マイクロサービス構築を始めるには

クラウドネイティブなアプリケーションの設計と実装における考慮点

インフラやソフトウェア・プラットフォームのサービス提供として始まったクラウドは、その後コンテナに代表される仮想化技術の進歩や、テスト自動化などのツールチェーンに支えられた継続的デリバリーの実現により、アプリケーションの開発スタイルそのものを変えつつあります。クラウドを活用することで迅速に開発でき、処理増加に対するスケーラビリティーを持ち、コンテナ上であればどこでも稼働することができるクラウドネイティブなアプリケーションへの移行を、多くのお客様が検討しています。

本稿では、クラウドネイティブなアプリケーションのスタイルとして特にフォーカスされることの多いマイクロサービスの構築を始めようとしている方々へ、構築の際に直面しがちな設計・実装上の考慮点などを紹介します。

1. マイクロサービスの構築を始めようとすると……

マイクロサービスの構築を始めようとするお客様からの問い合わせで多いのが、「マイクロサービスをどう設計すれば良いか分からない」ということです。

マイクロサービスのコミュニティーでは、“モノリス・ファースト”という考え方があります。これは、いきなりマイクロサービスを新規構築するリスクを避け、まずは普通にアプリケーションを構築し、それからマイクロサービスにリファクタリングする方が無難である、という考え方です。この方法にはリスク軽減というメリットがありますが、その反面、「普通に構築したアプリケーションを後から簡単に分割できるのか？」「初期構築後のマイクロサービス化がほぼ完了した際には、結局最初に構築したアプリケーション費用は無駄にならないのか？」という疑念が残ります。そのため、「最初からマイクロサービスで構築したい」「最初からマイクロサービスとは言わないまでも、後からマイクロサービス化しやすい設計をしたい」という要望があがります。

マイクロサービスは、リリース・サイクルの短縮によりビジネス変化への追従性が高いという多くの優れたメリットを持つ反面、前述したような設計方法に関する課題のほか、実装面、運用面においても課題が存在します。マイクロサービスは、サービスがサービスを呼び出しながら機能を実現する分散トランザクション構造であり、性能面や運用面での考慮が必要で、設計時においても考慮が必要となります。

本稿では、マイクロサービスの構築を始めようとする際に直面する、これらの課題に対する考慮点やIBMの取り組み・事例を紹介します。

2. マイクロサービスの設計における考慮点

2-1. 対象アプリケーションの選定

マイクロサービスの設計では、サービスの分割方法にフォーカスされがちですが、最初の重要な設計観点は、そもそもどのアプリケーションをマイクロサービスとするか、ということです。まずビジネス・ニーズを基にマイクロサービス化の対象を適切に検討することが重要であり、それをどのように設計するのかという検討は、その次の設計観点となります。

マイクロサービス化の対象とするアプリケーションの選定においては、以下のような特徴をもつアプリケーションが有力候補となります。

●成長が予測されるビジネスで活用するアプリケーション

今後のビジネス成長に伴い、システムの拡張／改修コ

ストが増加することを想定し、マイクロサービス化でリリース・サイクルの短縮や変更波及局所化によるコスト最適化を図ります。また成長に伴う処理量の増加にも容易に対応できるようになります。

●ビジネス要求による変更頻度が高いアプリケーション

ビジネス・モデルの変革へ追従するために頻繁に変更が行われている、また他社との競争や顧客要望の変遷への対応として定期的な進化を求められているアプリケーションは、マイクロサービス化により俊敏な機能追加／改善を図れるようになります。

テストやリリースの自動化を中心とした継続的デリバリーの実現によるリリース・サイクルの短縮や、開発チーム間の調整を最小限としたサービス単位での個別リリースなど、マイクロサービスのメリットには開発者観点のものが数多く挙げられます。しかし、サービス構築のコストが投資である以上、ビジネス観点での分析を欠かすことができません。

ビジネス成長領域や差別化のためのIT投資領域の分析としてIBMでは、例えばComponent Business Model（以下、CBM）というモデリング手法で、ヒートマップ分析と呼ばれる重要なビジネス機能を分析する手法があり、これは投資対象のシステムを識別する方法です（**図1**）。

CBMではビジネスを構成する機能の分割と、分割されたそれぞれのコンポーネントとITシステムとの対応・依存関係に関する分析などが実施され、この中で重要となるITシステムが識別され、マイクロサービス化の候補となります。

他の方法としてIBMでは、Strategic Capability Network（以下、SCN）という手法を課題、施策識別のために使用しています。SCNでは戦略目標のために必要な企業能力、またその企業能力を可能とする実現手段は何か、というブレークダウンを実施します。実現手段は知識、組織や体制、業務プロセス、テクノロジーに分類され、このときに実現テクノロジーとしてマイクロサービスが選択されたものが、マイクロサービス化の候補となります（**図2**）。

このようなトップダウン・アプローチ以外にも、例えば現状課題分析からマイクロサービス化の候補を識別するボトムアップのアプローチも有効です。現在の保守状況から、変更頻度やリリースに要する期間、またリリース時の障害の発生率などを分析し、マイクロサービス化が有効となるアプリケーションを識別します。マイクロサービスでの改善目標を数値化しやすいという面もあります。

2-2. 構築アプローチ

マイクロサービス化のアプリケーションが選定されたら、次の課題は構築方法です。

まずアプローチですが、大規模システムのビッグバン更改（または新規ビッグバン構築）はリスク面から推奨されません。ビッグバンを避けるということは、価値のある機能をなるべく早くリリースできるようにする、というマイクロサービスの思想を保守局面だけでなく、開発局面にも取り入れるためにも必要な考え方です。価値

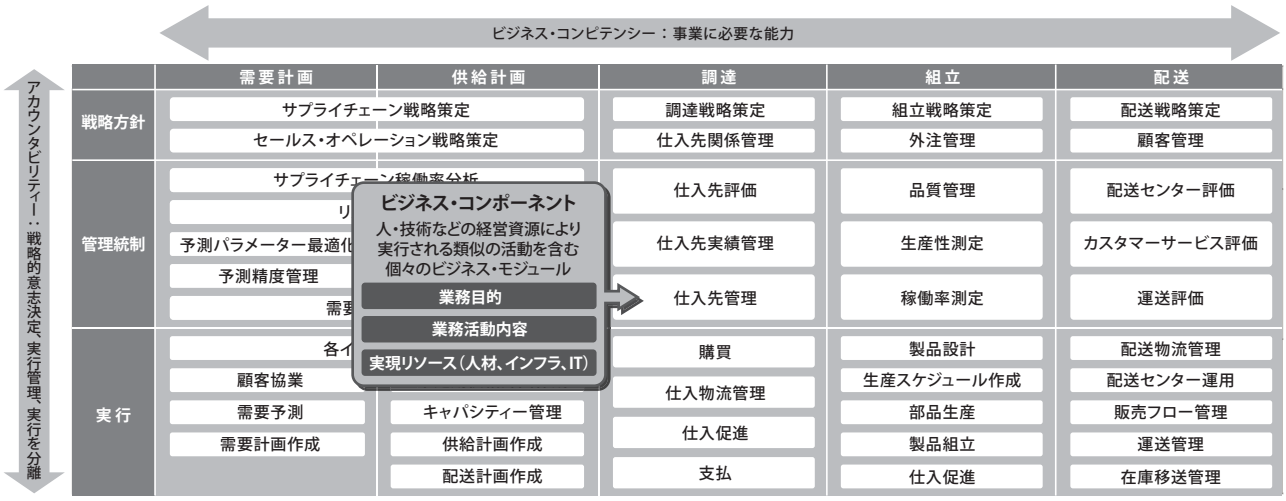


図1. CBM Map（サプライチェーンの例）

ある機能をなるべく早期から順次リリースしていくインクリメンタルな繰り返しアプローチが推奨されます。継続的なリリースのたびにテストが必要となりますが、マイクロサービスの運用前提となる継続的インテグレーション環境がこれらテストとリリースの負荷を軽減します。

このような段階的な構築アプローチとして、既存システムと共存しながら段階的に新システムを構築／移行し、最終的に新システムへの移行が完了した後に旧システムを停止するストラングラー・パターン[1]が推奨されています（**図3**）。新規構築の場合にも、ビジネス要求上の優先度で機能を分類し、優先度の高いものから段階的に構築・リリースを行うアプローチが推奨されます。この場合、段階的なリソース増強に適しているクラウドのメリットも享受できます。

2-3. サービスの分割

次に、具体的なサービスの設計に入ると、設計者は相反する要件を調整し、最適なサービスの分割境界を決定します。ここで「相反する」と述べたのは、サービスが小さくなれば機能変更時の影響スコープが小さくなりリリースの自由度が向上するといったメリットがある反面、システム全体の処理時間は増加傾向となり、障害頻度や考慮点が増加するというデメリットも増すことを指しています。このため、設計者は複数の観点から、「サービスが大きすぎず、小さすぎないこと」を検証しつつサービスのスコープを決定する必要があります。

観点として中心となるのはドメイン分析です。既存のドメイン・モデルがあればそこから開始することができますが、存在しない場合、新規に作成するか、業界モデルを参照することになります。

対象業務ドメインを境界づけられたコンテキスト[2]に分割し、各コンテキストをチームに割り当てます。コンテキストの分割においては、エンティティーの概念や関連で分割される一方、実際には担当部門や利用者のロールが切り替わる境界で分割されることも多く、ビジネス・プロセスやデータ・フロー・ダイアグラムによる分析も有用です。また、すべてのコンテキストをマイクロサービスにする必要はありませんが、マイクロサービスにしたいコンテキストを選定したら、そのコンテキストは一つのチームで構築・保守できる大きさに分割しておくべきです。

ドメイン分析を補完する観点として、ユーザー・ジャーニー分析も使用できます。これはユーザーに対して提供すべき価値あるサービスを識別・選定するための分析で、仮定したペルソナの行動と、各タッチポイントで必要となる機能からサービスが提供すべきAPIを定義していく方法です。採用されたAPIはマイクロサービスに割り当てられていきます。

もう一つ補完的観点で重要なのがリソース（データ）分析です。あるデータに対して責務を持つサービスは一つであることが理想です。サービスの粒度が比較的大きければ、かなり理想に近い設計も可能ですが、それでもコンテキスト境界をまたがったライフサイクルを持つデータの扱いや、参照のための他サービスへのデータ複製配布など、実運用のためのサービスへのデータ配置検討が

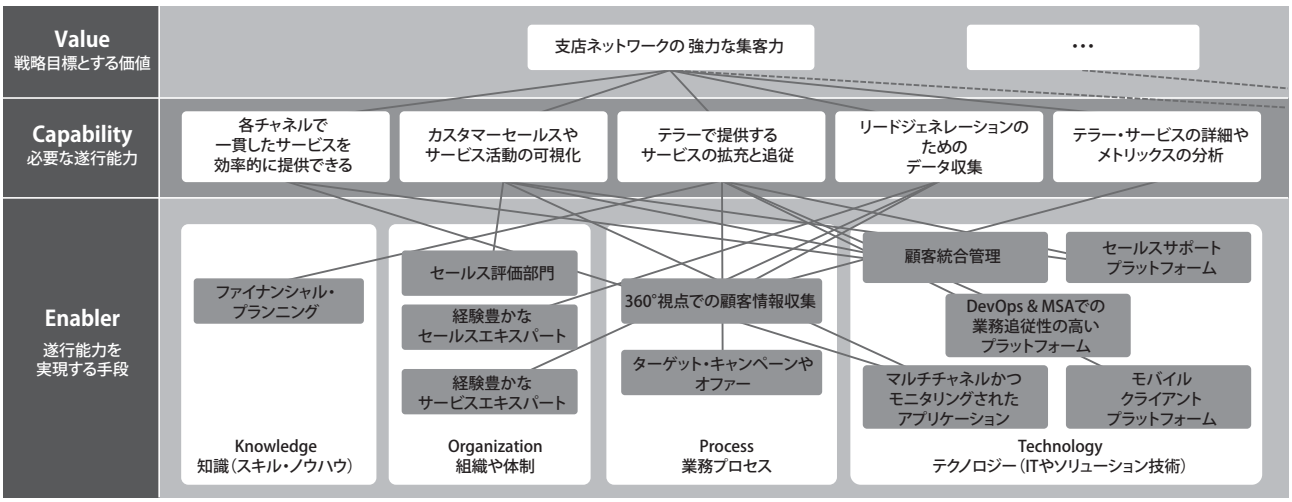


図2. SCN（金融業での分析・検討例）

必要となります。現実的な判断として、最初は複数のマイクロサービスがデータを共有する構成を採用することもあるでしょう。データ配置の設計はサービスの疎結合性と処理性能含めた運用性とのトレードオフのバランスを取る必要があり、各処理で必要となるデータ鮮度やデータ整合性の確保、レスポンス要件、障害時対応など非機能要件面を考慮した検討を行います。

2-4. 非機能側面の考慮

こうしたさまざまな観点からサービスの分割境界を決定し、各サービスの役割や提供するAPIを定義したら処理設計が始まります。

マイクロサービスは複数のコンテナ上のサービス協業により処理を実現する分散トランザクション・アーキテクチャーであり、処理設計においても業務仕様以外の非機能面の考慮点が増えています。障害発生を前提としたリトライ処理や補償処理の考慮は必須であり、またリトライされることを前提に処理に冪等性（べきとうせい）を持たせることが重要になります。マイクロサービス間の連携は一般的にRESTや軽量なメッセージングを使用しますが、耐障害性などで非同期メッセージングによる連携の採用が有効なケースが多々あります。こうした実装面の考慮により設計を最適化することが重要です。旧来の局面化された設計・開発では上流と下流でのスキル分業が主流となっていましたが、明らかに近年では設計者に業務／ドメインの知識とシステム設計の知識双方が必要となっています。

ここまで述べてきたように、マイクロサービスの構築にはビジネス分析、業務ドメイン分析、ユーザー分析、データ分析、非機能要件分析と、さまざまな分析に基づいた設計が求められることになります。

3. マイクロサービスの実装における考慮点

設計が終われば、次は実装となります。システムをマイクロサービスで構築することにより、ビジネス変化へ素早く対応するための変更リスクを低減できる反面、以下の考慮が必要となります。

●耐障害性の考慮

モノリスなシステムの同一プロセス内でのサービス呼び出しに比べ、マイクロサービスのネットワークを介したリモート呼び出しは、対象マイクロサービスのプロセス停止やハングアップ、過負荷など、失敗する可能性が高くなります。また、複数のサービスに依存したサービスの信頼性は、その乗算となり、マイクロサービスを複数経由する分、モノリスなシステムと比べて信頼性が劣ります。このため、一部のサービスのプロセスダウンやスローダウンによる影響が、サービス全体に及ばないようにするため、サービス呼び出し側で、耐障害性を高めるための対策が必要となります（Fault Tolerance）。

●運用の自動化、ツール化

マイクロサービスのシステムは、複数のサービスによって成り立っているため、監視や障害対策、リリースの運用もサービスごとに行う必要があります。サービスが増加するにつれて、従来の方式では運用負荷が高くなるた

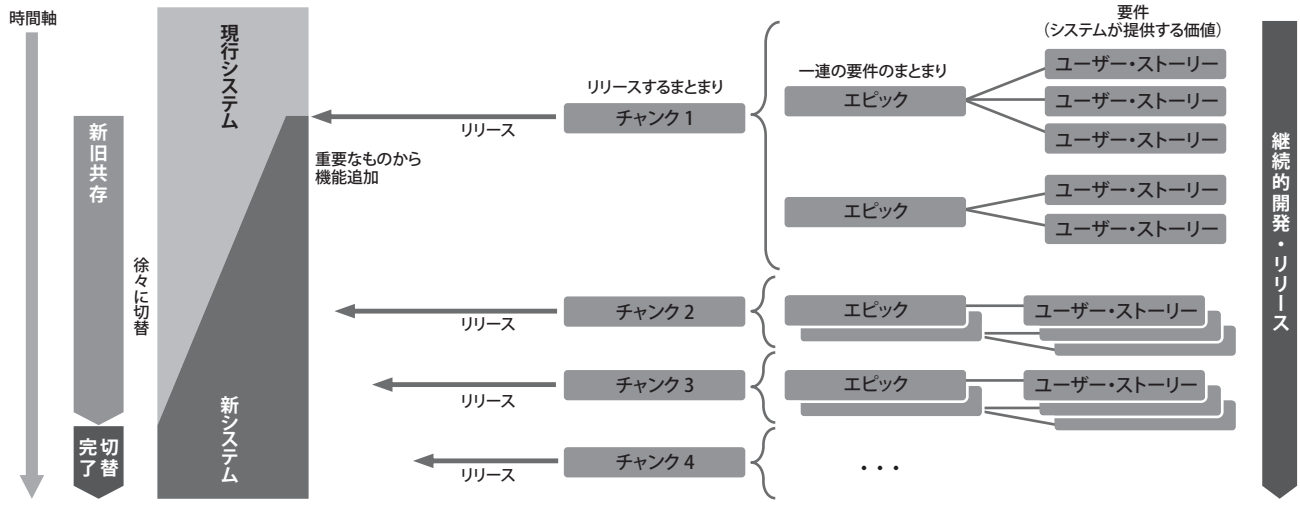


図3. ストラングラー・パターン

め、運用負荷を低減する対策が必要となります。障害時はできるだけ自動復旧するよう、各サービスの監視、ダウンした際の自動リカバリー、高負荷時の自動スケーリングが必須の機能となります（Health Check、Auto Healing、Auto Scaling）。

モノリスなシステムでは、一つのシステムの状態を確認しさえすれば、障害時の調査、サービスの状態を把握することができました。しかし、マイクロサービスのシステムでは、すべてのサービスの状態を確認する必要があります。数十、数百のサービスのログや、サーバーの稼働状態を人手で調査するには大変な労力を要します。このため、マイクロサービスでは、各サービスのログを収集し、分析できるサービスが必要となります（Distributed Tracing、Metrics、Log Collection、Log Analyze）。

本章では、Fault Tolerance、Distributed Tracingの実装について説明します。

3-1. Fault Tolerance

他のサービスをリモート呼び出しする際、呼び出し側のサービスは、以下を考慮した設計、実装とする必要があります。

■Timeout

同期処理でのマイクロサービス呼び出し時、呼び出し先のマイクロサービスが過負荷やハングなどで応答が返ってこない場合、呼び出し側のマイクロサービスも応答を待ち続ける状態となり、エンドユーザーへ応答が返らずハングしているように見えます。後続の処理も同様に対象サービスの応答待ちとなるため、呼び出し側のサービスもハング状態になり、影響が連鎖的に全体に及びます。適切な時間内に応答が返らない場合は、タイムアウトするようにし、全体がハングしないようにします。

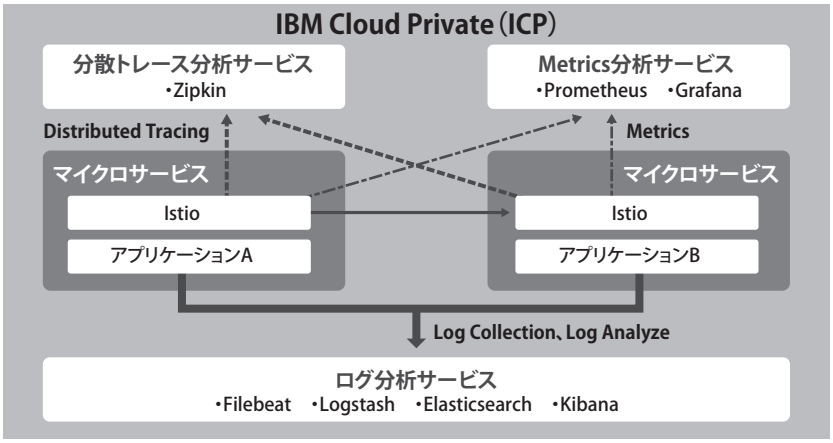
■Circuit Breaker

一般的に、処理内容やシステム状態によりシステムのレスポンス・タイムは変化するため、タイムアウト値を「レスポンス・タイムの期待値＋予備時間」とするのが一般的です。また、タイムアウト値を設定していたとしても、一部のサービスが一時的な過負荷によりスローダウンした場合、タイムアウト待ちのスレッドでサーバーのリソースが占有され、他の要求を処理できなくなります。このため、一定の割合で呼び出しが失敗したサービスに対しては、一定時間そのサービスへの要求は行わずにエラーとし、不必要なリソース占有状態を回避します。一定時間経過後に一部対象のサービスに要求を送信してみて、正常処理が確認できたら通常状態に戻し、正常処理が確認できなければ再度そのサービスへの要求は行わないよう制御します。

■Bulkhead

タイムアウトしないまでも、呼び出し先のサービスがスローダウンした場合には、呼び出し元が応答待ちのスレッドで占有されてしまい、サービスが提供できなくなってしまいます。ある一つのサービスのスローダウンがボトルネックとならないよう、サービスごとに同時呼び出し数を制限し、影響を局所化します。

Fault Toleranceの実装は、OSS（Hystrix）や製品などライブラリーが多数提供されているため、スクラッチ



- Istioにより、Metrics情報をMetrics分析サービスに送信
- Istioにより、分散トレース情報を分散トレース分析サービスに送信
- Istioにより、他のマイクロサービス 呼び出し時、Fault Tolerance対応
- ICPにより、マイクロサービスのログは、ログ分析サービスに送信

ICP+Istioにより
マイクロサービス・アーキテクチャーに必要な機能は、アプリケーション改修なしに実装可能

図4. IBM Cloud PrivateとIstioを用いたマイクロサービス実装例（ログ分析サービスを内包）

で開発する必要はありません。「IBM WebSphere Application Server Liberty」(以下、WAS Liberty)ではMicroProfile Fault Tolerance 1.0仕様に準拠したフィーチャーを提供しています。MicroProfileは、Java Enterpriseでマイクロサービスのアプリケーションを開発するための共通仕様です。また、Hystrix、MicroProfile共に、呼び出すマイクロサービスごとにクライアントを作成、Fault Toleranceの設定が必要ですが、AIF(API Integration Framework:WAS Liberty上で稼働するIBMアセットのJavaフレームワーク)では、一つの汎用的なクライアントを提供しており、設定のみでFault Toleranceを実現可能です。

3-2. Distributed Tracing

エラーやスローダウン発生時には、あるサービスから呼び出された複数のサービスの処理結果および処理時間のデータをリアルタイムで可視化し、障害時のリードタイムを短縮することが必要です。このためには、各マイクロサービスが実行された際に、実行されたサービスの処理情報と処理時間を、これらの情報を収集するためのサーバー(分散トレースサーバー)にリアルタイム送信します。分散トレースサーバーは、Zipkin、JaegerといったOSSで構築が可能です。各マイクロサービスから分散トレースサーバーへ送信する機能は、同じくZipkin、Jaegerが提供するクライアント・ライブラリーを使用し、各マイクロサービス側で実装する必要があります。WAS Libertyでは、MicroprofileのOpenTracing 1.0仕様に準拠したフィーチャーと、そのフィーチャーを使いZipkinに情報を送信するサンプルを提供しています。さらにAIFを組み合わせ、Zipkinへ送信する情報にAIFが対象リクエストを実行したログの追跡IDを付与し、アプリケーション・ログとのトレーサビリティーを強化しています。

上記ではコンテナ内のアプリケーション、ミドルウェアによる実装方法を記載しましたが、「IBM Cloud Private」上でKubernetesを使用した構築マイクロサービスを実装する場合、待望のバージョン1.0がリリースされた「Istio」を導入することで、アプリケーション自体に手を入れることなく、Fault Tolerance、Distributed Tracing、Metrics、Log Collection、Log Analyzeの機能を実装することができます(**図4**)。

このように、マイクロサービスの実装構築には、その欠点を補うための手段が必要であり、IBMではこれらを提供するアセットの開発とOSSの利用による対応を行っています。本稿では記載していないマイクロサービス構築に必要となる機能、Health Check、Auto Healing、Auto Scaling、Metrics、Log Collection、Log Analyzeについても、IBMアセットとして、設計ガイド、実装ガイドを作成中で、2018年内にリリース予定です。

4. おわりに

以上、マイクロサービスの構築時に、設計・実装面で直面する課題への取り組みについて紹介しました。冒頭で述べたようにマイクロサービスは仮想化や自動化技術の進化を取り入れながらも、過去のソフトウェア工学が築いてきた進歩の先へ向かうものの一つであり、これからの展開が望まれています。

[参考文献]
[1] IBM DeveloperWorks:マイクロサービス・アプリケーションにストラングラー・アプリケーション・パターンを適用する, Kyle Brown, https://www.ibm.com/developerworks/jp/cloud/library/cl-strangler-application-pattern-microservices-apps-trs/index.html
[2] エリック・エヴァンス:ドメイン駆動設計, Addison-Wesley (2003)

日本アイ・ビー・エム株式会社
グローバル・ビジネス・サービス事業部
クラウド・アプリケーション・サービス, MSA&モダナイゼーション 担当
シニア・アーキテクト/コンプレックス・エンタープライズ・アーキテクト

松本 龍幸
Tatsuyuki Matsumoto

1993年日本IBM入社。金融業、製造業のお客様を中心にホスト基幹系からWeb基幹系まで大規模システム構築を主体として、アーキテクトとしてアプリケーション基盤の構築や開発手法の適用に従事。現在はクラウドの提案とデリバリー適用を担当。

日本アイ・ビー・エム株式会社
グローバル・ビジネス・サービス事業部
クラウド・アプリケーション・サービス, API&インテグレーション
ITスペシャリスト

大和田 弘成
Hironari Ohwada

2002年日本IBM入社。金融のお客様を中心にWebアプケーション開発案件のシステム構築に従事。ホストGWシステム開発などを経て、現在は、APIシステム構築の提案、構築支援、APIアセットの推進を担当。