HITACHI
Inspire the Next

COBOLのOLTPシステムをモダナイズする勘所
ークラウドネイティブなトランザクションと共に

株式会社 日立製作所

西谷淳平

問合せ先は、P.44ページにあります。

自己紹介

HITACHI
Inspire the Next

名前　西谷 淳平
所属　株式会社 日立製作所
仕事　Microservices周辺技術の調査・開発・検討

今まで行ってきたこと

2007年～2012年ごろまで　Webサーバの開発・保守

2012年～2018年ごろまで　アプリケーションサーバの開発・保守

2018年～2019年ごろまで　クラウドマイグレーションの企画

2020年～　Microservice Transactionの検討

話すこと

HITACHI
Inspire the Next

「2025年の崖問題」を乗り越えるためには、既存資産を最新のテクノロジーにモダナイズする必要がありますが、それを阻む要因は多々存在します。特に、既存のOLTP(Online Transaction Processing)システムは、そのテクノロジーの特性としてモダナイズしにくい状況にあります。

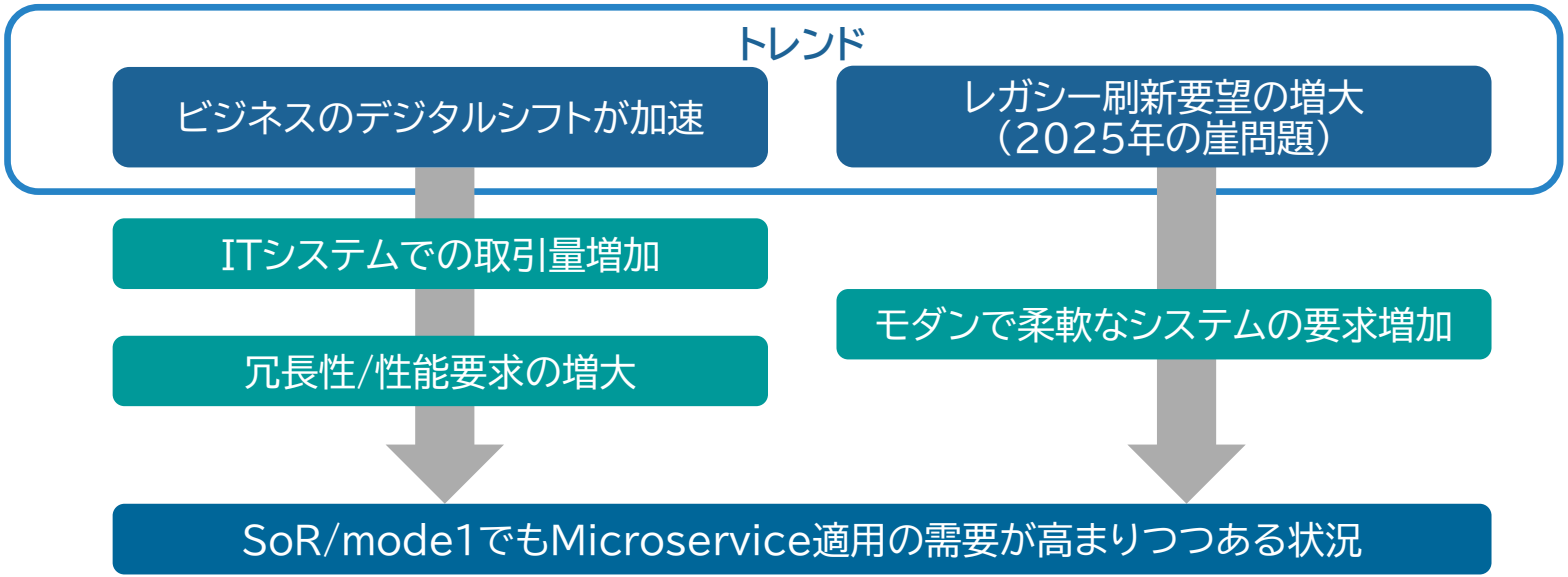
本セッションでは、その問題点を明らかにしたうえで、現在日立が考えているOLTPのモダナイゼーションについて紹介いたします。

1. 2025年の崖問題とレガシーモダナイゼーションの課題
2. Cloud Nativeでトランザクションを実現するときの課題
3. 故障、そして可用性の実現
4. 日立の取り組み:Paxos CommitとCOBOL Microservices

2019年-2021年のIT国内状況の変化

以下要因によって、SoRのMicroservice適用需要が増加傾向

- 不確実性の高まり等による、ビジネスのデジタルシフト加速
- 旧来型システムの刷新要望の増大



「2025年の崖」問題と対策

HITACHI
Inspire the Next

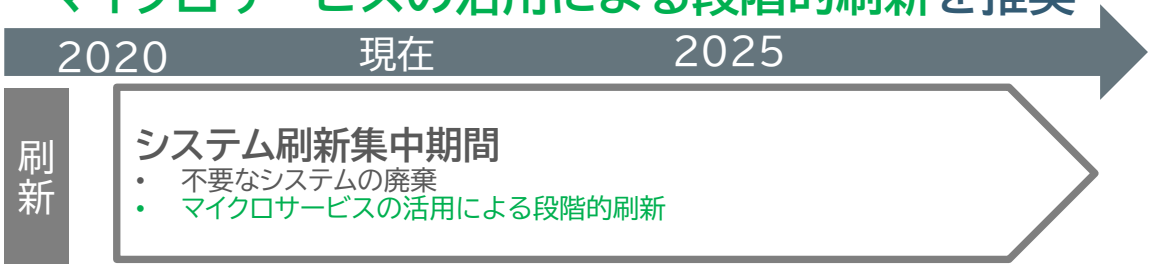
2025年の崖問題

古い基幹システムがレガシー化した結果2025年以降12兆円／年の国内経済損失

対策

ITユーザ企業の基幹システムの

マイクロサービスの活用による段階的刷新を推奨



出典：経済産業省「DXレポート ～ITシステム「2025年の崖」克服とDXの本格的な展開～ 38P」

レガシーモダナイゼーションの課題

HITACHI
Inspire the Next

レガシーのOLTP(Online Transaction Processing)システムが
モダナイズの置き去りになっている状況が散見

課題1

モダナイズ先
Cloud Nativeにおける
Transaction Processingの
課題

課題2

モダナイズ先
インフラストラクチャの
可用性の問題

課題3

レガシー/モダナイ間の技術差異
- 使用プログラミング言語の差
- DBアーキテクチャの差

レガシーモダナイゼーションの課題1

レガシーのOLTP(Online Transaction Processing)システムが
モダナイズの置き去りになっている状況が散見

課題1

モダナイズ先
Cloud Nativeにおける
Transaction Processingの
課題

以下に課題あり

- Microservicesで分散トランザクションを組む複雑さ
- 分散トランザクション自体の不完全性

モダナイズ先
インフラストラクチャの
可用性の問題

レガシー/モダナイ間の技術差異
- 使用プログラミング言語の差
- DBアーキテクチャの差

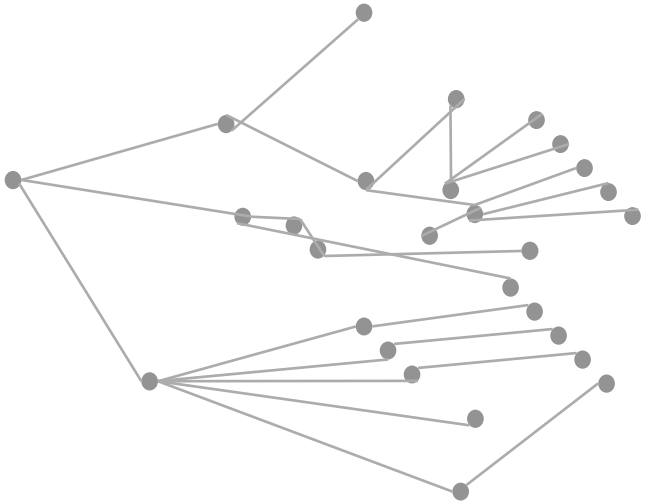
マイクロサービスとは何か

HITACHI
Inspire the Next

ビジネス機能にしたがった複数の小さなサービスでシステムを構成するアーキテクチャ・ソフトウェア開発技法

役割/責務境界にそった小さいサービスを結合

- サービスの理解やサービスの開発が簡単に
- システムの部分的な改善・デプロイが容易
- サービスそれぞれの役割に最適な環境を選択可



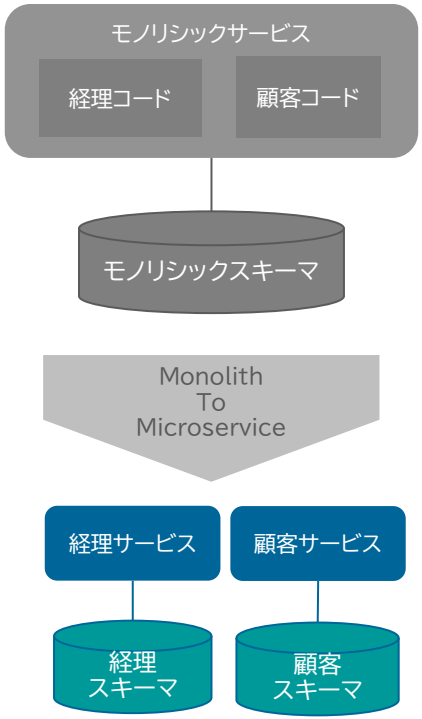
Microservice Data Managementの難所

ビジネス要件・業務要件の変更に応じたアーキテクチャ変更を迅速に行いたい。
そのために、エンタープライズシステムにMicroserviceを導入したい。

エンタープライズシステムでMicroserviceを導入するとき、
業務データの機敏性を高めなければ、業務アプリの機敏性が高まらない

責務境界にしたがって業務アプリとDBを分割し、サービス化。
サービス間の依存性を下げて改修効率を高める。(疎結合・DB per Service)

DBの分散化が始まる。
分散したDBのデータを更新すると、矛盾が発生するリスクが高まる。

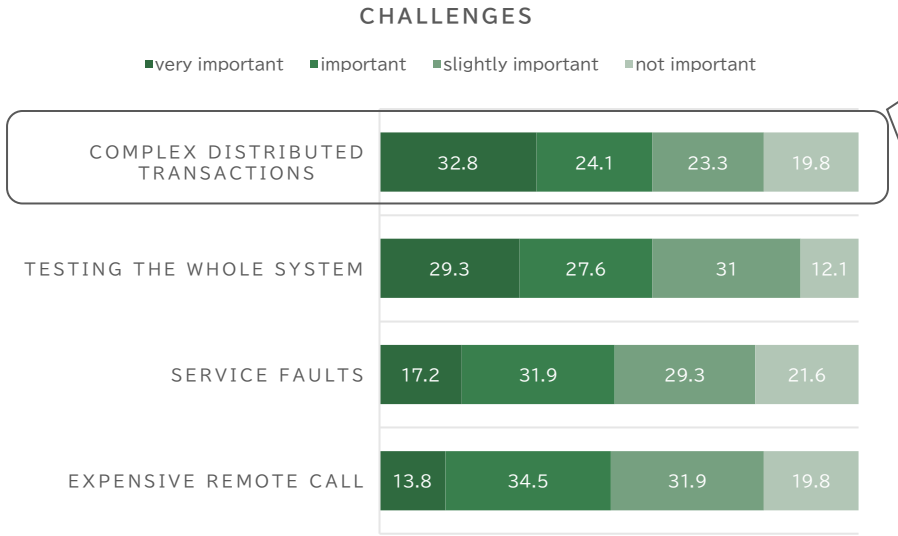


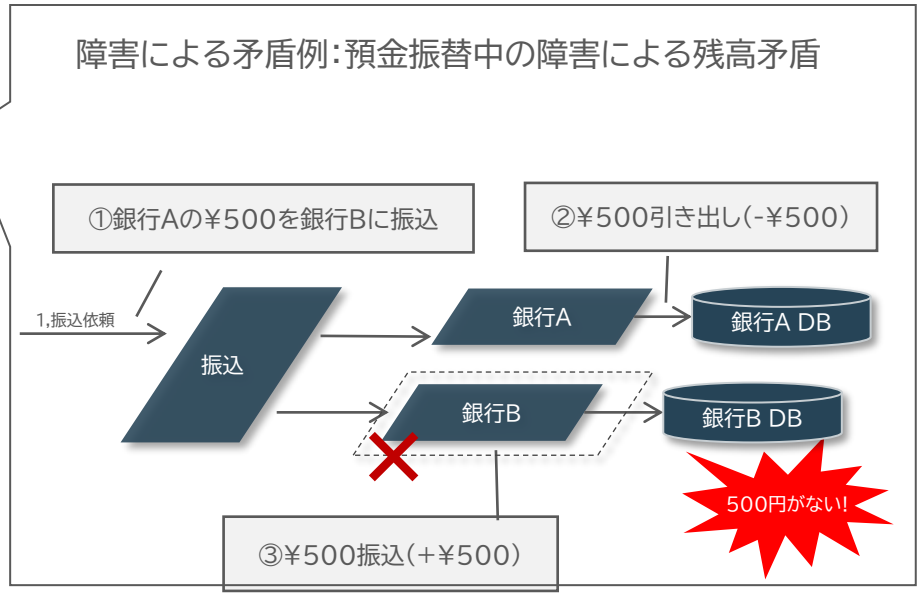
DB per Service
出典:「マイクロサービスアーキテクチャ 104P」:Sam Newman

Microservice Data Managementの問題点

HITACHI
Inspire the Next

複数リソースを更新するときに発生する矛盾防止、整合性を確保するために
システムにトランザクション管理を導入するとシステムが複雑になってしまう





出典:Microservice in Practice: A Survey Study arXiv:1808.04836v1 14 Aug 2018 stack overflow エンジニアへのサーベイ論文

Microserviceの利点と分散トランザクションの課題

Microservicesの教科書「Microservices Patterns: With examples in Java」には、Microservicesの利点と、Global Transaction Manager（GTM）をMicroserviceに適用する課題が定義
そのため、Semantic ACID論文を引用しつつ、アプリでトランザクションを実現する手段を説いた。

Microserviceの利点

- Continuous DeliveryおよびContinuous Deploymentを大きく複雑なサービスで可能にする。
- サービスが小さくなりメンテナンスが容易になる
- サービスを独立してデプロイ・スケールできる
- 新しいテクノロジーの適用を容易にする
- 故障の隔離が簡単になる。

Global Transaction Managerの課題:

1. DBがXAに準拠している必要があり、NoSQL等のモダンなDBを選べない
2. 技術スタックがJava EEやSpringのようなTransaction Monitorを具備するものに固定される
3. CAP定理より、分散システムではConsistencyはある程度諦める必要がある

Saga Patternの適用を推奨

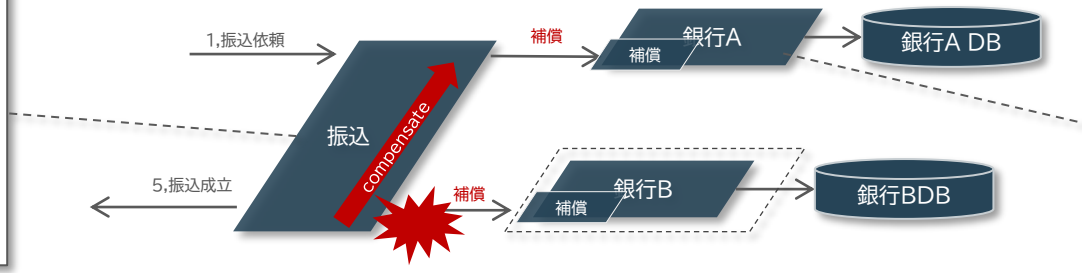
問題点1:システムの複雑化を招く

HITACHI
Inspire the Next

Sagaパターン

- 各DBへのローカルトランザクションをつなげてデータを更新
- ローカルトランザクションが失敗した場合，サービスに実装した補償コードを呼び出して、実行済み更新を取り消す

```
DistributedTransaction(){
 try{
  entity1.update();
  entity2.update();
 }catch(BusinessException e){
  entity1.cancel();
  entity2.cancel();
 }catch(RuntimeException e){
  if(e=xxxx) xxxxx;
  if(e=xxxx) xxxxx;
  if(e=xxxx) xxxxx;
 <snip>
 }
}
```

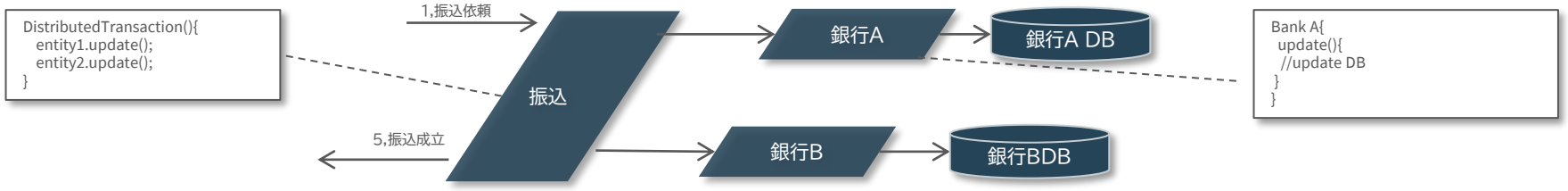



```
entityX{
 update(){
  getCriticalSection(FileldA);
  //update
  freeCriticalSection(FiledA);
 }
 cancel(){
  getCriticalSection(FileldA);
  //CompensationUpdate
  freeCriticalSection(FiledA);
 }
 getCriticalSection(Fileld){
   //CuncurrencyControlImpl
 }
 freeCriticalSection(Fileld){
   //CuncurrencyControlImpl
 }
}
```

アプリの業務コードへ不整合防止用コードを組み込むと、

補償トランザクションコードが業務コード/ビジネスルールを汚染する

複雑化の要因



リソース間不整合が発生する原因は三つ

分散リソースの不整合

業務エラー

例:
- 上限以上の金額を引き落とした
- 残高不足になった

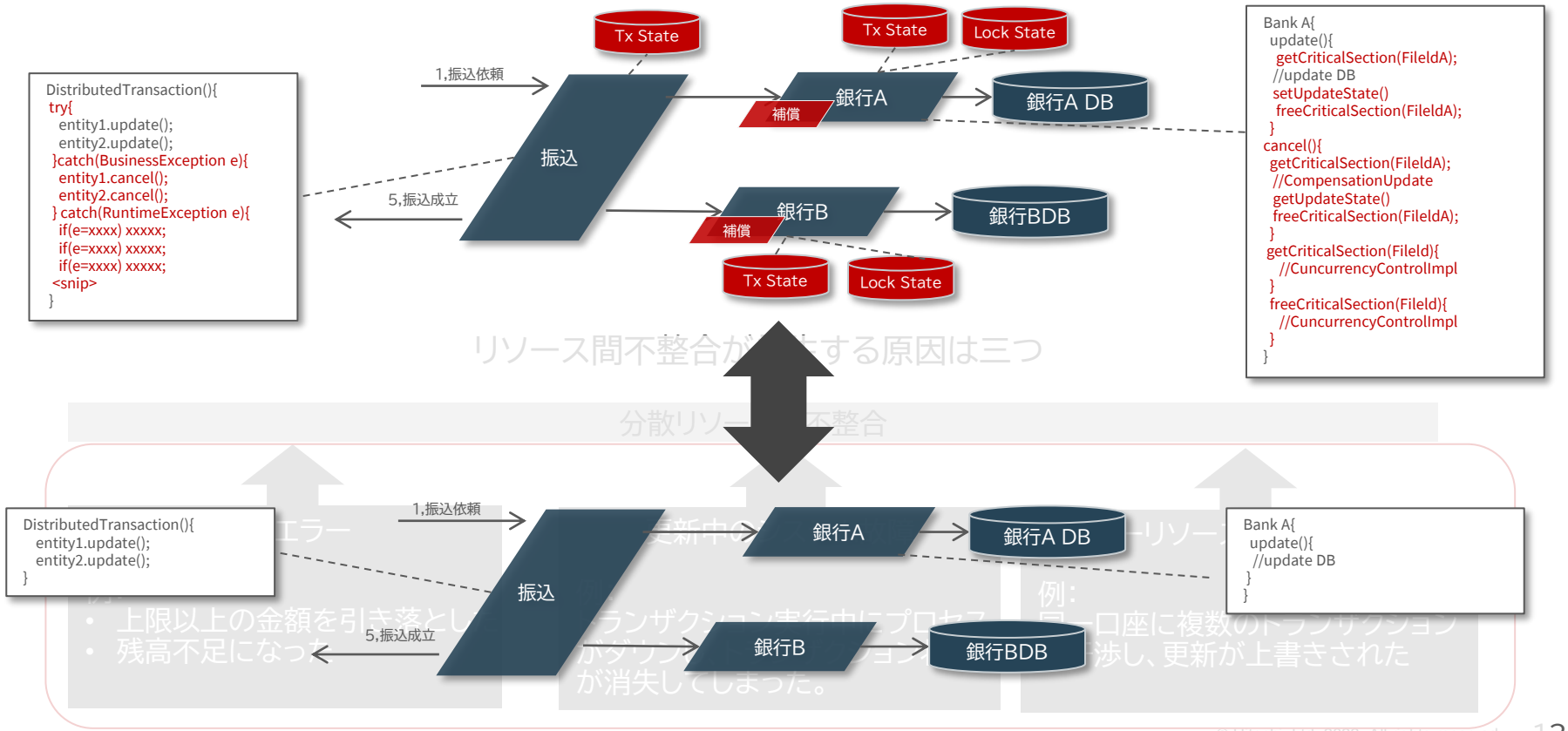
更新中のシステム故障

例:
トランザクション実行中にプロセスがダウンし、トランザクション状態が消失してしまった。

同一リソースへの並列更新

例:
同一口座に複数のトランザクションが干渉し、更新が上書きされた

複雑化の要因



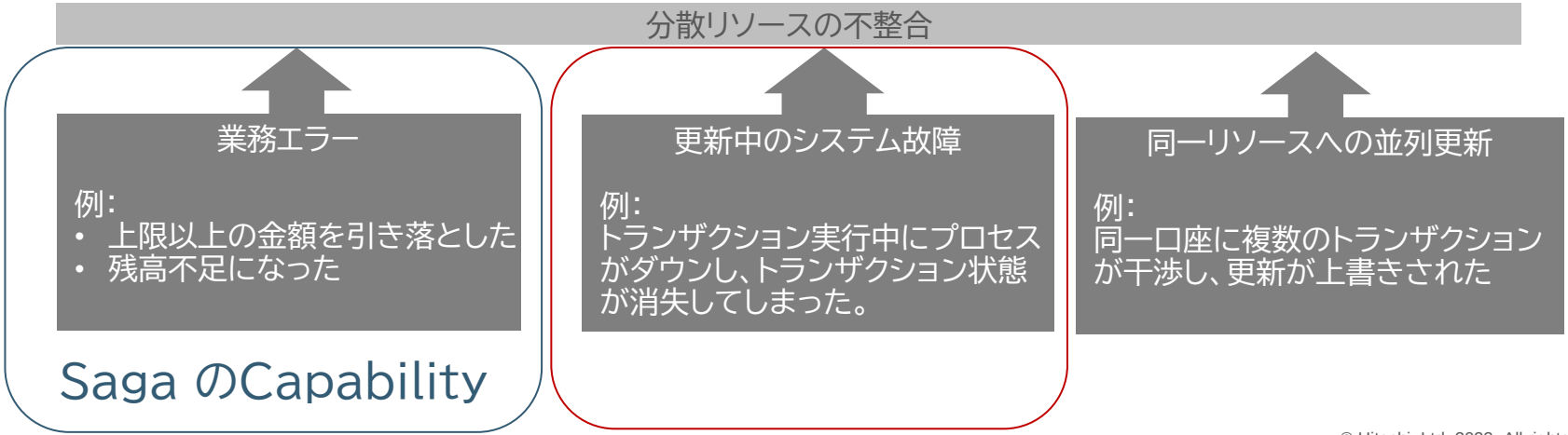
問題点2：Saga パターンのみでは矛盾発生ケースを救いきれない

HITACHI
Inspire the Next

リソース間不整合が発生する原因は

「業務エラー」「同一リソースへの並列更新」「更新中のシステム故障」

- 「同一リソースへの並列更新」による不整合は、**Sagaでは救いきれない**
- 「更新中のノード故障」による不整合も、複雑ケースではSagaのみで防ぎきれない



レガシーモダナイゼーションの課題2

課題

信頼性の高いインフラストラクチャで構築していたシステムはクラウドで同等の信頼性を確保することができるのだろうか？

課題1

モダナイズ先
Cloud Nativeにおける
Transaction Processingの
課題

課題2

モダナイズ先
インフラストラクチャの
可用性の問題

課題3

レガシー/モダナイ間の技術差異
- 使用プログラミング言語の差
- DBアーキテクチャの差

可用性とは何か

- 利用者がシステムを利用できる度合い。Availabilityの日本語訳。
- 信頼性指標RASISにおいて、可用性は稼働率で計測する。

$$\text{可用性} = \text{稼働率} = \frac{\text{稼働時間}}{\text{稼働時間} + \text{停止時間}}$$

稼働時間: MTBF, Mean Time Between Failure ,平均故障間隔
停止時間: MTTR, Mean Time To Repair ,平均修復時間

可用性を高める考え方1

時間的稼働率で考える場合、以下のアプローチとなる

1. 稼働時間を上げる ↑
2. 停止時間を下げる ↓

$$\text{可用性} = \text{稼働率} \uparrow = \frac{\uparrow \text{稼働時間}}{\uparrow \text{稼働時間} + \text{停止時間} \downarrow}$$

可用性を上げるアプローチ

HITACHI
Inspire the Next

可用性を上げる ＝ 稼働率を上げる

1. **稼働時間を上げる**
   - バグを失くす。
   - **単一障害点を避ける。冗長性を高める。**
   - **停止せずに変更できるようにする。**

寄与

- サーキットブレイカー
- ローリングアップデート
- ブルーグリーンデプロイメント
- カナリアリリース

2. **停止時間を下げる**
   - 停止を未然に防ぐ。
     - **停止を監視する。**
     - **停止の予兆を検知し、切り替える**
   - 停止したとき、迅速に復旧する。
     - **原因を迅速に特定する。**
     - **迅速に対策する。**

寄与

- メトリクス監視
- トレース
- 障害自動復旧
- 障害運用自動化
- Primary-backup

可用性を高める考え方2

ノードの並列性を高める。

->システム全体の稼働率 = 1 - 全ノードが同時に停止する確率

->全ノードが同時に停止する確率 =(1 – ノード1稼働率) ×(1 – ノード2稼働率) …

上記並列システムの稼働率=1-（1-0.9）^{3}=**0.999**

アプリケーションで可用性を高めるときの課題

HITACHI
Inspire the Next

アプリケーションは、「状態を持たないステートレスアプリケーション」と、

「状態を持つステートフルアプリケーション」に大別できる。

ステートフルアプリケーションは可用性を高めにくい。

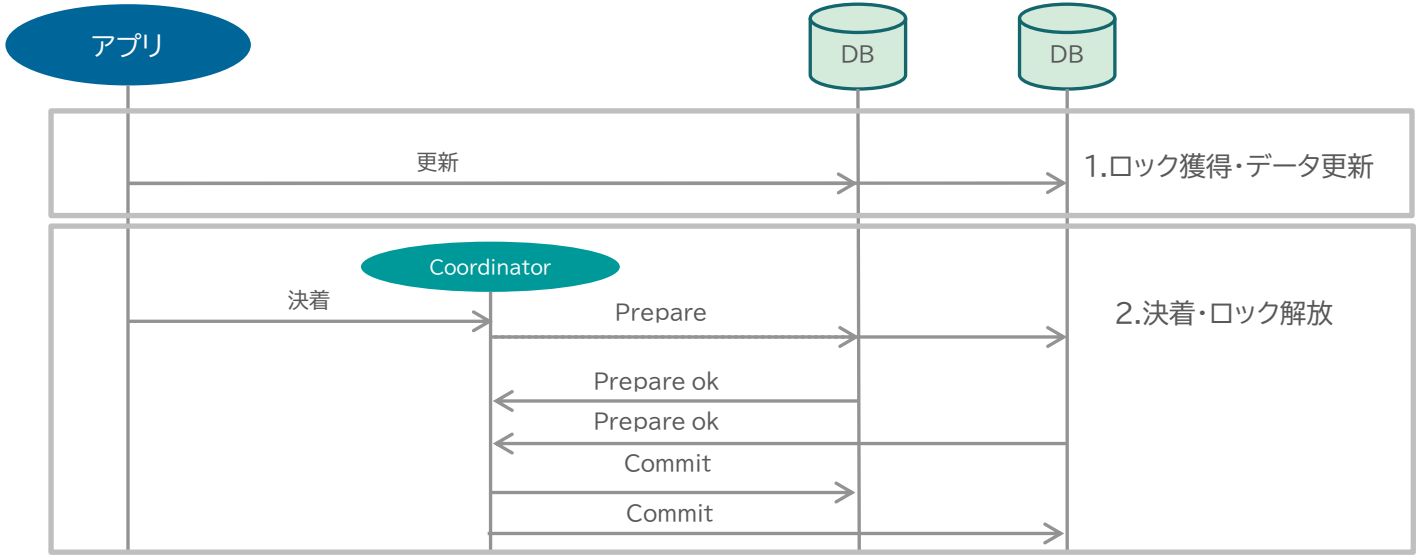
- ステートレスアプリケーションは、単純な冗長化で稼働率が向上
- ステートフルアプリケーションは、単純に冗長化してしまうと、並列ノードで状態が揃わずノードによって挙動が変わってしまう
- ステートフルアプリケーション例：HTTP セッション、Transaction Processing、等々

2Phase Commitによるトランザクション制御

Microserviceではアンチパターンとされている2Phase Commitには以下2つのメリットがあり。

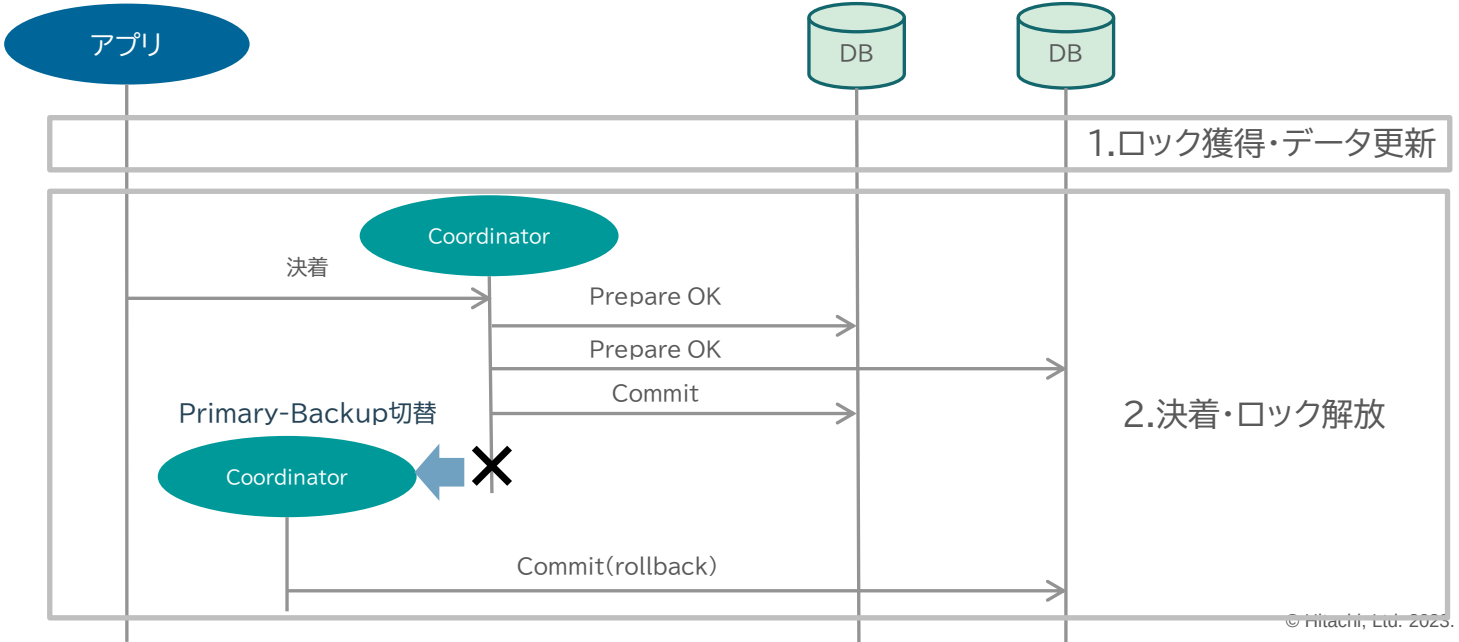
◎ 2Phase Lock(更新～状態確定(commit or rollback)の間を排他区間の設定)搭載。コンカレンシーによる不整合防止。

◎ JTA/@Transactionalのように、業務アプリに対するTransaction制御機能の露出を最低一行程度と極めて少なくできる。疎結合。

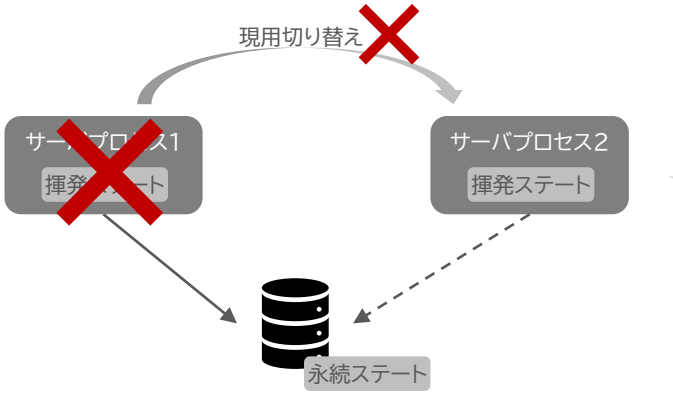


ヒューリスティック対策の系切り替え（Primary-Backup）

Transaction Manager/Coordinatorの単一障害を防ぐために、トランザクションログをストレージに永続化し、
Primary-Backupによる系切り替えを行う運用が一般的。
ただし、「信頼性の低いインフラストラクチャ（後述）」でPrimary-Backupすると、系切り替えに想定より時間がかかる場合がある。
結果、ロック解除に時間がかかったり、最悪ヒューリスティックになり、可用性低下の原因となり得る。



ステートフルアプリケーション可用性向上手段1:シェアードディスク・スタンバイ切替方式（Primary-Backup）

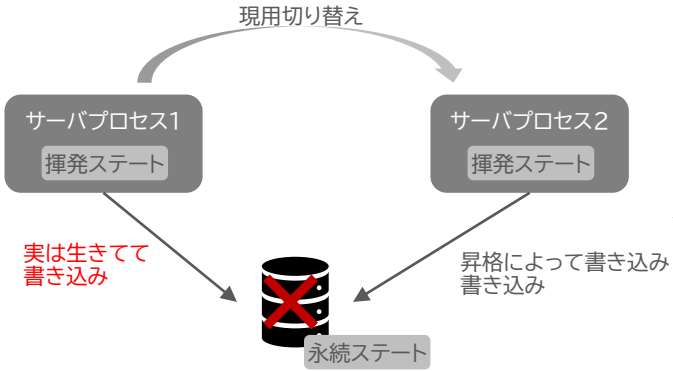


バックアッププロセスの昇格失敗例1:

- 現用系の故障通知が遅延し、待機に切り替変わらない

例）

平成24年 東京証券取引所 株式売買システム障害



バックアッププロセスの昇格失敗例2:

1. 現用系の故障誤検知によって、待機に切り替え
2. 現用系が実は生きていて共用ストレージに書き込み
3. 待機系も共用ストレージに書き込むため競合。データ破壊発生

Primary-Backupとは、**故障の通知が即座に正しく系に伝達されること**を前提とした形態

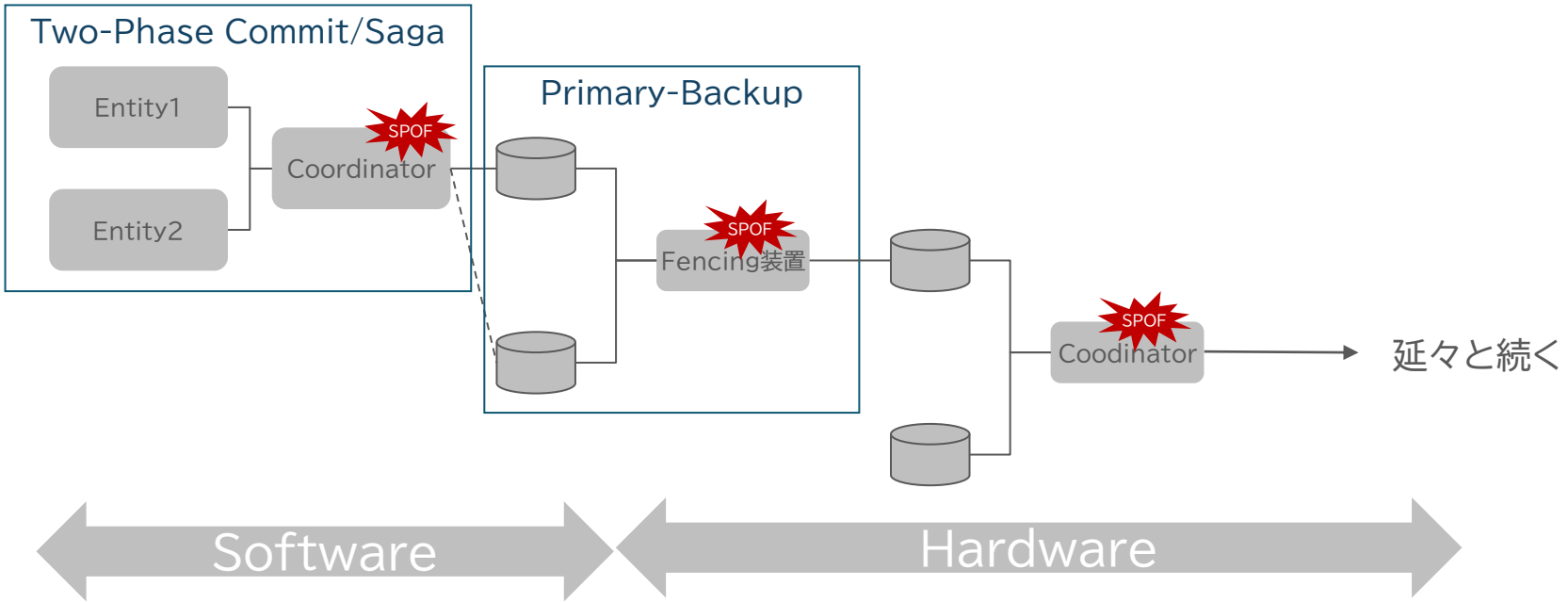
参照元:Google - Site Reliability Engineering (sre.google)

Primary-Backupの単一障害点問題

HITACHI
Inspire the Next

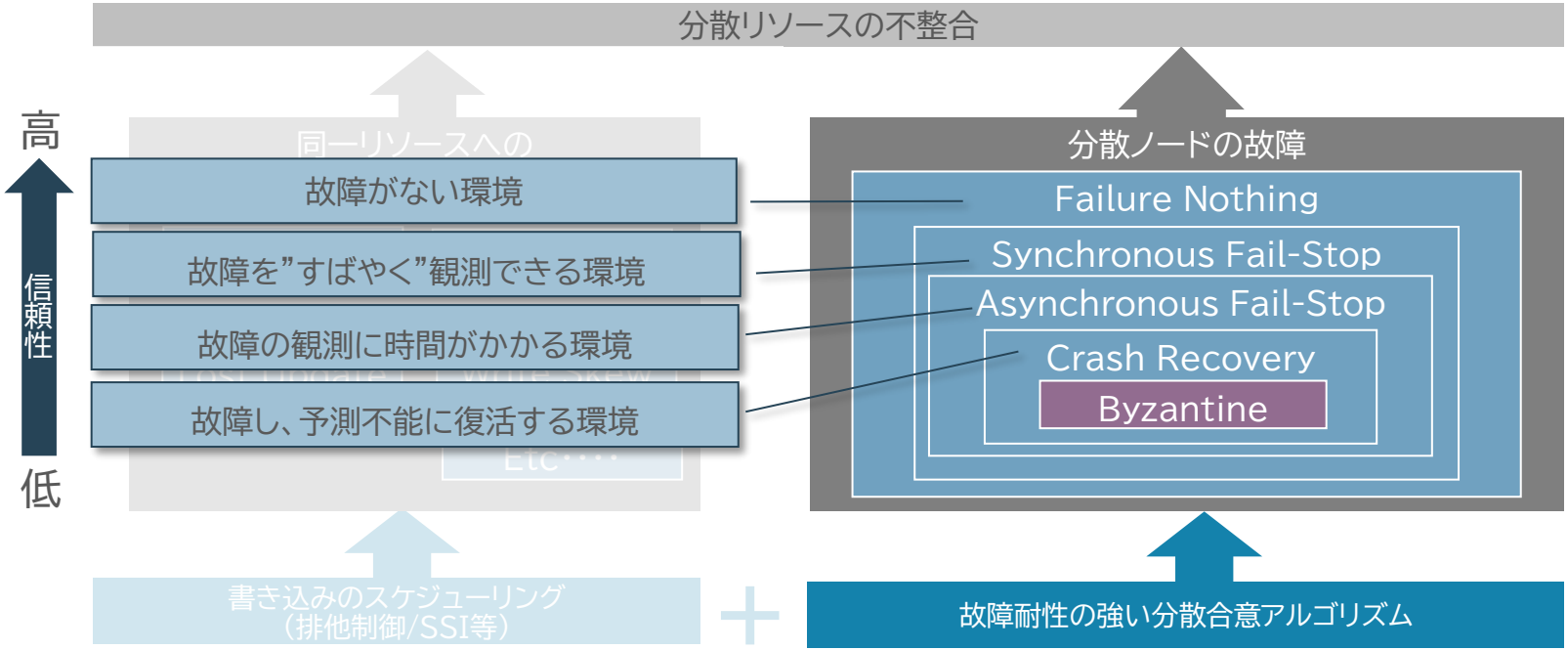
◎ 単一障害点を防ぐ仕組み自体が単一障害点となる

◎ 単一障害点を防ぐ仕組みがハードウェアへの依存性を高める



故障モデル(インフラストラクチャの信頼性段階)

- ノードが壊れる状況の対処の厳しさを段階的に切り分けたモデル
- 信頼性の高い環境で動く合意プロトコルは、信頼性が低い環境で動作するとは限らない

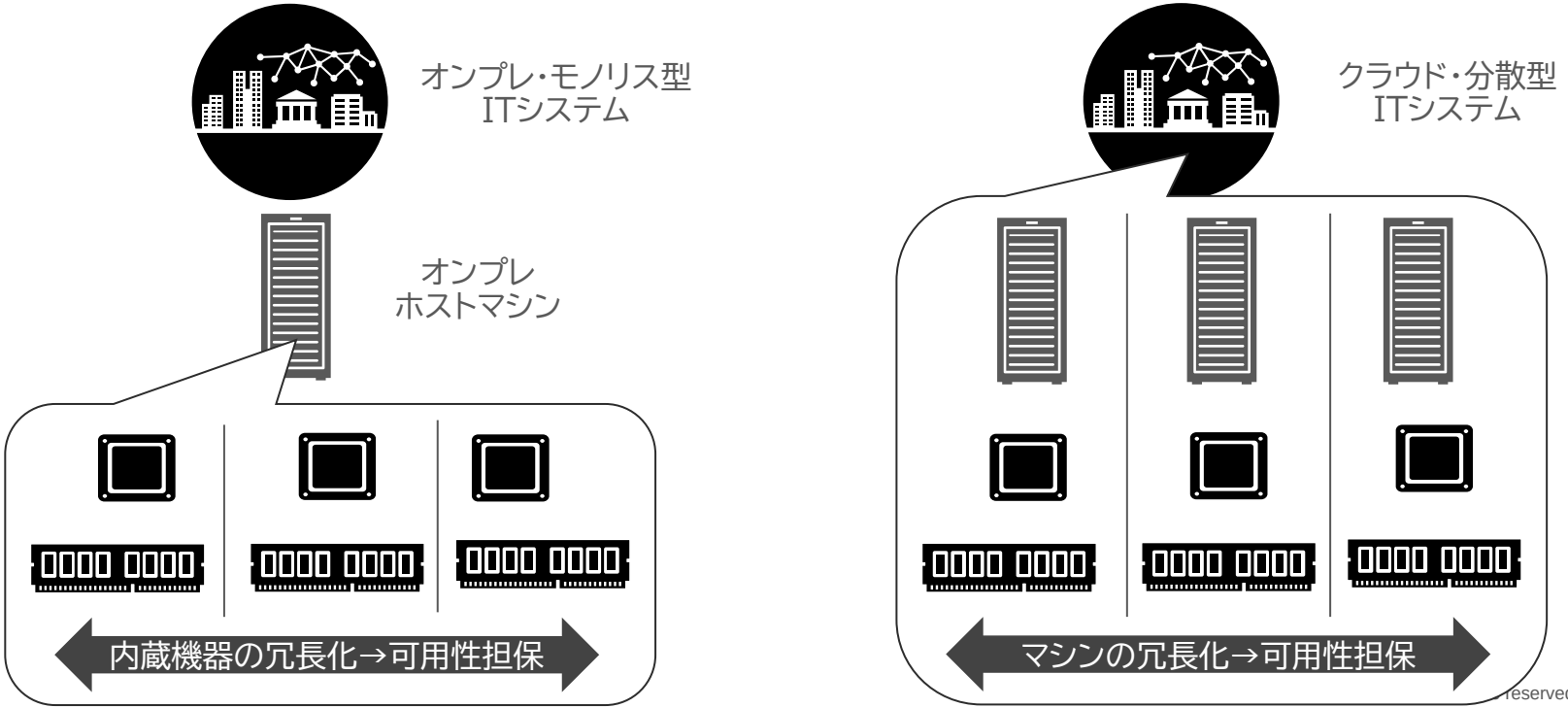


可用性担保に関するオンプレミスホストとクラウドの考え方の違い

HITACHI
Inspire the Next

◎ オンプレミスホスト：マシン筐体内の部品を冗長化して可用性を担保

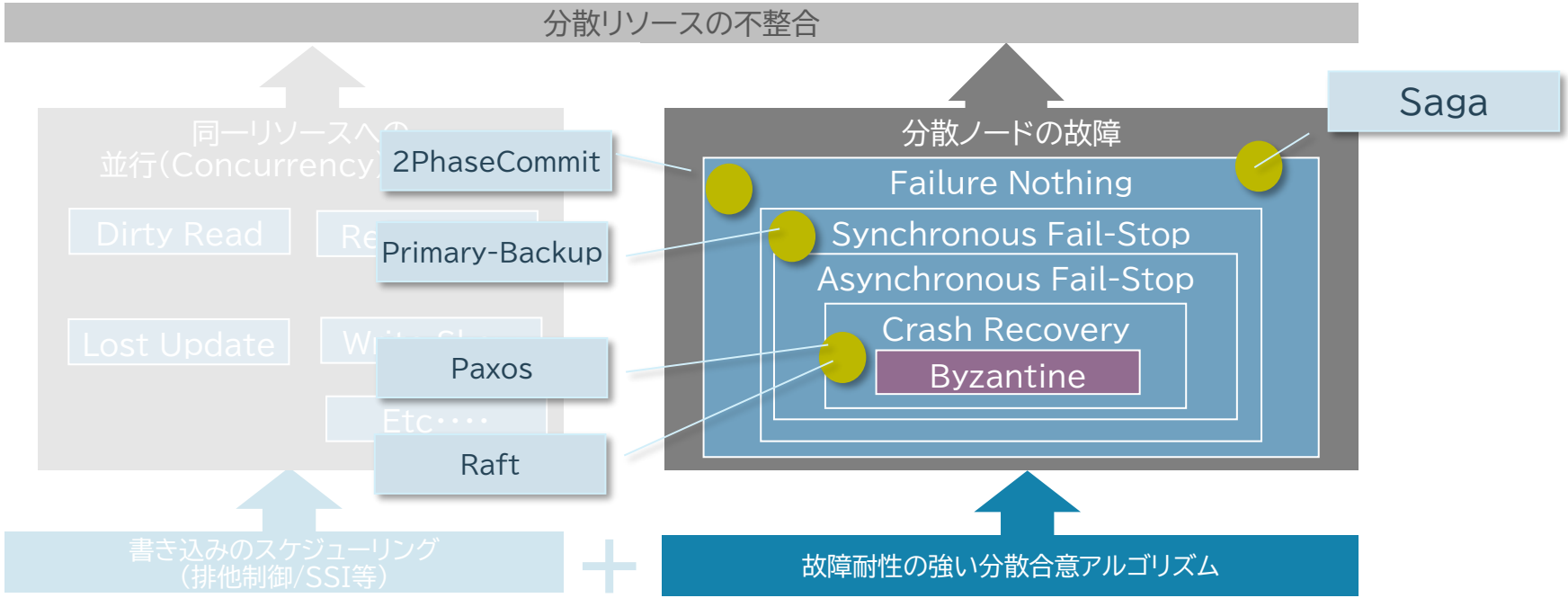
◎ クラウド：マシン自体を冗長化して可用性を担保



故障モデルにおけるSaga/2PhaseCommit/Primary-Backup

- Saga/2Phase Commit は「故障がない環境」が前提で動作可能
- Primary-Backupは「 Synchronous Fail-Stop」で有効な分散合意アルゴリズム

Saga/2Phase Commit/Primary-Backupの組み合わせは、「Asynchronous Fail-Stop」で動作しない

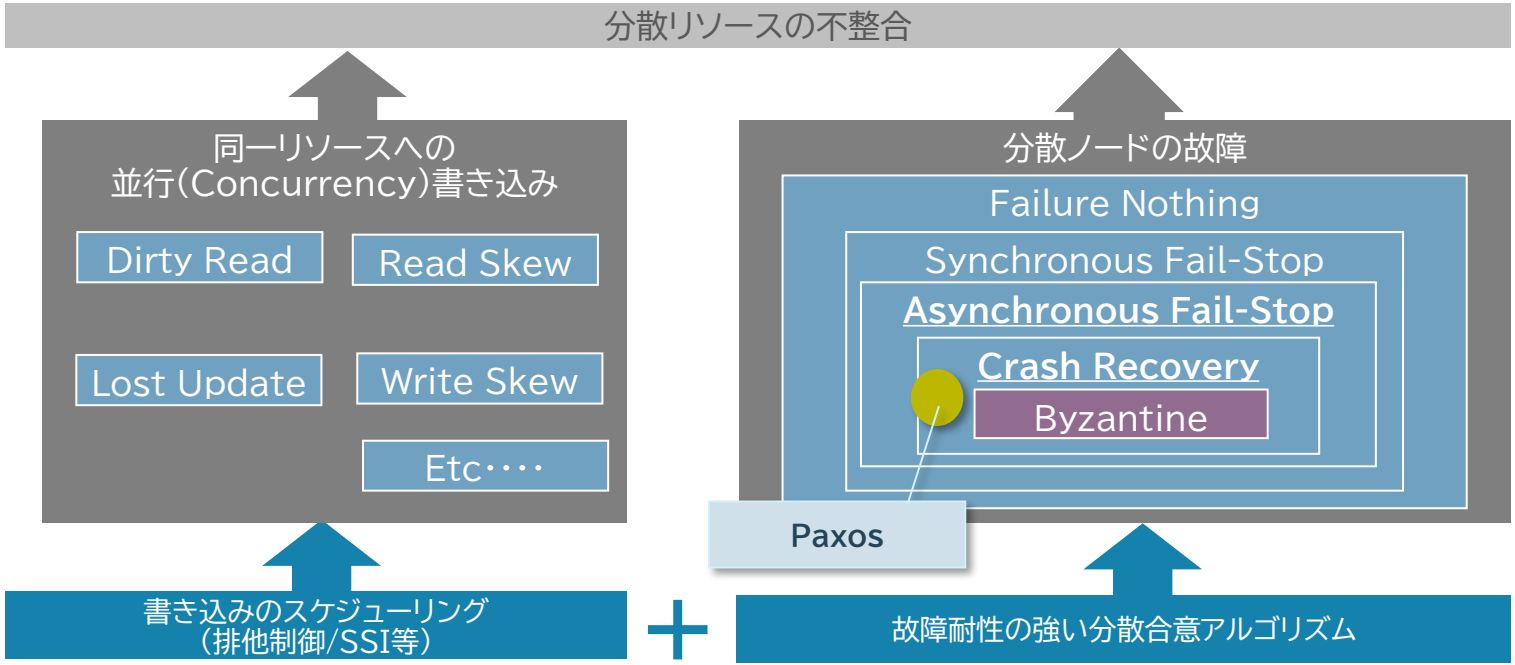


現代ITシステムの信頼性をどのように判断すべきか

HITACHI
Inspire the Next

Asynchronous Fail-Stop/Crash-Recoveryと同程度の信頼性になることも視野に入れるべき

- オンプレミス環境においても、Primary-backup昇格失敗事例が現在でも起きている
- クラウド環境において、システムのレジリエンシーを高めるために、システムを分散させはじめると通信の遅延が発生し始める。(例:リージョンを跨ったDisaster Recovery構成等)



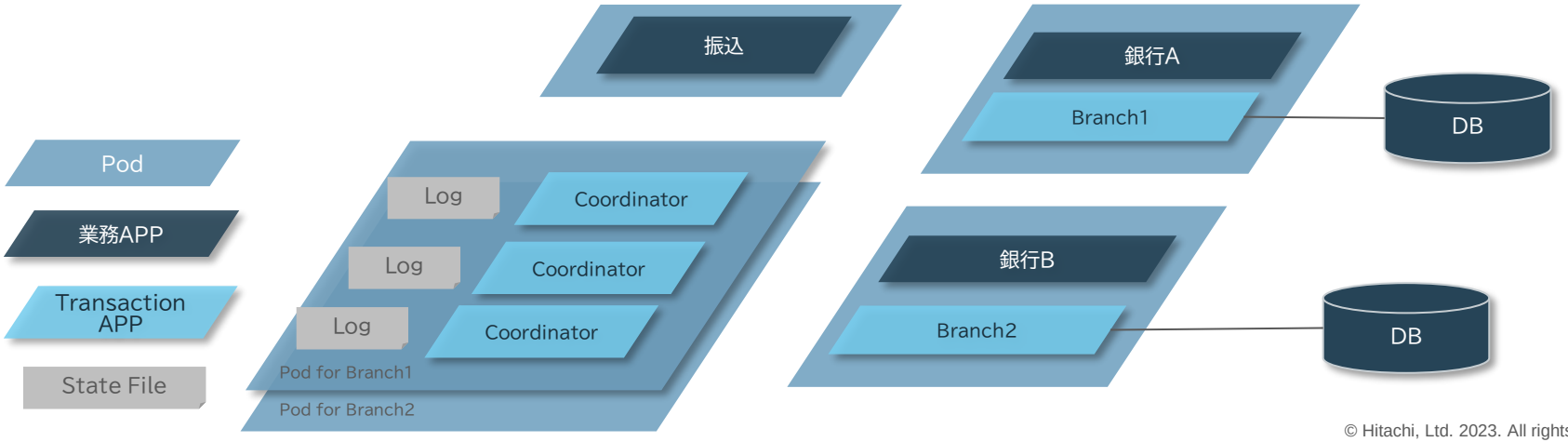
Microservices Transactionの課題と解決

Microservices Engineer のトランザクション課題:

1. 分散トランザクションを組むとアプリケーションが複雑になる

Global Transaction Managerの課題:

1. 技術スタックがJava EEやSpringのようなTransaction Monitorを具備するものに固定される
2. CAP定理より、分散システムではConsistencyはある程度諦める必要がある
3. DBがXAに準拠してる必要があり、NoSQL等のモダンなDBを選べない



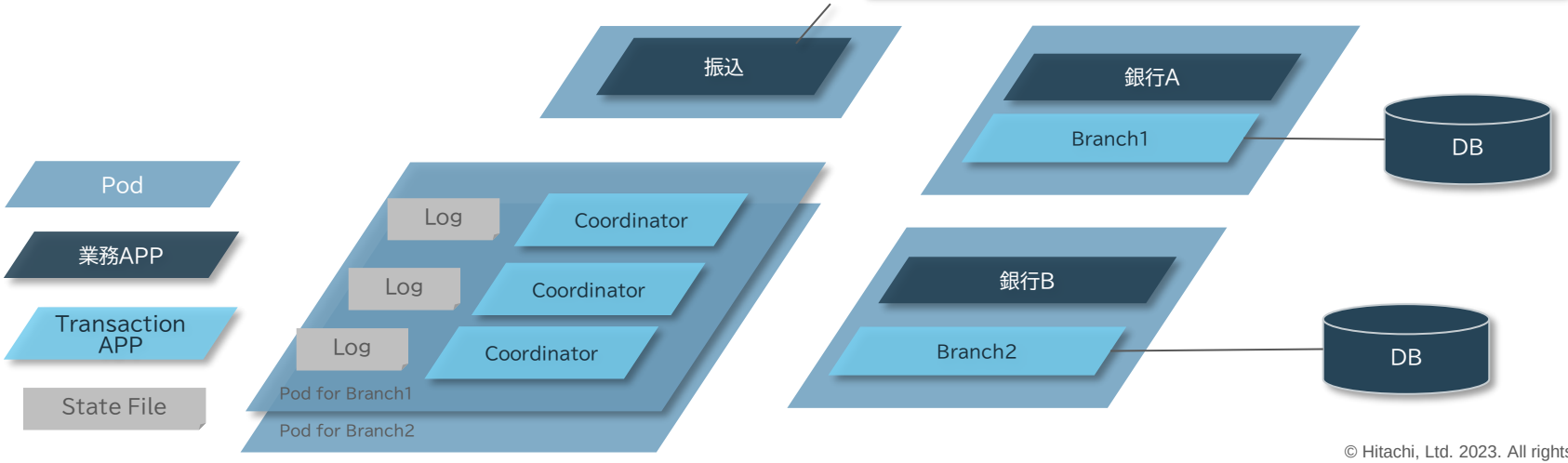
Microservices Engineer のトランザクション課題:

1. 分散トランザクションを組むとアプリケーションが複雑になる→@Transactionalレベルでシンプルに記載

Global Transaction Managerの課題:

1. 技術スタックがJava EEやSpringのようなTransaction Monito
2. CAP定理より、分散システムではConsistencyはある程度諦める必
3. DBがXAに準拠してる必要があり、NoSQL等のモダンなDBを選べ

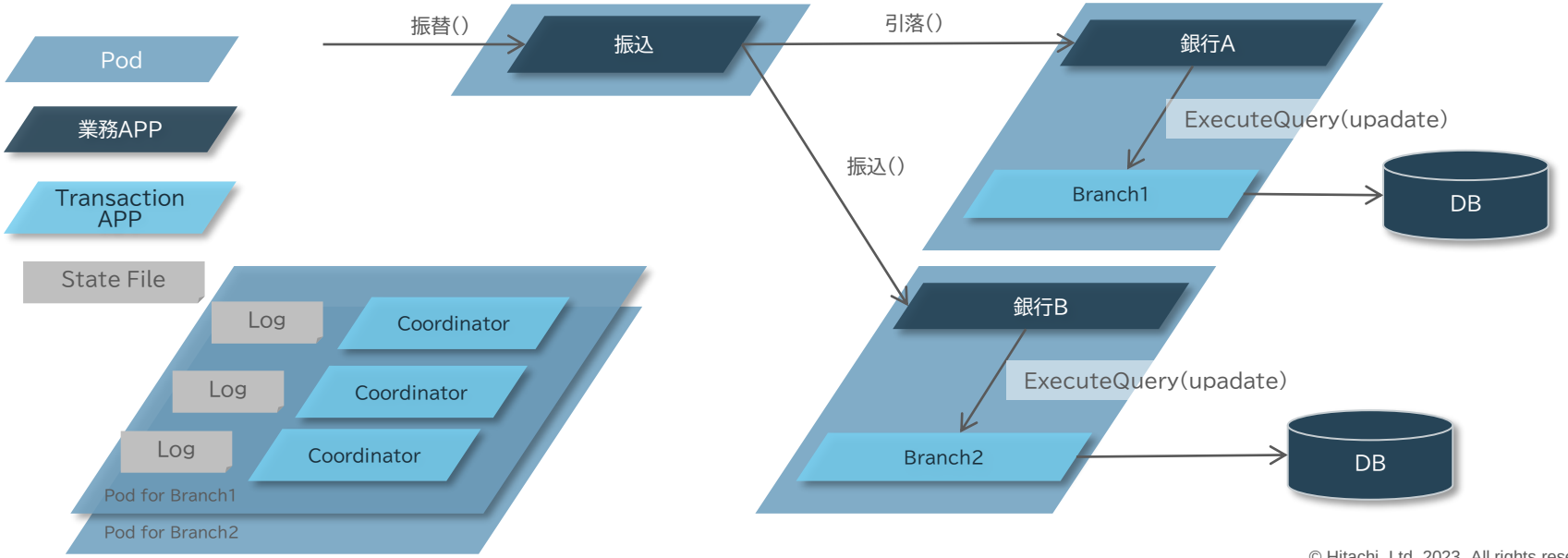
```
@Transactional
public String TransferMoney(int mount){

    Bank1.witdraw(mount);
    Bank2.deposit(mount);
}
```



Paxos Commit におけるトランザクション決着

HITACHI
Inspire the Next

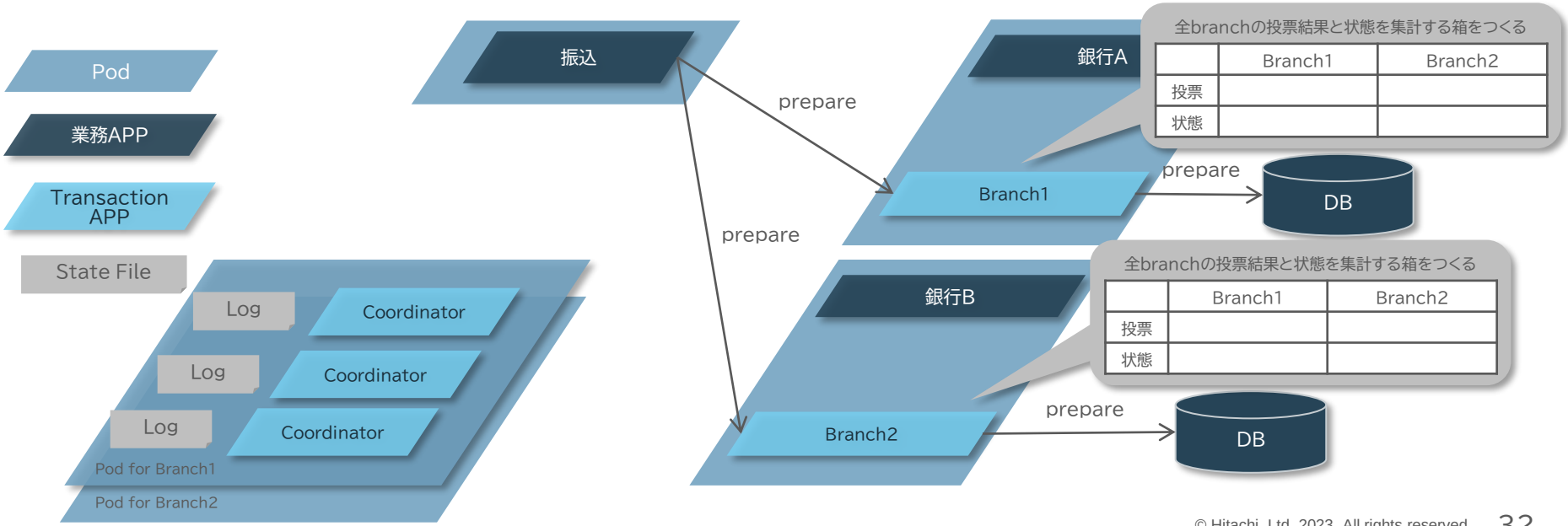
DB更新 → Atomic Commit 下準備 → Commit/Rollback の集計 → Atomic Commit の決着



Paxos Commit におけるトランザクション決着

HITACHI
Inspire the Next

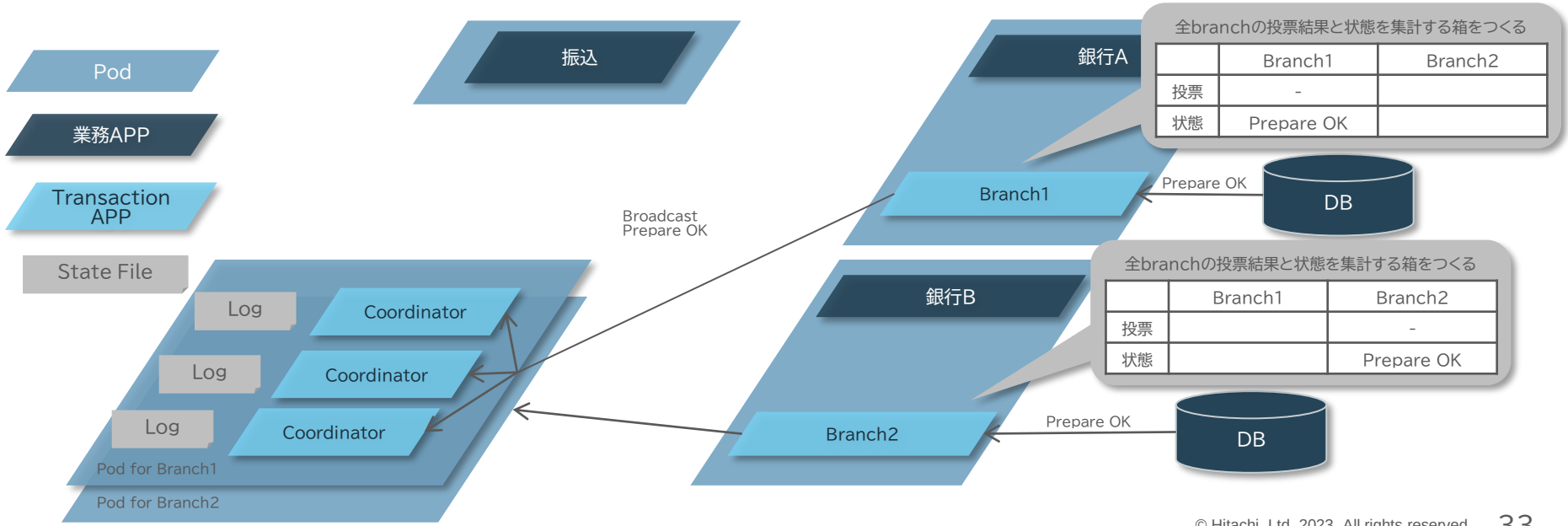
DB更新 → Atomic Commit 下準備 → Commit/Rollback の集計 → Atomic Commit の決着



Paxos Commit におけるトランザクション決着

HITACHI
Inspire the Next

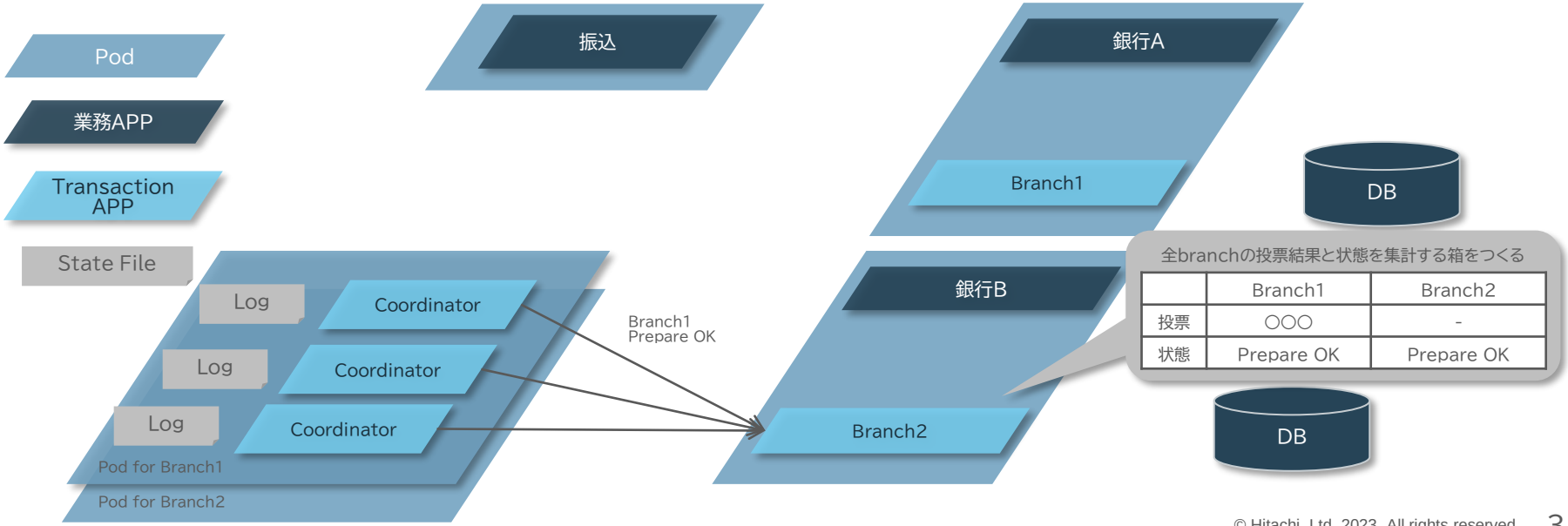
DB更新 → Atomic Commit 下準備 → Commit/Rollback の集計 → Atomic Commit の決着



Paxos Commit におけるトランザクション決着

HITACHI
Inspire the Next

DB更新 → Atomic Commit 下準備 → Commit/Rollback の集計 → Atomic Commit の決着

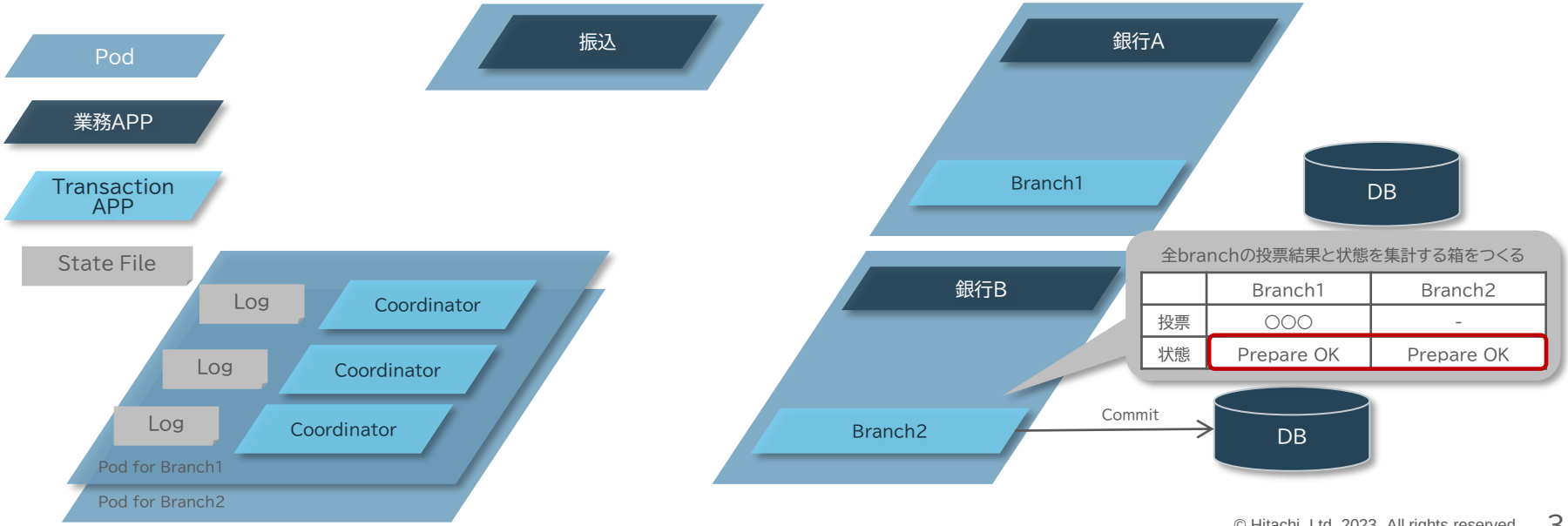


	Branch1	Branch2
投票	○○○	-
状態	Prepare OK	Prepare OK

Paxos Commit におけるトランザクション決着

HITACHI
Inspire the Next

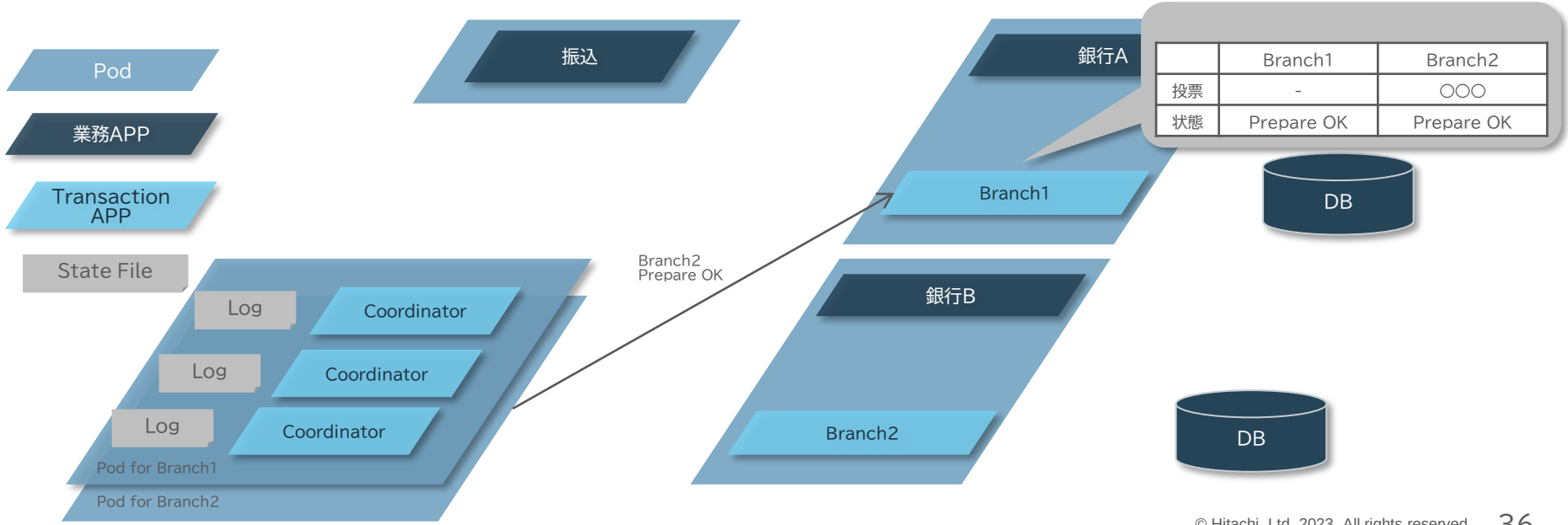
DB更新 → Atomic Commit 下準備 → Commit/Rollback の集計 → Atomic Commit の決着



Paxos Commit におけるトランザクション決着

HITACHI
Inspire the Next

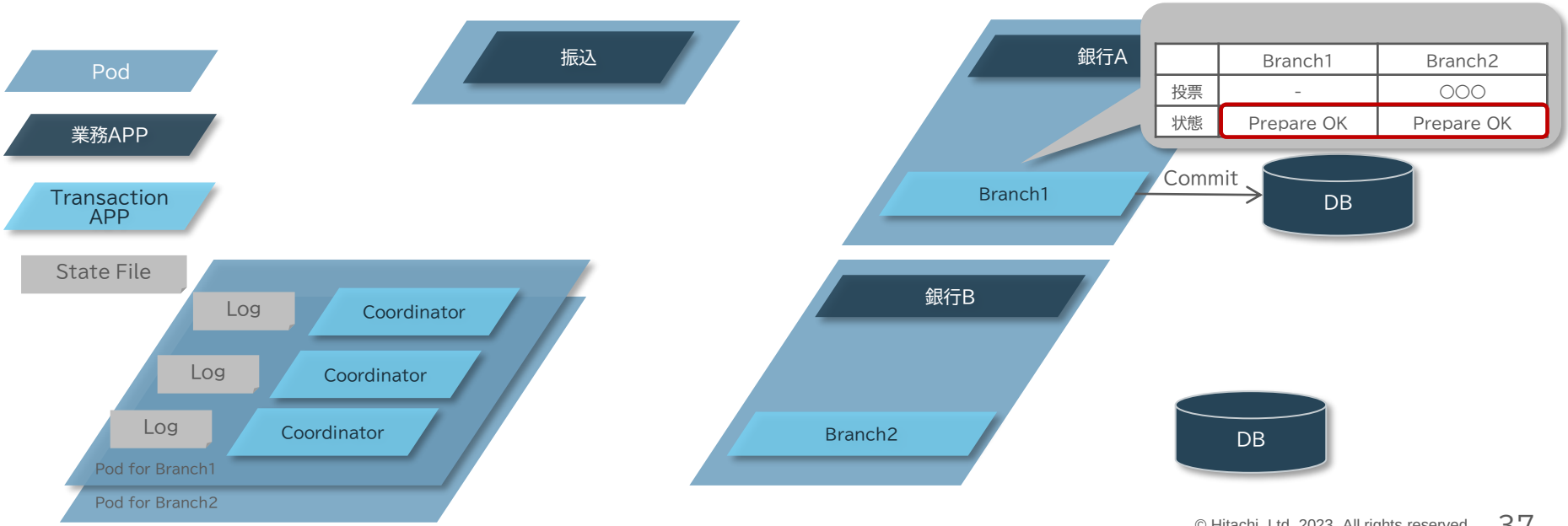
DB更新 → Atomic Commit 下準備 → Commit/Rollback の集計 → Atomic Commit の決着



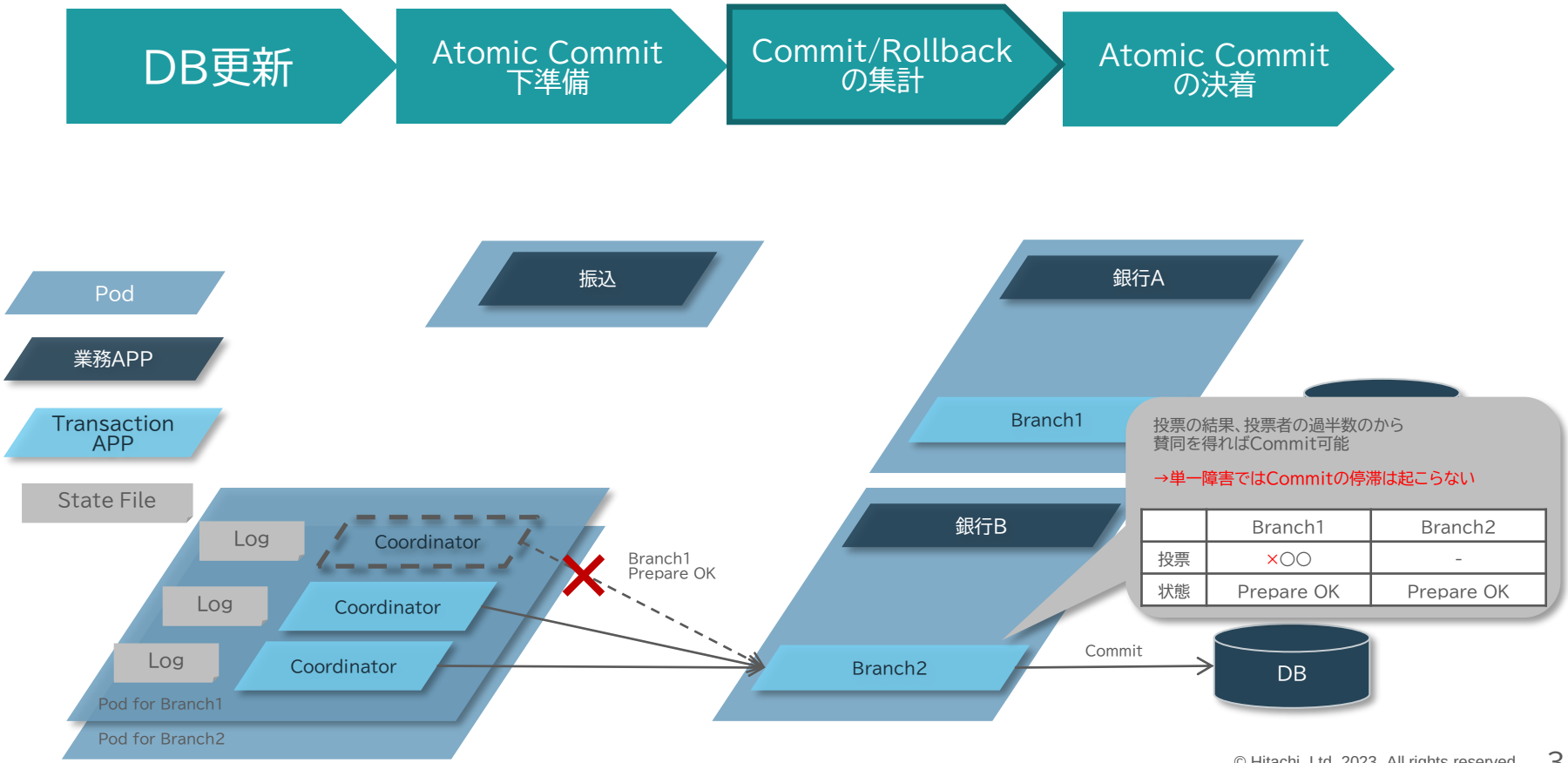
Paxos Commit におけるトランザクション決着

HITACHI
Inspire the Next

DB更新 → Atomic Commit 下準備 → Commit/Rollback の集計 → Atomic Commit の決着



Paxos Commit の単一障害点抑止



COBOLのリビルド方式案

cessing)システムが

アプローチ

- 言語体系の異なるプログラムの移行は大変
- Microservicesの作法に則り、**COBOLプログラムをなるべくそのまま利用する形でMicroserviceを作成する方式**を検討中

課題3

レガシー/モダナイ間の技術差異
- 使用プログラミング言語の差
- DBアーキテクチャの差

モダナイズ先
Cloud Nativeにおける
Transaction Processingの
課題

モダナイズ先
インフラストラクチャの
可用性の問題

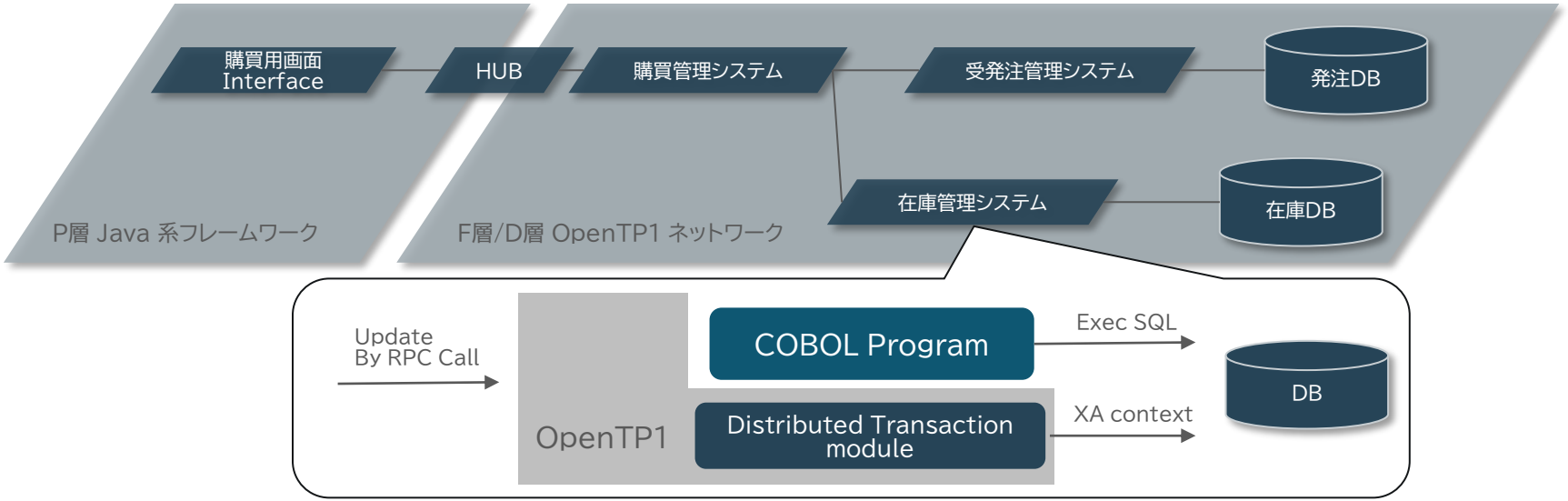
OpenTP1による更新系システム

HITACHI
Inspire the Next

購買管理システム

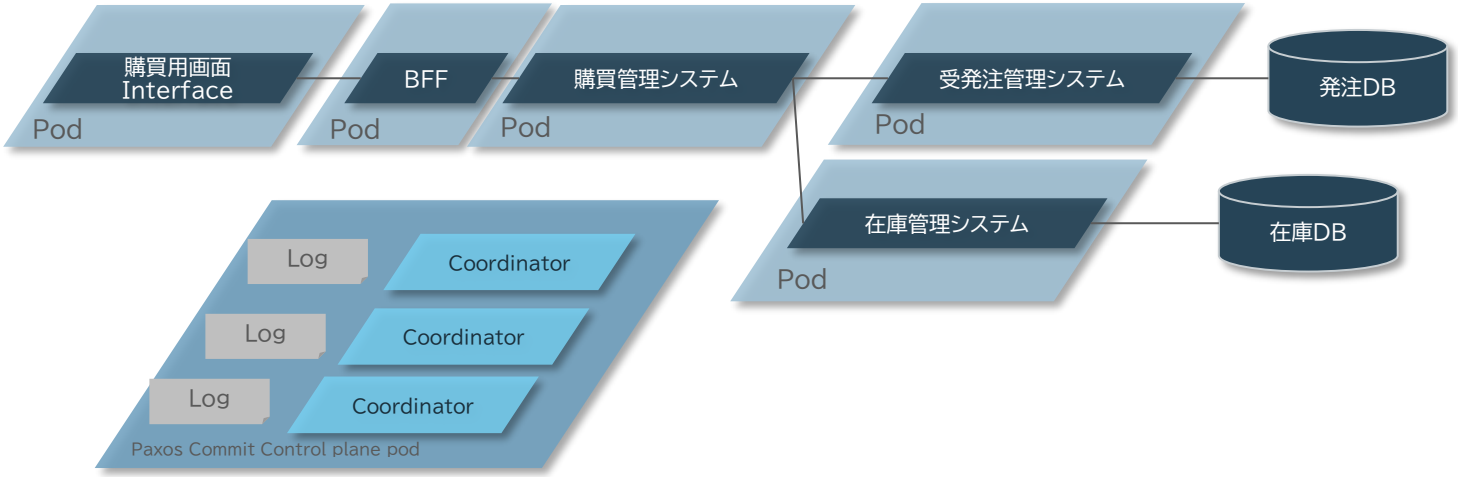
◎ 受注業務と在庫引き当て業務の間で原子性が必要

◎ 在庫がないのに受注することを防ぐため



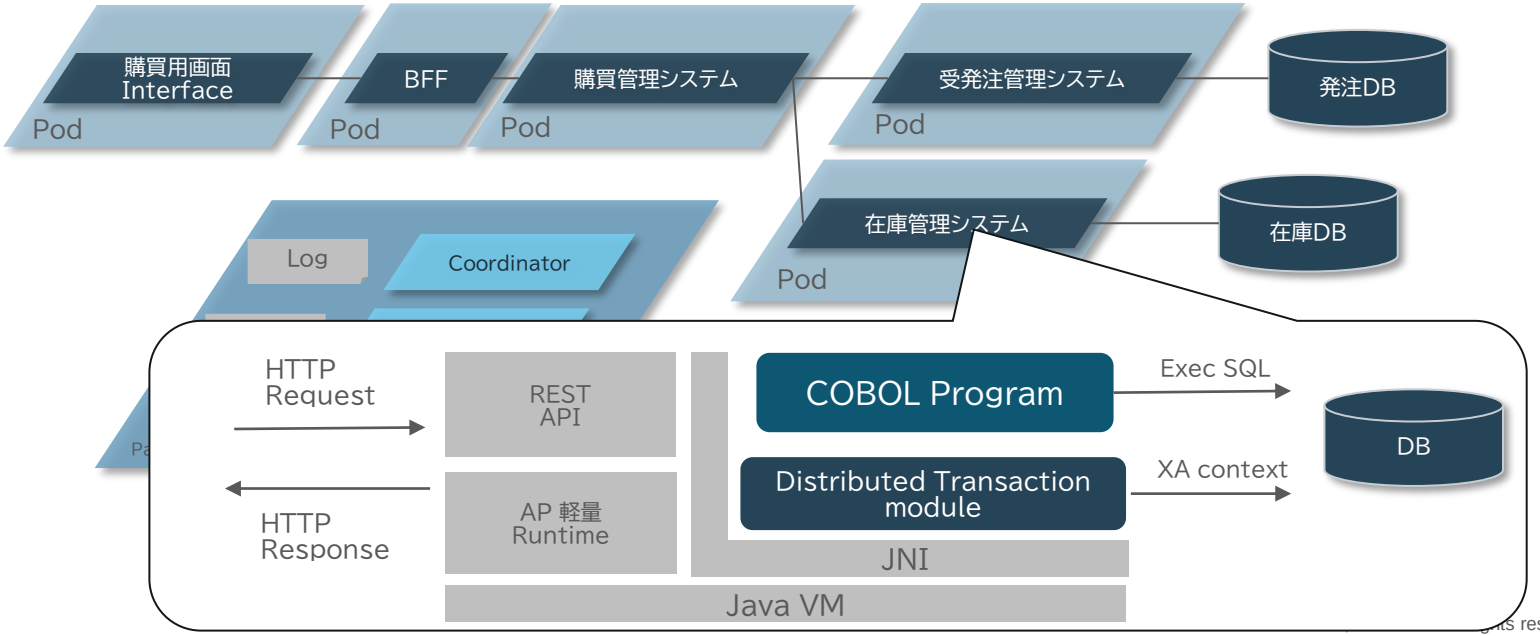
Microservices 分割とKubernetes化

1. 業務の責務境界を単位にMicroservices分割
2. それぞれを Kubernetes Pod に割り当て
3. OpenTP1でマネージしていた分散トランザクションはControl Plane に分離



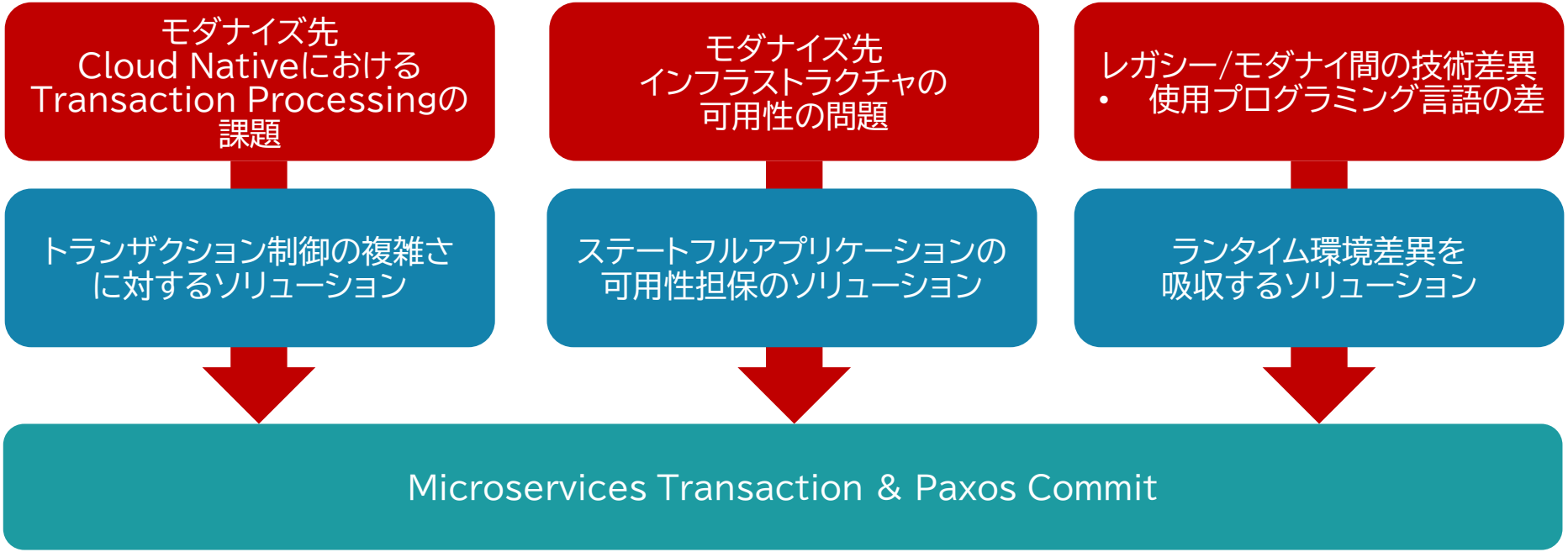
COBOL Microservices の構築

1. OpenTP1で動いていたCOBOLはJNIを通してJava Runtimeに移植
2. OpenTP1で使用していた分散トランザクション用部品もJNI経由で実行するように改修



最後に

今後も日立はリビルドモダナイゼーションの方向性を提案していきます



関連情報

本資料についてのお問い合わせ

■ 株式会社 日立製作所 デジタルプラットフォーム事業部

COBOL拡販担当　E-mail: COBOL2002@itg.hitachi.co.jp

※お問い合わせいただく前に、「個人情報保護に関して(*)」をお読みいただき、記載されている内容に関してご同意いただく必要があります。

ご同意いただけない場合には、お問い合わせに回答できない場合があります。お問い合わせへの個人情報のご提供は、ご本人さまの任意です。

ご提供されない場合は、お問い合わせに回答できない場合があります。

個人情報保護に関して(*)」をよくお読みいただき、ご同意いただける場合のみ、 上記のアドレスに送付ください。
(*)「個人情報保護に関して」:https://www.hitachi.co.jp/utility/privacy/index.html

関連するURL

■ COBOL2002　https://www.hitachi.co.jp/soft/cobol/

■ ITモダナイゼーション ソリューション　https://www.hitachi.co.jp/soft/modernization/

■ ミドルウェア移行支援 ソリューション　https://www.hitachi.co.jp/soft/legacy/


Hitachi Social Innovation is
POWERING GOOD