BtoB SaaSにおける

大規模データとの戦い方

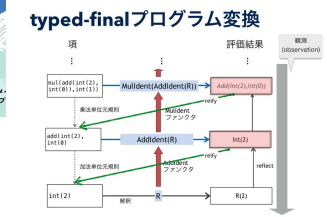
Kenichi Suzuki
Loglass Inc.

株式会社ログラス
鈴木健一 / Kenichi SUZUKI

X: @_knih

新卒でNTTデータに入社。アーキテクチャ設計や統括業務に従事した後、より堅牢な開発を求めてプログラム言語理論（型システム、関数型等）の研究に踏み込む。
その後、よりセキュアな世界を目指して、Visional（BizReach）にてサイバーセキュリティ事業の立ち上げや開発をリード、ContractSにて開発部長、技術戦略室長、VP of Development を経て、2023年に株式会社ログラスにジョイン。

Publications



Finally, safely-extensible and efficient language-integrated query
PEPM '16: Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program ManipulationJanuary 2016 Pages 37–48https://doi.org/10.1145/2847538.2847542

Multi-Stage Programming、そして脆弱性レジリエンス× Clean Architecture
https://engineering.visional.inc/blog/212/scalamatsuri2020-interview-msp/

脆弱性レジリエンスを高めるためのClean Architecture
https://yamory.io/blog/architecting-for-resilience/

エンタープライズアジャイル開発における品質管理とセキュリティ対策をyamoryで考える
https://yamory.connpass.com/event/198588/

Scala3でコードは爆速になるマルチステージプログラミングの考え方
https://logmi.jp/tech/articles/324146

Dotty ではじめるマルチステージプログラミング入門
https://www.youtube.com/watch?v=gpQHgcIIzFY

From Tagless-Final to Typed-Final: Program Transformations in the Final Style
https://speakerdeck.com/dcubeio/from-tagless-final-to-typed-final-program-transformations-in-the-final-style

ContractS Tech Book Vol.1
第8章　型の力で高品質にドメインをモデリングする―関数型DDDに向けて―.
https://nextpublishing.jp/book/14565.html

良い景気を作ろう。

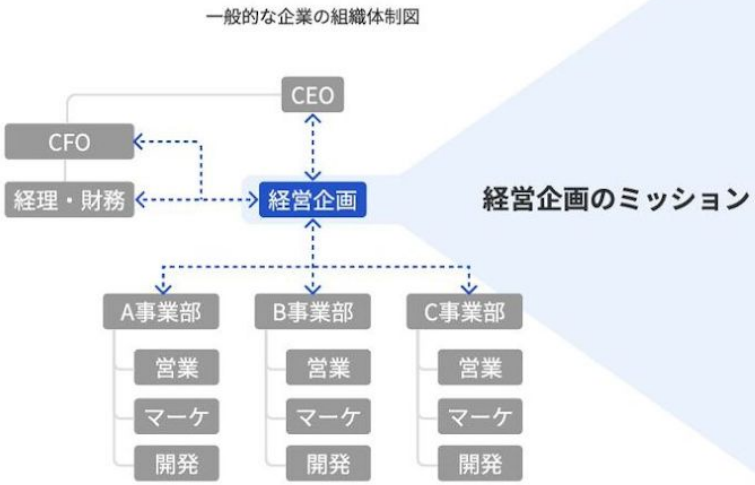
管理会計100年の歴史に終止符を打ち、新しいデータ経営の在り方を生み出す。
そのために、ログラスは立ち上がりました。「失われた30年」日本は悔しい思いをしてきました。更に今後は、労働人口が減少し、経済競争力を失っていくと言われています。でも、本当にそうでしょうか。テクノロジーの力で、人間を超える速さ、正確さでデータを導き出し、経営の新しい答えに辿り着く。そんな未来が訪れたとしたら、強い経済はまた必ず作れる、世界と戦えると私たちは確信しています。
パートナーたちと一つ一つの壁を越え、良い景気を作る。私たちはログラスです。

Loglass

Loglass

次世代型 経営管理クラウド

経営企画は「企業価値の向上」をミッションに、企業経営にまつわるあらゆる業務を担っている



経営企画のミッション

企業価値の向上

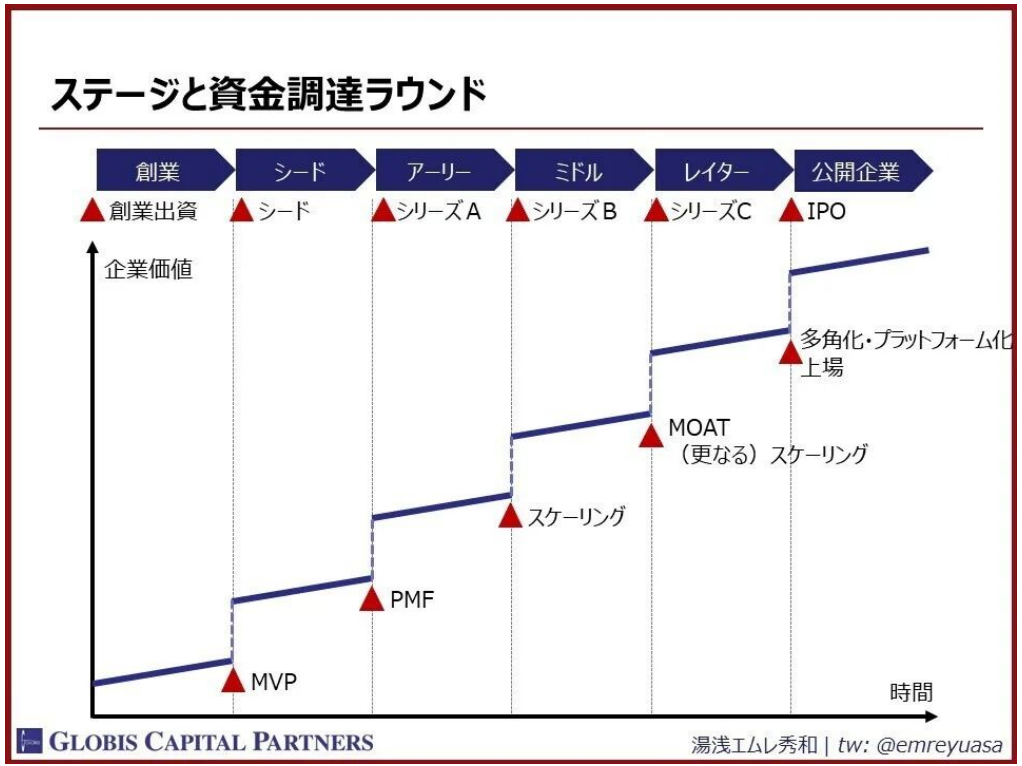
攻めの業務	守りの業務
• 中長期の戦略策定 • 新規事業開発 • M&A • DX推進 • 各種調査・分析 ⋮	• 経営管理(財務・管理会計) • IR・開示対応 • 内部統制 • 社内会議体の運営 ⋮

前提

- 本発表ではBtoB SaaSプロダクトを前提とします
  - とはいえ、アーキテクチャについて言及するためBtoCでも通用するところは多い

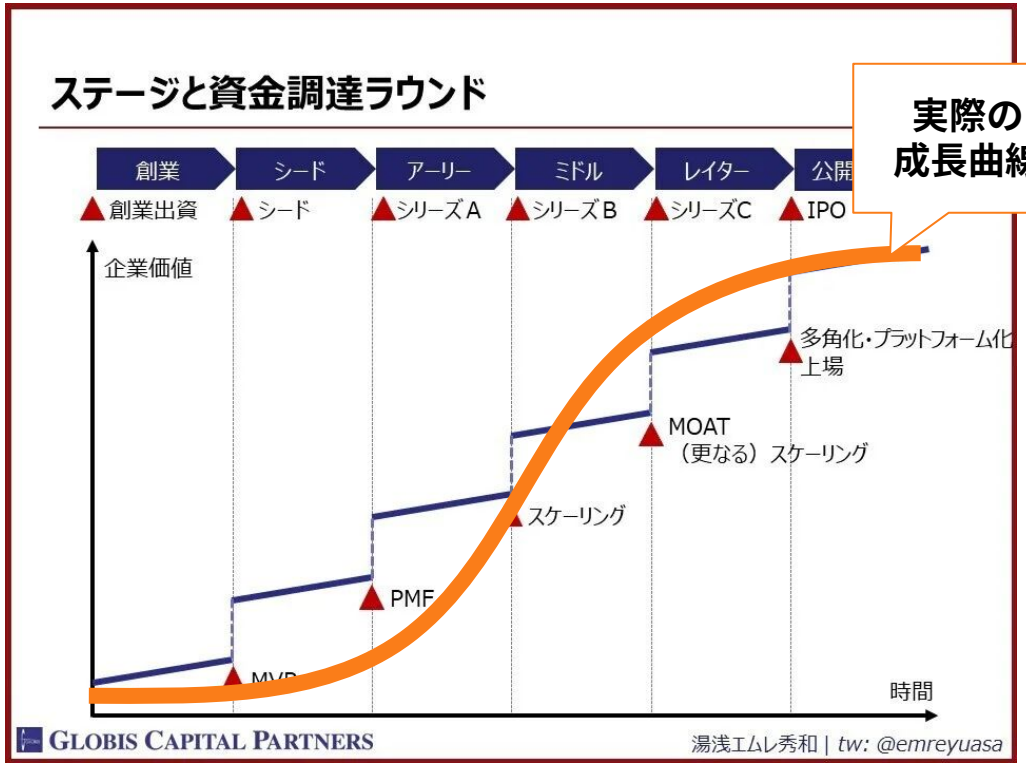
データ増大による問題

プロダクトスケール



湯浅エムレ秀和（2021）. シリーズA.
https://note.com/emreyuasa/n/n03aca5b7a981

プロダクトスケール



リクエスト数やデータ量が
増えていく

事業成長にあわせて、
プロダクトをスケール
していく必要がある

湯浅エムレ秀和（2021）. シリーズA.
https://note.com/emreyuasa/n/n03aca5b7a981

データの増大

データが増える要因

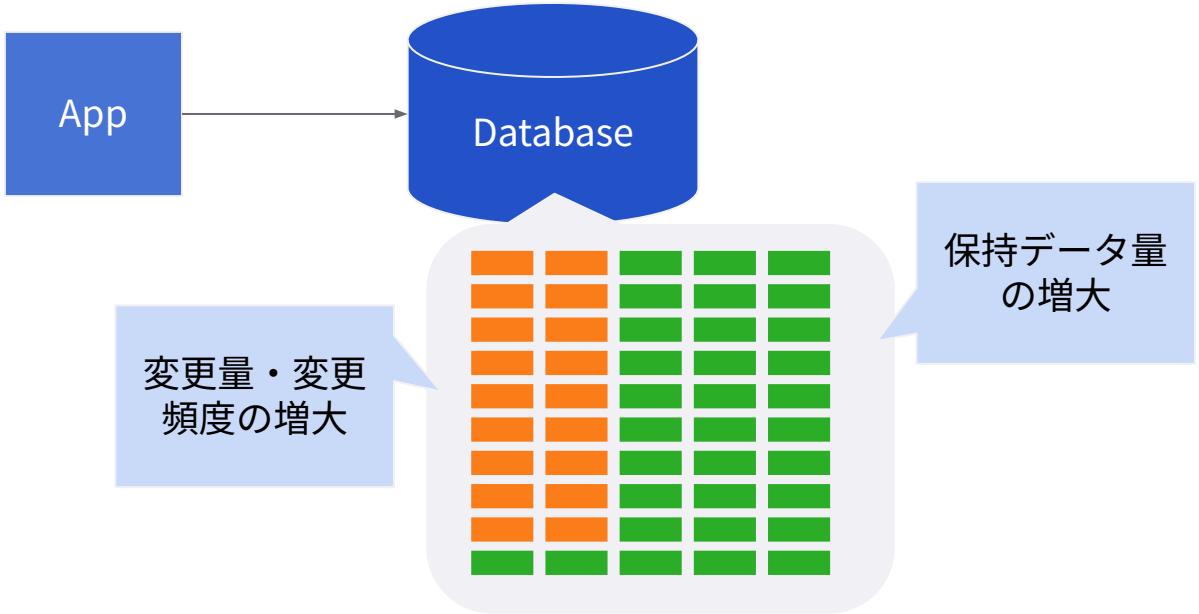
- ユーザー数の増加
- 機能の拡充
- 利用頻度の増加
- 新規プロダクトの追加
- ユーザー規模の拡大
- 顧客分析の強化

・・・等々

データスケールを考慮していない
アーキテクチャだとどうなるか？

データの増大

データボリュームだけでなく、トランザクションボリュームも増加



パフォーマンス問題

パフォーマンスが劣化すると、様々な問題が発生

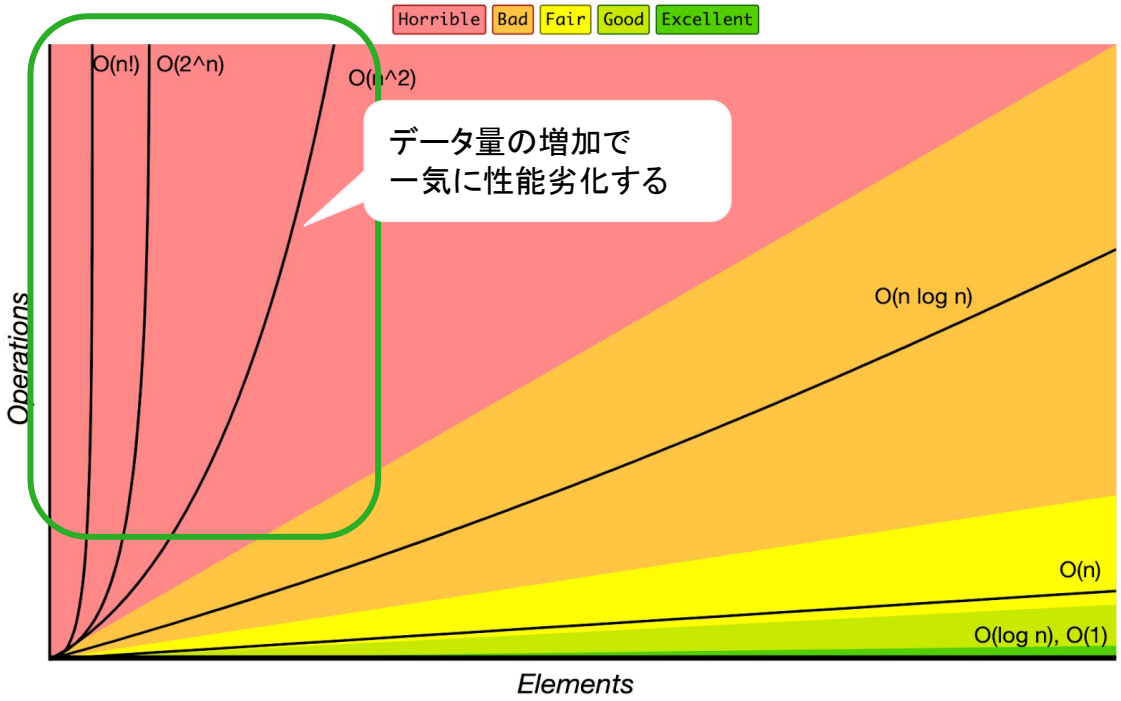
- **DB・CPU負荷の増大**
  - インフラコスト増大
  - リクエストが詰まりやすい状況を誘発
- **タイムアウトによるユーザーリクエストの失敗**
  - 機会損失
- マテリアライズド・ビューの更新遅延
  - **タイムリネス（データ鮮度）の低下**
- パフォーマンスチューニングの工数が増えることで、**開発が遅くなる**

参考：マテリアライズド・ビューの功罪

- マテリアライズド・ビューは、事前に計算されたデータセット
- 増分更新ではない場合、**全件アップデートするためデータ量の増加につれ、更新負荷があがっていく**
  - PostgreSQLでは基本的に非増分更新
    （増分更新をサポートするアドオンはある）
- さらに、集約関数を使うようなSQLは、メモリに乗らなくなった瞬間に急激に遅くなる

データ増加に伴い、
データの鮮度が落ちやすくなる

パフォーマンス問題を誘発しやすい作りになっていないか



問題が起きてから対処しては遅い

パフォーマンスの
抜本的な改善には時間がかかる

パフォーマンスの抜本的な改善

- アーキテクチャの変更
- データモデルの修正、インデックスの再設計
- アプリケーション仕様の変更

パフォーマンスの検討に早過ぎるということはない

> “最大の理由は、どのような変更を加えた時にパフォーマンスが急降下したかがわかることです。これならパフォーマンス問題に直面した時に、アーキテクチャー全体を相手にせず、最近に加えた変更に焦点を絞り込んでいけるのです。

Rebecca Parsons (2009). 97 Things Every Software Architect Should Know - Chapter 13. O'Reilly Media.

スケールを成功させるためには洗練されたアーキテクチャが必要

> “ソフトウェア・システムの規模が大きく、
> 複雑であればあるほど、成功のためには
> **よく練られたアーキテクチャ**が必要になる
>
> The greater the size and complexity of a software system,
> the more you will need a **well thought-out architecture**
> in order to succeed.

Joseph Ingeno (2018). Software Architect's Handbook: Become a successful software architect by implementing effective architecture concepts. Packt Publishing.

パフォーマンスを検討するといっても、
デリバリが遅くなってはいけないのでは？

開発速度をキープしつつ、
スケールしていきたい

そこで

データスケールを考慮した継続的アーキテクチャ

Minimum Viable Architecture（MVA）

- 実用に足る最小限のアーキテクチャ
- MVP（Minimum Viable Product）
- 最低限の品質要件を満たしている
- プロトタイプアーキテクチャ
  - 理想的には特別な技術を必要としない（ペーパープロト等）
- Just Enoughアーキテクチャ
  - 高速な学習と改善、スケーリングしない

参考：Randy Shoup. Minimum Viable Architecture. YOW! Brisbane 2022.
https://yowcon.com/brisbane-2022/sessions/2358/minimal-viable-architecture

事業成長をキープし続けるためには

事業が急速に成長するよりも**前に**、
次のアーキテクチャに進化させたい

継続的アーキテクチャとその原則

以下の6原則に従ったアーキテクチャアプローチ

1. プロジェクトではなく、プロダクトを設計
2. 機能要件ではなく、品質属性にフォーカス
3. 必要になるまで、設計の決定を遅らせる
4. 変化のために設計する — "小さな力 "を活用する
5. ビルド、テスト、デプロイのための設計
6. システムの設計後に組織をモデル化

顧客に集中する

品質属性の要件がアーキテクチャを推進

使われない無駄を避ける

小規模で疎結合に

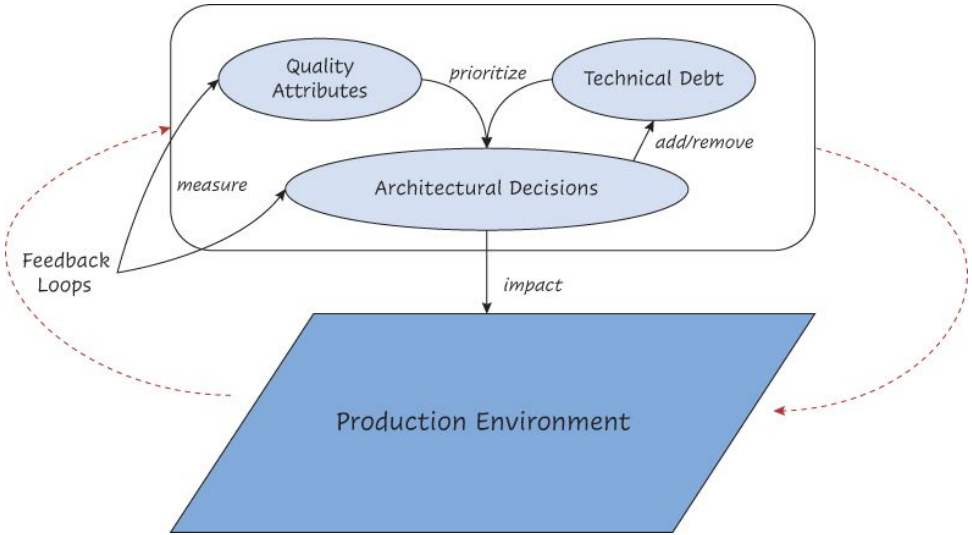
継続的デリバリーも考慮

チームの編成方法がアーキテクチャと設計を駆動する

参考：Murat Erder (2015). Continuous Architecture: Sustainable Architecture in an Agile and Cloud-Centric World. Morgan Kaufmann.

継続的アーキテクチャの主要な活動

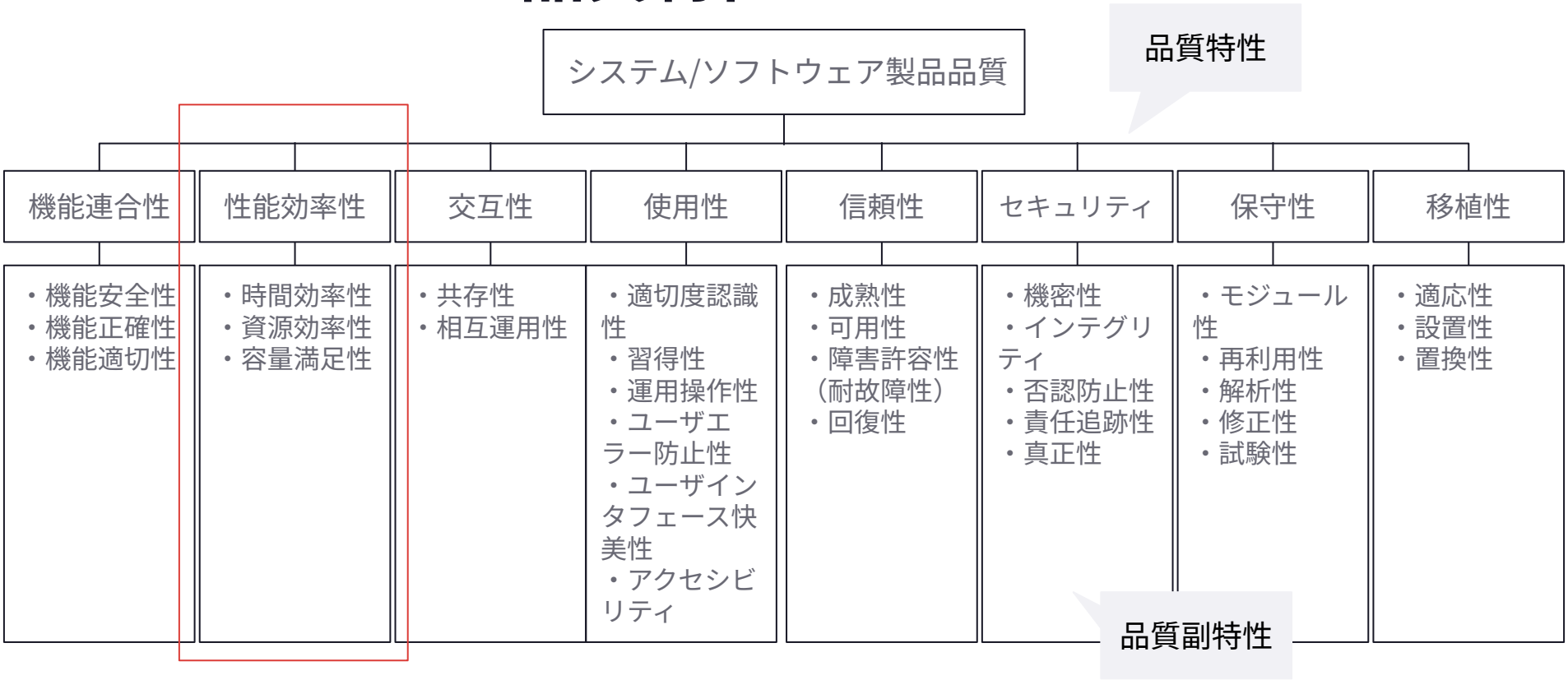
- 品質属性を計測し、フィードバックループをまわす
- 技術的負債を管理する



Murat Erder, et al. Continuous Architecture in Practice. Addison-Wesley Professional. 2015.

品質属性と見積もり

ソフトウェアの品質特性



システム・ソフトウェア製品の品質モデル（JIS X 25010:2013（IEC25010：2011）に基づく）

ソフトウェアの品質属性

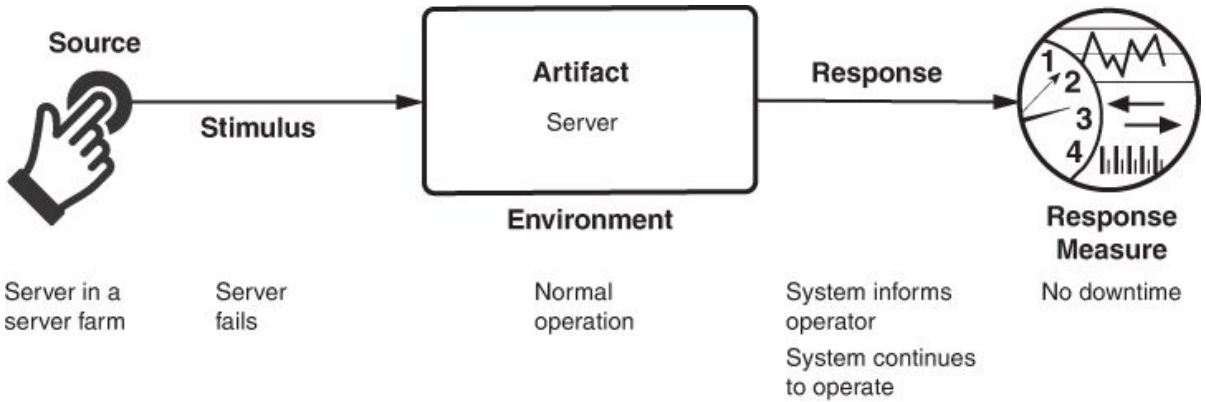
- 品質属性＝品質特性を細分化した末端
  - 性能→時間効率→**レスポンスタイム、スループット**等
- 測定可能、テスト可能
- 品質属性要件はアーキテクチャと相互に影響しあう
- 品質属性間にトレードオフがある
  - 例：セキュリティを高めるとユーザビリティが低下する

品質属性要件がアーキテクチャの肝になる

品質特性シナリオ

- 品質要求を言語化するためにシナリオを検討する
- シナリオ
  - 要求を具体的なストーリーにしたもの
- シナリオの種類
  - **利用シナリオ**、**成長シナリオ**、調査シナリオ（問題予測）

ユーザー体験・機能を軸にするため、フィーチャー開発チームが主体となるか、連携できる体制で臨むのがよい



Len Bass, et al. Software Architecture in Practice, 4th Edition. Addison-Wesley Professional. 2021.

品質特性シナリオの例

- 品質特性
  - パフォーマンス
- シナリオ例
  - 登録データが10億件以内である場合、ユーザーには3秒以内で一覧が返される

想定されるシナリオをリスト化し、
優先順位をつけるとよい

想定リクエスト数、データ量

基礎データ（要件）

- 当該ユースケースに対するリクエスト数
- データ量

SQL計算モデル

- 計算量見積もり
- インデックス設計
  - 結合、フィルタ、ソート
- MENTORの法則

参考：結合の計算コスト

掛け算が多いと、データ量の増加に比例して急激にコストが跳ね上がる

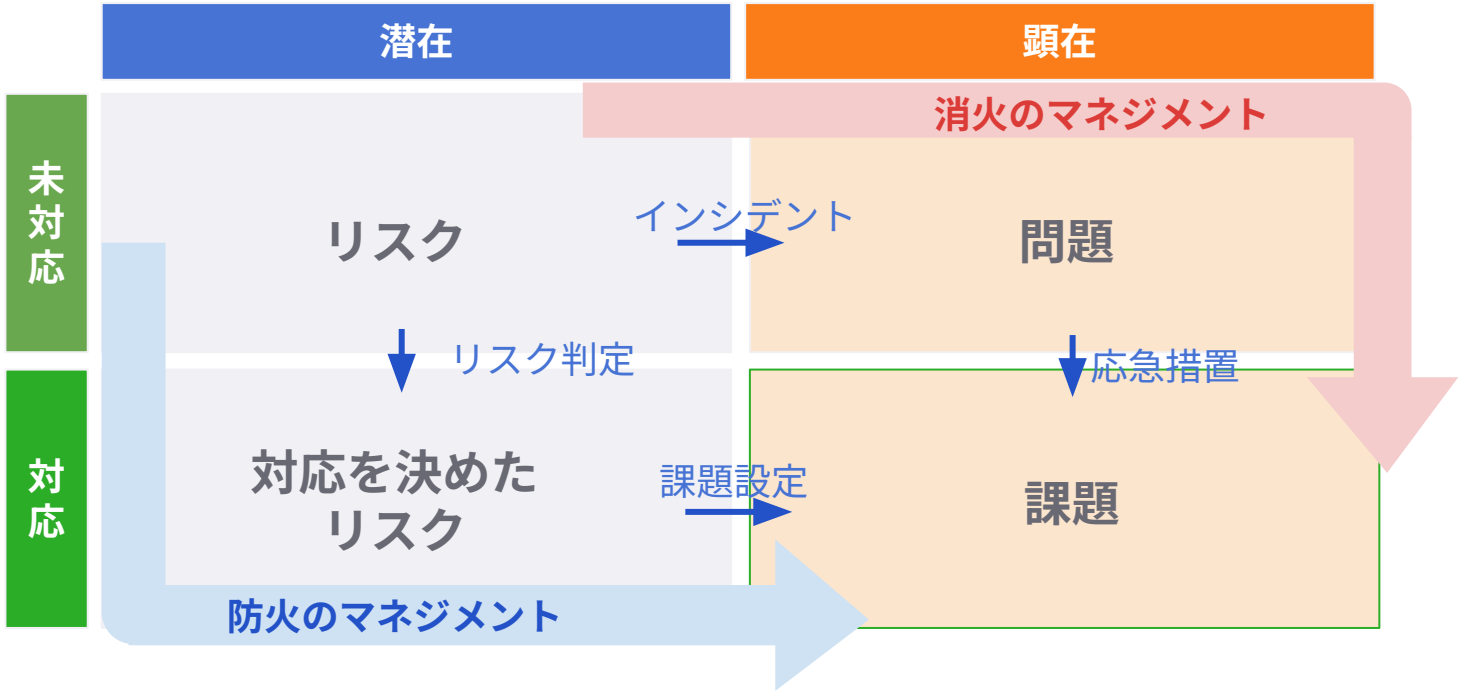
ネステッドループ結合	$O(N \times M)$
ソートマージ結合	$O(N \log N) + O(M \log M) + O(N + M)$
ハッシュ結合	$O(N) + O(M)$

どこまでアーキテクチャを設計・実装すればよい？

問題ではなくリスクに着目する

リスクと問題

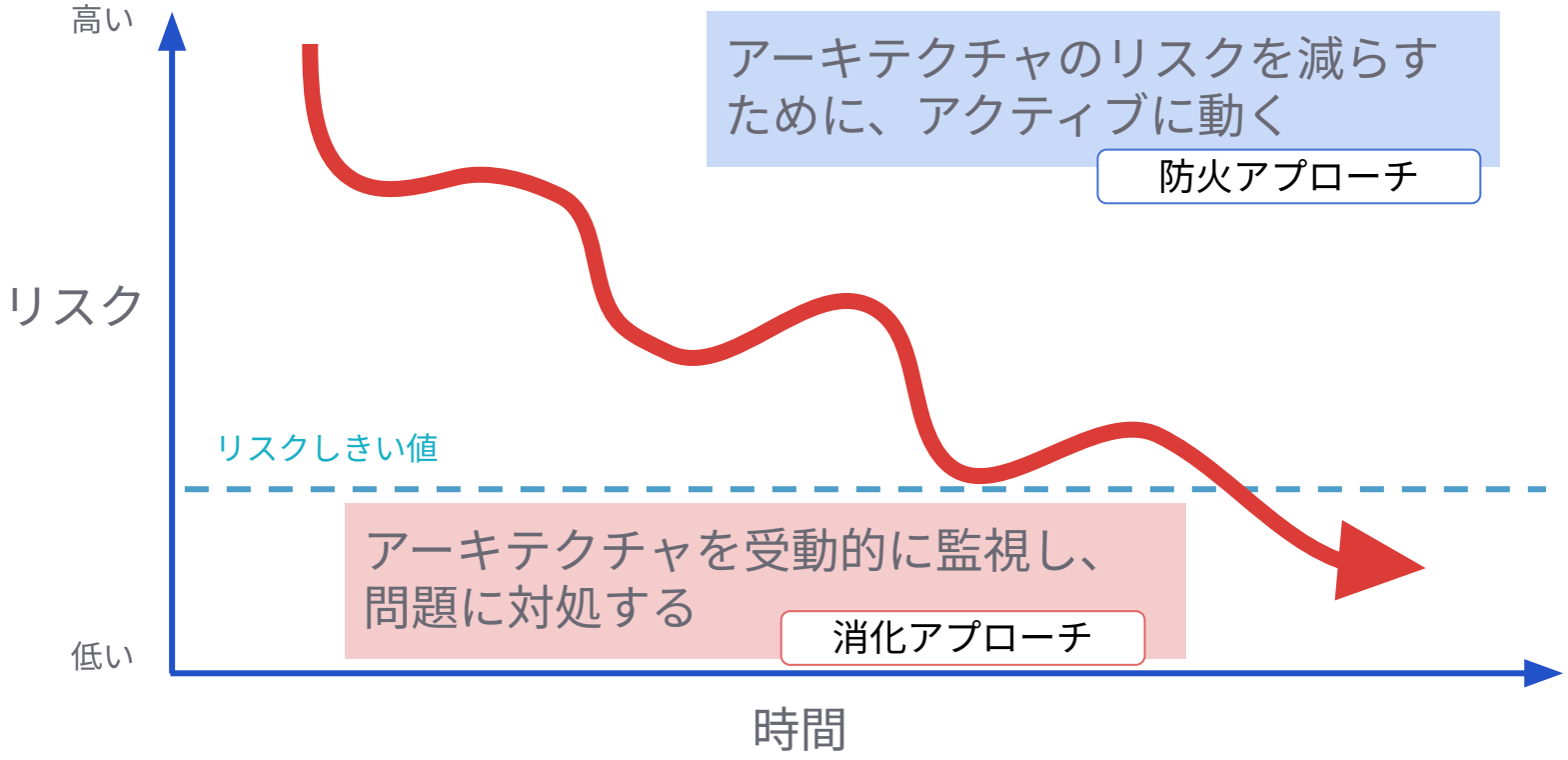
状況に応じて消化・防火それぞれのアプローチをとる



起こり得る可能性で見極める

- リスクが顕在化したときの脅威の大きさと、可能性の高いシナリオに着目
- リスクが複数ある場合は、アーキテクチャバックログを作成すると良い

リスクが低減したらパッシブモードへ



Michael Keeling (2017). Design It!: From Programmer to Software Architect. Pragmatic Bookshelf.

大規模データを考慮した
アーキテクチャ観点

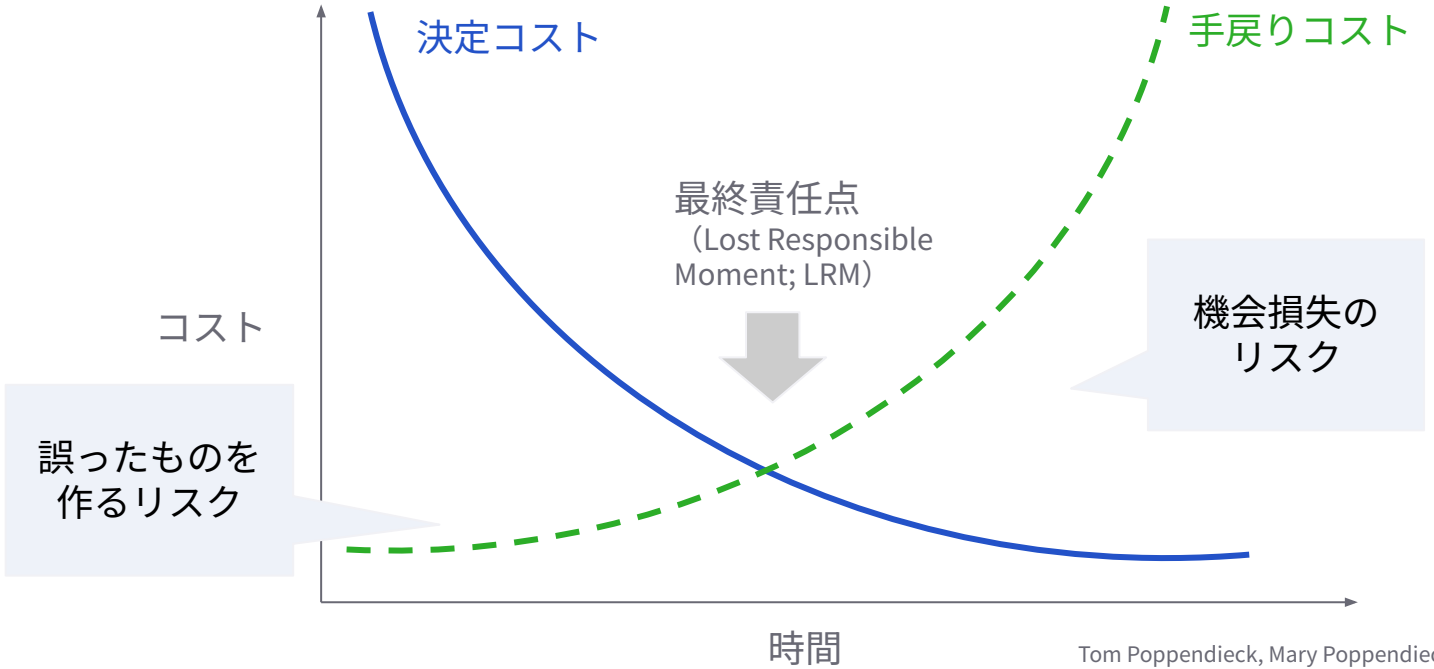
検討しておくべきリスク

決定を遅らせる≠検討しない

必要になった段階で決める
≠
問題が起こってから決める

決定が遅くなると・・・

意思決定のジレンマ、手戻りが発生しないようにしたい

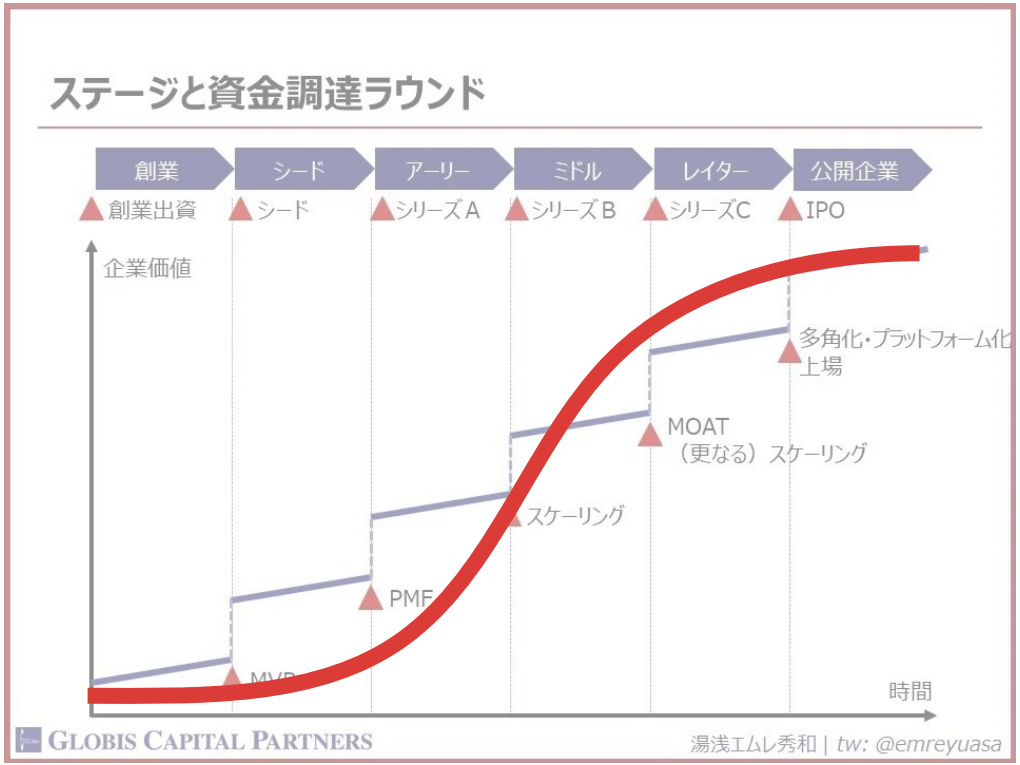


Tom Poppendieck, Mary Poppendieck. Lean Software Development: An Agile Toolkit. Addison-Wesley Professional. 2003.
を参考に作成

早期に考慮しておいたほうが良い観点

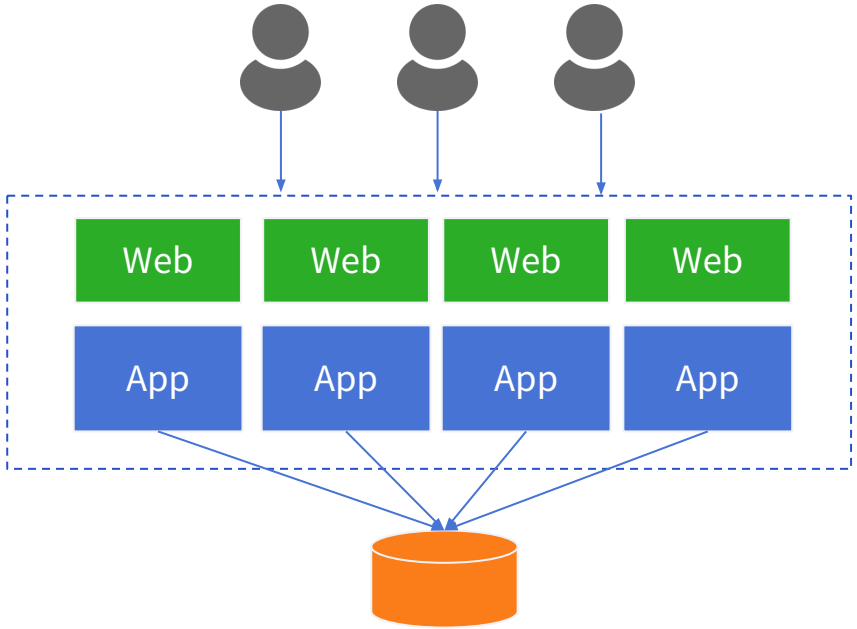
アーキテクチャを作り込むのは
適切なタイミングで

（再掲）プロダクトスケール



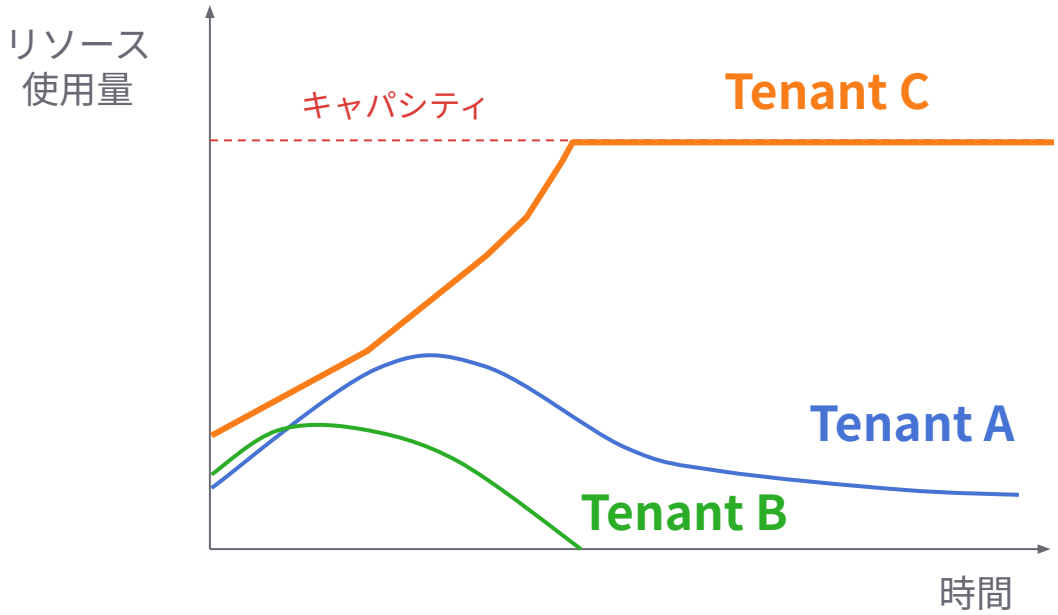
湯浅エムレ秀和（2021）. シリーズA.
https://note.com/emreyuasa/n/n03aca5b7a981

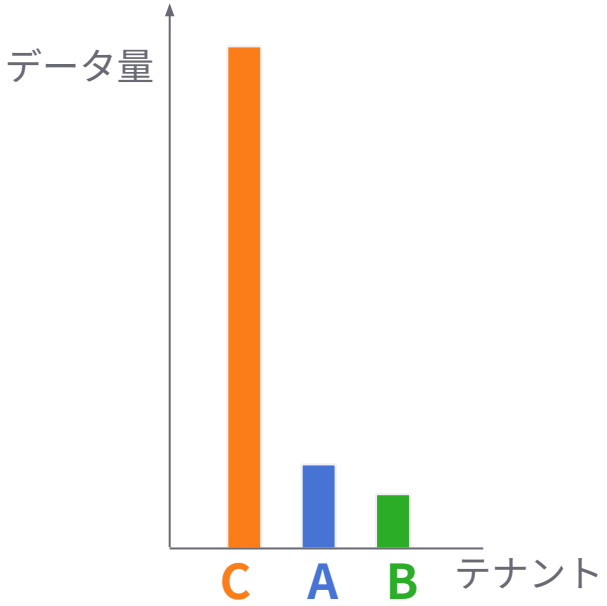
マルチテナント



マルチテナント戦略

うるさい隣人（ノイジーネイバー）アンチパターン

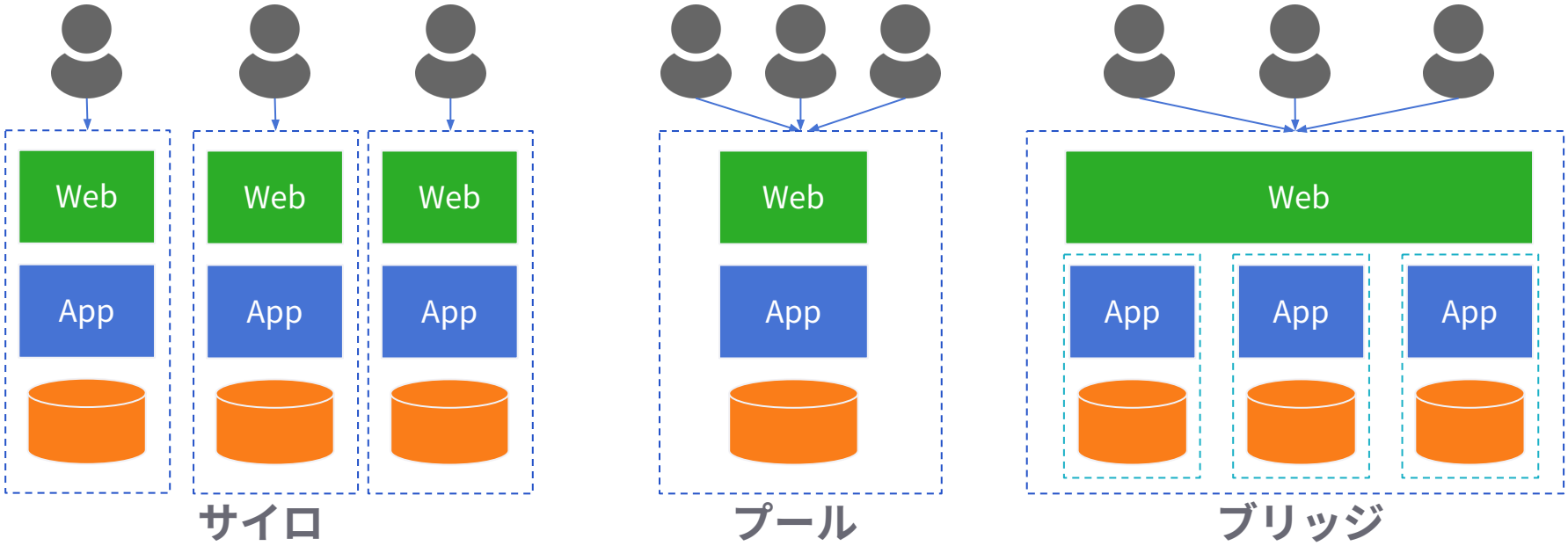




テナントの分離

テナント別サブドメイ化もあわせて検討しておくとよい
https://xxx.example.com

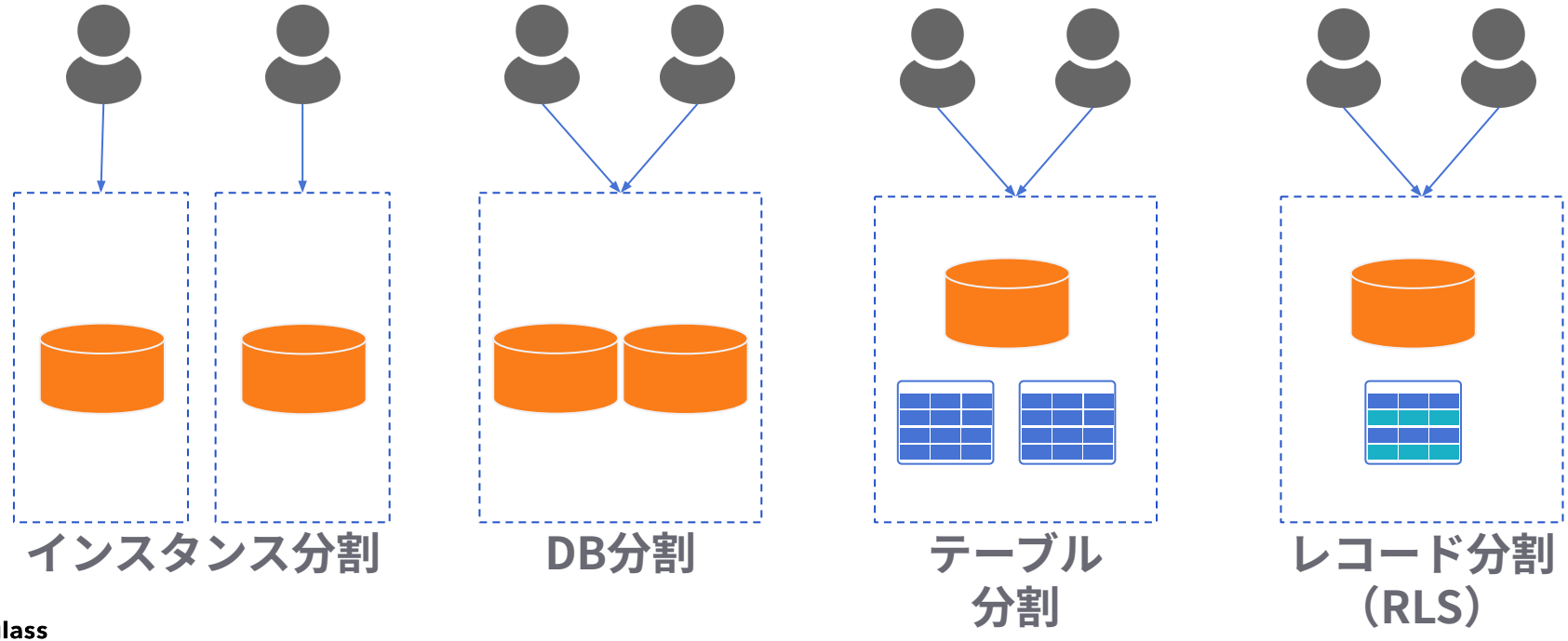
プールパターンはノイジーネイバーのリスクがある



データの分離

インスタンス・DB分割は、後述の論理分割方式によって決定を遅らせることが可能

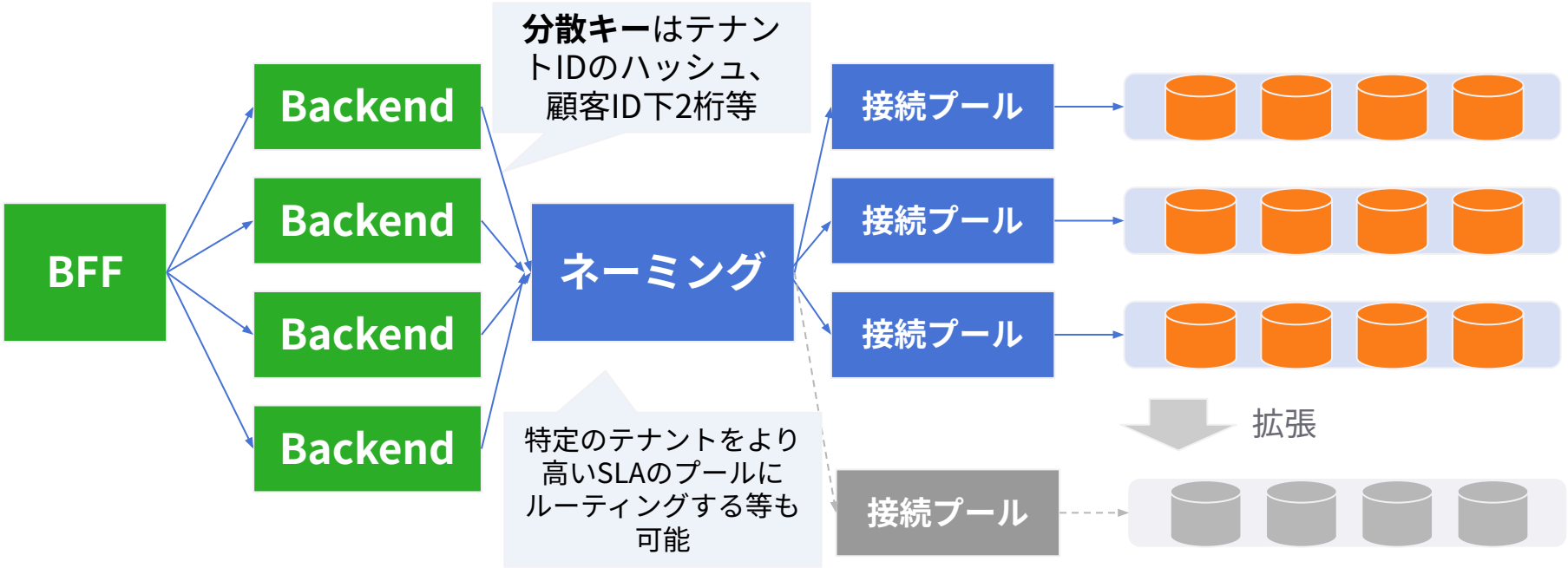
アクセス制御の開発コスト、スキーマバージョン管理の運用負荷を考慮しておく



アプリケーションによる論理分割

大規模であれば**コンテンツベースルーターパターン**とあわせて検討したほうがよい

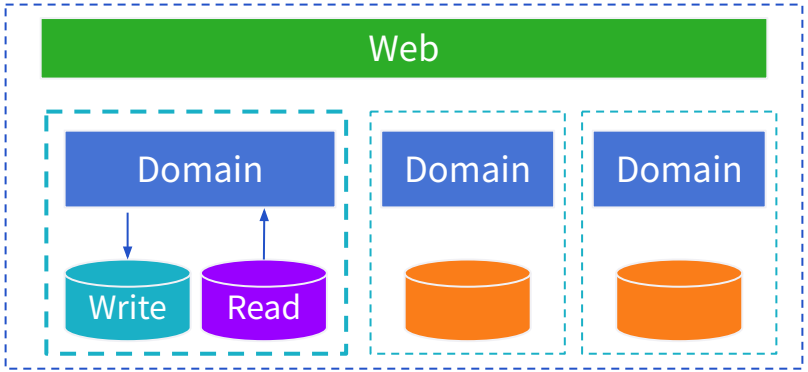
論理分散キーによるアプリケーション制御による分散



イベントソーシング・CQRS

Data Vaultパターンは運用コストは高いが、後付けでInsert-onlyな分析データベースを構築可能

- 複数の境界付きコンテキストがあり、将来的にイベントソーシングにしていく可能性があるなら、ドメインデータプロダクトとして分離しておくと良い
  - イベントソーシングへの変更はデータモデルに大きく影響するため、依存を分離しておいた方が良い
- 大規模なシステムに、可能な限りリアルタイム性を持たせたいなら検討する

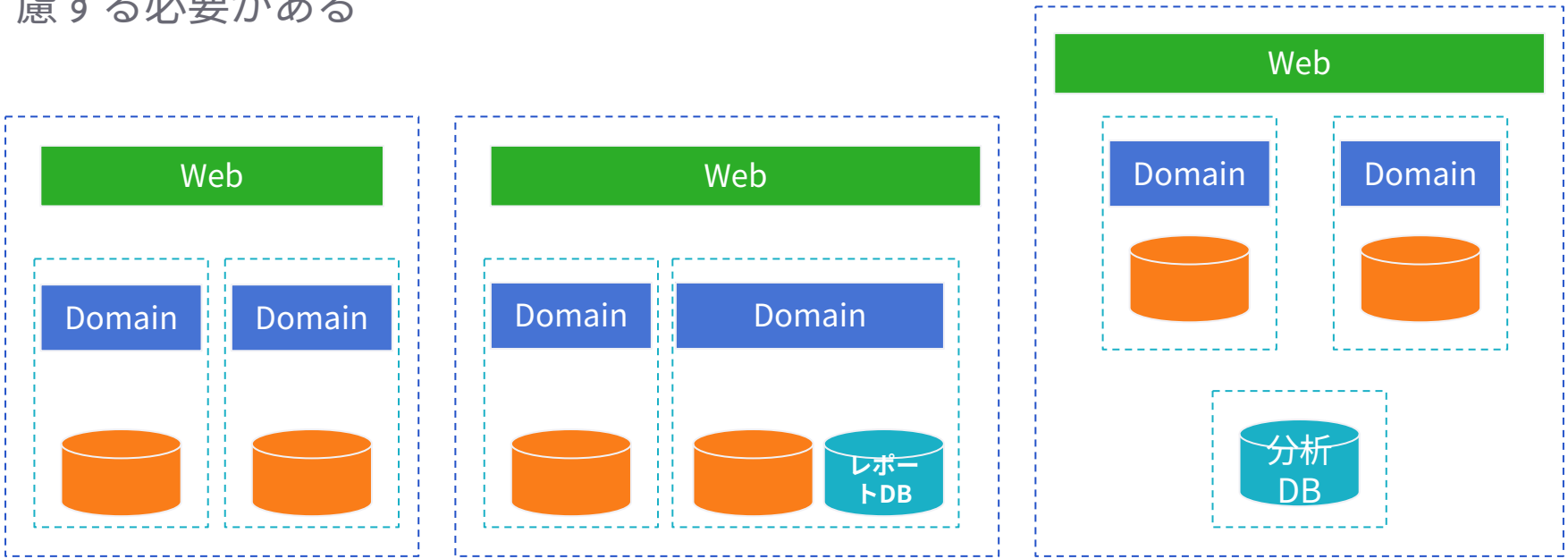


データ分析

- ダッシュボードやレポート機能等、機能を横断したデータ分析をどうするか
  - データから重要な意思決定のための知識を抽出するニーズ
- パフォーマンス要件次第
  - オペレーショナルデータベースをそのまま使う
  - データプロダクト内部に分析データベースを持つ
  - 分析データベースを共通化する
- ポリグロット永続性もあわせて検討
  - サービスごとに最適なデータベース

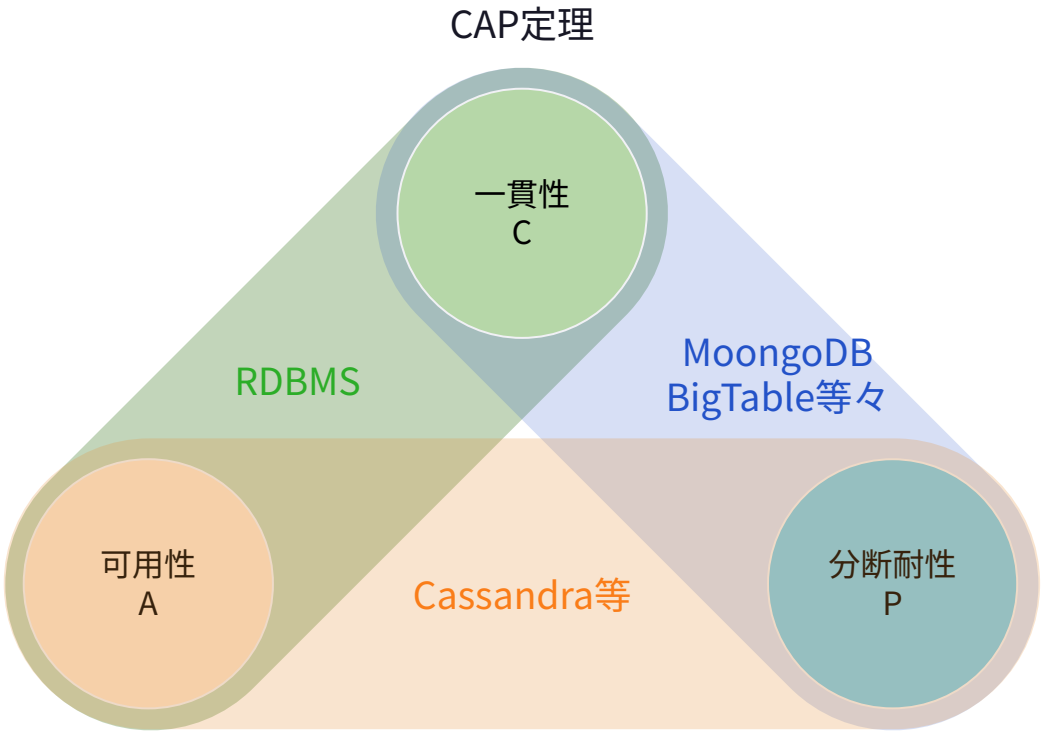
分析データベース

データベースの持ち方によって、ETLやレプリケーションを通した全体の整合を考慮する必要がある



ケース1　ケース2　ケース3

CAP定理

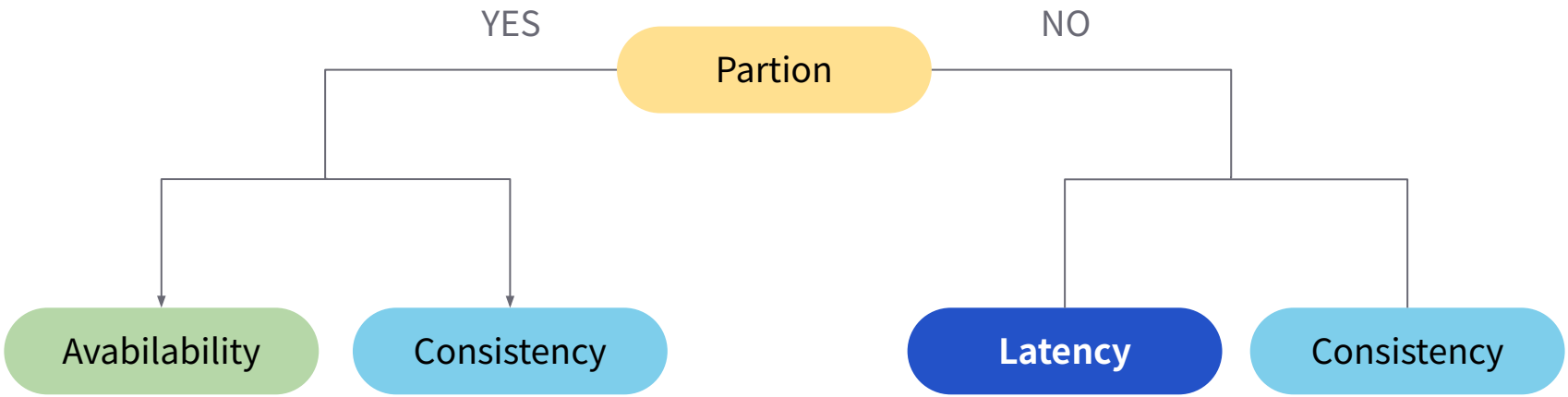

CAP定理
一貫性
C
RDBMS
MoongoDB
BigTable等々
可用性
A
Cassandra等
分断耐性
P

CAP定理からPACELC定理へ

- PACELC定理
  - メリーランド大学のDaniel Abadi先生が提唱
  - CAP定理に対して**遅延（Latency）**の軸を含めたもの

参考：Daniel Abadi (2010). Problems with CAP, and Yahoo’s little known NoSQL system.
https://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html

PACELC定理

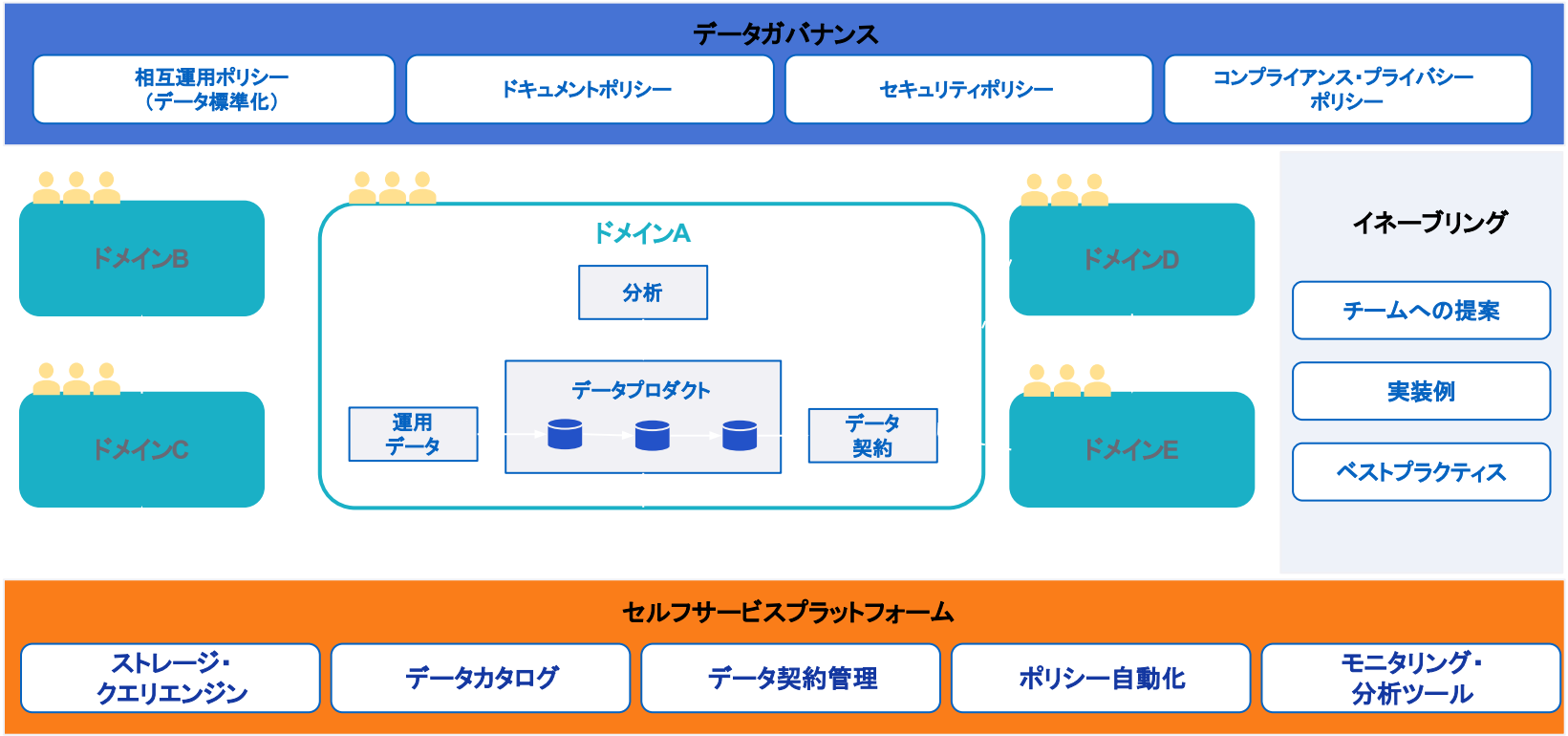


CAP Theorem

PACELC Theorem

参考：Daniel Abadi (2010). Problems with CAP, and Yahoo's little known NoSQL system.
https://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html

データメッシュ



Jochen Christ, ,Larysa Visengeriyeva, Simon Harrer, et al. DATA MESH ARCHITECTURE
https://www.datamesh-architecture.com/ を参考に作成

既存のデータベースに対する分解手順

- データドメインの分析
- テーブルをデータドメインに割り当てる
- データドメインへのデータベース接続を分離する
- スキーマを分離されたデータベースに移動
- 独立したデータベースクラスタに分離する

変化を設計する（適応可能アーキテクチャ）

- シナリオ
  - データ量の増大
  - テナント数の増大
  - トラフィック・リクエスト数の増大
- シングルインスタンス　→　マルチインスタンス
  - データ量の限界
  - 論理DB
- 分散DB

アーキテクチャ設計のアジャイルへの組み込み

- アーキテクチャの決定
- アーキテクチャガバナンス

アーキテクチャの決定スタイル

人月の神話	継続的アーキテクチャ
少数で決める	コミュニケーション努力によって概念を整合
概念が整合しやすい スケールしづらい	コミュニケーション難易度が高い スケールできる

概念の整合が重要

まとめ

- 事前にリスクを起点としてアーキテクチャを検討する
- MVAからはじまり、スケールに備えられるよう継続的アーキテクチャを実践する
  - BtoBではデータ増大が主要リスク
  - マルチテナント戦略とデータ分割は先に検討したほうがよい
- アーキテクチャガバナンスで民主化と概念の整合を担保する

We are hiring!

一緒にプロダクト開発の次元をあげていきたい、良い景気づくりにご興味のある方を絶賛募集しています！！

https://job.loglass.jp/

カジュアル面談Welcomeです！

良い景気を作ろう。