WHITE PAPER | 2014 年 2 月

アプリケーション・リリースの完全自動化

Daneil Kushner と Eran Sher
アプリケーション・デリバリ

ca technologies

目次

概要 3

セクション 1: 4
アジャイル開発を採用している組織でデプロイを自動化し、結果を出すには

アジャイルな反復 : マイクロ開発ライフサイクル

継続的インテグレーション : アジャイル手法のテストとインテグレーションへの拡張

セクション 2: 6
アジャイルな組織におけるゼロタッチ・デプロイメントの目的

デプロイを自動化する複雑さ

自動デプロイは本番環境だけではない

すべてに共通するデプロイ言語

概要

課題

アジャイル開発を導入している組織では、運用チームが大きな課題に直面しています。それは、開発やテストが完了した後、直ちにリリースを本番にデプロイするということです。アプリケーションが正しくデプロイされるためには、自動で透明なプロセスが必要で、私たちはこのプロセスを「ゼロタッチ・デプロイメント」（手作業を介さないデプロイ）と呼んでいます。

本資料では、ゼロタッチ・デプロイメントへの 2 つの手法、スクリプト・ベースのソリューションとリリース自動化プラットフォームについて検討します。アジャイル開発を採用している組織が自動リリース・システムの実装に着手する場合に、これらの手法が、どのように技術的および組織的課題を解決できるかについて説明します。

まず最初に、リリース自動化ソリューションを求めるに至ったビジネスおよび技術的背景について考えてみましょう。

セクション 1:

アジャイル開発を採用している組織でデプロイを自動化し、結果を出すには

エンタープライズ・ソフトウエア業界は競争が激しく、組織は、新しいサービスを素早く本番へと移行できるようにする開発方法を採用する必要があります。

組織は、ビジネス目標を満たすため、継続的に改善と確信を求める顧客の期待を満たし、市場と開発チームの間のフィードバック・ループを短縮する必要があります。

開発チームは、画期的な新機能を展開することへのプレッシャーに対応するうえで、現行のソフトウェア開発方法が十分ではないことを認識しています。実現させるためのソリューションの 1 つが、ソフトウェア・リリースを短縮するために新しいソフトウェア開発方法を構築することでした。

このような開発フレームワークは通常、「アジャイル開発」と呼ばれ、開発プロセスに反復とインクリメンタルな手法を実現することで、ソフトウェア開発ライフサイクルを短縮することを目的とします。

アジャイルな反復 : マイクロ開発ライフサイクル

反復型でインクリメンタルな開発の基本原理では、ソフトウェア開発を小さなサイクルまたは反復で進めていきます。各反復は、一般的なソフトウェア開発ライフサイクルのマイクロ（小単位）実装となります。ライフサイクルは、計画から始まり、開発とテストへと続き、デプロイで終了します。最初の反復では、ソフトウェア要件のサブセットを実現し、その後の反復では、アプリケーションを徐々に肉付けしていきます。

このサイクリックなアプローチにより、現在の反復に関するユーザのフィードバックが今後の開発計画に影響を及ぼすため、組織は、顧客ベースに一層注意を払うようになります。アプリケーションの機能が常に評価、変更および改善されることで、アプリケーションのビジネス価値が拡大します。また、反復型アプローチには、従来のウォーターフォール型開発方法と比べて、より早い段階でアプリケーションが収益を生み出せるようにするという大きなメリットがあります。

反復型アプローチは、設計から開発、テストおよび最後のデプロイまで、開発プロセス全体に渡ります。しかし、このような変更を実現するのは決して簡単なことではありません。1 つの反復の範囲内で、新機能のテスト、汎用ビルドへの統合、ステージングおよび本番環境へのデプロイを実行するため、より素早い方法を確立する必要があります。テストおよび統合ステーションの場合、ソリューションは継続的インテグレーション（CI）に基づきます。

継続的インテグレーション：アジャイル手法のテストとインテグレーションへの拡張

継続的インテグレーションは、テストとインテグレーションをひとつの反復に組み入れるために、自動化技術を利用します。開発者が新しいコードをバージョン管理システムにコミットすると（できれば、1 日に 1 回以上）、CI サーバが自動的に新しいビルドを実行し、すべての自動テストを適用します。このテストには、ユニット・テスト、コンポーネント・テスト、統合テスト、機能および非機能受け入れテストが含まれます。なお、手動テストは機械的に実行できないテストのみに限られます。また、テストの失敗として特定された問題は、次の反復に進む前に、開発チームが修正できます。この説明はもちろん概略的なものですが、ビジネスでの生き残りが、新機能を継続的に本番にリリースできる機能に左右される今、多くの組織が現実に直面する問題を理解するのに役立てればと思います。

しかし、ソフトウェア開発サイクルは、テストおよび統合ステーションで終わるわけではありません。アジャイル開発を完遂するには、組織は、新機能をアプリケーション・ライフサイクルのすべての環境に導入できるように、反復を拡張する必要があります。少なくとも理論的には、導入は CI フレームワークの一部ですが、ソフトウェア開発ライフサイクルのこの最後の段階は、アジャイル開発エコシステムに対応するための最後の段階でもあると考えられます。

ボトルネックが運用へと移動

現在、組織は、アジャイル開発を開発とテストに採用すると、デプロイする直前の中止は不可能になります。アジャイル開発を完璧に実行し、すべての反復が正しく機能したとしても、予定通りに導入できなければ意味がありません。運用チームはキャパシティの過多に陥り、新しいアップデートを処理できなくなり、開発とテストから溢れ出したボトルネックが、運用へと流れ込んできます。

すなわち、運用チームは、増え続けるデプロイ回数を除外しても、複雑な課題に直面していることになります。複数階層とサービス指向アーキテクチャにより、デプロイのフローはさらに複雑化しています。物理および仮想環境が同時にサポートおよび維持され、仮想化やクラウド・コンピューティングの登場により、デプロイ済みサーバ数が飛躍的に増加しています。こういったビジネスの現実に基づき、アジャイル開発の採用と継続的インテグレーションは、すでに複雑な組織および技術環境をさらに複雑化するため、一層問題が深刻になります。

組織において、アジャイル開発が遅れて運用に採用されることで、アジャイル開発の本来の目的が損なわれます。1 つの専用リリースにデプロイされるのではなく、単独の反復で開発された製品が、他の反復で開発された製品にパッケージされるのです。このような望ましくない結果は、本番での実行、収入の創出および顧客からの評価の前に新機能がまだ導入されていないため、アジャイル開発の基本特性に適合しません。

セクション 2:

アジャイルな組織におけるゼロタッチ・デプロイメントの目的

アジャイル開発を実施している組織は続いて、運用における大きな課題に直面します。即ち、開発とテストが完了すれば、直ちにアプリケーションを本番へとリリースし、ユーザからの評価を得て、可能であれば収益を創出することです。言い換えれば、デプロイ・プロセスは、そのプロセスが起動されると同時に、完全に自動化され透明でなければならないのです。

極端に言えば、ゼロタッチ・デプロイメントとは、新しいビルドを受け入れテストから正しく展開することにより、本番へのリリースが自動的に開始されるプロセスと定義できます。通常、複雑なリリースに対してゼロタッチ・デプロイを実現するには、カスタム・メイドのデプロイメント・マニフェスト・ファイルを作成します。マニフェスト導入は、アプリケーション・リリースの動的要素を定着したプロセスから分離します。マニフェストによるデプロイは、アプリケーションのリリースにおける動的な要素を固定的な要素から分離します。これは、あらゆる動的で変更可能なデプロイの要素を XML ファイルや他のデータソースに定義することを意味します。これには、どのアプリケーション・リソースが利用されるかであるとか、アプリケーションの具体的な位置やバージョンといった詳細情報も含まれます。これらの詳細情報は、リリース毎に素早く簡単に変更することができます。リリース・プロセスがどのように実行されるかを定義したシンプルな固定プロセスを動的な要素と別に管理することで、繰り返し利用することができるようになるのです。

図 1：

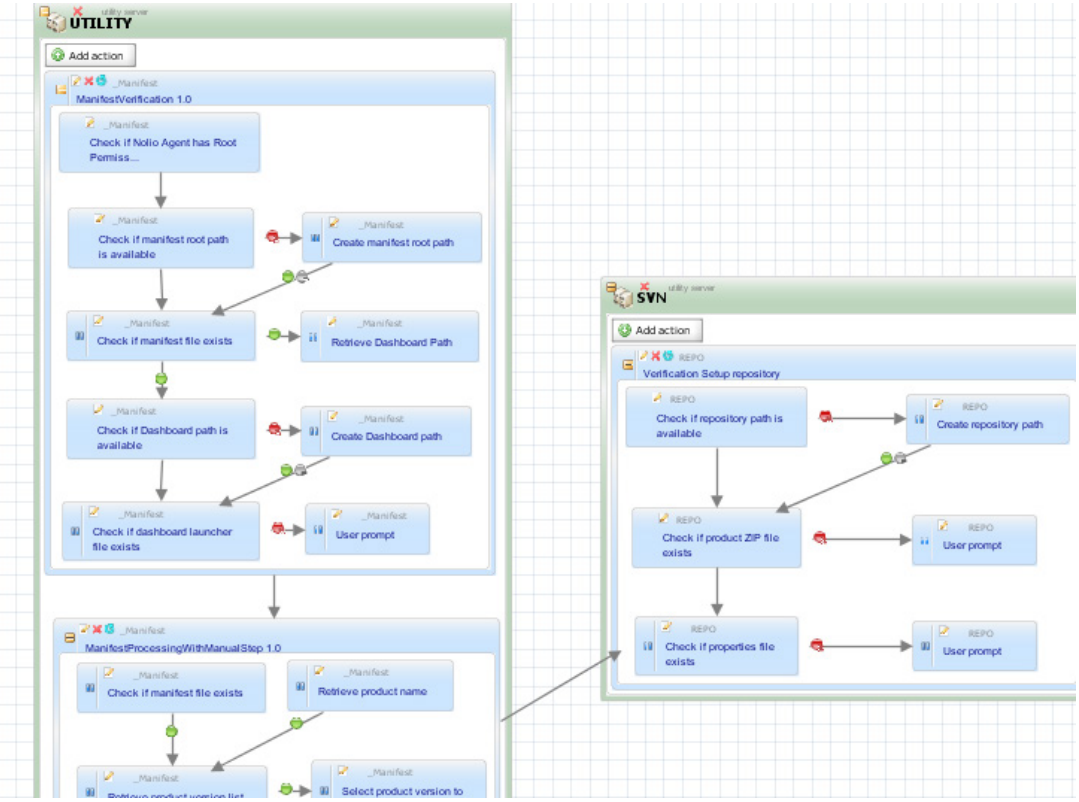
XML デプロイメント・マニフェスト：

すべての反復において、本番環境と利用者に対して通路を開くこれらのプロセスによって、アジャイルなライフサイクルをサポートするデプロイメント・ステーションの実現が可能になります。

```
<topology name="RDL">

    <server-type name='MASTER-DB'>
        <Artifacts>
            <Artifact artifact-type='ces_master'    deployment='sql' schema-type='master'   />
        </Artifacts>
    </server-type>

    <server-type name='THREAD-DB'>
        <Artifacts>
            <Artifact artifact-type='ces_portal'    extension='zip' deployment='sql'    db-type='ces'   />
            <Artifact artifact-type='ct_portal'     extension='zip' deployment='sql'    db-type='ct'    />
            <Artifact artifact-type='dw_portal'     extension='zip' deployment='sql'    db-type='dw'    />
        </Artifacts>
    </server-type>

    <steps>
        <step name='CUSTOMERS-INIT'     process='01_INIT-CUSTOMERS 1.0'         count='1'   dependency='NONE'                retries='1' />
        <step name='ARTIFACT-DELIVERY'  process='01_DISTRIBUTE-ATIFACTS 1.0'    count='1'   dependency='CUSTOMERS-INIT'      retries='1' />
        <!-- <step name='WEB-SWITCH-THREAD'     process=''                      count='4'   dependency='CUSTOMERS-INIT'      retries='1' /> -->
        <step name='MASTER'             process='02_MASTER-DB 1.0'              count='1'   dependency='ARTIFACT-DELIVERY'   retries='3' />
        <step name='DB-EXEC-THREAD'     process='03_DEPLOY-DB 1.0'              count='4'   dependency='MASTER'              retries='1' />
        <step name='WRAPUP'             process='05_WRAPUP 1.0'                 count='1'   dependency='ALL'                 retries='1' />
    </steps>
```

図 2：

CA Release Automation での XML マニフェストの利用



デプロイを自動化する複雑さ

しかし、大規模データセンタへで利用されるようなアプリケーションのデプロイを自動化することは単純な作業ではありません。特に、アジャイルな組織におけるデプロイのニーズに対して真に完全な答えを提供することは困難を極めます。一連のデプロイメント・スクリプトで構成するか、専用のアプリケーション・リリース自動化プラットフォームで管理するのかに応じて、デプロイ自動化システムは、複数レベルで多数の複雑化要因に対処する必要があります。

複雑性は、アジャイル組織が生み出す複数の異なるタイプのデプロイ・イベントによって引き起こされます。つまり、アプリケーションの完全インストールのような大規模なイベントから、構成ファイルをひとつ更新するようなマイナーな更新に至るデプロイ・イベントを処理するためには、様々な自動デプロイ・プロセスを作成しメンテナンスしなければならないことを意味します。自動デプロイメント・システムは、アプリケーションのデプロイ作業を簡素化し、リスクを低減し、複雑な手作業を信頼できる繰り返し可能でエラーのないプロセスへと転換できなければなりません。自動デプロイメント・システムによって、個々のリリースに要する時間が短縮され、複数アプリケーションの同時並行デプロイが可能になるのです。

さらに、デプロイメント・イベントには複数のアプリケーション層を含めることができます。バグ修正などのマイナーなデプロイメント・イベントには、ミドルウェア・サーバの構成ファイル内のパラメータ値の変更や、アプリケーション・サーバのライブラリ・ファイルへのパッチ適用を含めることができます。アプリケーションを最初からインストールするなどのメジャーなデプロイメント・イベントでは通常、データベース・サーバやアプリケーション・サーバ、Webサーバに対してアプリケーションを同時並行でデプロイします。

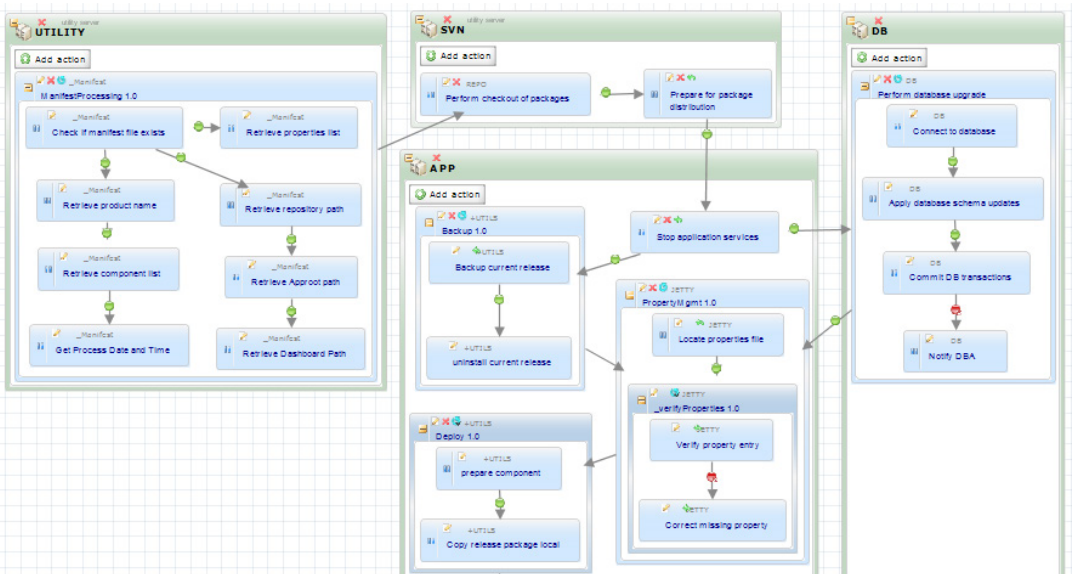
いかなる自動デプロイ・プロセスも、各アプリケーション・コンポーネントや階層に対してひとつずつ異なるデプロイ・フローを構成する必要があることを意味します。したがって、多層アプリケーションのデプロイを実現するには、自動デプロイ・システムが、デプロイされるアプリケーションの階層ベースのアーキテクチャを捕捉できなければなりません。スクリプト・ベースのシステムでは、個々のデプロイ・スクリプトを各階層にマッピングさせます。専用の自動リリース・プラットフォームは、グラフィカルなアプローチで、デプロイされるアプリケーションの個別階層独自のフローをユーザがビジュアルに表示し作成できます。

階層固有のデプロイ・フロー間に相互依存の関係がある場合は、自動デプロイ・プロセスが含めることができなければならない別の複雑化要因が生じます。 別のフローが完了した後でのみ、あるフローが開始できるデプロイ・プロセスは相互依存にあるといわれます。データベースのデプロイ・フローが完了するまでWebサイトのデプロイ・フローが開始できないような例がそれにあたります。

自動デプロイのメカニズムには、相互依存の階層固有のデプロイ・フローの実行順序を調整できることが求められます。スクリプト・ベースのシステムでは、ビルド管理ツールを使用して、異なる階層のデプロイ・スクリプトの実行を調整できますが、自動リリース・プラットフォームの場合は、異なる階層別フローを示すノード間にコネクタを描くことで、ユーザが依存関係を指定することができます。

図 3：

アプリケーションのデプロイ中の多階層の依存性



デプロイ・イベントとフローの多様性や、その実行管理や調整に伴う課題に加え、自動デプロイ・システムは、複数のデプロイ先を管理しなければなりません。グローバル規模のユーザーを抱えるオンライン・アプリケーションの登場と、仮想化およびクラウド・ベースのテクノロジーの普及により、ひとつのアプリケーション・リリースが対象とするインスタンス数は桁違いに増加しました。その結果、自動デプロイ・システムは、ひとつのデプロイ・イベントを、複数のデータセンタに存在する膨大な数のサーバに対して実行できなければなりません。

それぞれが固有の構成情報を持つ多数のサーバに対してアプリケーションのデプロイを管理できるようにするには、自動デプロイ・システムは、デプロイ・プロセスと構成情報を明確に分離できなければなりません。同一のデプロイ・プロセスのインスタンスは、各デプロイ先に固有の構成情報を使って実行されます。

このような分離をサポートするには、スクリプト・ベースのシステムでは、厳密な構成管理ポリシーを設定しなければなりません。しかし、自動リリース・プラットフォームの場合は、アプリケーションがデプロイされる環境をマッピングし表示する専用のグリッドがあります。物理/仮想化/クラウド・サーバはそれぞれ、該当する環境やデプロイ・プロセスとの関係に基づいて、グラフィカルに表示されます。各サーバの構成情報は、この表示に基づいて、表現、保管、メンテナンスされます。

ゼロタッチ・デプロイメントの原則に基づいてデプロイを自動化すると、組織は、アジャイル開発のフレームワークでデプロイメント・ステーションの目的を実現できます。すなわち、新しいコードの開発およびテストが完了した後、直ちに本番環境へとデプロイできるのです。

自動デプロイは本番環境だけではない

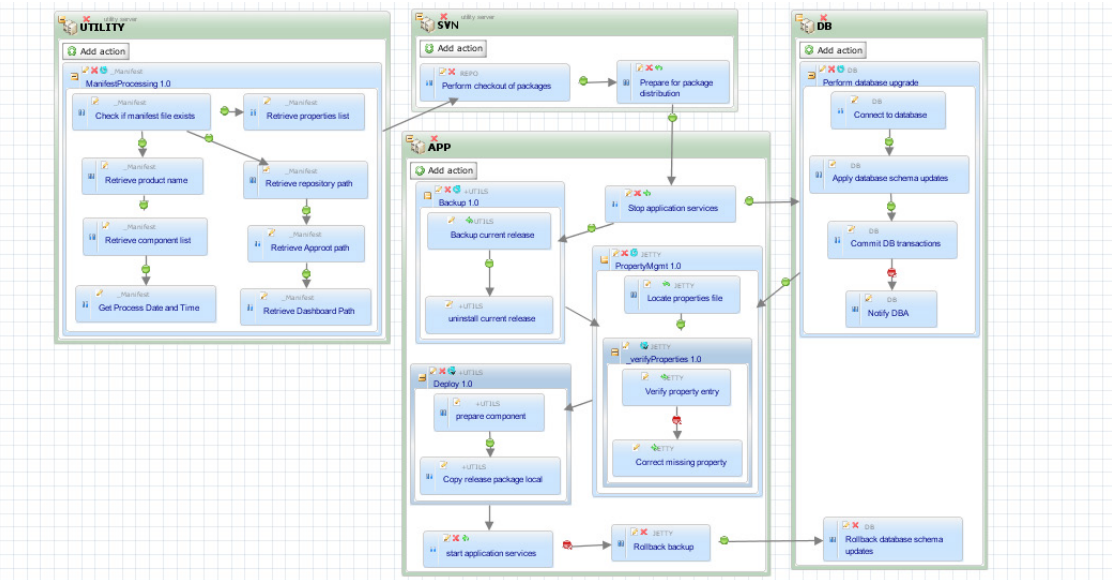
しかし、自動デプロイ・システムが包括的で、適応性があり、堅牢で、アジャイルな組織で生じる課題に十分に対応できるとしても、そのプロセスは常に調整と改善が要求されます。このため、自動デプロイ・プロセスの使用を、反復の終わりにデプロイ・ステーションだけに制限してはいけません。それどころか、本番環境へのデプロイに使用した自動化プロセスを、反復全体で開発およびテスト環境にも使用できることが必要です。この手法により、リリース日の前にデプロイ・プロセスが幅広く試行され、土壇場での問題を回避することができます。

開発チームは、アプリケーションをローカル開発ステーションにデプロイするときはいつでも自動プロセスを使うべきですし、テストチームは、テスト環境を作成するときはいつでも自動プロセスを使うべきです。こうすることで、新しいコードが自動デプロイ・プロセスに及ぼす悪影響を、比較的修正しやすい早期に発見できるようになるのです。

開発環境とテスト環境は、本番環境とは異なるため、より本番環境に近づけるための修正を施さなければなりません。この修正にはある程度のコストがかかりますが、リリースが失敗した場合のコストと比べれば少ない経費だといえるでしょう。

図 4 :

アプリケーションのデプロイ・プロセスのグラフィカル表現

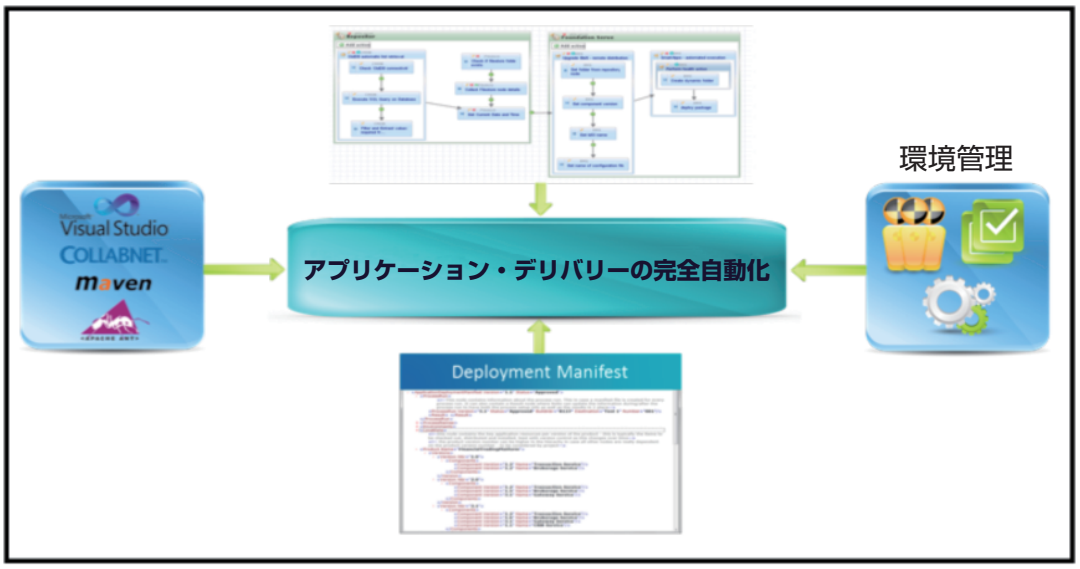


すべてに共通するデプロイ言語

ここで、自動導入システムの「所有」者は誰か?という疑問が生じます。デプロイ作業は従来は運用部門の単独責任でしたが、現在のアジャイルな組織では、ソフトウェアの製造に携わる全員がデプロイに関与することになります。開発者、テスターおよび運用スタッフが同じように、日常業務で自動デプロイ・システムを使用する必要があるため、毎日の実行や保守において、テクノロジ、用語、作業手順およびリリース戦略に関する知識を共有しなければなりません。

図 5 :

アプリケーション・デプロイのエコシステム



自動デプロイ・システムの所有権を組織全体で共有するのは、開発ライフサイクルに関与する全員がシステムにアクセスできるようにするためです。スクリプト・ベースのシステムの場合は、コーディングの技術を持つスタッフだけがデプロイ・スクリプトを開発・保守できます。これは組織の観点から見ると、システムのアクセス性が制限されることになります。しかし、専用のリリース自動化プラットフォームの場合は、スクリプトをコーディングしなくても、タスクに特有のGUIを使用してユーザが自動デプロイ・プロセスを定義、構成、保守できます。これにより、開発、テストおよび運用チームの間で共通の言語が形成され、アジャイルな開発ライフサイクルにおいてデプロイ・ステーションの統合に成功するようになるのです。

ca technologies

ca.com/jpからCA Technologiesにアクセスしてください

Agility Made Possible : CA Technologies のメリット

CA Technologies (NASDAQ:CA) は、複雑な IT 環境の管理と保護に役立つ IT 管理ソリューションを提供し、俊敏性に優れたビジネス・サービスを支援します。CA Technologies のソフトウェアと SaaS ソリューションを活用することで、データセンタからクラウドに至るまでイノベーションを加速し、インフラストラクチャを変革し、データとアイデンティティを保護できます。CA Technologies はそのテクノロジにより、お客様が必要な成果と期待どおりのビジネス・バリューを実現できるようにします。CA Technologies の詳細については、**ca.com/jp** をご覧ください。

Copyright © 2013 CA. All rights reserved. 本書に記載されているすべての商標、商号、サービス・マーク、ロゴは、該当する各社に帰属しています。本文書は、情報として使用することを目的としており、準拠法で認められる限り、CAが「現状有姿のまま」提供するものであり、いかなる種類の保証（市場性または特定の目的に対する適合性、他者の権利に対する不侵害についての黙示の保証が含まれますが、これに限定されません）も伴いません。本書に記載された情報は、幅広い開発および顧客環境における参照ソフトウェア製品からCAが得た経験に基づいています。このような開発および顧客環境におけるソフトウェア製品の過去のパフォーマンスが、同じ環境、類似した環境、または異なる環境で、かかる製品の将来のパフォーマンスを示すものではありません。また、本文書の使用が直接または間接に起因し、逸失利益、業務の中断、営業権の喪失、業務情報の損失等いかなる損失または損害が発生しても、CAは一切責任を負いません。CAがかかる損害の可能性について明示的にあらかじめ通告されていた場合も同様とします。CAは法的な助言は行わないものとします。本書または本契約で言及されるソフトウェア製品は、本契約で言及されるいかなる法律（法令、法規、規定、規則、指令、基準、政策、行政規定、行政命令、その他（「法律」と総称される）への遵守またはサードパーティとの契約上の義務に置き換わるものではありません。そのような法的または契約上の義務に関しては、資格を有する弁護士にご相談ください。

CS4080_0613