83

エンタープライズシステムへの マイクロサービスアーキテクチャー適用の実践[1]

1. 概要

クラウドの普及によりシステムのあり方や開発スタイルは大きく変化した。従来のシステム開発は、全ての要件や仕様を確定した後に、長い期間をかけてシステムを開発するウォーターフォール型が主流である。ウォーターフォール型は、仕様や設計の変更が発生すると上流工程への後戻りコストが高く、変更を受け入れ難い開発方法である。また、開発したシステムも、長期間変更されることなく、利用され続けることが想定される。

クラウドは必要なときに必要なプラットフォームを提供でき、不要になったら破棄することができる環境である。変化に対して柔軟な対応ができるクラウドにより、変化を許容するシステム開発が行われるようになっている。開発手法はウォーターフォール型ではなくアジャイル開発が採用され、開発されるシステムはマイクロサービスアーキテクチャーの構成をとり、短い期間で機能変更や追加が行われる事例が出ている。

しかし、事例の多くは SNS(Social Networking Service)等のネットサービスやエンターテイメント向けのシステムであり、企業向けのエンタープライズシステムのものはまだ少ない。本稿では、エンタープライズシステムでは例が少ないマイクロサービスアーキテクチャーを当社のシステム開発で実践した結果と、得られた知見や留意点を紹介する。

2. 事例プロジェクト「SPINEX™」

東芝では IoT(Internet of Things)を活用したデジタルトランスフォーメーションを推進し、顧客のビジネス成長を実現するため、東芝 IoT アーキテクチャー「SPINEX™」を提供している。SPINEX は「エッジコンピューティング」「デジタルツイン」「アナリティックス AI/コミュニケーション AI」という 3 つの特長を持つ。

- エッジコンピューティング

  現場(エッジ)でのリアルタイム処理とクラウド処理を最適に協調させる仕組み。東芝の半導体・組込ソフトウェア技術により現場のデータを素早く判断し、最適な処理を実現する。

- デジタルツイン

  デジタル上に現場環境を忠実に、リアルタイムに再現することで、機器の状況を正確に見守り、いち早く兆候を捉えて改善、新たなサービスへとつなげる。

---

[1] 事例提供：東芝デジタルソリューションズ株式会社 斉藤 稔 氏

● アナリティクス AI/コミュニケーション AI

ものづくりの知見で磨かれた分析技術と AI(人工知能)を融合して識別、予測、要因推定、異常検知、故障予兆検知、最適化、自動制御に活用。また音声認識・画像認識技術と AI 技術を融合して音声や映像などの情報を解析し、人の意図や現場の状況までを理解する。

図 83- 1 東芝 IoT アーキテクチャー SPINEX™

SPINEX の基本構成要素の 1 つに、機器管理や機器の見える化を実現する標準アプリケーションがある。標準アプリケーションが IoT を実現するために必要な要件は次の通りである。以降、SPINEX の標準アプリケーションを「対象アプリケーション」と記載する。

(1) 変更容易性

2 種類の変更を想定している。 1 つ目は、東芝としてサービスの拡大を進めるため、対象アプリケーション自体の機能拡充や変更である。2 つめは、顧客毎に行うカスタマイズである。顧客の要望や環境に柔軟に対応できることは、顧客にとってもサービスを提供する当社にとっても重要な要件である。

(2) スピード

IoT では、機器のセンシングデータを活用して改善をしたいという要望はあるが、具体的な要件が明確になっているとは限らない。仮説をたて、実際の機器のセンシングデータを収集/分析し検証を繰り返して問題や要件を明確化してく PoC(Proof of Concept)という手法がとられることが多い。PoC を行うためには、センシングデータの収集/分析/検証を行うためのシステムを、短期間で繰り返してリリースできる必要がある。

(3) 拡張性(スケール)

初期導入は投資を抑えて小規模で開始して、効果を確認後に適用範囲を拡大していくというアプローチが取られることが想定される。適用範囲の拡大に合わせてシステムが拡張できる必要がある。

(4) 可用性

さまざまな機器がつながり、多く人が利用するサービスとして提供することを想定し、24 時間 365 日止まらない稼動が必要である。

3. マイクロサービスアーキテクチャー

3.1. マイクロサービスアーキテクチャーとは

マイクロサービスアーキテクチャーは、小さなマイクロサービスを組み合わせて構築されるアーキテクチャーである。個々のマイクロサービスは独立して起動/停止が可能なものであり、HTTP やメッセージング機構により通信を行う。データモデルはマイクロサービス毎に管理され、原則的にデータモデルはマイクロサービス間で共有せず独立している。

マイクロサービスアーキテクチャーに対して、1 つのアプリケーションに全ての機能が詰め込まれたシステムの構成をモノリシックと呼ぶ。図 83- 2 はモノリシックとマイクロサービスアーキテクチャーの違いを図示したものである。

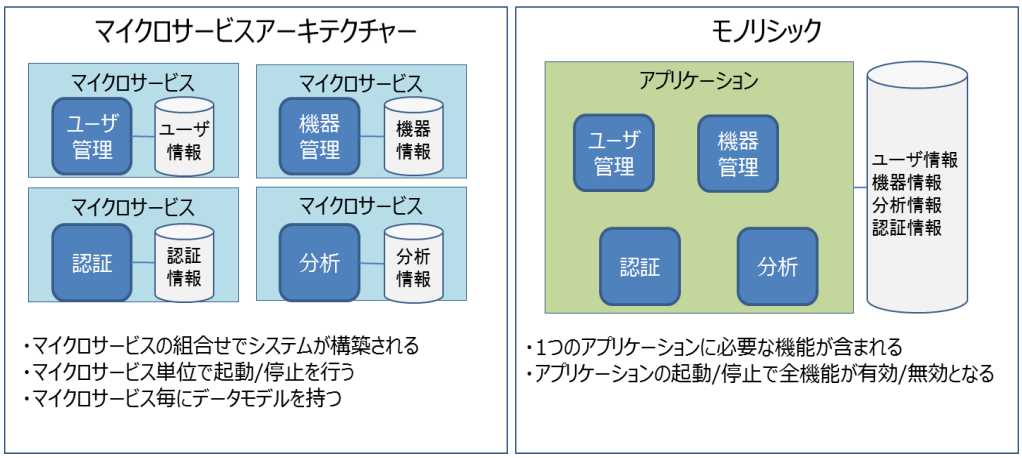

図 83- 2　マイクロサービスアーキテクチャーとモノリシックの比較

マイクロサービスアーキテクチャーは独立性の高いマイクロサービスを組み合わせた構成のため、モノリシックに比べて変更容易性と拡張性が高いといわれる。変更容易性についてモノリシックとの比較を図 83- 3 に、拡張性について図 83- 4 に示す。

一部の機能に変更が発生した場合、モノリシックでは変更していない機能も含めてアプリケーションの作り直しになる。そのため変更していない機能に対するテストや確認も行う必要が

ある。また拡張(スケールアウト)する場合も、拡張不要な機能も含めて拡張することになり不必要にリソースを必要とする。モノリシックに対してマイクロサービスアーキテクチャーでは、変更や拡張をマイクロサービス単位で行えるので無駄無く効率的に実現可能である。

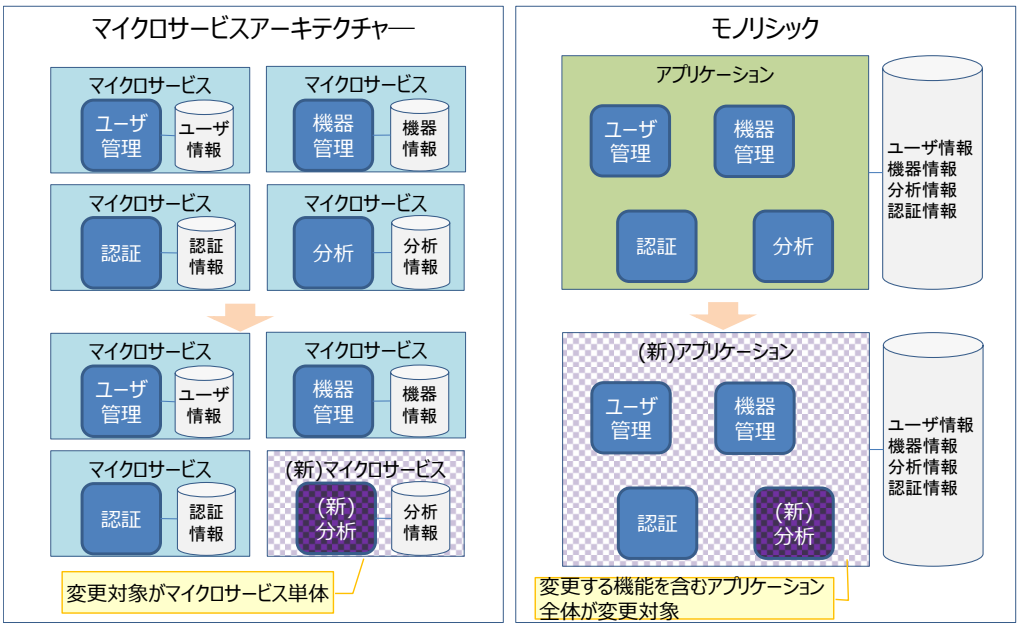

図 83- 3 変更容易性の比較

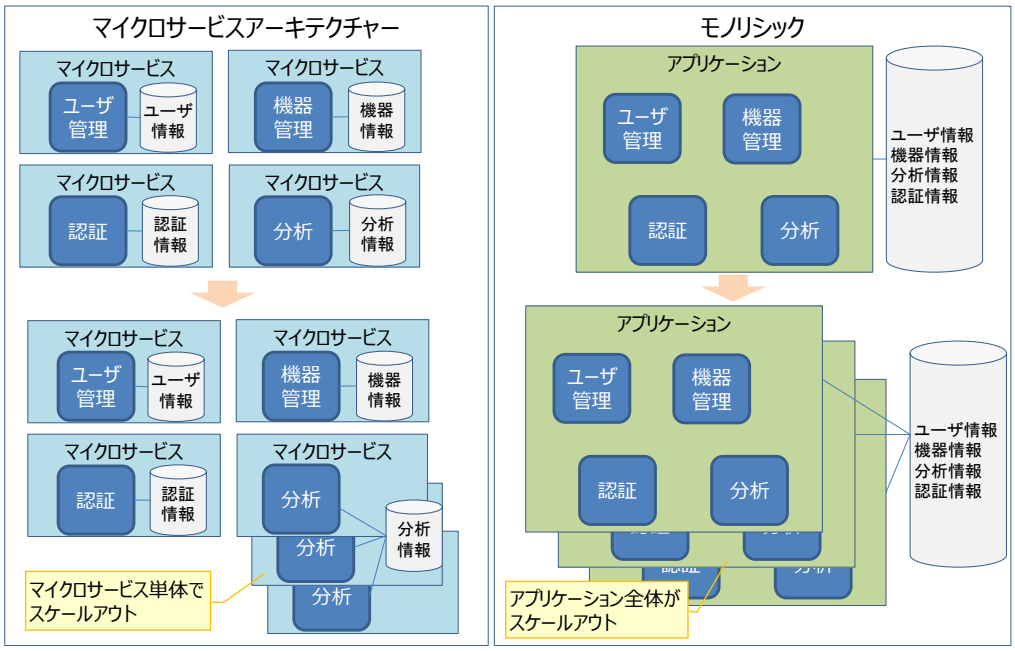

図 83- 4 拡張性の比較

もちろんマイクロサービスアーキテクチャーはメリットだけでなく、デメリットもある。メリットとデメリットを表 83- 1 にまとめる。

変更や拡張容易性以外のメリットとしては、技術の多様性が上げられる。個々のマイクロサービスは HTTP やメッセージングにより連携できるインタフェースを備えていればよく、具体的な実装技術の制約は低い。実現したい機能に適した技術の選択や、開発チームが得意な技術の選択が可能である。

デメリットは、複数の小さなマイクロサービスの組合せでシステムが構成されるため、システムが複雑化することである。構成が複雑化することで通信のオーバーヘッドによる性能の低下、障害点の増加、運用の複雑化がおきる。またマイクロサービス単体での変更/追加は容易に行えるが、システム全体としての変更/追加による影響範囲の確認などは必要になる。

表 83- 1 マイクロサービスアーキテクチャーのメリット／デメリット

メリット	デメリット
● 強固なモジュール境界 部分的な変更/追加がし易い ● 個別デプロイ可能 継続的デリバリがし易くなり、サイクルタイムを短縮可能 ● 技術の多様性 実現したい機能・サービスに適した言語、ライブラリ、ミドルウェア、ツールを選択可能	● 性能 分散したサービスの呼出が多くなり性能に影響する ● 障害点の増加 リモート呼出になることで障害が発生しえる箇所が増加する ● 整合性の実現 個別サービスの変更がシステム全体としてどう影響するかの考慮が必要 ● 運用の複雑化 管理対象の増加、サービス間の関係の複雑さの増加

3.2. アーキテクチャーと開発手法

本開発では、表 83- 1 に示したメリットにより対象アプリケーションへの要求を満たせると判断し、マイクロサービスアーキテクチャーの採用を決定した。また開発は、マイクロサービスアーキテクチャーに適したアジャイル開発の 1 つであるスクラムにより進めることとした。

4. 課題

対象アプリケーションの開発チームには、Java や Web 技術に対する高い技術力を有するが、マイクロサービスアーキテクチャーやアジャイル開発の経験がないメンバがアサインされた。

アジャイル開発については、社内のアジャイル開発支援メンバからサポートを得られたが、マイクロサービスアーキテクチャーついては、社内でもノウハウが少ない状態であったため、マイクロサービスアーキテクチャー自体の調査から行った。調査した結果、開発チームとして解決しなければならない表 83- 2 に示す３つの課題を洗い出した。

マイクロサービスアーキテクチャーは”小さな”マイクロサービスの組合せにより構成されるが、そのマイクロサービスの粒度に定量的な指針はない。対象アプリケーションにとって適切な粒度を決めることを課題とした。

マイクロサービスの粒度と合わせて、マイクロサービスの構成を課題とした。構成とはマイクロサービスの組み合わせ方であり、単純な構成であれば、全てのマイクロサービスが一律フラットに配置される。また、多段構成にして階層毎に役割を持たせるという考え方もある。構成を検討する観点として、性能とシステムのカスタマイズ性に着目した。マイクロサービスアーキテクチャーは、変更/追加が容易な特徴があるが、変更/追加のルールは必要であり、ルールは文書や規約だけでなくマイクロサービスの構成で明確に示すべきと考えた。

開発手法としてアジャイル開発を採用したことにより、マイクロサービスを順次開発してシステムを組み合わせていくことになるのだが、マイクロサービスアーキテクチャーの経験がない開発チームでは、どのような順で開発していくべきかを課題として検討した。

表 83- 2 課題

課題 1	**マイクロサービスの粒度** マイクロサービスは小さすぎると構成が複雑化し、大きすぎると変更容易性や拡張性が低下する。適切な粒度をどのように定義したらよいか。
課題 2	**マイクロサービスの構成** マイクロサービス間の通信数増加による性能問題を抑制するための構成とは。 システムの可変箇所と固定箇所を構成により明確に定義するにはどうすればよいか。
課題 3	**開発順序** 初めて行うマイクロサービスアーキテクチャー開発において、効率的に且つリスクを最小限にするためには、どのような順序で開発すればよいか。

5. 課題の検討結果

表 83- 2 の課題を解決するために検討した結果を示す。

5.1. (課題 1) マイクロサービスの粒度

マイクロサービスの粒度の決め方として、表 83- 3 に示す観点が一般的に示されている。

表 83- 3 マイクロサービスの粒度を決める観点

- **サービスのライフサイクルの観点**

  マイクロサービスが変更されるサイクルの観点である。長期的に変わらない機能と、ニーズによって短期的に変更がある機能は 1 つのマイクロサービスにせず、別のマイクロサービスとして定義すべきである。

- **スケーラビリティの観点**

  拡張(スケール)するタイミングの観点である。システムに対する負荷が増加した場合にマイクロサービスを拡張する。負荷の増加の要因には、利用者数の増加や月末の集中アクセスのような時期的なものがある。拡張のタイミングが異なる機能は同じマイクロサービスにすべきではない。

- **組織の観点**

  サービスを提供/保守する組織やグループに対応させて粒度を決めるという観点である。マイクロサービスの変更や問題があった場合に、他の組織やグループの業務に影響を与えないようにすべきである。

ライフサイクルやスケーラビリティの観点は、システム利用者のニーズや状況を考慮して粒度を決めるアプローチである。組織の観点は、システム提供者側からの指定で粒度を決めるアプローチである。

対象アプリケーションの場合、サービスを提供する組織は同一であったため、システム利用者を考慮したアプローチを取り、「利用される単位」という粒度を決定した。図 83- 5 に示すように対象アプリケーションの想定されるアクターと機能を整理して、アクターが利用するタイミングや利用するアクターの種類から「利用される単位」を定義し、その単位でマイクロサービスを開発した。

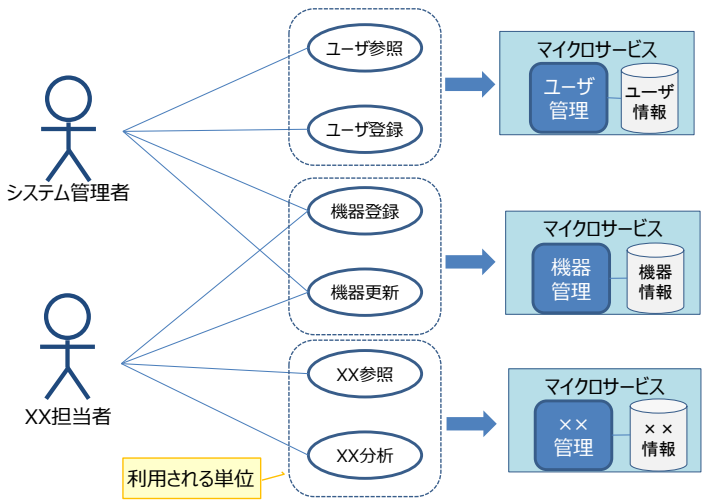

図 83- 5 マイクロサービスの粒度

5.2. (課題２) マイクロサービスの構成

マイクロサービスの組み合せ方は、システムの性能や顧客毎のカスタマイズ性に影響すると考えた。

複数のマイクロサービスを組み合わせるため、モノリシックに比べて通信数が増えることは必然である。対象アプリケーションはクラウド上に配置する予定であったため、システムの性能に着目すると、特にインターネットを介する通信の増加は影響が大きい。インターネットを介した通信の起点になるのは画面のため、画面からの通信数を最小限にすることが、全体の性能向上に寄与すると考えた。

カスタマイズ性に関しては、まずカスタマイズの定義を決めることから行った。対象アプリケーションでは 2 種類の変更が想定され、1 つは対象アプリケーション自体の機能変更や追加のための変更で、２つ目は対象アプリケーションをベースに顧客の要件に合わせて行う変更である。後者の顧客の要件に合わせた変更をカスタマイズと定義した。カスタマイズにはルールや規約が必要であり、カスタマイズできない部分とできる部分を明確に定義することが保守性や機能拡張には重要である。ルールとして対象アプリケーションの機能はカスタマイズできない部分とし、機能を使って提供する画面や業務のシナリオはカスタマイズできる部分と定義した。

性能やカスタマイズ性を検討した結果、機能を提供するマイクロサービスと業務シナリオを提供するマイクロサービスの２段構成とした。業務シナリオを提供するマイクロサービスは画面からのリクエストに応じて機能を提供するマイクロサービスを順次呼び分ける役割を担い、プロジェクトでは UI サービスと命名した。UI サービスには、マイクロサービスアーキテクチャーのデザインパターンの１つである API Gateway の要素も取り込み、表 83- 4 に示す効果を出すことを狙った。

表 83- 4 UI サービスに期待した効果

- **画面とサーバ間の通信量削減**
  画面から直接マイクロサービスを実行する場合、1 つの業務を行うためにマイクロサービスの呼び出しを複数回行う可能性があるが、画面からのリクエストを一旦 UI サービスで受信して、UI サービスがマイクロサービスを複数回呼び出す構成をとることでインターネットを経由する通信を削減する。
- **マイクロサービスの資産化とカスタマイズ箇所の明確化**
  顧客毎のカスタマイズは、画面やマイクロサービスの利用方法に反映されると想定し、画面からのリクエストに応じてマイクロサービスを実行する UI サービスをカスタマイズ箇所として配置する。UI サービスから呼び出されるマイクロサービスは変更不可の資産と定義した。
- **マイクロサービスのロケーション仮想化**
  マイクロサービスの変更や拡張等によりマイクロサービスの配置や数が変ったとして

も、UI サービスで吸収する。顧客が操作する画面はマイクロサービスの変更や拡張を考慮することなく常に UI サービスだけを通信相手とするため、マイクロサービス側の変更が画面に影響されず、結果として顧客への影響が少なくなる。

- **セキュリティ対策の局所化**
  全てのアクセスが UI サービスを経由するため、認証/認可等のセキュリティ対策を UI サービスで局所的に行えばよく、個々のマイクロサービス開発でセキュリティを考慮する必要がなくなる。

マイクロサービスアーキテクチャーの構成の概略を図 83- 6 に示す。インターネットを経由する画面からのリクエストは UI サービスで受信しマイクロサービスの呼び出しを行う。認証機能等のセキュリティ機能は UI サービスに対して横断的に適用してバックエンドのマイクロサービスへの不正なアクセスを遮断する。顧客毎のカスタマイズは画面と UI サービスに対して行い、マイクロサービスには顧客独自の要件は混入させずに、対象アプリケーションに共通の機能を提供する。

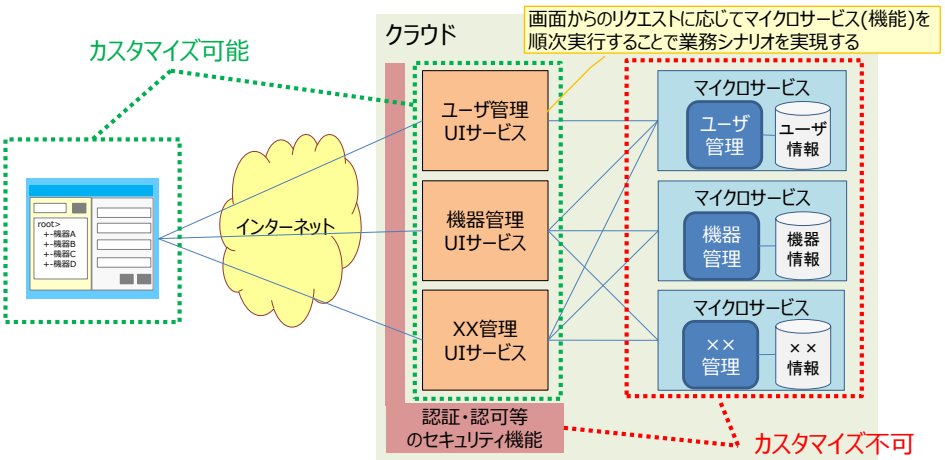

図 83- 6 マイクロサービスの構成

5.3. (課題 3) 開発順序

アジャイル開発のスクラムの理想では、開発チームが抱えるタスクを開発メンバの誰でも処理できることである。しかし、実際は全ての種類のタスクを処理できるような人材を集めることは困難である。本開発でも、開発メンバには得意分野と不得意分野がありスキルも様々であった。そのため開発チーム内を開発技術に合わせて、マイクロサービスを担当するチームと画面周りを担当するチームに分けた。

また、初めて行うマイクロサービスアーキテクチャーのリスクを最小化する目的で、アプリケーションの 1 パスをマイクロサービスチームと画面チームで協力して開発し、ノウハウを蓄積した上で横展開するという開発順序をとった。

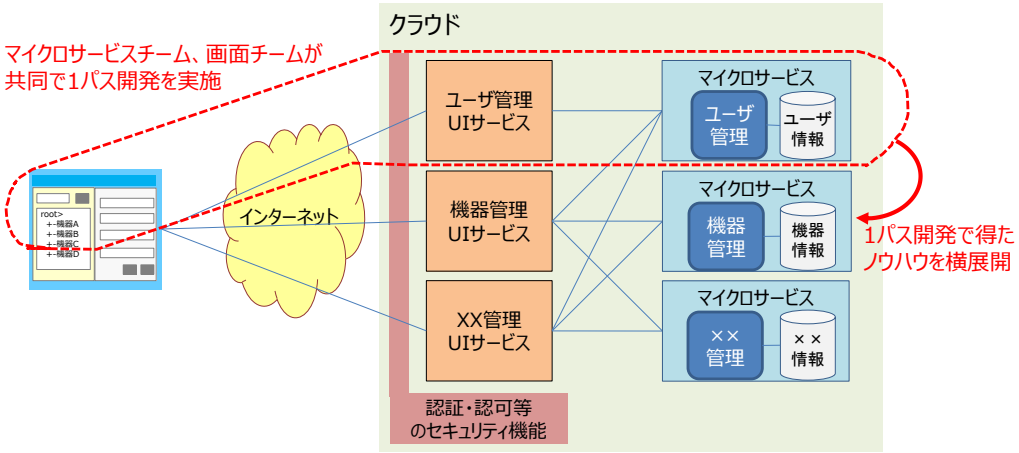

図 83- 7 開発順序

6. マイクロサービスアーキテクチャー適用の評価

5 章で示した課題解決を行って開発した対象システムにおいて、マイクロサービスアーキテクチャーの効果と、開発の進め方に関して評価する。

6.1. マイクロサービスアーキテクチャーの効果

マイクロサービスアーキテクチャーにより対象アプリケーションに要求された 4 つの要件が満たされたかを評価する。評価した結果を表 83- 5 に示す。

表 83- 5 要件に対する結果

要件	結果
変更容易性	UI サービス、マイクロサービスといった役割を持たせ、変更の方針を定義した。各サービスの変更はマイクロサービスの特徴である独立性を維持しているため、他のサービスへの影響なく変更する構成にできた。
スピード	開発中は、UI サービスやマイクロサービス毎に開発/デプロイし、できたものから順次確認をするというサイクルを繰り返して行うことができた。特定のマイクロサービスに不具合があったときは、全体を再ビルドすることなく、不具合のあるマイクロサービスだけを差し替えることができた。
拡張性	マイクロサービスの単位で多重化しスケールアウトすることができた。
可用性	マイクロサービスの単位で多重化することで可用性を高めることができた。

表 83- 5 に示したとおり、マイクロサービスアーキテクチャーについて、期待した効果を得られたことを確認できた。

マイクロサービスアーキテクチャーの効果が分かる事例を図 83- 8 に示す。対象アプリケーションに対して、ある顧客向けの機能を追加する開発が行われた際、対象アプリケーションのマイクロサービスや UI サービスは無停止のまま、顧客向け画面と UI サービスを追加して開発することができた。これは既に稼動しているサービスを停止することなく、システムに対して機能追加ができる事を示しており、可用性を確保しながら変更容易性も持つシステムであることを意味する。

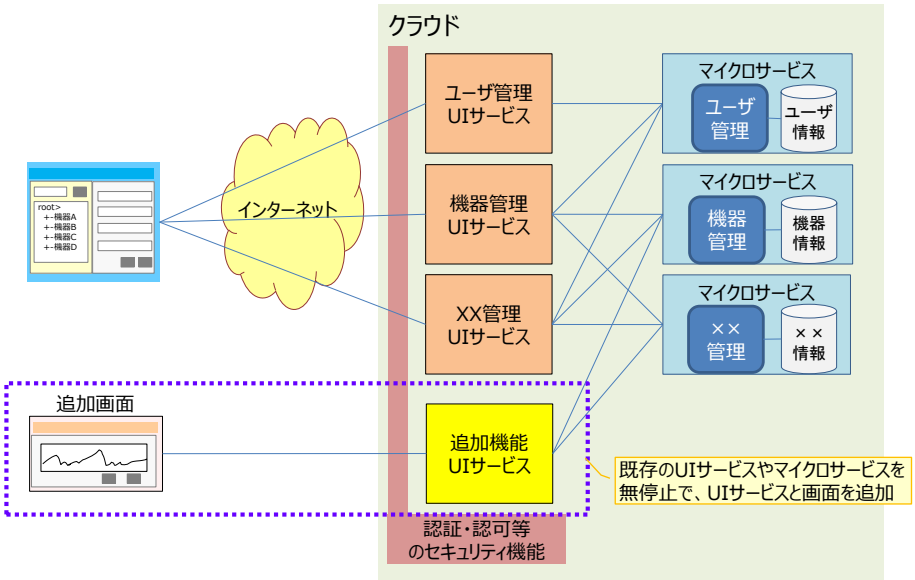

図 83- 8 機能追加の事例

6.2. 開発の進め方の評価

初めて行ったマイクロサービスアーキテクチャーの開発の進め方について評価する。開発順序として 1 パスを作成して横展開するという進め方は、新しい技術を習得しながら開発するプロジェクトには適していた。特にマイクロサービスアーキテクチャーのように独立したマイクロサービスを組み合わせてシステムを実現するアーキテクチャーには、アジャイル開発と組み合わせた親和性が高いと感じた。

しかし、今後アジャイル開発でマイクロサービスアーキテクチャーを実現するには課題もある。開発チームが短い期間で小さなマイクロサービスを開発するためには、チームメンバがあらゆる作業を臨機応変に処理することが期待される。現実的には、開発メンバには得意不得意があり、あらゆるスキルを持った開発メンバを集められることは稀である。本プロジェクトにおいても、開発の効率化を狙ってマイクロサービスチームと画面チームというメンバのスキルに合わせたチーム構成をとった。その結果、個々の技術は高度化したが、スキルの幅を広げる

ことができずアジャイルに期待される人材の育成はできなかった。

マイクロサービスアーキテクチャーは、開発完了後も環境や要望に合わせて変化を受け入れられるものであり、変化を受け入れるためには開発チームもアジャイル開発によって対応できる必要がある。今後の機能拡張や保守において、今回の開発チームメンバ全員を確保できない場合もあるため、アジャイル開発に期待されるスキルを持った技術者の育成が課題となった。

7. マイクロサービスアーキテクチャーの評価

対象アプリケーションの開発において、マイクロサービスアーキテクチャーのメリットを享受することで要件を満たすことができた。本章では、マイクロサービスアーキテクチャーを実践した経験から分かったマイクロサービスアーキテクチャーの難しさについて評価する。

7.1. マイクロサービスの粒度

対象アプリケーションは、「利用される単位」という観点により粒度を決めて要件を満たす事ができたが、他のプロジェクトでは必ずしも容易に粒度を決めることはできないと感じた。マイクロサービスはその独立性を実現するためにマイクロサービスの単位でデータモデルも分離される。そのためマイクロサービス間で RDB のリレーションが利用できず、トランザクションも分散される。

図 83- 9 にマイクロサービスの単位でデータモデルが分離された場合の問題点を示す。デバイス情報とユーザ情報という２つのデータモデルを検索して、デバイス情報とユーザ情報から取得したデバイスの管理者情報を一覧検索するという例である。リレーションが存在する場合、デバイス情報とユーザ情報を Join した SQL 文を１度実行するだけで、検索結果を取得するこができる。対して、マイクロサービスによりデバイス情報とユーザ情報が分離されていると、デバイス管理マイクロサービスからデバイス情報を取得し、取得したデバイスの管理者をユーザ管理マイクロサービスから取得するという処理をデバイス情報分繰り返し行う必要がある。複数回の通信が発生するため、リレーションが利用できる場合に比べて性能が低下するは明白である。

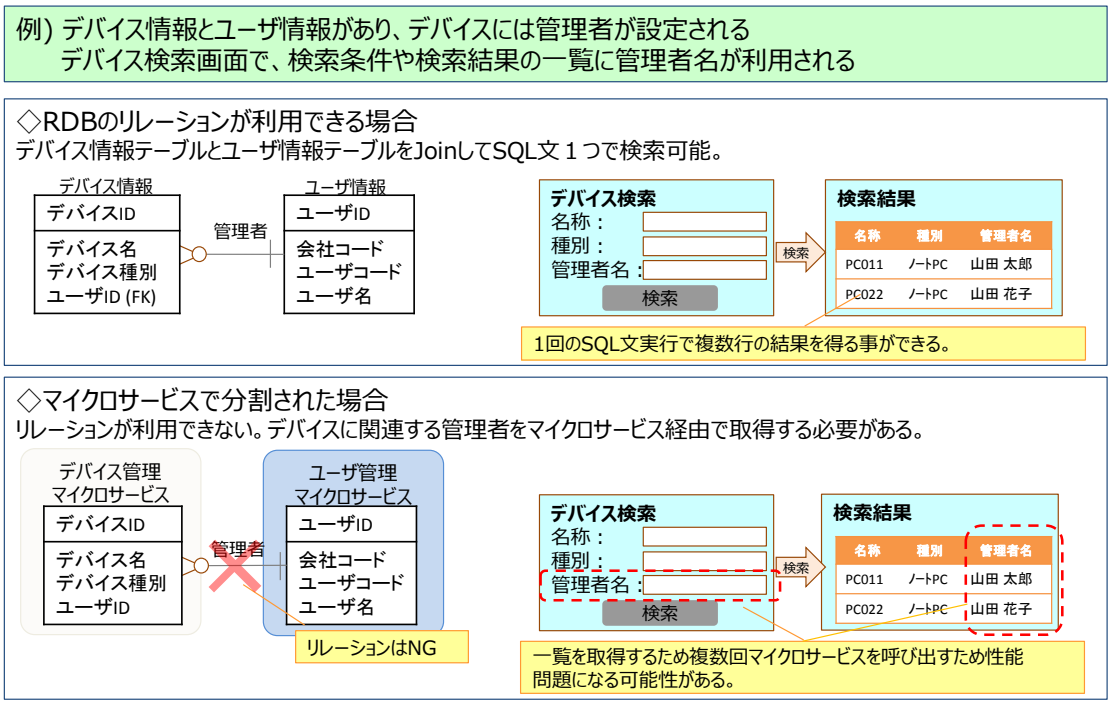

図 83- 9 マイクロサービスによりデータモデルが分離された場合の問題点

エンタープライズシステムの開発でデータモデルを設計した場合、多数のデータモデルが登場し、各データモデル間に関係が張られているような ER 図を多く目にする。つまり、エンタープライズの領域で扱うデータモデルは複雑で互いに関連し合っている場合が多い。そのままマイクロサービスアーキテクチャーを適用した場合、不適切な粒度でマイクロサービスが作られ図 83- 9 に示す性能問題を起こすか、全てのデータモデルを内包した巨大なマイクロサービスが作られて拡張性や変更容易性といったメリットを享受できないものとなる恐れがある。

マイクロサービスの粒度を決める観点は、表 83- 3 で前述した”ライフサイクル”、”スケーラビリティ”、”組織”が一般的に示されているが、複雑なエンタープライズシステムを対象にするには、多くの経験とノウハウ、さらに開発した後にシステムがどのように成長するかまで考慮する必要があり容易ではないと考える。

7.2. マイクロサービスの配置

マイクロサービスは機能的には小さなマイクロサービスとして定義されるが、フットプリントは決して小さくない。近年のアプリケーション開発においては、多数の OSS(Open Source Software)のライブラリを活用して開発するのが一般的であり、1 つのアプリケーション内に数十もの OSS ライブラリが同梱されるのは珍しくない。また、マイクロサービスは単独で起動/停止するために必要なアプリケーションサーバとしての機能も併せ持つ必要もある。以上のことから、たとえ機能的には小さくともアプリケーションとしてのサイズは小さくない。特に OSS のライブラリやアプリケーションサーバの機能などはモノリシックのアプリケーション

であれば共通的に利用できたものが、マイクロサービス毎に分割して必要になるため、全体としてのサイズは大きいものとなる。

クラウド環境を利用する場合、開発したマイクロサービスは仮想マシン(VM)やコンテナ上に配置される。複数のマイクロサービスを 1 つの仮想マシンやコンテナに上に配置することは可能だが、拡張性や変更容易性のメリットを活かすためには 1 つの仮想マシンやコンテナ上に 1 つマイクロサービスを配置することになる。

図 83- 10 に配置と必要なリソースの関係を示す。モノリシックに比べてマイクロサービスアーキテクチャーにした時点で多くのリースを必要とする。さらに拡張性や変更容易性を考慮して個別の仮想マシンやコンテナを用意した場合はさらにリソースが必要となる。クラウドは、容易にリソースを確保できるが、当然のことながら費用が発生するためトレードオフとなる。実際、対象アプリケーションの開発においても当初想定していた以上にリソースが必要になりメモリを追加した経緯がある。

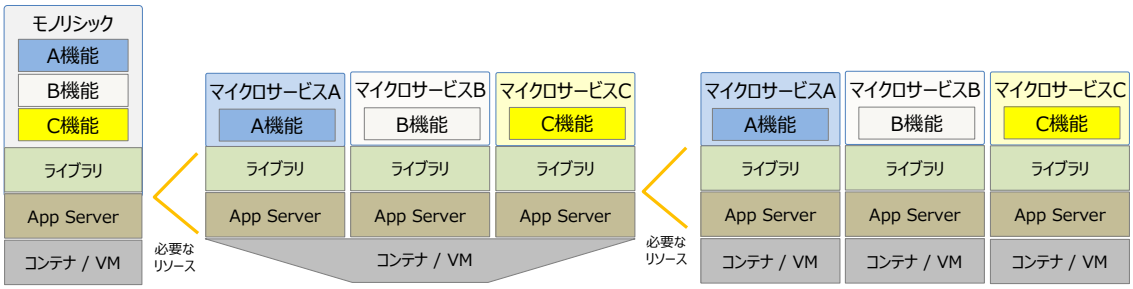

図 83- 10 配置による必要なリソースに違い

8. まとめ

エンタープライズシステムへのマイクロサービスアーキテクチャー適用と当社の今後の取組みついてまとめる。

8.1. エンタープライズシステムへ適用について

対象アプリケーションは、マイクロサービスの粒度が決めやすく、マイクロサービスアーキテクチャーに適したアプリケーションであった。しかし多くのエンタープライズシステムでは、7.1 節に示したように粒度決めは難しい場合があると推測する。しかし、ビジネス環境の変化に俊敏に対応するためには有効なアーキテクチャーであり、また技術に敏感な顧客への反応もよいため、エンタープライズシステムへ適用は広がると考える。

ただし、エンタープライズシステムの複雑さを考慮すると、全ての機能をマイクロサービスアーキテクチャーで実現するにはハードルが高いと感じる。解決策の一案としては、全てをマ

イクロサービス化するのではく、拡張や変更の要望の高い部分のみを抜き出して部分的にマイクロサービス化を行うといったアプローチも有効であると考える。

8.2. 今後の取組み

対象アプリケーションの開発を通してマイクロサービスアーキテクチャーに関する技術的な難しさはないと感じた。エンターテイメントやネットサービス系のシステムでマイクロサービスアーキテクチャーを先行しており、開発するためのライブラリの充実や技術なノウハウが公開されているという要因が大きい。しかし、当社のメインターゲットであるエンタープライズシステムへの適用については、設計におけるノウハウや事例が少ない。今回の事例で得られたノウハウや手法はもちろん、今後マイクロサービスアーキテクチャーを適用した他のプロジェクトで得られた知見も含めて、体系立て形式知化して社内に展開する活動を開始している。

参考文献

[1] Microserivces, http://martinfowler.com/articles/microservices.html, (参照 2016/09/18)

[2] Sam Newman 著,佐藤 直生 監訳, 木下 哲也 訳,『マイクロサービスアーキテクチャ』オライリー・ジャパン,2016 年

[3] Toshiba IoT Architecture SPINEX, http://www.toshiba.co.jp/iot/spinex/ (参照 2016/09/18)

独立行政法人情報処理推進機構 技術本部 ソフトウェア高信頼化センター (IPA/SEC)