CTC Engineer's Insight # 5

現場エンジニアの構成判断は、なぜつまずく？

― クラウド・アーキテクチャ改善と設計思考のリアル

インフラも考慮せざるをえない アプリケーション構築のノウハウ

領域間のエアポケット・狭間を埋める視点から

伊藤忠テクノソリューションズ株式会社
金融開発第１部
河口　晃一郎

自己紹介

氏名：河口　晃一郎（かわぐち　こういちろう）

経歴：
1998年からSIerにてシステム開発に従事
2013年にCTCへ中途入社

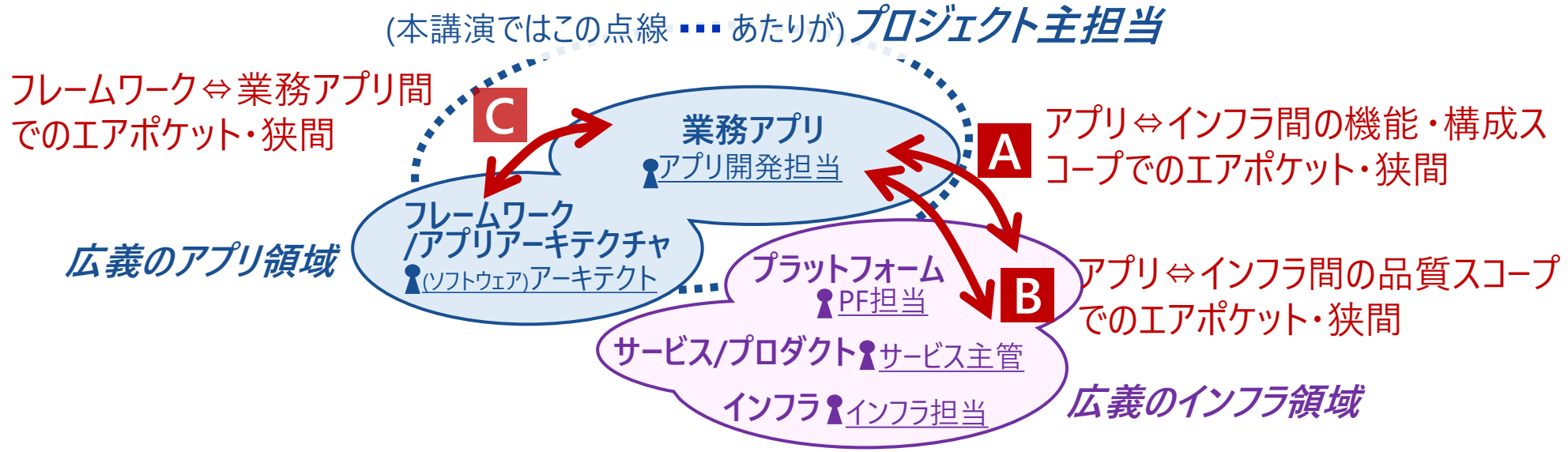
主に金融機関（保険・共済、クレジットカード・信販、FinTech系、銀行）の
基幹システムや周辺システムの構築案件に従事。

アプリケーション領域に重心を置き、PM/PL/方式設計者/アプリ開発者etcとして、
よく言えば 幅広く、 正直に言えば 特定の業務や技術に特化せず にやってきました。

AGENDA

#1-1. 本日、お話しすること

- 「インフラ⇔アプリ間」や「フレームワーク⇔業務アプリ間」の**領域間**で発生してしまう**エアポケット・狭間**について**対策**してきた事例や**気づき・学び**をお話しさせていただきます。



- **エアポケット・狭間**は、**技術的**な面もあるが、**組織・文化的**にもサイロ化して**発生しやすい**。
  ①各領域には専門性があり**各担当**がいる。
  ②インフラ / プラットフォーム / サービスは、アプリとは**構築時期**や**ライフサイクル**が**異なり**、複数のアプリを稼働させるために**基盤**として**汎化**している。
  ③**アプリ領域内**でも、**フレームワーク⇔業務アプリ**では**要件・特性**が**異なり**、アーキテクトとアプリ開発者の**立場・役割の違い**がある。　等々

#1-2. 課題

- サイロ化しやすい要素があるとはいえ、**厳格な計画**を立案し**堅実なシステム**の構築・運用することをなによりも**重んじる金融機関**のシステム開発で**領域間**の**エアポケット・狭間**が**発生**すること**自体**が**おかしいのでは？**

プロジェクト計画/作業項目/WBS

計画、
もちろん
立ててます!!!

役割分担表/RACIチャート

		チームA	チームB	チームC	…
hoge	piyo	◎	△	-	
	…				
foo	bar	△	◎	○	
	…				

役割分担、
しっかり
やっています!!!

▶ では、なぜ
エアポケットが ？

- **そのとおり。**　プロジェクト**開始時**の見積もり・計画段階から、**解像度**が**高く**、**具体的**な作業内容で**役割分担**を**見極められていれば、エアポケット・狭間は生じないはず**。
  なぜなら、全てお見通しで計画通りに粛々と遂行するのだから。
- **しかし現実**には(後知恵で振り返ると)**プロジェクト当初**は・・・

誤読、認識違い	理解が浅い、解像度が低くてサービス仕様を誤読していた
分担の勝手な期待	他の領域担当チームが分担して対応してくれると勝手に期待
経験からの思込み	過去の経験から〇〇〇が準備されていると思い込み、 現実の既存IT環境や運用・制約の見込み違い

そもそも Unknown-Unknown な側面もある

- **本日のお話し**は、プロジェクト**開始時**は**クリアにリアルに見通せておらず**、プロジェクトを**進めていく中で検知・対応してきた事項**についてとなります。

#1-3. 結論サマリ

- 同一前提・超類似案件でもない限り、特に**新技術の採用**や**複数領域に跨るプロジェクト**では、個々要素が単純は足し算や合体しやすいブロックにはならず**もつれ合う**ので、**初期**段階から**クリア**(おおざっぱでも)**に見通す**のは**困難**。

> 「*幸福な家庭はすべて互いに似かよったものであるが、不幸な家庭はそれぞれその不幸のおもむきが違うものである。(Lev Tolstoy『Anna Karenina』藤沼貴 訳,新潮文庫*」と通じるものがあり、
> **うまく**いくには**条件**が**揃**っていなければならず、**うまくいかない**原因は**千差万別**でどの部分にも**潜みうる**。

- その経験からの主な**気づき・学び**を**キーワードでサマリ**すると以下となる。

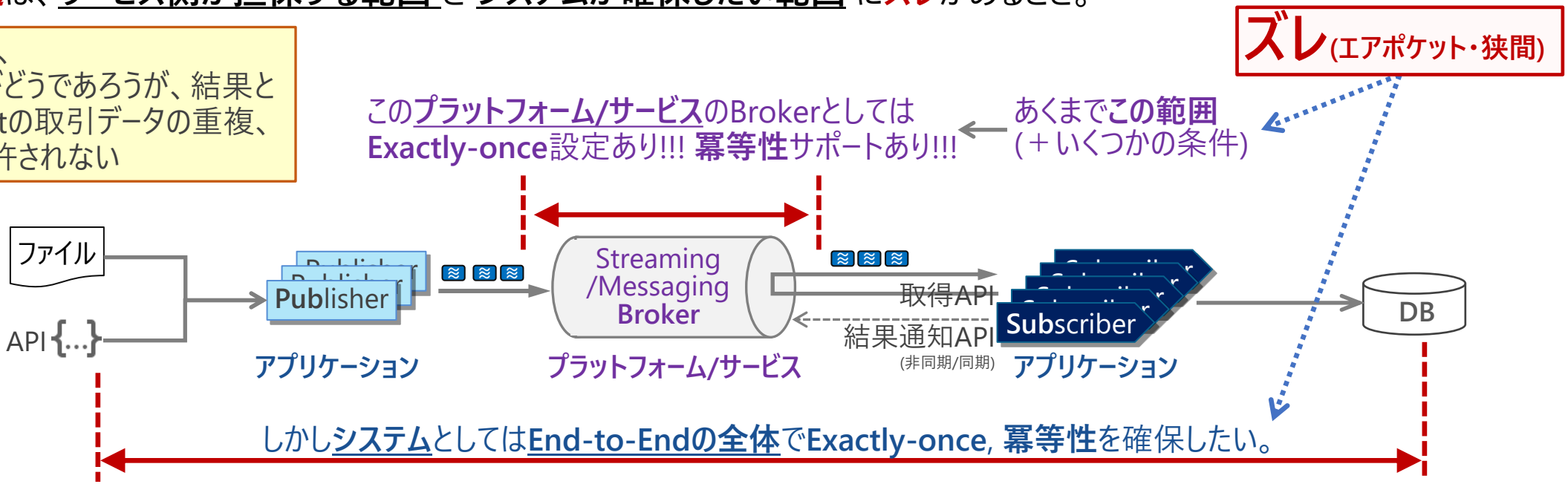
オーナーシップ、自分たちで漕ぐ	解像度のズームイン/ズームアウト	敬意ある領空侵犯	捨てる・畳む
インテグレーション・結合の複雑さ・数に嵌らずコントロール	絞込	段階的	基礎力
ドメイン駆動設計を参考にアプリケーションアーキテクチャ導入の実践からの学び・気づき	同時並行でのプロジェクト/コスト管理	スケール	「なぜこの方式・手法なのか」のクリアな言語化、浸透

赤点線---部は、**いまでも持ち続けている課題感**

このあと、**#2～#4.事例A,B,C)** および **#5.結論)気づき・学び** で **中身**をみていきます。

動機 #2-1. 事例A)アプリとインフラのエアポケット・狭間 - 構成編

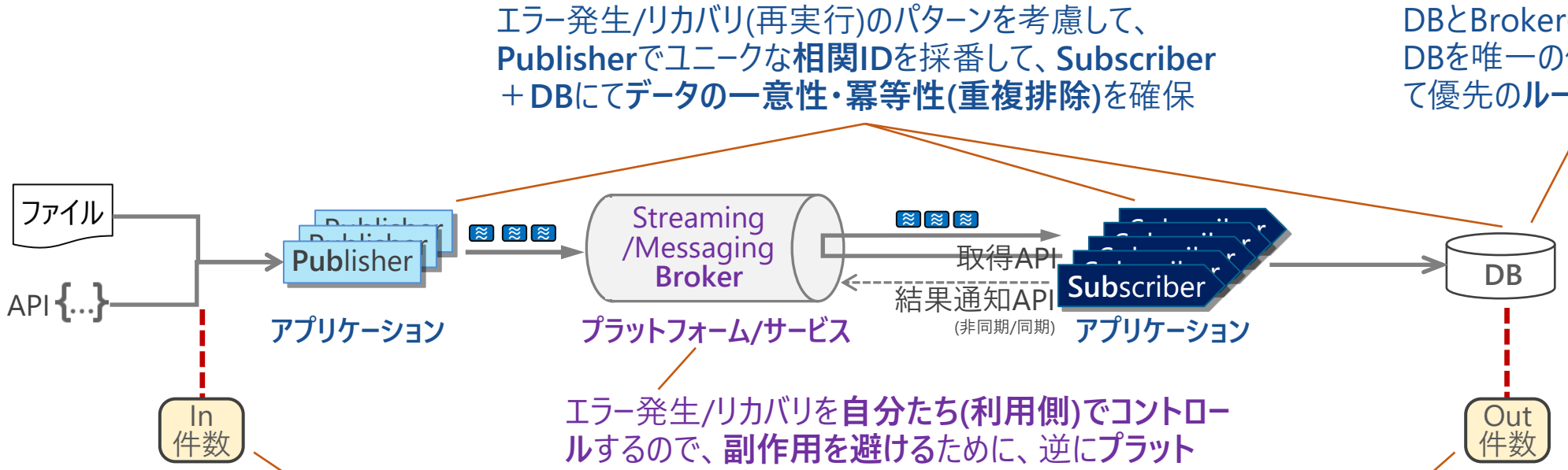
- 金融機関のシステムでは、**取引データ**の**重複**や**ヌケ・モレ**は当たり前だが**絶対許容されない**。
- 業務的なデータ(ログやCDCではなくて、業務ど真ん中の取引データ)の処理に**ストリーミング/メッセージング**を採用する場合、どうしても分散システムのネットワーク/複数ノードの不確実性からくる**データの信頼性**(障害・回復における処理中のデータがどうなるか)が付きまとう。
  ➡多くの**ストリーミング/メッセージング**のサービス (Apache Kafka, Google Cloud Pub/Sub, Amazon SQS(FIFO)など) は**サービス仕様**として「**Exactly-once**(正確にデータは一度だけ処理される)」や「**冪等性**」をサポートをうたっている。一見良さそう。
- **ここで問題**は、**サービス側が担保する範囲** と **システムが確保したい範囲** に**ズレ**があること。



- このズレ(エアポケット・狭間)に**落ち込まないように対応**が必要

対応 #2-2. 事例A)アプリとインフラのエアポケット・狭間 - 構成編

- 個々のサービスだけが保証できれば良いわけではなく、**システム全体の組み合わせで保証**できるようにプロデュース
  ➡以下の主な**対応**を**実施**。

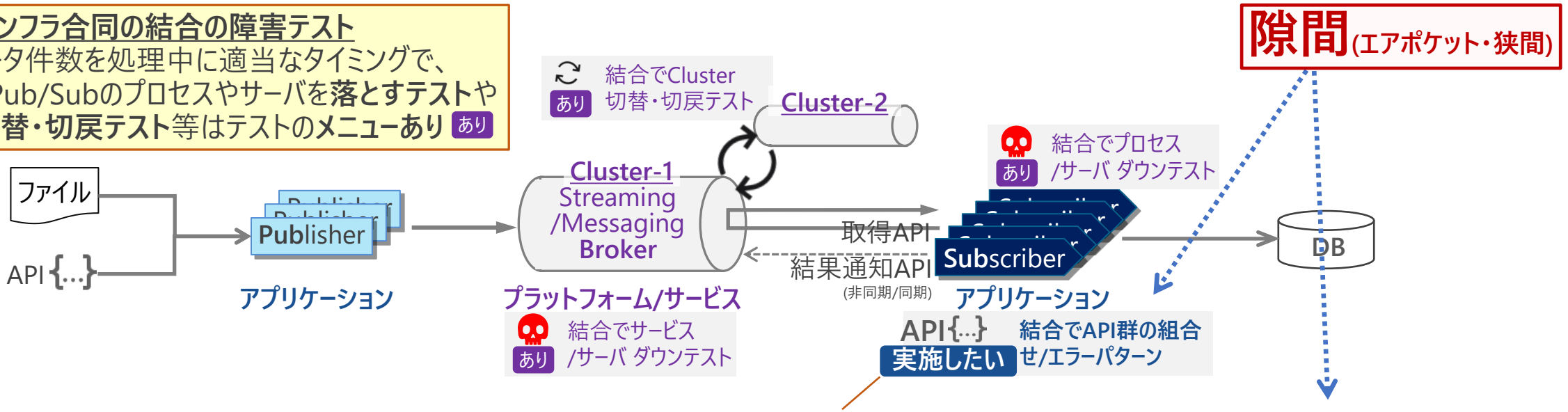


- **アプリ＋プラットフォーム/サービス ＋ DB＋別機能(件数突合せ)**を含めた**システム全体でExactly-once,冪等性**を**確保**した。

動機 #3-1. 事例B)アプリとインフラのエアポケット・狭間 - 品質編

- 金融機関のシステムでは、**障害テスト**(障害を発生させて、リカバリーまで検証)は当たり前に実施する重要なフェーズであり、**計画時**から障害テストは組み込んでおり、**関係者間**で役割分担を含め**総論**としては**齟齬なし**。
- **ここで問題**は、いざテストが近づいてくると、どこまで、どのように品質保証をするかの**具体レベル**での**隙間**があった。

アプリ⇔インフラ合同の結合の障害テスト
大量のデータ件数を処理中に適当なタイミングで、BrokerやPub/Subのプロセスやサーバを**落とすテスト**や**Cluster切替・切戻テスト**等はテストの**メニューあり** あり



しかし、**サービス(Broker)の利用側(アプリ)**としては結合レベルで、アプリからサービスを利用する**API群の組合せ/エラーパターンのレベル**で**障害テスト**を実施し**品質を確保したい**
➡**狙ったAPI呼出しの条件**で**Faultをインジェクトしたい**

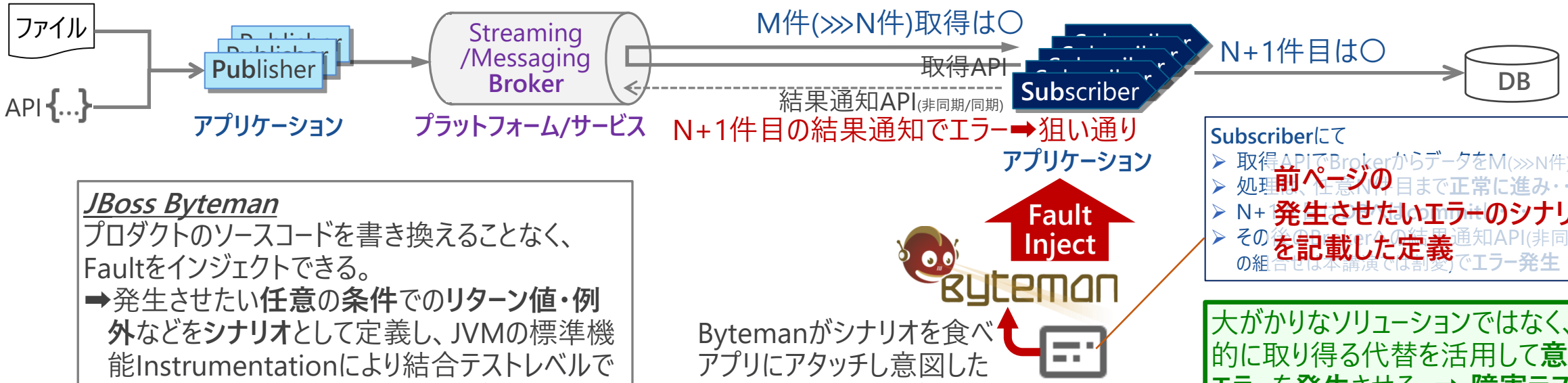
発生させたい**シナリオ**、例えば**Subscriber**にて
- **取得API**で**Broker**からデータを**M件**(>>>N件)取得
- 処理は、任意N件目まで**正常に進み・・・**
- N+1件目は**DBへはcommit**し・・・
- その後の**Broker**への**結果通知API**(非同期/同期の組合せは本講演では割愛)で**エラー発生**

➡このとき、**データの状態、ハンドリング**、システム全体での**リカバリー**が**設計通り**にできるか?

Faultをインジェクトできる機能・サービス
Chaos Mesh® Gremlin なし
等は(アプリも)インフラも提供していない...

対応 #3-2. 事例B)アプリとインフラのエアポケット・狭間 - 品質編

- スタブやモックやDebuggerでのUTレベルではなく、ソースコードを改変することもなく、実環境の結合レベルで検証して、品質保証の**隙間（エアポケット・狭間）を解消**したい。
- サービスの狙ったAPI・タイミングレベルで障害を発生させてデータ状態、エラーハンドリング、リカバリーを検証するために、以下の主な対応を実施。



JBoss Byteman
プロダクトのソースコードを書き換えることなく、Faultをインジェクトできる。
➡発生させたい**任意**の**条件**での**リターン値・例外**などを**シナリオ**として定義し、JVMの標準機能Instrumentationにより結合テストレベルでも動的にアタッチして**Faultをインジェクト**

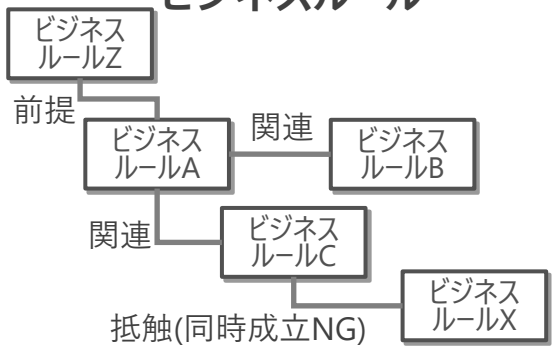
大がかりなソリューションではなく、現実的に取り得る代替を活用して**意図したエラーを発生**させる ➡ **障害テストを突き詰めて実施して品質を確保!!!**

- 障害テストとして、API・タイミングレベルでの**詳細パターン**を、当たり前ことだが**実施・検証**することで**品質の確保**と、ここまで確実に**結合レベルでテスト**を実施したという**安心感**を**開発サイド**にもたらした。

動機 #4-1. 事例C)アプリ内の業務ロジックとフレームワークのエアポケット・狭間

- 保険・共済の加入 / 保全 / 収納/ 給付を担うシステムの**特性** (これは、ほんのいちぶ) として以下がある。

複雑、多数、関連しあうビジネスルール

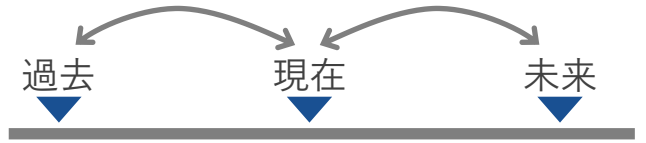


多様な商品(契約)。共通部や派生、新商品の追加や変更

息の長い商品性(契約)

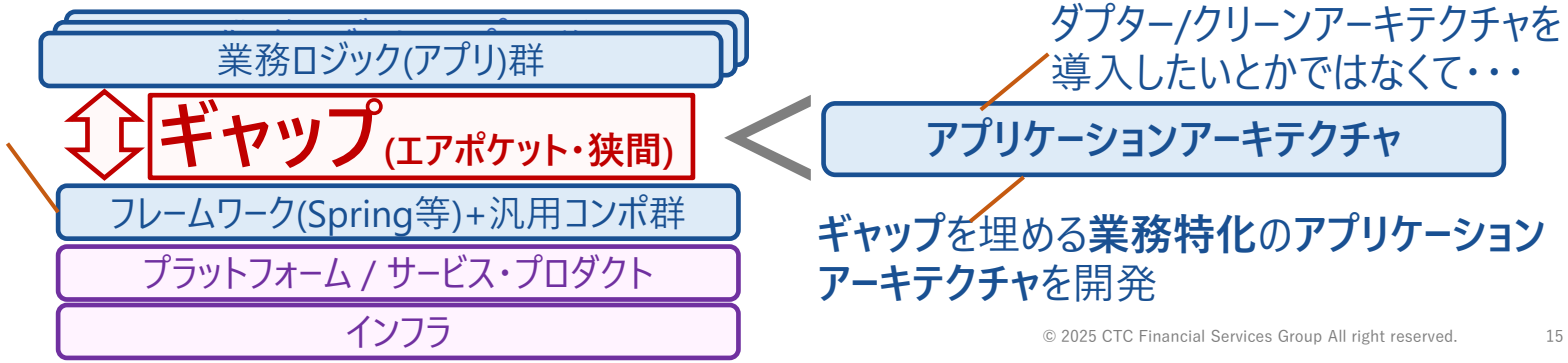
- 被保険者の属性の変更、家族や口数の増減
- 未来日付の契約変更、過去日付の死亡・解約
- 掛金請求や給付のための掛金の履歴管理 … etc

➡**過去・現在・未来**を**行き来**する**時間軸**を意識した保険料・掛金の**異動**などの処理あり



- 上記の**特性**も踏まえ、通常開発の画面・バッチの1画面/1機能ずつに似たような処理をベタベタ実装する**トランザクションスクリプト(手続き型)**での開発では**なく**、**構造化/共通化**して開発・運用の**生産性**や**保守性**を向上する**手法**を**模索**。
- その検討のなかで、通常の**汎用的なフレームワーク**では上記の**特性を構造化/共通化**には**薄い**と考え、**ギャップ**を埋めるように業務特化の**アプリケーションアーキテクチャ**の**開発**するに至った。

SpringやJakarta EE/MicroProfile等の**汎用フレームワーク** ＋ 簡易な文字列,日付,計算処理の共通化を目的とした**汎用的なコンポーネント群だけ**では、**業務処理の構造化/共通化の基盤**としては**薄い**

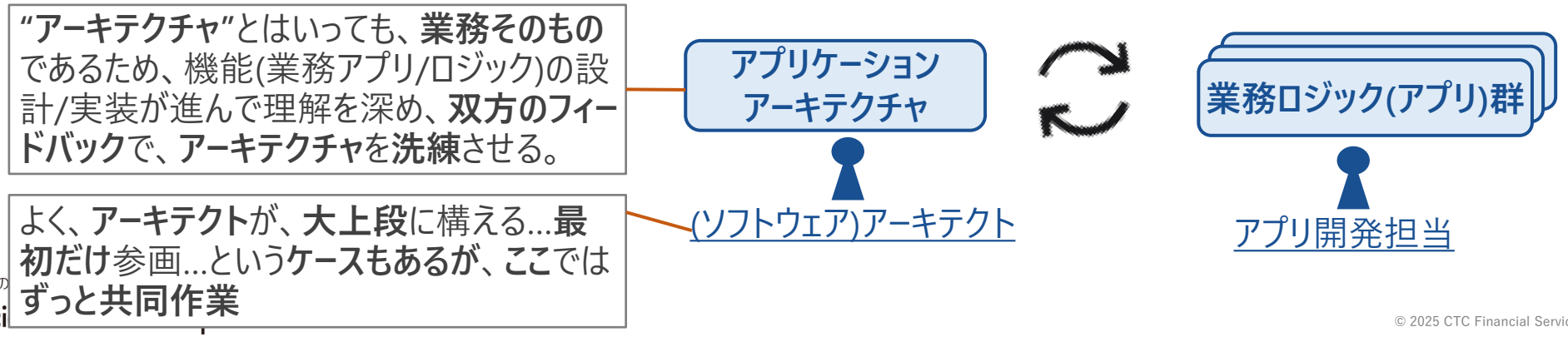


対応 #4-2. 事例C)アプリ内の業務ロジックとフレームワークのエアポケット・狭間

- **アプリケーションアーキテクチャ（および業務ロジック部分も）**としては、**ドメイン駆動設計（DDD）**の考えを**参考**にした。

ドメインモデル (DBではないEntity/Aggregates)	ビジネスロジックのコア部分。契約＋商品群、掛金、請求、給付、左記に含有される規約群・・・ 該当ドメインの個社色なく一般的な簡易イメージ⇒スライド 24
値オブジェクト	プリミティブの直利用を減らし、レンジなどをもった日付系、金額系、会員・契約の属性系、・・・
アプリケーションサービス	バックエンドで複数のドメインモデルを呼び出すユースケース
コントローラ	フロントエンドとバックエンドのつなぎ
その他、DTO/EO、会計など他システム連携、DB操作リポジトリなど	フロントエンドの画面 /他システム連携のIF/インフラのDB等と、ドメインモデルの依存関係を極力減らすことも意識したコンポーネント群（レイヤー間のデータの詰め替えは多め）

- ファーストユーザのお客様は巻き込めきれなかったが、**保険/共済の業務現場**を熟知している**メンバー**がプロジェクト**参画**にしており、**モデリング力/業務理解**を**大きく底上げ**。
- アーキテクトとアプリ開発者のチームとしての**狭間ができないよう**に、**双方フィードバック**による**洗練**や、**アーキテクト**もアプリ部分の**設計・実装**を担い**理解を深める施策**を実施。



対応 #4-3. 事例C)アプリ内の業務ロジックとフレームワークのエアポケット・狭間

- 開発プロセスについては、ビジネスルールをカテゴライズ・ナンバリングして、追跡マトリクス（要件⇔設計⇔実装⇔テスト）でビジネスルール番号で紐付け、追跡性の向上や影響調査の手段とした。

仕様番号	1 No	階層1 仕様名	階層1 内容 2 No	階層2 仕様名	階層2 内容	処理 INPUT	処理 OUTPUT
A9-1	1	調査対象外契約・商品	調査の対象外ある否かを判定・・・				
			【目的】	契約/商品毎に調査が要否があるため、・・・			
A9-1-1			1-1	対象外契約判定	**判定ルール**「M-1.契約・・・	XXX.照会区分	見込額判定対象
A9-1-2			1-2	対象外商品判定	**判定ルール**「M-2.商品・・・	YYY.商品番号	見込額判定対象
A9-2	2	契約_予定額 算出	契約の予定額を、今回申込分予定額＝申込受付時に作成するPIYOスケジュールの初回から12か月分の・・・			「PIYOスケジュール」算出ロジックを使用	契約_予定額
			【理由】	本合算値でXXXの判定を行・・・			

…

要件・設計書
ビジネスルール番号(仕様番号)

ソースコード
ビジネスルール番号(仕様番号)

テスト仕様書
ビジネスルール番号(仕様番号)

ビジネスルールの追跡性を向上

- **DDD原理主義に陥らず**、業務ロジックのない一覧系や帳票出力などは**トランザクションスクリプト**も現実解として**採用。**

業務の**メイン機能群**

DDD(ドメインモデル)で開発

一覧表示や帳票出力etcの**シンプルな機能群**

トランザクションスクリプトで開発

DDD原理主義に陥らず、
完全な統一感よりも経済的効率性を考慮して、前向きな落としどころでの開発

- DDDの考えを多いに参考にしつつも、現実的な手法・施策を取り入れ、業務特化型のアプリケーションアーキテクチャを開発できた。（残念ながら、DSLの導入までは至らず。あと マイクロサービスは やっていません!!!）
- ここで事例C)は結びますが、**関連する課題感**を#5-3にて**後述**します。

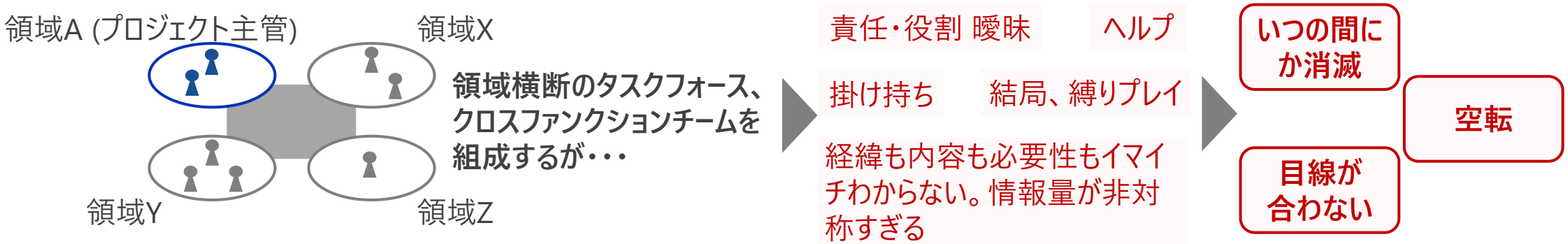
#5-0. 結論)気づき・学び

以下の3軸で、気づき・学び　をみていく。

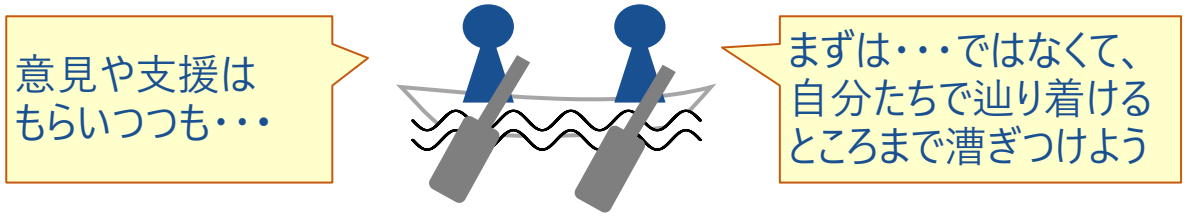
1. 「インフラ/プラットフォーム⇔アプリ領域横断」での気づき・学び
2. 「インテグレーション、コントロール」での気づき・学び
3. 「アプリアーキテクチャ」導入の実践からの気づき・学び

#5-1-1. 結論)「インフラ/プラットフォーム⇔アプリ領域横断」での気づき・学び

- 横断的な**タスクフォース**や**クロスファンクション**チームを組成しても、責任の曖昧さ、ヘルプに入ったというモチベーション、立場・情報の非対称さetcから**形骸化**しやすい。フワフワと途中で**消滅**、**目線が合わない**、双方の時間が**空転**する。



- 課題を解決する**推進力**は、強い**意志**と高い**解像度**を**持つ**、少人数のチーム（**プロジェクトの主管チームのメンバー**、ときには**ひとり**）から生むしかない。
- **役割分担**や諸々の**組織論・チーム論**が提唱される**令和**の時代に**逆行**するが・・・
  意見の拝聴や作業自体の支援を受けつつも、「**救援は来ない**」という**前提**で、**当事者が自らオールを握って漕ぐ**「**単独航海**」**の覚悟**を持つアプローチ。この**オーナーシップ**こそが、課題解決、エアポケット・狭間の**解消**につながる。

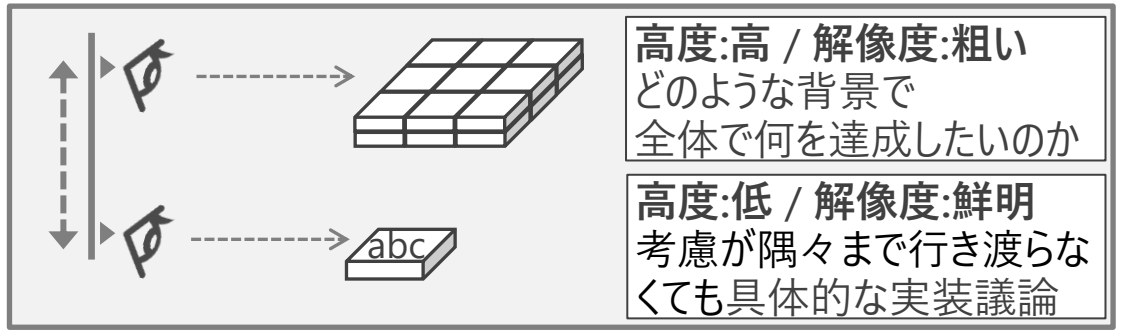


#5-1-2. 結論)「インフラ/プラットフォーム⇔アプリ領域横断」での気づき・学び

- そのためにも、以下が不可欠と考える。

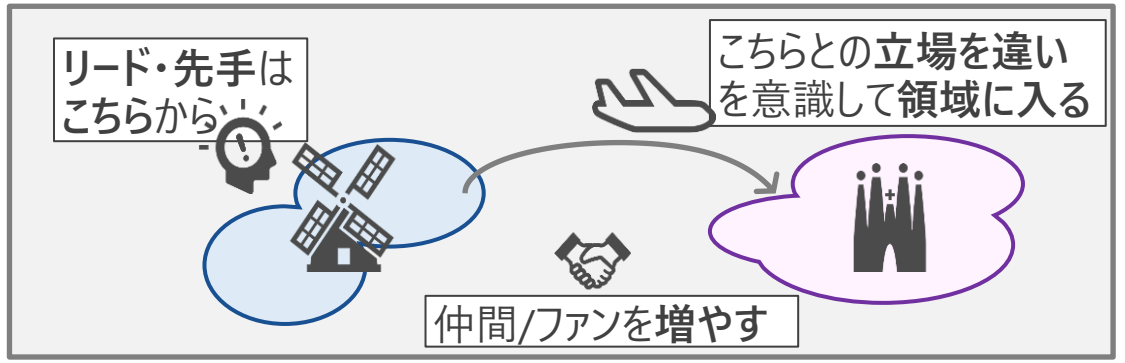
解像度のズームイン／アウト

多方面の**関係者**(上位層も含む)への**説明・説得**だが、**情報量**や掛けられる**時間**に**差異**がある
➡「要求・実現したいこと」から「実装議論」まで、**視点の高度/解像度**を**上げ下げ**して、自分のためにも相手にあわせる柔軟さ



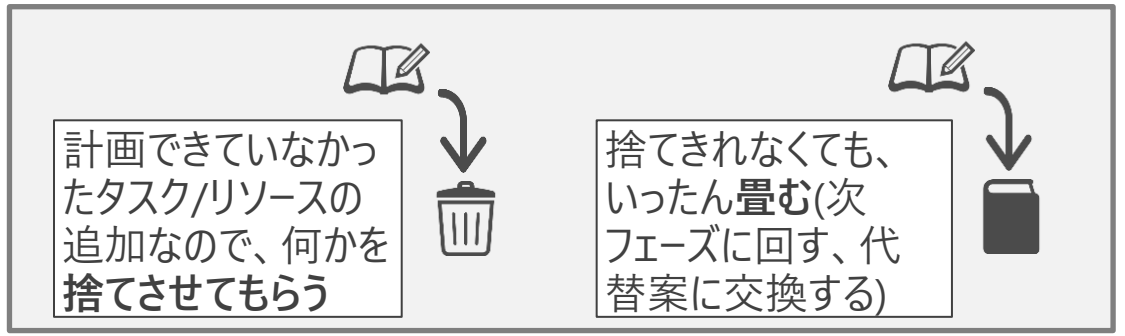
敬意ある領空侵犯

他チームの領域に**踏み込む**際、
「**要求だけ**に終始**しない**」「相手の立場を**尊重**する」「完璧案ではなくても**事前準備・対処案**を作成・持参する」で**リード・先手**はこちらから出す作法。➡**仲間**が増えて、支援ではなく **当事者**に**切り替わって**いってくれる（たぶん）。



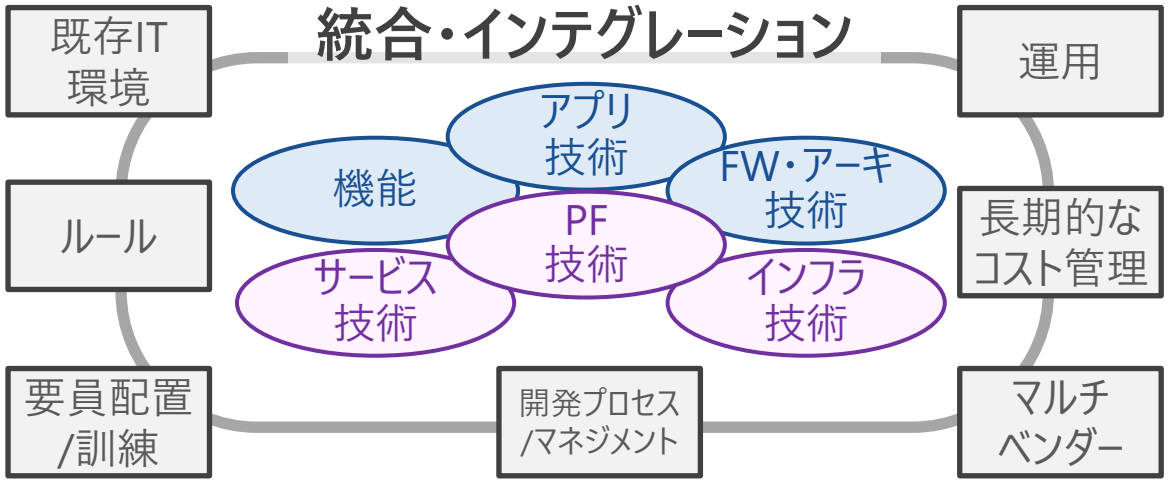
捨てる・畳む

本講演では、当初計画からは残念ながら想定外の状況を取り上げている。**無理**は**禁物(延焼**する)なので、プラスマイナス0にはできなくても「**やらないこと**」**(捨てる)** 、「**まずは着地させること**」**(畳む)** をステークホルダーと握る**調整**。（困ったことに、これも当事者として実行する必要あり）



#5-2-1. 結論)「インテグレーション、コントロール」での気づき・学び

- 近年、**クラウド / コンテナ / 機能特化のOSS**などの技術は、**個々のサービス**や**特定の組み合わせ**では、調達の効率、俊敏さ、エラスティック、マネージドといった**強力なメリット**をもたらした。
- しかし、**本当の課題**は、個々の優れたサービスだけではなく、**既存IT環境**とも**組み合わせ**、**品質**を担保し、**運用**に乗せる**インテグレーションという重責は、すべて我々自身が負わなければならない**こと。➡そこが**一番のノウハウ**になる



実現したい要件や技術以外の**周辺**の多くの事項も含めて、**統合・インテグレーション**しないといけない

- これはシステム開発に限ったことではない。アメリカ海軍の新型空母「ジェラルド・R・フォード」級の**開発の事例**を見てみる。23の新規技術を一度に投入したこともあり、個々の**技術**および**インテグレーション・運用開始**に**失敗**。計画は大幅な**遅延**(4年以上)、莫大なコスト**超過**(28億ドル)を招いた。
  ➡アメリカ海軍のように空母を設計・製造・運用の**実績の豊富な組織**であっても、プロジェクトの**「コントロール」**を**失った**。
  ➡その後、アメリカ海軍は、**今後の新造艦**では新規技術の採用を数個に**絞る**という**方針転換**を余儀なくされた。

出典: https://www.popularmechanics.com/military/navy-ships/a37093943/uss-gerald-ford-aircraft-carrier-problems/

#5-2-2. 結論)「インテグレーション、コントロール」での気づき・学び

- この端的な**教訓**、および前述 #5-1の「当事者達が自らオールを握って漕ぐ・・・」から、**私たち**の扱える**複雑さ・数**には**限界**がある。
- 「**複雑さ・数の沼**」に**嵌らず**、**コントロール**を**維持**するための**気づき・学び**は、身も蓋もない話になるが、以下となる。

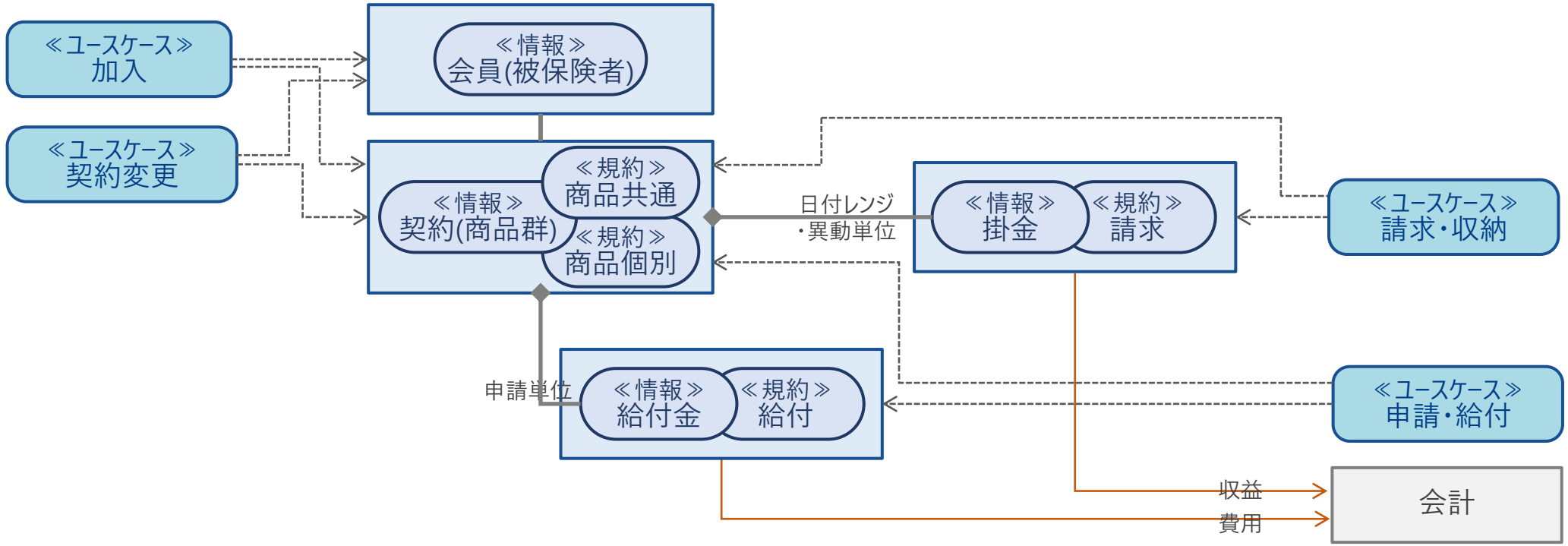
一度に導入する新規技術の数を、コントロール可能な範囲（数個）に意図的に絞り込む。

既存IT環境とのインピーダンスミスマッチを見極め、決して一気に行わず、段階的に導入を進める。

最新技術も土台は「基礎」（データ処理、ネットワーク、分散技術、ソフトウェアアーキテクチャ、設計/開発方法論、etc）の組み合わせであるため、遠回りでも、いまから (問題に直面したその時点から) でも、これらを学習する。

#5-3-1. 結論)「アプリアーキテクチャ」導入の実践からの気づき・学び

- **アプリケーションアーキテクチャ**として**DDD**の考えを参考に、汎用フレームワーク と 業務ロジック(アプリ) の間に存在する「ギャップ(エアポケット・狭間)」を**埋める**こと、**より良い**モノと考えて**システム**の構築を実践。
  ➡DDDを完全に咀嚼して自分達のものにできたか・・・は難しいところだが、**業務に根差した**モデルの具現化を**試行錯誤**しながらも実施できた、と考えている。（下図は、該当ドメインの、個社色なく一般的な簡易イメージです）



- 技術以上に、**幸運**にも**開発チーム**に**業務精通者**(元業務現場 and 火力 高め)が**在籍**していたという**人的要因**に**支えられた**面が強い。

#5-3-2. 結論)「アプリアーキテクチャ」導入の実践からの気づき・学び

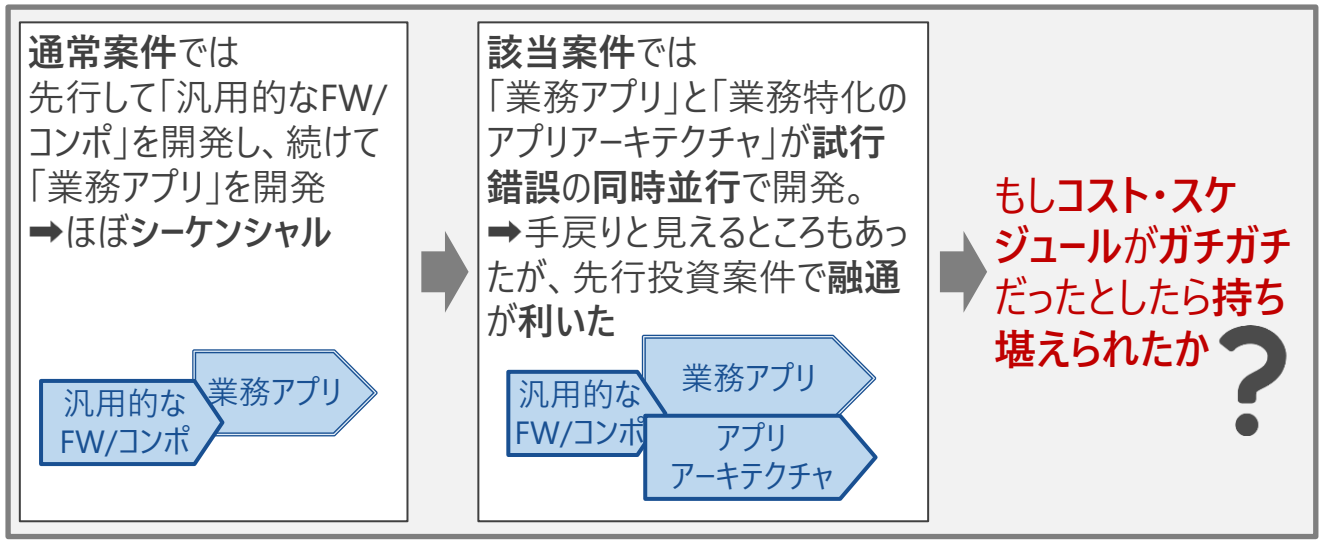
- 一方で、**実践**から以下の**課題**が**浮き彫り**となった。　1/3, 2/3

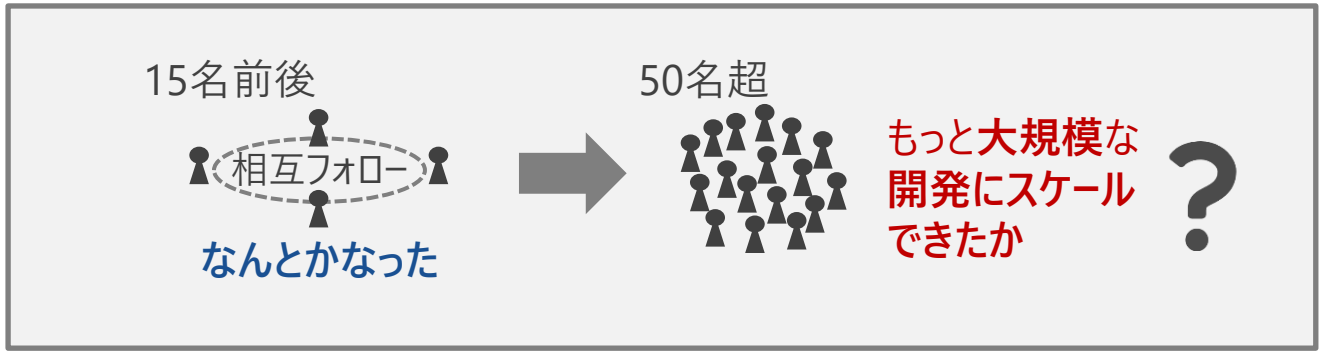
同時並行でのプロジェクト進捗/コスト管理

アプリケーション**アーキテクチャ**を、**業務アプリ**とほぼ**同時並行/フィードバック**を受けながらの開発で**試行錯誤**（良く言えば"洗練" ⇔ 管理者からすると"手戻り"）
➡該当案件は先行投資の色合いが強かったので融通が利いたが、**もしコストや作業計画**が**計画通り**の**ガチガチ案件・環境**では**難航**（特に調整事や説得）、と想定される。

スケール

DDDを十分に取得・体現できたかは別としてもプロジェクトが体制が15名前後のチームで**相互フォロー**することができた。しかし、**もし**大規模案件に**スケール**(50名超)する際は、**進め方**を相当**工夫**しないと、**考え方/設計/実装**の**方向性・均質性**が**混沌**として**中途半端**になる、と想定される。





#5-3-3. 結論)「アプリアーキテクチャ」導入の実践からの気づき・学び

- 一方で、**実践**から以下の**課題**が**浮き彫り**となった。　3/3

「なぜこの方式・手法なのか」のクリアな言語化、浸透

プロジェクトの**最終**局面でも、**コアメンバー**から「従来の方式通りでもよいのでは」「何をもって正解とするのかグレー」など**正直**な**リアクション**があった

➡

- 相応の理由や必然性の言語化の弱さ
- 合意形成の不十分
- ・・・

DDDに限定したことを言いたいのではなく、性能面やコスト面etcで明確な競争力の高いものでもない限り、（該当環境において）**新たな**方式や手法を**採用**する際には**ついてまわるテーマ**かと思います。

➡

特に、『「**なぜこの方式・手法なのか**」というテーゼに対する、相応の**理由**や**必然性**の**言語化の弱さ**』は、何年経過してもずっと自分の頭から離れず思案していますが、**答えは見つけられていません。**

コアメンバーから「設計書もソースコードも**飛び飛び**」「**まどろっこしい**」「何をもって**正解**とするのか**グレー**」「もっとシンプルに**共通化だけ**で良いでは」云々の正直な リアクション あり	●「**なぜこの方式・手法なのか**」というテーゼに対する、**相応の理由**や**必然性の言語化**の**弱さ**	●納得感・腹落ちする**合意形成**の**不十分**
	●育てていくロードマップの**途上**での**スッキリしない**感	●ビジネスなのだから経済的なバリューが第一ではあるが、**効果測定**が**欠如**
	●「ビジネスルールの追跡性」だけではなくて、「**データの追跡性**」も**欲しかった**なぁ…	……。振り返るといろいろある。

#5-4. 最後に

最後は、
反省色の強い**振り返り**での**締めくくり**になってしまいましたが、
本講演は**以上**となります。

ご清聴、ありがとうございました。

無限の未来と、
幾千のテクノロジーをつなぐ。

CTC Financial Services Group

CTC
Challenging Tomorrow's Changes