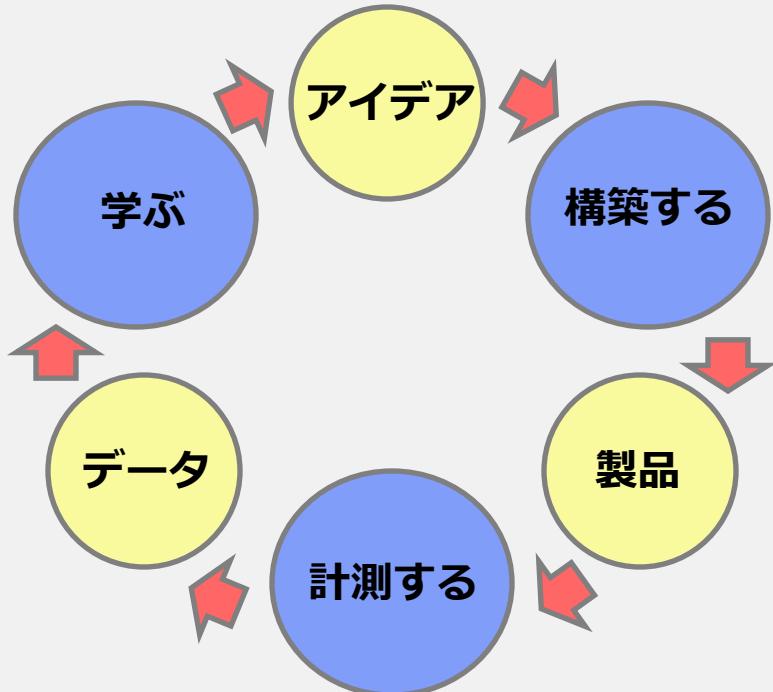
フィードバックループの高速化・効率化

- 進化・変化への対応・戦略の変更 スピードの高速化
- ユーザの価値に直結しないあらゆる“ムダ”の排除と生産性の向上



IT ビジネス運営のフィードバックループ

IT ビジネス運営のプラクティス - 検証による学び と MVP (Minimum Viable Product)



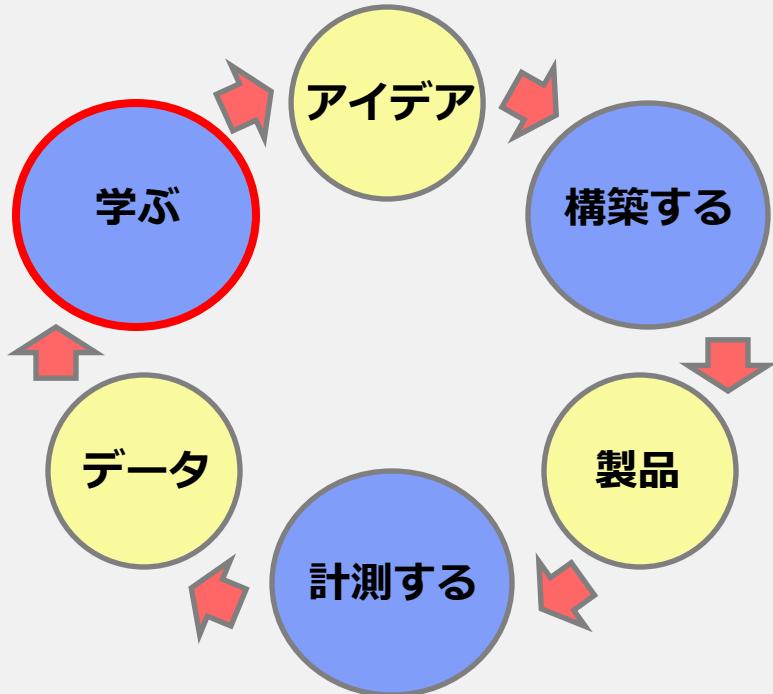
検証による学び

- ・ 仮説 (アイデア) の定量・定性的な効果計測
- ・ ユーザが欲しない機能を作る“ムダ”の排除
- ・ ユーザの価値に直結しないあらゆる“ムダ”の排除と生産性の向上
- ・ A/B リリース

MVP (minimum Viable Product)

- ・ ユーザの価値に直結する必要最小限の機能 (アイデア) のリリース

IT ビジネス のフィードバックループ



構築・計測・学習 ループに要する時間を最小化



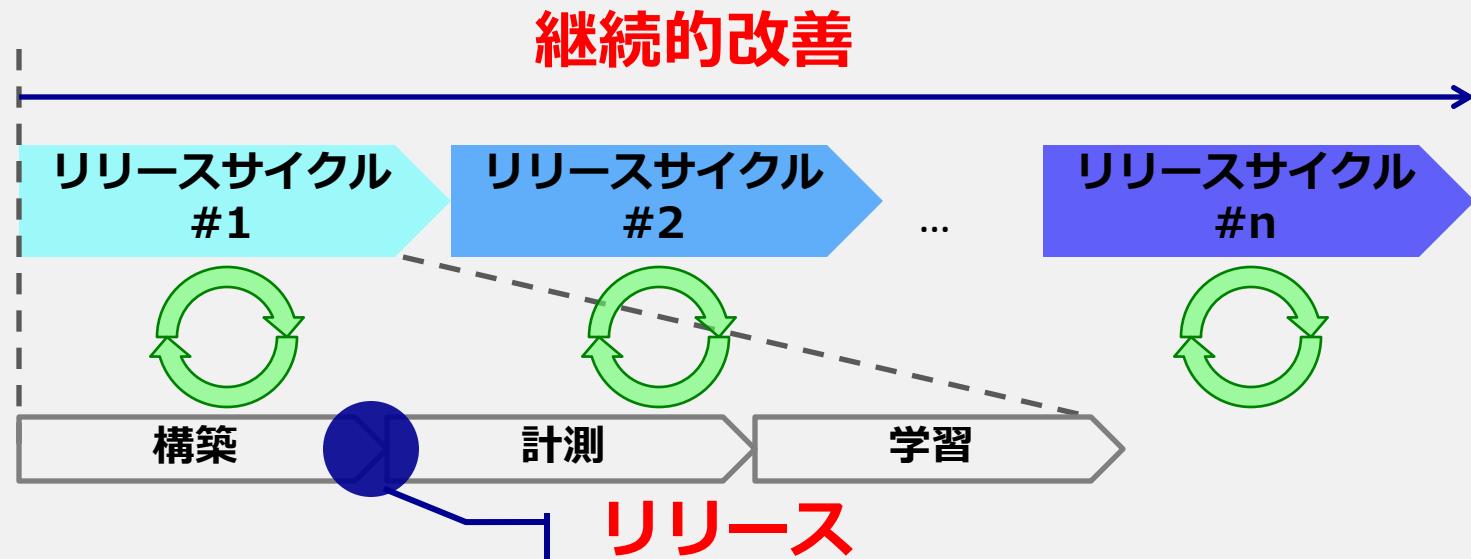
改善・変化への対応の高速化

改善・変化への対応の機会の増加



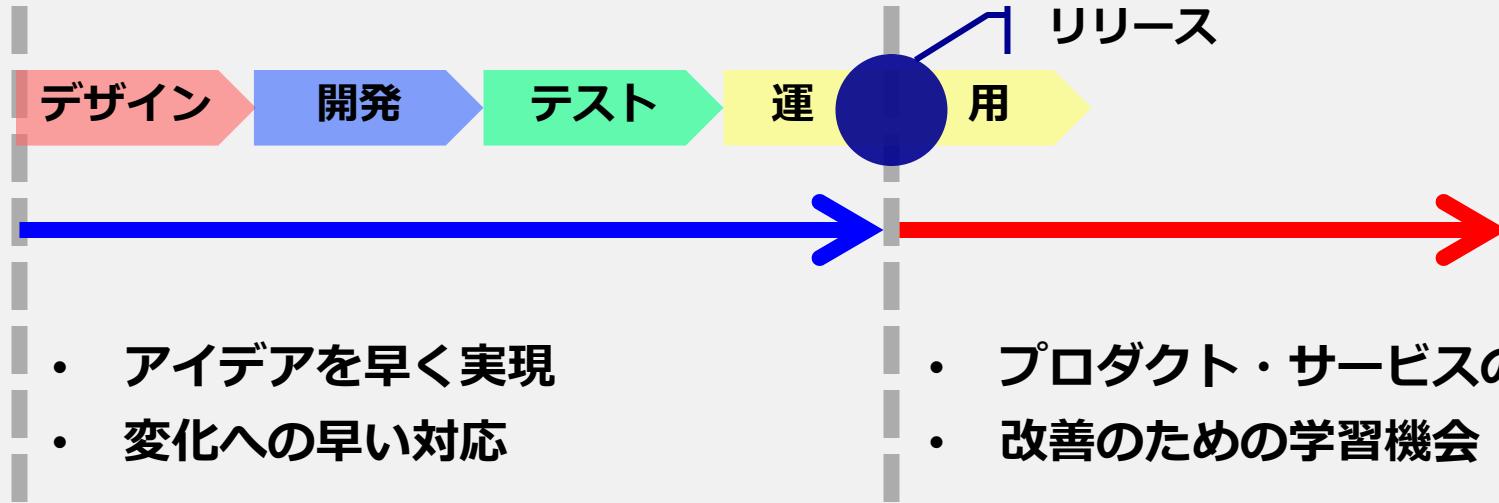
プロダクト・サービスの価値の向上

フィードバックループと IT サービス運営



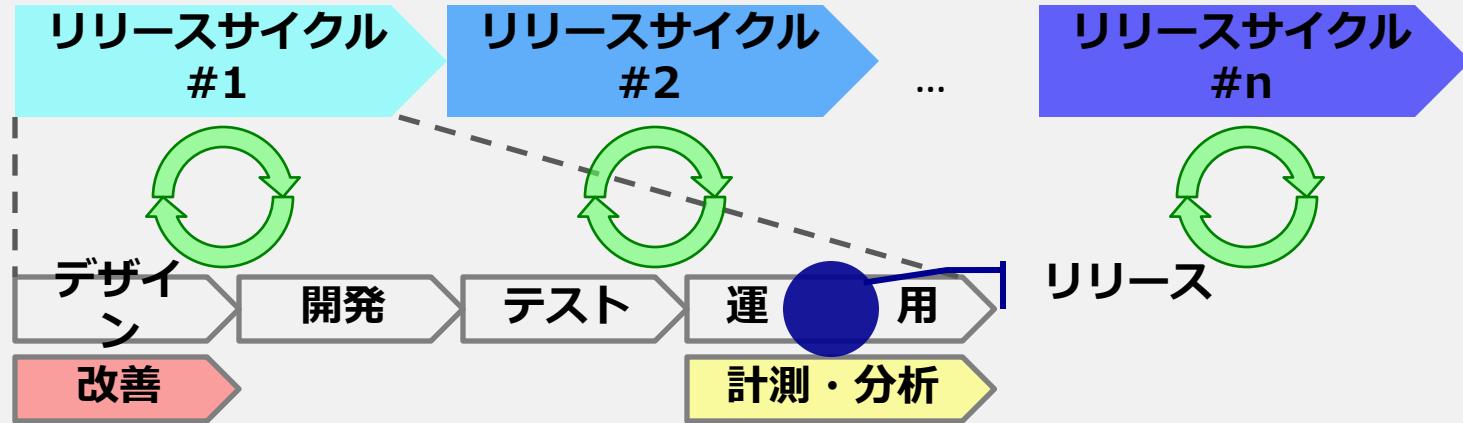
DevOps な IT サービス運営 – リリース

IT 運営の考え方



DevOps な IT サービス運営 – 繼続的改善

IT 運営の考え方



- ・ サイクル数 を増やし 分析・改善・学習 の機会を得る
- ・ サイクル数 を増やし変化に対応する機会を得る

DevOps の目的

- ・ ビジネスの価値・競争力の向上・成長
- ・ ビジネスニーズの変化に迅速に対応



- ・ より早くサービスをリリースする
- ・ より多くフィードバックをし継続的にサービスを改善

DevOps は
ビジネスの価値・競争力の向上・成長
を目的とした
リードタイム短縮 および 継続的改善
のための IT 運営プラクティス

DevOps と IT 運営プロセス

DevOps ≈ Agile, Lean

DevOps 周辺の歴史

DevOps と プロセス



Agile

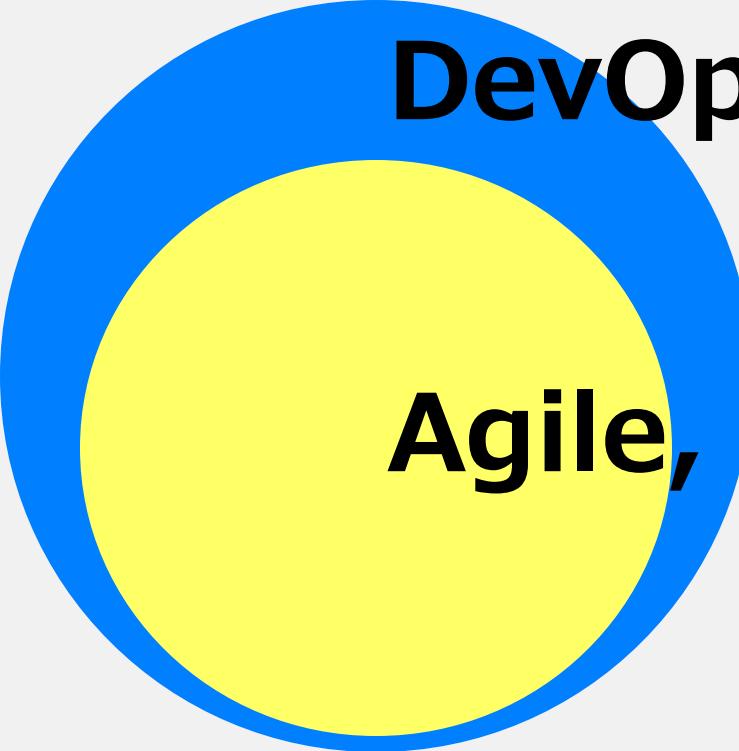
- ・ 顧客満足を最優先
- ・ 価値のあるソフトウェアを早く継続的に提供
- ・ 要求の変更を歓迎
- ・ 短い時間間隔でリリース
- ・ 持続可能な開発を促進
- ・ 定期的な振り返りと最適な調整

Lean

- ・ 顧客重視
- ・ 知識の創造
- ・ 早く提供 (リードタイムの短縮)
- ・ 全体最適化
- ・ フィードバック
- ・ イテレーション

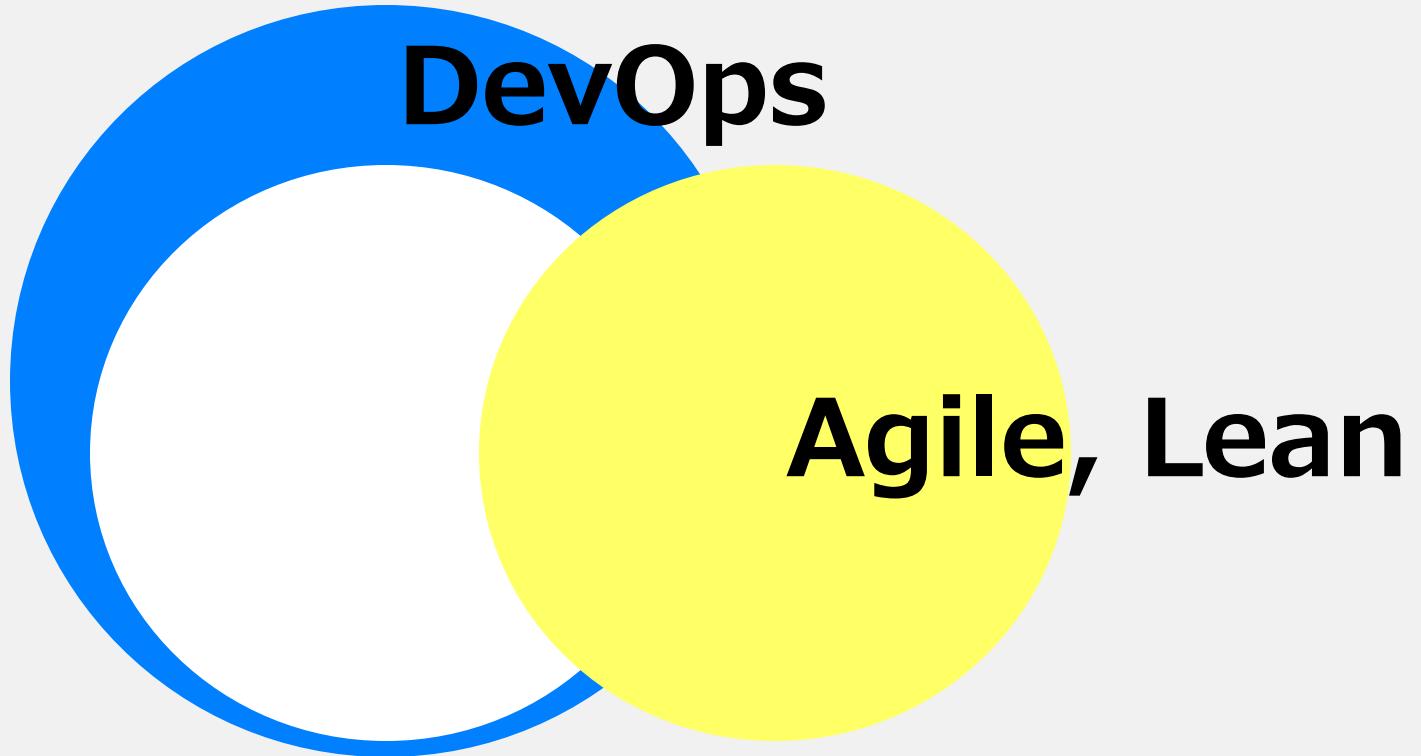
※ アジャイルソフトウェア開発の 12 の原則 [8] の一部抜粋

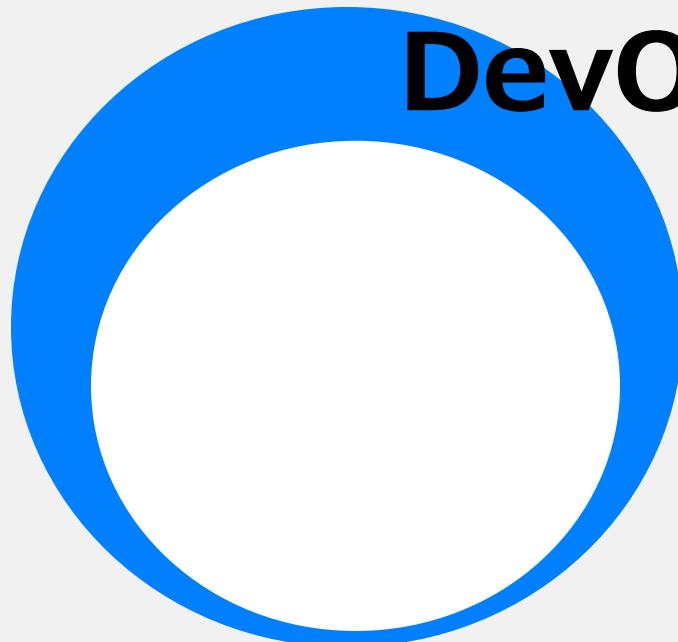
※ リーンソフトウェア開発から抜粋 [4], [14]



DevOps

Agile, Lean





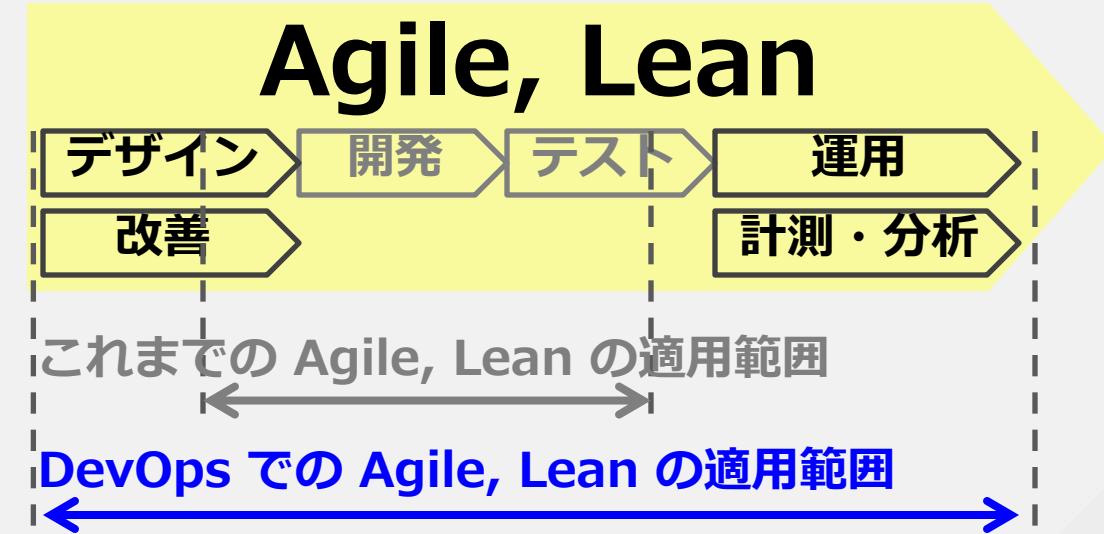
DevOps

=

?

DevOps

=



DevOps の本質

プロセス

Agile, Lean



Agile, Lean の IT 運営プロセス 全体への適用

Agile, Lean



Biz



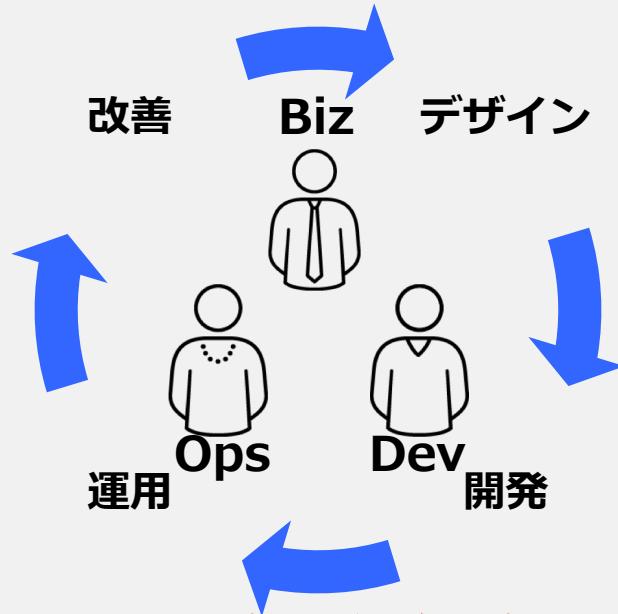
Dev



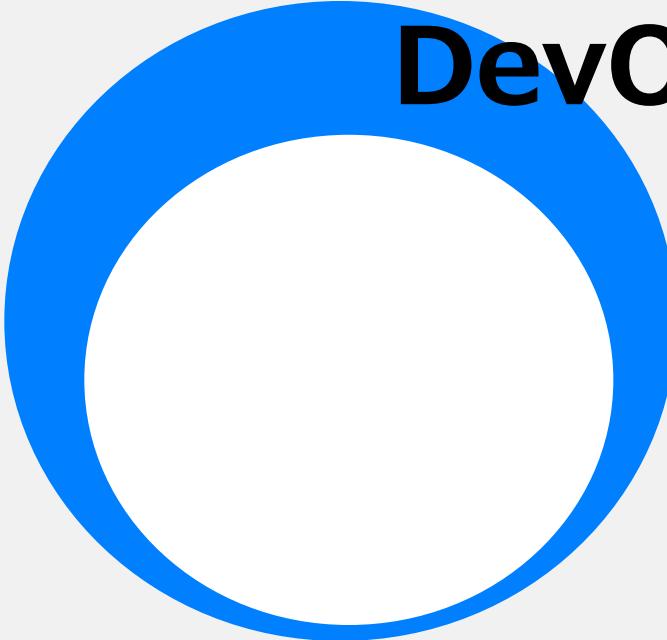
Ops

DevOps の本質

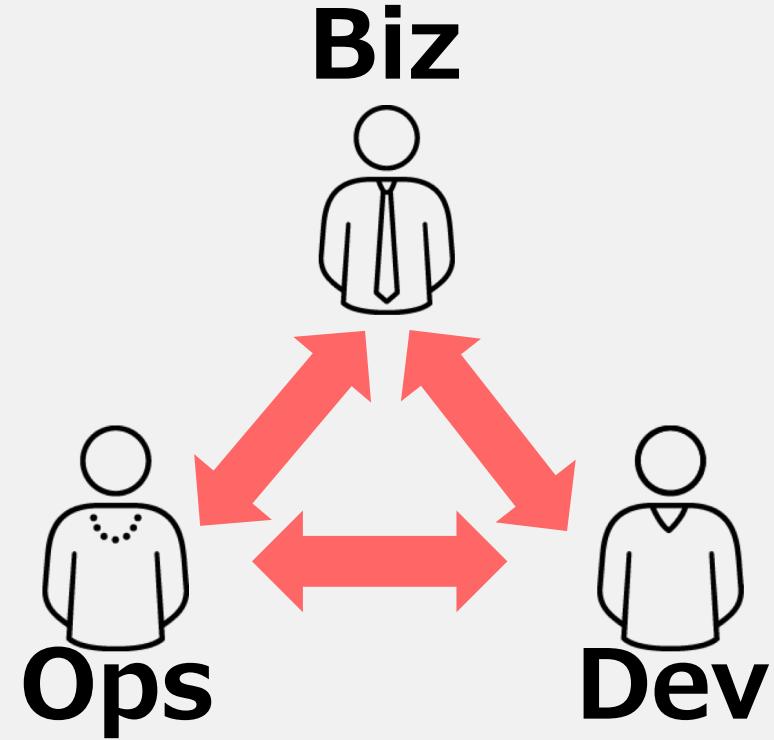
プロセス



- Biz, Dev, Ope 横断的な継続的改善活動
- Agile, Lean の Biz, Dev, Ops 横断的な実践

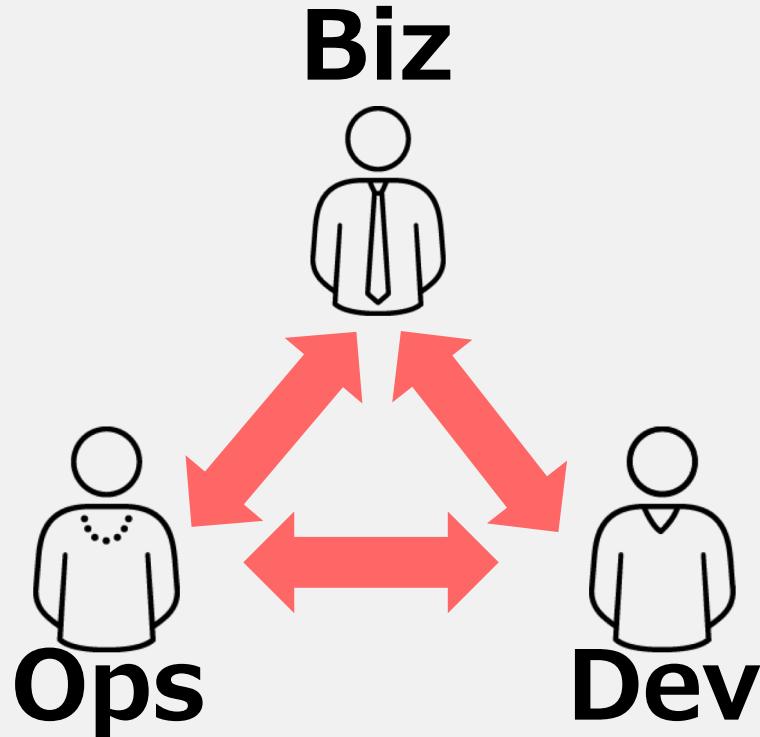


DevOps



DevOps の本質

人・文化



- ビジネスファースト
- 変化・改善に前向きなマインド
- Biz, Dev, Ops の協調
- 協調のためのマインド
- Biz, Dev, Ops チームドリブン

DevOps と 技術・ツール

DevOps が 技術・ツール に求めるもの

DevOps と 技術・ツール

適切な 品質、コスト を伴った

- 繼続的変更 (継続的改善) の実現
- 変更スピード (リードタイム短縮) の向上
- チーム・プロセスへの適合

継続的変更・変更スピードに関連する品質特性

保守性

生産性

モジュール性

変更の影響範囲が局所化されている度合いやモジュール間の依存性の度合

一貫性

記法・用語・概念が一貫していること

再利用性

ソフトウェアコンポーネントが他のシステムを構築する際に利用できる度合

解析性

障害・変更箇所の識別のし易さや変更の影響範囲の特定のし易さの度合

変更容易性

欠陥や品質の低下なく変更が効果的・効率的に行える度合

テスト容易性

テストポリシー・テスト評価・テスト実装の効果性・効率性の度合

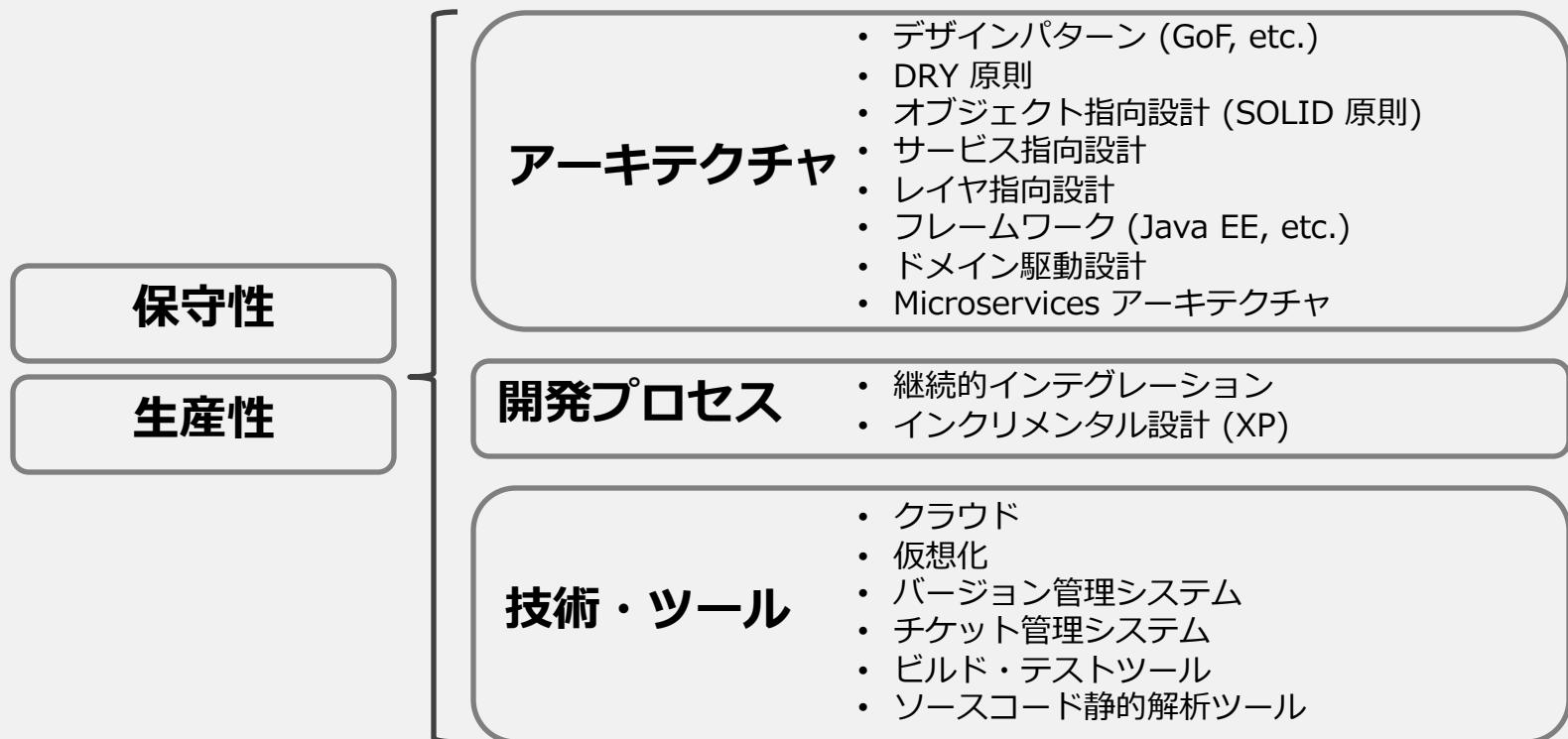
可読性

ソースコードを読む際の、その目的や処理の流れの理解のし易さの度合

簡潔性

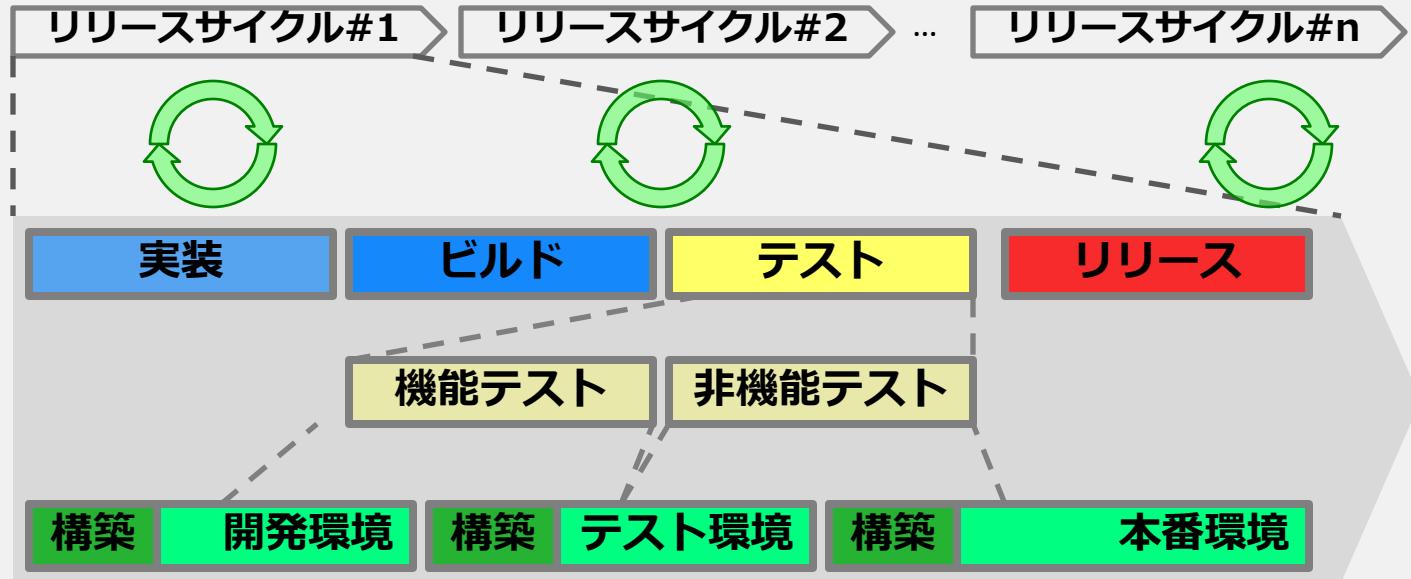
実行されない・冗長性・複雑性の少なさの度合

品質特性を担保するためのアプローチ



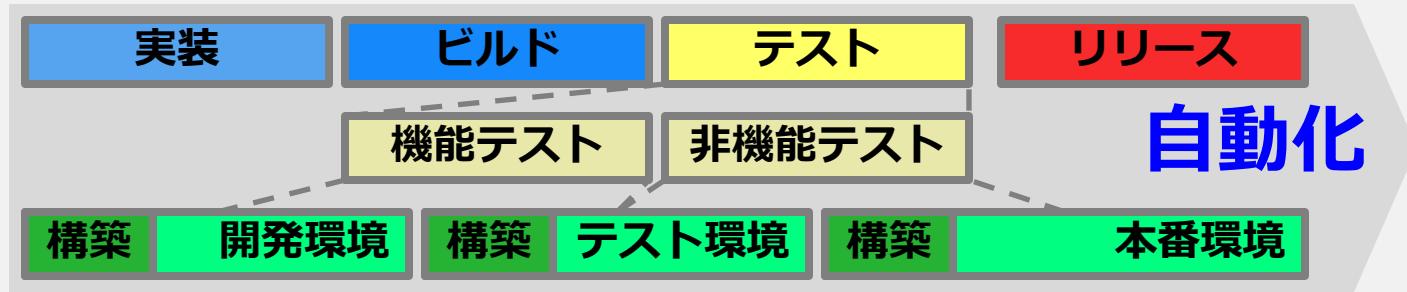
システムインテグレーション

DevOps と 技術・ツール



CI/CD パイプライン

DevOps と 技術・ツール



繰返し行われる手順の**自動化**

- ・ ビルド・構築 の 保守性・生産性 の向上
- ・ テスト・リリース の 保守性・生産性 の向上



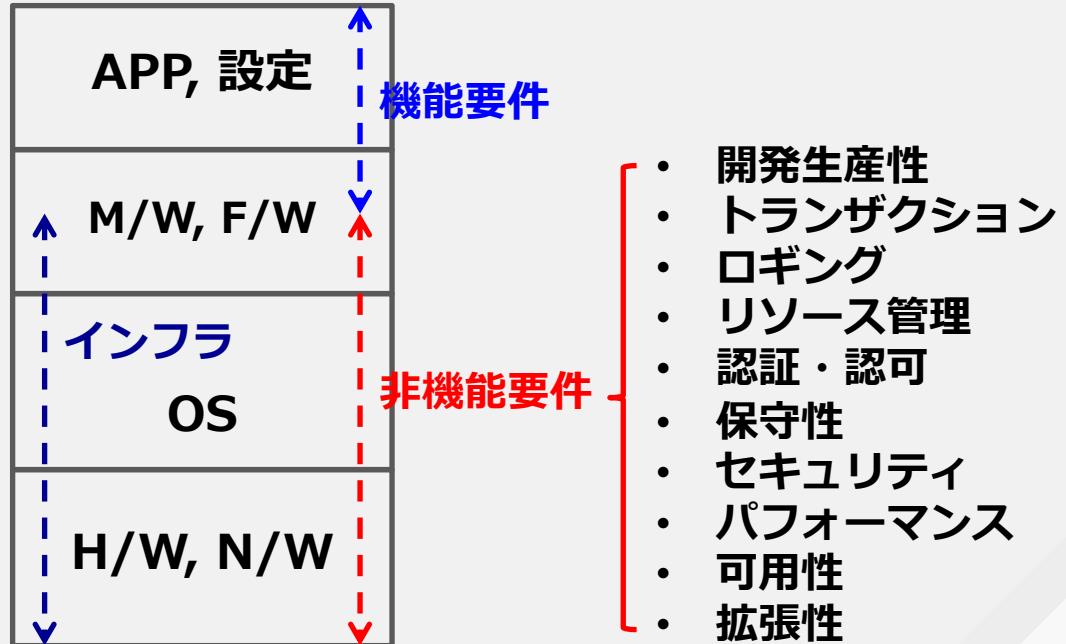
継続的変更の効率と変更スピードの向上

自動化とインフラレイヤの特性

DevOps と 技術・ツール

インフラレイヤの特性

- ・ 構築のコード化：難
- ・ 非機能要件のコード化：難
- ・ 環境依存性：高
- ・ 移植性：低
- ・ テスト容易性・効率性：低
- ・ 構成管理の容易性：低



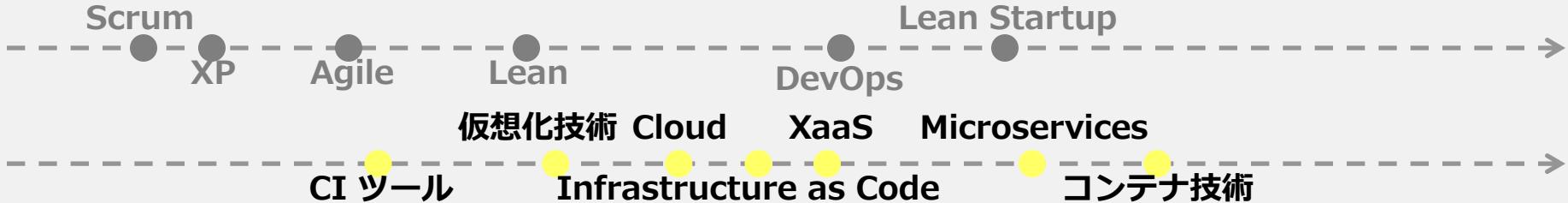
自動化における技術的課題

DevOps と 技術・ツール

- ・ インフラ構築の自動化
- ・ インフラ要件のコード化
- ・ インフラに依存するテストの自動化
- ・ インフラの構成管理の効率化

DevOps 周辺の歴史

DevOps と 技術・ツール



コンテナ

- Docker
- Kubernetes
- OpenShift

インフラの自動化

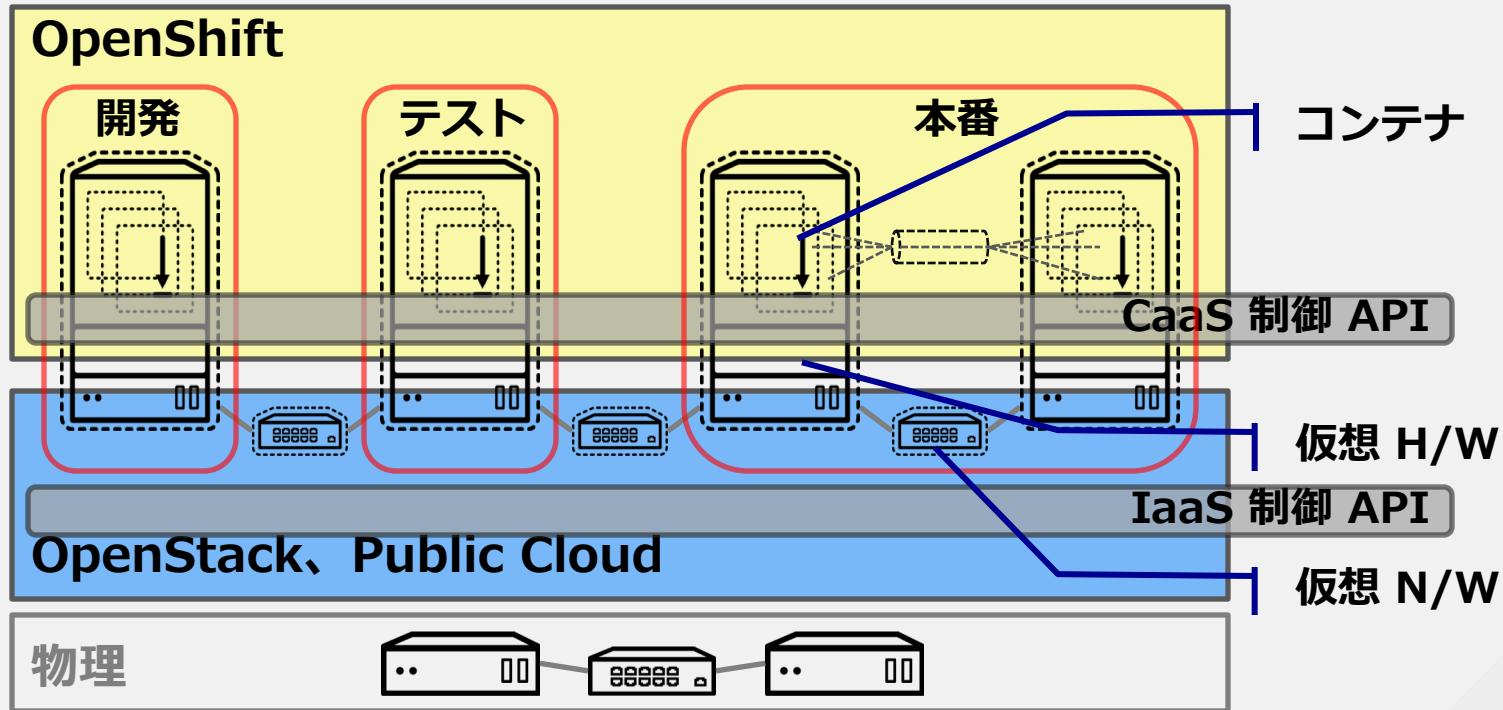
- Ansible
- OpenStack

その他

- Git, Jenkins, Serverspec

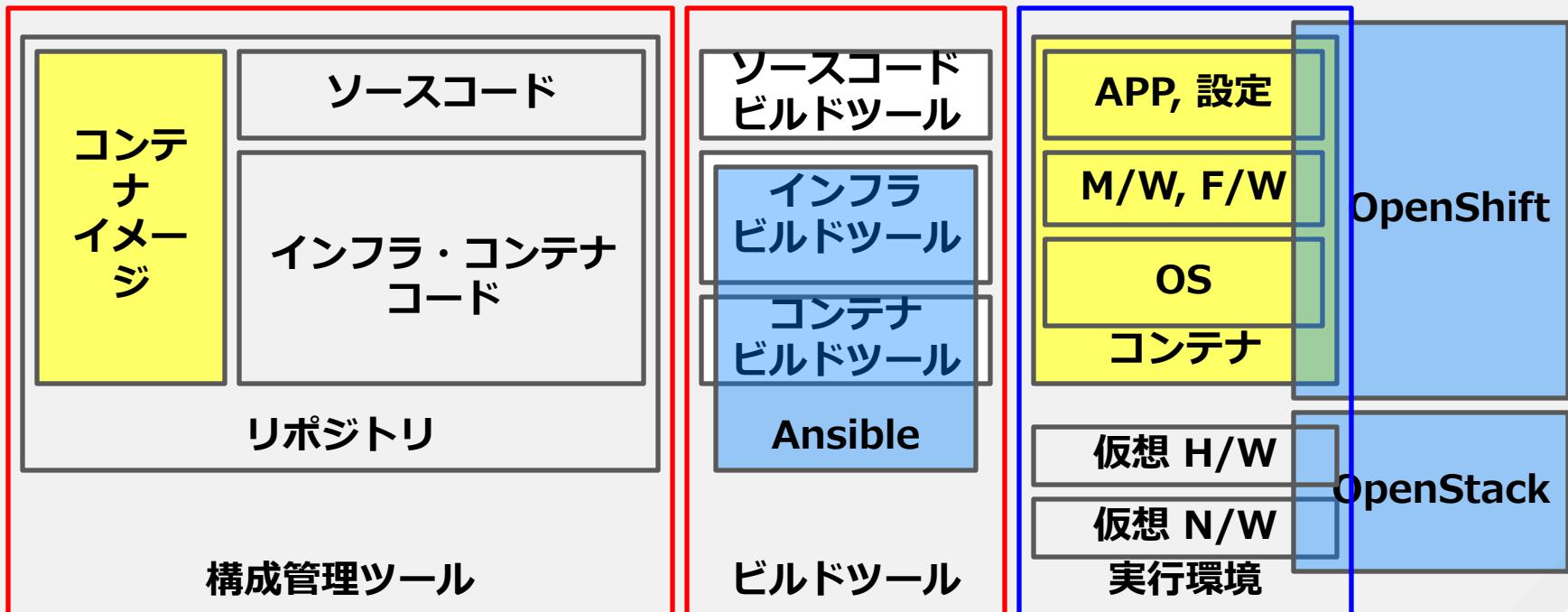
DevOps なインフラ

DevOps と 技術・ツール

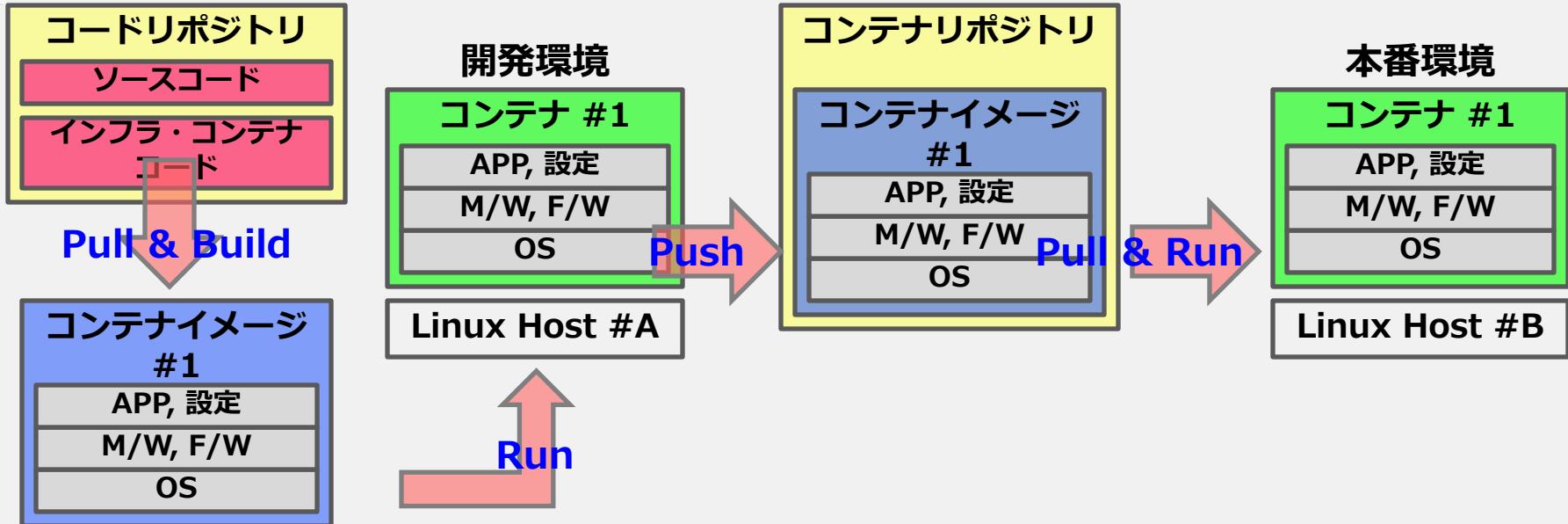


DevOps な 構成管理・ビルド

DevOps と 技術・ツール



コンテナ技術の特徴



- ・ システムをコンテナイイメージとしてパッケージング
- ・ コンテナイイメージはコンテナリポジトリ上で 名前:タグ名 で管理可能
- ・ ホストの異なる任意のコンテナ環境でコンテナイイメージを実行可能

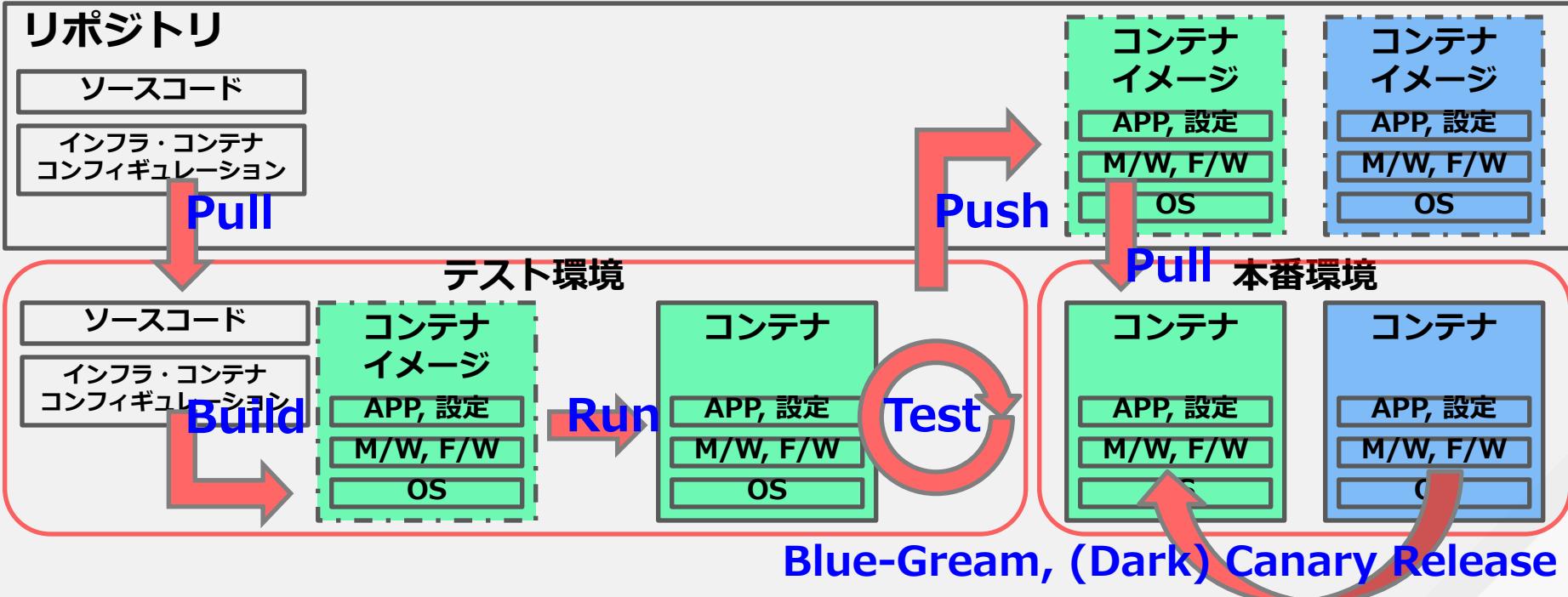
DevOps なインフラの特徴

DevOps と 技術・ツール

- ・ インフラ構築 の コード化・自動化
- ・ インフラ要件 の コード化
 - ・ 制御 API を使用した 非機能要件 の コード化
- ・ インフラ要件 の テストの自動化
- ・ システム の 高い移植性 による 保証性の高いテスト
 - ・ コンテナイメージの push&pull via コンテナリポジトリ
- ・ インフラ の 構成管理 の 効率化

DevOps な CI/CD

DevOps と 技術・ツール



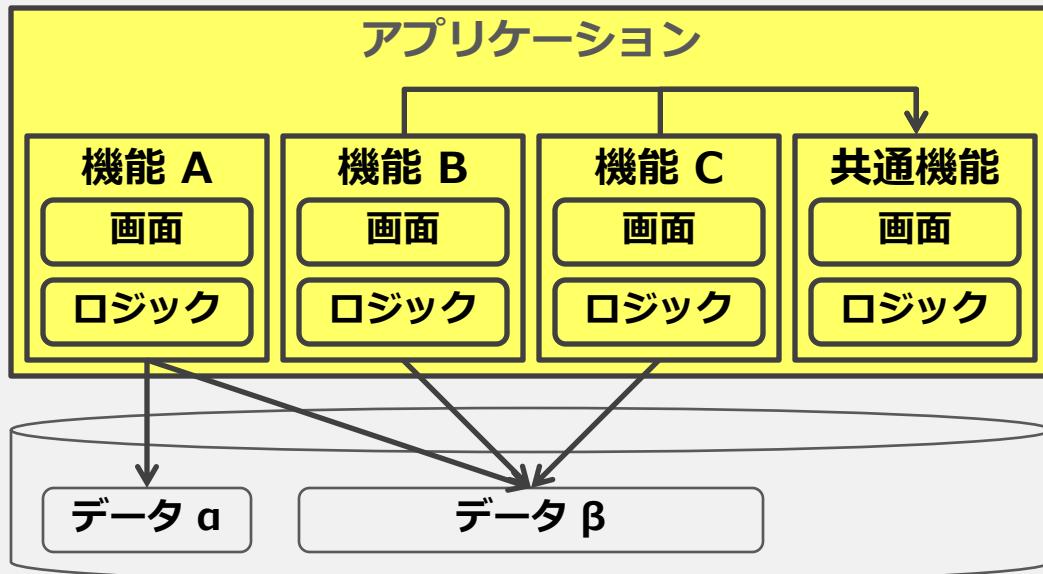
システムの“変更のしやすさ”とアーキテクチャ

DevOps とアーキテクチャ



DevOps と アーキテクチャ

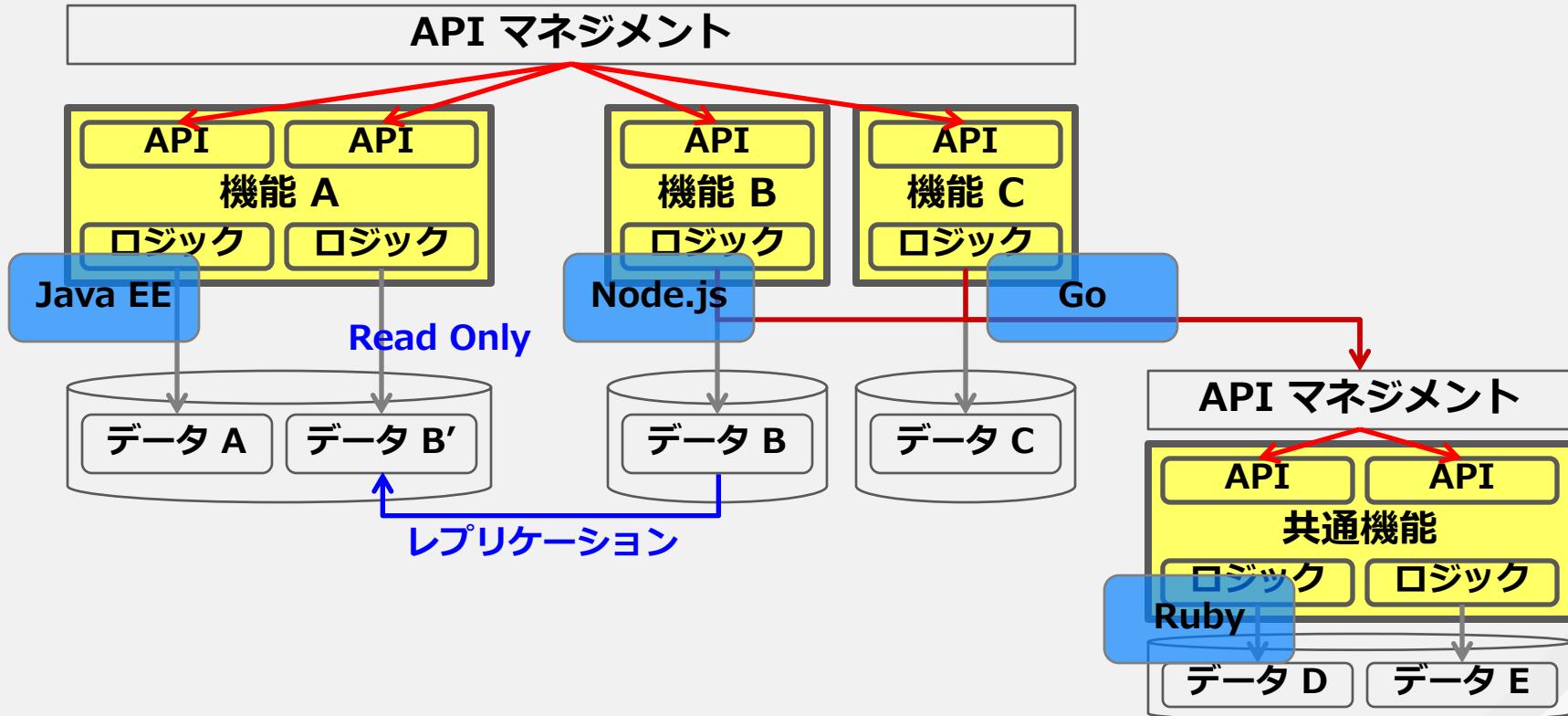
Monolithic アーキテクチャ



問題点

- 機能間の依存性
- 巨大な影響範囲
- 異なる機能間で以下を共有
 - リリースサイクル
 - インフラ

Microservices アーキテクチャ



Microservices アーキテクチャの利点

DevOps と アーキテクチャ

- 疎結合
- 小さな影響範囲 = 変更容易性
- 小さな関心事 = 変更スピード
- 個別のリリースサイクル
- 個別のインフラ = 個別に柔軟に非機能要件に対応
- チームにフィット (コンウェイの法則 [15])
- 技術異質性

DevOps の 主要成功要因

DevOps の 主要成功要因

ビジネス上の戦略的目標

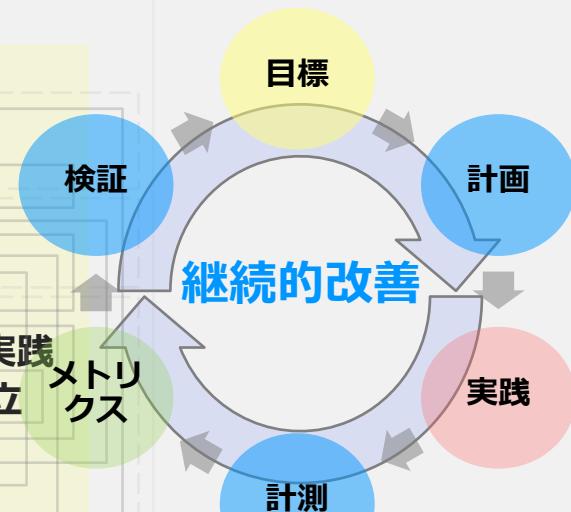
生産性、リードタイム、リリース頻度、品質、UX や機能の有効性 等に関する定量的・定性的な目標

DevOps 実践における優先順位や方向付けの定義

改善効果の目標へのフィードバック

DevOps の主要成功要因

	Biz	Dev	Ops
組織・文化	Agile/Lean の実践 メトリクス戦略		
プロセス	テスト戦略 自動化戦略	Infrastructure as Code の実践 Cloud/Container 基盤確立	
技術	CI/CD の実践 アーキテクチャ戦略		



ビジネス目標を以下の観点で分解し、計測・検証（メトリクスとその分析）によるフィードバック

- ・ 目標の組織（チーム）単位でのブレークダウン
- ・ 目標達成のための要因のブレークダウン
- ・ イテレーション毎の小さな目標へブレークダウン

DevOps 主要成功要因 の概要

主要成功要因	概要
Agile/Lean の実践	<ul style="list-style-type: none">継続的改善・リードタイム短縮を効果的・効率的に行うために組織に Agile/Lean の原則を導入する。Biz,Dev,Ops から構成される自己組織化したチームが Agile/Lean の原則に則った IT 運営を行うための組織・文化を醸成する。Scrum/XP を参考に個々のビジネス・環境・戦略に合った Agile/Lean の原則に則ったプロセスを構築・実践し継続的改善を行う。
CI/CD の実践	<ul style="list-style-type: none">フィードバックと継続的改善を効果的・効率的に行うために CI/CD を実践する。CI/CD の効果性・効率性向上のため、アーキテクチャ、テストに関する戦略の検討、IaC の実践、Cloud/Container 基盤の確立を行う。
アーキテクチャ戦略	<ul style="list-style-type: none">以下の観点でのアーキテクチャの検討・改善を行う。<ul style="list-style-type: none">継続的改善(変更)のための保守性の向上テスト自動化のためのテスト容易性の向上以下の観点での Microservices アーキテクチャの導入検討を行う。<ul style="list-style-type: none">小規模で自己組織化されたチームへの適合リリース頻度、リソース使用量、セキュリティレベル等の違いによるシステムの分割と保守性の向上
テスト戦略	<ul style="list-style-type: none">手動テストの目的・スコープ・プロセスを検討する。単体・結合・システム・回帰 テストの自動化を効果的・効率的に行うためのテストの分類・ポリシー策定を行う。
自動化戦略	<ul style="list-style-type: none">IT 運営の効果性・効率性向上のために、CI/CD、リリース、フォールバック、メトリクス・ログの収集と可視化、各種運用・保守手順等の自動化範囲の拡大の方針・方式の検討を行う。
IaC の実践	<ul style="list-style-type: none">CI/CD、テスト自動化、各種運用・保守手順の自動化の効果性・効率性向上のための Infrastructure as Code の方針・方式の検討を行う。
Cloud/Container 基盤	<ul style="list-style-type: none">IaC、CI/CD、テスト自動化、各種運用・保守手順の自動化の効果性・効率性向上のための Cloud/Container 基盤確立の方針・方式の検討を行う。
メトリクス戦略	<ul style="list-style-type: none">IT 運営の継続的な改善、効果的・効率的・客観的な分析とフィードバックを行うために以下の観点で定性的・定量的なメトリクスの整理と分析・フィードバックプロセスの検討を行う。<ul style="list-style-type: none">ビジネスメトリクス：CVR、機能使用数、行動データ、UX 等のプロダクトのビジネス上の効果測定・改善のためのメトリクスプロセスマトリクス：活動基準原価、サイクルタイム、リリースタイム、ペロシティ、スクラップレート等ソフトウェア(テスト)メトリクス：ソフトウェア静的解析、テスト網羅率等システムメトリクス：CPU, Memory, Disk, I/O 使用率等のリソース使用状況、障害(エラー)発生状況等

実践編

DevOps の実践モデル

DevOps の実践モデル

理解

- DevOps の目的を知る。
- DevOps の網羅的・体系的な知識を得る。
- DevOps の成功要因を知る。
- DevOps の実践方法を知る。

現状分析

- DevOps の成功要因の観点での現状の文化・プロセス・技術の状況を知る。

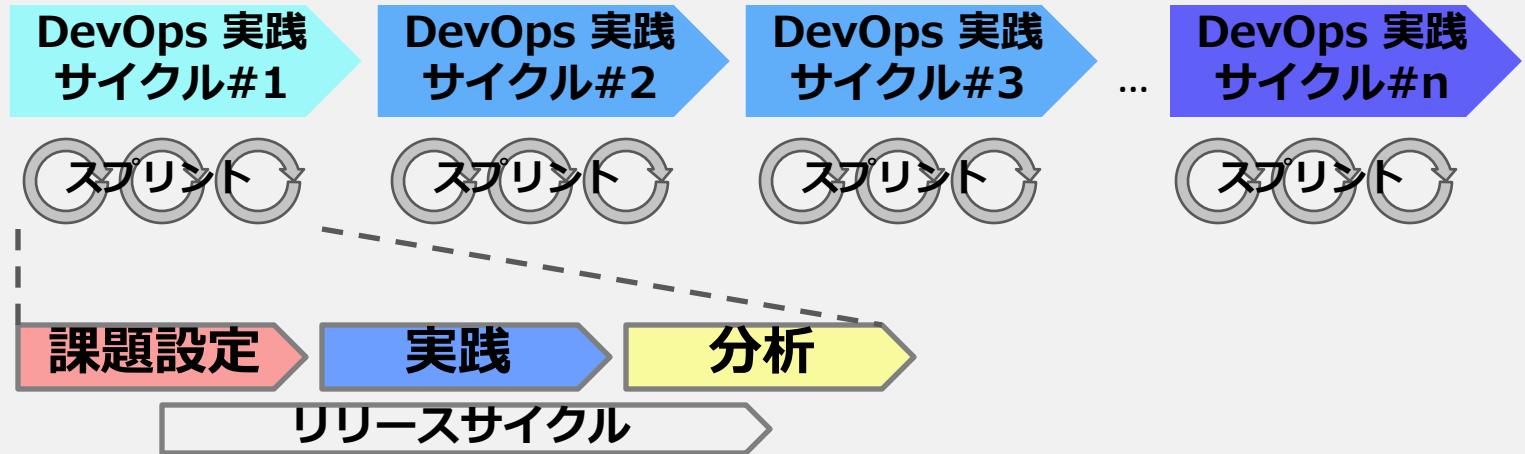
課題整理

- DevOps の成功要因の観点での文化・プロセス・技術、Biz・Dev・Ops の課題を整理する。
- 短期・中期・長期的なゴールを設定する。

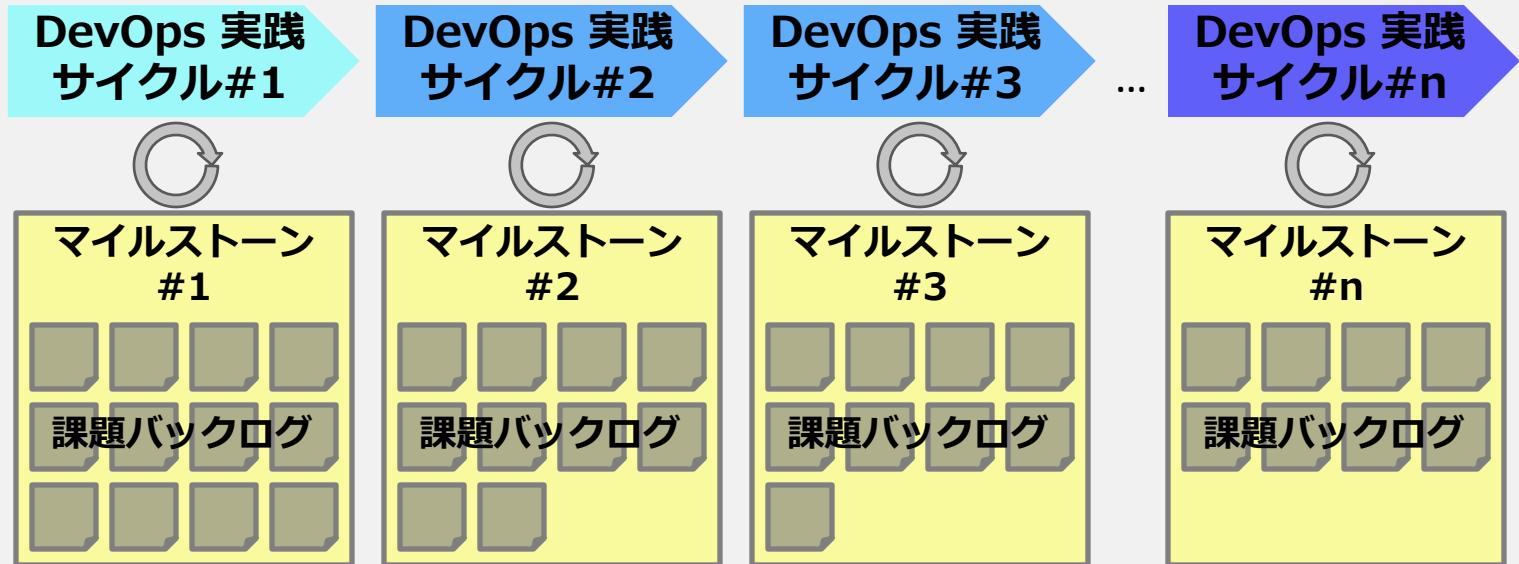
継続的改善

1. マイルストーンの設定
 - DevOps 実践の短期・中期・長期的なゴールからマイルストーンおよび KPI を設定する。
 - DevOps 実践における課題をマイルストーンと関連付けて整理する。
2. スクラム型実践
 - スクラム型 DevOps チームを構成する。
 - マイルストーン・課題に従いリリースプランニングを行う。
3. 継続的改善

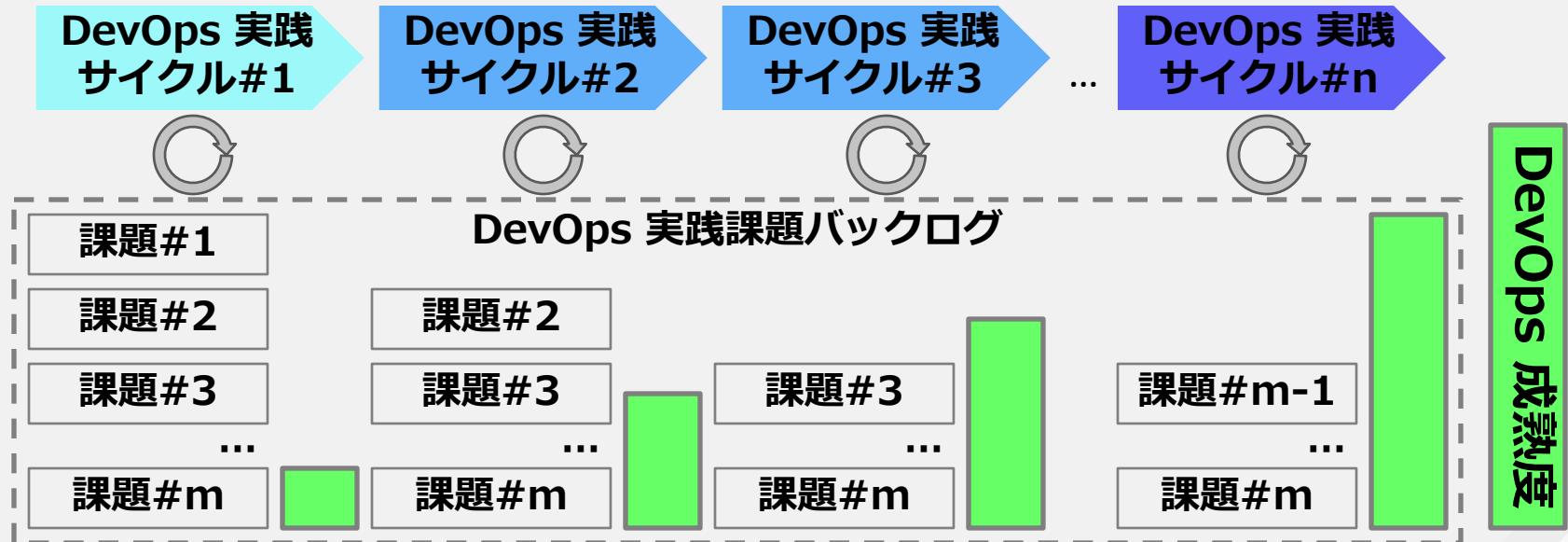
DevOps のスクラム型継続的改善



スクラム型継続的改善とマイルストーン



スクラム型継続的改善と成熟度



Agile/Lean の実践

DevOps と Agile

IT ビジネスの運営

Why

- 顧客の要求・環境の変化スピードの加速
- 顧客の要求・環境の多様化

How

- アイデアの早急な市場投入
- フィードバック（検証による学び）による継続的改善

仮説/検証・漸進/探索 的アプローチ

IT の運営

Why

- プロダクトが顧客に届ける価値を最優先
- 変更スピード（リードタイム短縮）
- 継続的改善（継続的変更）

How

- Agile/Lean 原則に則った IT 運営

プロセス・ドキュメント・契約交渉・計画の遵守より実際に顧客に提供する価値（顧客満足）とスピードを最優先、変更に前向き

アジャイルソフトウェア開発

歴史：アジャイルソフトウェア開発宣言

- 2001/02 に 17 名の著名なソフトウェアエンジニアが検討し文書化・公表 ([1], [2] 参照) したソフトウェア開発における原則（文化・価値観）

概要

- 特定の開発手法を指すのではなく「アジャイルソフトウェア開発宣言」をベースとした一群のソフトウェア開発手法を表す総称
※ Agile : 俊敏な、明敏な、頭の回転が早い、etc.
- 実際に動くソフトウェアを素早く提供しフィードバックと変更を実施（適応的、探索的）
- 短い反復単位（イテレーション）で 設計、実装、テスト、リリース を実施（反復的）
- イテレーションを繰り返すことによりソフトウェアの拡張と変更を実施（継続的、漸進的）

アジャイルソフトウェア開発手法

- スクラム
- XP（エクストリーム・プログラミング）、etc.

アジャイルソフトウェア開発宣言

私たちは、ソフトウェア開発の実践あるいは実践を手助けをする活動を通じて、よりよい開発方法を見つけだそうとしている。この活動を通して、私たちは以下の価値に至った。

プロセスやツールよりも**個人と対話を**、
包括的なドキュメントよりも**動くソフトウェアを**、
契約交渉よりも顧客との協調を、
計画に従うことよりも**変化への対応を**、

価値とする。すなわち、左記のことながらに価値があることを認めながらも、私たちは右記のことながらにより価値をおく。

Kent Beck	James Grenning	Robert C. Martin
Mike Beedle	Jim Highsmith	Steve Mellor
Arie van Bennekum	Andrew Hunt	Ken Schwaber
Alistair Cockburn	Ron Jeffries	Jeff Sutherland
Ward Cunningham	Jon Kern	Dave Thomas
Martin Fowler	Brian Marick	

© 2001, 上記の著者たち
この宣言は、この注意書きも含めた形で全文を含めることを条件に
自由にコピーしてよい。

アジャイルソフトウェア開発の特徴 - 1

動くソフトウェアを重視

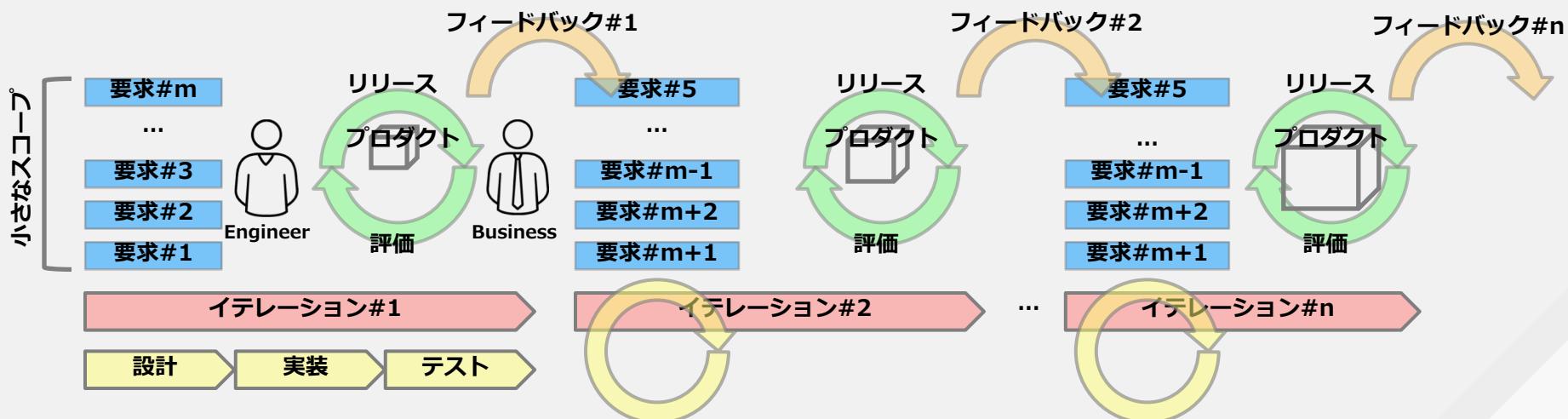
- 動くソフトウェアを顧客に素早く提供、動くソフトウェアを対象にした評価とフィードバック

イテレーション

- 定期で短い反復単位（イテレーション）で小さなスコープ（小さな機能セット）の設計、実装、テスト、リリースを行い、リリース可能なプロダクトを反復的・継続的に提供

変化への対応と改善

- イテレーション毎にプロダクトの改善・変更要求の取込を実施 ⇒ 漸進的・適応的・探索的な改善



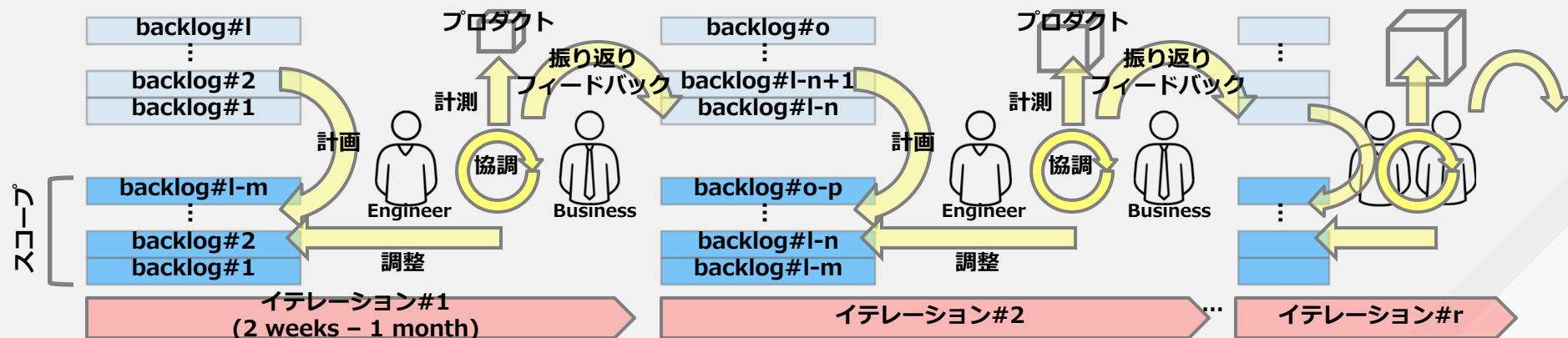
アジャイルソフトウェア開発の特徴 - 2

組織・文化

- ビジネス担当とエンジニアが日々同一の環境（近い距離）で（上下関係なく）協調し、チーム全員参加（Face to Face）によるプロダクト・課題・タスク管理を実施、変更の歓迎と変化によるプロダクトの価値向上

マネジメントモデル

- スコープ、ゴール：プロダクトに対する要求事項をバックログとして管理
- スケジュール：定期の固定されたイテレーション（2週間～1ヶ月）単位毎にバックログに対応
- リソース：固定されたリソース（通常10人未満のビジネス担当とエンジニアから構成されるチーム）
- 計画：イテレーション開始時に優先度に従いイテレーションで対応可能なバックログを見積
- 計測：動くソフトウェアを対象にチーム全員で実施
- 調整、フィードバック：イテレーション開始時の“振り返り”によるバックログの管理（追加/削除・優先度設定）



ソフトウェア開発における困難

不可視性

- ・ S/W の 実際の 外見・動作・得られる体験・効果 を**予め可視化することの困難さ**
- ・ S/W の 設計工程 における中間生成物(設計書)による**最終成果物をイメージすることの困難さ**

単一性

- ・ (SaaS、パッケージ製品 等を除く) S/W は**それぞれがただ一つの構造物**

変更性

- ・ (H/W 等に比べ) S/W は 環境・要求 の変化 に対して**容易に変更可能**であるべき
- ・ S/W をとりまく**環境(ビジネス、市場)の変化の早さ**
- ・ S/W に求められる**変化の圧力・頻度・対応速度** の高さ

複雑性

- ・ S/W の**構造と動作の複雑さ**
- ・ S/W の**開発工程の複雑さ**

不確実性(非予見的)

- ・ **目的(What)と手段(How)の不確実性**
- ・ S/W ライフサイクルにおける**変更要求を予測することの困難さ**
- ・ S/W ライフサイクルにおけるその**構造や統合の最適化を行うことの困難さ**
- ・ S/W の**開発計画(スコープ、スケジュール、リソース)を予測することの困難さ**

アジャイルのアプローチ

Why : ソフトウェア開発の困難

- ・ 不可視性
- ・ 単一性
- ・ 変更性
- ・ 複雑性
- ・ 不確実性 (非予見的)

How : 困難への対応

- ・ 予見性を前向きに否定
- ・ 実際に動くソフトウェアを素早く提供
- ・ 実際に動くソフトウェアで評価
- ・ 反復的・継続的・漸進的・適応的な開発手法
- ・ 短い反復単位で開発とリリース
- ・ 短い反復単位と小さなスコープと優先度
- ・ ビジネス担当者とエンジニアの協調
- ・ 反復的・継続的・漸進的・適応的な開発工程

ウォーターフォール・モデル（開発）

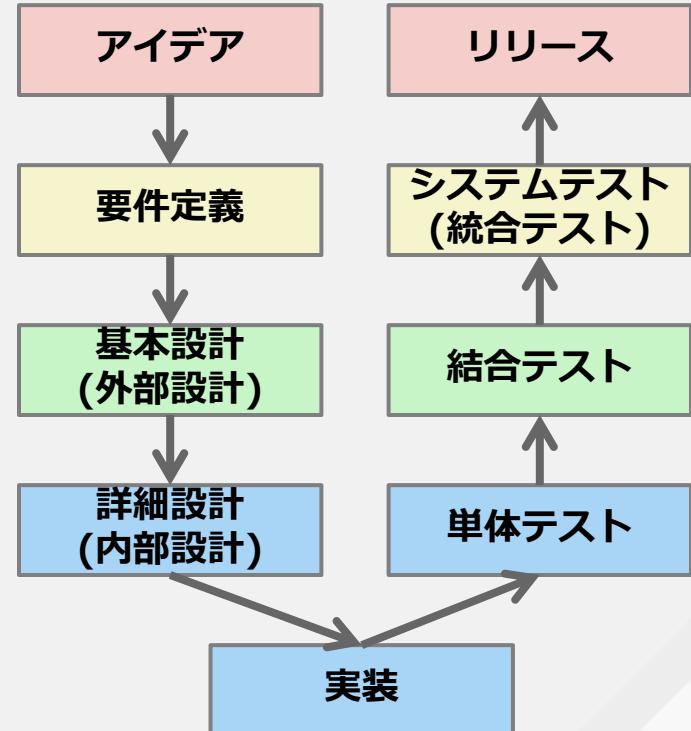
歴史

- 「ウォーターフォール・モデル」という用語は、文字通り「滝」を意味し、W.W.ロイスによって1970年に発表された論文「Managing the Development of Large Software Systems」の内容が元になったとされる。この論文において、「大規模ソフトウェア開発には、製品製造過程のようにいくつかの工程に分けたトップダウンアプローチが必要」と述べている。しかし論文には「ウォーターフォール・モデル」という記述は無く、また、前工程への後戻り（見直し）も提唱されており、元の論文の内容とは異なっている。
- 初めて「ウォーターフォール」という用語を用いたのはT.E.BellとT.A.Thayerによる1976年に発表された論文「Software Requirement」であり、B.W.Boehmが1981年に出版した本「Software Engineering Economics」においてウォーターフォールモデルのオリジナルはRoyceだと述べ、ウォーターフォール・モデルの起源がRoyceであるという誤解を広めた。

[ウォーターフォール・モデル (Jul. 30, 2016, 15:34 UTC). In Wikipedia: The Free Encyclopedia. Retrieved from <https://ja.wikipedia.org/wiki/ウォーターフォール・モデル>]

概要

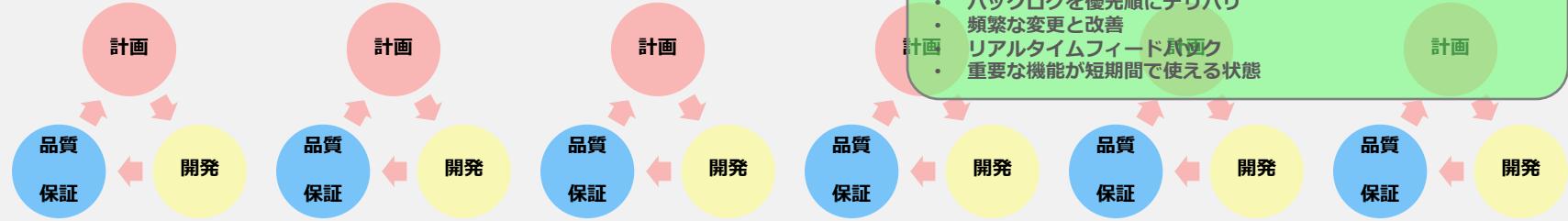
- 予見的・計画的アプローチ
 - 計画工程においてスコープを予見的に定義
 - スコープからスケジュール・リソースを計画
- 各工程を完了（仕様書・ドキュメントの品質検査等）しぬの工程に進む（※ある工程は全行程が完了していることが前提で一回のみの実施）
- アイデア～リリースが長期（実際に動くS/Wは後工程で提供）
- 設計工程以降の“要求変更”は工程の後戻りとなる



Agile と Waterfall の違い

アジャイル

- 反復的・継続的・漸進的・適応的なアプローチ



ウォーターフォール

- リソース、スコープ、スケジュールを固定するため、十分に事前計画を行う 計画的・予見的 アプローチ



ウォーターフォール・モデルの課題

ソフトウェア開発の困難への対応

- 予見性の前提に基づく計画性という非現実的なアプローチ
 - 見積りが不正確、進捗が不透明、期限遅れ、テストの短縮
 - チームの肥大化と生産性の低下

要求仕様抽出の効率性と“ムダ”な機能実装（機能の肥大化、投資の損失）

- ドキュメントからプロダクトを予想することは現実的に困難
- 変更が現実的に不可能 -> 要求仕様の詰め込み
- 実質的なフィードバックは開発の後工程

開発期間の長期化

- アイデアをリリースする段階でビジネス環境が変化
- 顧客価値（動く S/W）の長期化、開発中の仕掛品（在庫）

変化への対応性・俊敏性

- 設計フェーズ以降の変更は現実的に不可能

ビジネス担当者とエンジニアの分断

- ビジネス担当とエンジニアのゴールの不一致

統合と不具合の検知

- 後工程における大規模な統合、統合で具現化する不具合
- 不具合の作り込み～検知の時間の長期化

全ての IT 運営においてウォーターフォール・モデルが抱える問題ではなく web 上の IT ビジネスの開発プロセスとしてのウォーターフォール・モデルの課題であることに注意！！

ウォーターフォール・モデルの課題 - 統合と不具合の検知

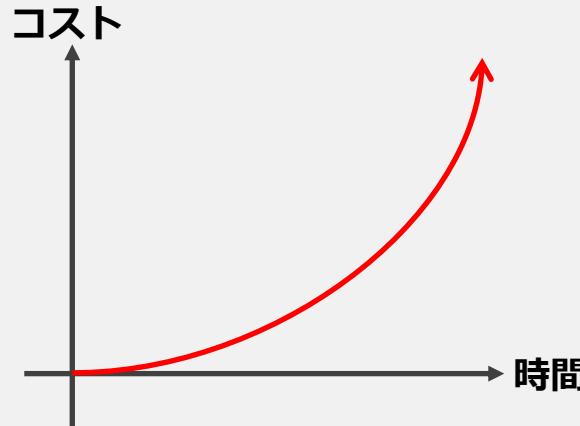
後工程での大規模な統合

- ・後工程でのシステム統合 (コンパイル、ビルド) と動作検証による**想定外の事象・不具合の頻出**

不具合の検知と修正

- ・大規模な統合に対する**不具合の検出・原因の特性・修正の効果性・効率性の低下**
- ・不具合の作り込み～検知の長期化による**不具合の原因の特定・修正の効果性・効率性の低下**
- ・**不具合の修正における手戻り**

不具合の修正に関する時間とコストの関係



アジャイルの開発手法

Scrum (スクラム)

- ・プロセスに関する手法
- ・アジャイル開発手法として最も普及

XP (eXtrem Programming)

- ・開発の仕方に関する手法 (プラクティス)
- ・概要
 - ・ケント・ベックらによって定式化され、提唱されているソフトウェア開発手法、1999年に書籍『XPエクストリーム・プログラミング入門—ソフトウェア開発の究極の手法』によって発表。XPは、軽量開発手法あるいはアジャイルソフトウェア開発手法と呼ばれる、同種の開発手法のなかで代表的
 - ・継続的インテグレーション (CI)、テスト駆動開発、ペアプログラミング、リファクタリング、YAGNI 原則が普及
- ・アジャイルを実践するにあたり CI (Continuous Integration) は必須