AWS ホワイトペーパー

AWS での継続的インテグレーションと継続的デリバリーの実践

AWS での継続的インテグレーションと継続的デリバリーの実践: AWS ホワイトペーパー

Table of Contents

AWS での継続的インテグレーションと継続的デリバリーの実践

公開日: 2021 年 10 月 27 日 (改訂履歴)

要約

このホワイトペーパーでは、お客様のソフトウェア開発環境において継続的インテグレーションと継続的デリバリー (CI/CD)、およびアマゾン ウェブ サービス (AWS) のツールを使用することの特徴と利点について説明します。継続的インテグレーションおよび継続的デリバリーは、DevOps のベストプラクティスであり、DevOps イニシアチブの重要なパートです。

ソフトウェアデリバリーの課題

今日のエンタープライズは、急激に変化する競争環境、進化し続けるセキュリティ要件、およびパフォーマンスとスケーラビリティの確保という課題に直面しています。エンタープライズは、オペレーションの安定性と迅速な機能開発のギャップを埋める必要があります。継続的インテグレーションと継続的デリバリー (CI/CD) は、システムの安定性とセキュリティを維持しながら、迅速なソフトウェアの変更を可能にするプラクティスです。

Amazon は、Amazon.com のリテールの顧客、Amazon の子会社、アマゾン ウェブ サービス (AWS) が新しく革新的なソフトウェアのデリバリー方法を必要とするであろう、機能のデリバリーに関するビジネスの要求について、早い時期に認識しました。Amazon のような規模では、迅速、安全かつ確実に、さらに機能停止を許容せずにソフトウェアをデリバリーするために、何千もの独立したソフトウェアチームが並行して作業しなければなりません。

ソフトウェアを高いスピードでデリバリーする方法を学ぶことによって、Amazon やその他の先進的な組織は DevOps の先駆者となりました。DevOps は、組織が迅速にアプリケーションとサービスを提供する能力を高める、文化的な考え方、プラクティス、およびツールの組み合わせです。DevOps の原則を使用することで、組織は、従来型のソフトウェア開発およびインフラストラクチャ管理のプロセスを使用する組織よりも速いペースで製品を進化させ、改善することができます。組織はこの迅速さのために顧客をより良く支え、市場での競合に勝ち抜くことができます。

「2 枚のピザ」チームやマイクロサービス/サービス指向アーキテクチャ (SOA) などの DevOps の原則のいくつかは、このホワイトペーパーでは取り扱っていません。このホワイトペーパーでは、Amazon が築き上げ、継続的に改善している CI/CD の機能について説明します。CI/CD は、ソフトウェア機能を迅速かつ期待通りにデリバリーするための鍵です。

現在 AWS は、AWS CodeStar、AWS CodeCommit、AWS CodePipeline、AWS CodeBuild、AWS CodeDeploy、AWS CodeArtifact などの CI/CD 機能を一連のデベロッパー向けサービスとして提供しています。DevOps を実践するデベロッパーや IT 運用のプロフェッショナルは、これらのサービスを使用して迅速、安全かつ確実にソフトウェアをデリバリーできます。これらを組み合わせることで、アプリケーションのソースコードの安全な保存とバージョン管理を行うことができます。AWS CodeStar を使用することで、これらのサービスを利用してエンドツーエンドのソフトウェアのリリースワークフローを迅速にオーケストレートすることができます。既存の環境に対しては、AWS CodePipeline が既存のツールによる独立したサービスを統合できる柔軟性を持ちます。これらは、他の AWS のサービスと同様に、AWS マネジメントコンソール、AWS アプリケーションプログラミングインターフェイス (API)、AWS ソフトウェア開発ツールキット (SDK) を通じてアクセスできる、可用性が高く、簡単に統合できるサービスです。

継続的インテグレーションと継続的デリバリー/デプロイとは

このセクションでは、継続的インテグレーションと継続的デリバリーの実践方法、そして継続的デリバリーと継続的デプロイの違いについて説明します。

継続的インテグレーション

継続的インテグレーション (CI) とは、デベロッパーが定期的にコードに対する変更をセントラルリポジトリにマージし、その後自動化されたビルドとテストが実行されるソフトウェア開発手法です。CI は、ほとんどの場合ソフトウェアリリースプロセスのビルドまたはインテグレーションステージを指し、オートメーション要素 (CI やビルドサービスなど) と文化的要素 (頻繁に統合することを学ぶなど) の両方を必要とします。CI の主な目標は、バグを早期に発見して対処し、ソフトウェアの品質を向上し、新しいソフトウェアアップデートの検証とリリースにかかる時間を短縮することです。

継続的インテグレーションでは、より小さなコミットと、統合するためのより小さなコードの変更に焦点を当てます。デベロッパーは、定期的な間隔で、少なくとも 1 日に 1 回コードをコミットします。デベロッパーは、コードリポジトリからコードをプルし、ビルドサーバーにプッシュする前に、ローカルホスト上でコードがマージできることを確認します。このステージでは、ビルドサーバーで様々なテストを実行し、コードのコミットを許可または拒否します。

CI を実装する際の基本的な課題には、共通のコードベースへのより頻繁なコミット、単一のソースコードリポジトリの維持、ビルドの自動化、テストの自動化などがあります。さらに、本稼働環境に類似した環境でのテスト、チームへのプロセスの可視性の提供、デベロッパーがアプリケーションのすべてのバージョンを容易に取得できるようにすることなどの課題があります。

継続的デリバリーおよびデプロイ

継続的デリバリー (CD) とは、コードの変更によって自動的なビルド、テスト、本稼働環境へのリリースの準備を実施するソフトウェア開発のプラクティスです。ビルドステージが完了した後でテスト環境または本稼働環境、あるいはその両方にすべてのコード変更をデプロイすることで、継続的インテグレーションを拡張します。継続的デリバリーは、ワークフロープロセスを使用して完全に自動化するか、重要なポイントで手動ステップを使用して部分的に自動化することができます。継続的デリバリーが適切に実装されている場合、デベロッパーは、標準化されたテストプロセスをパスしたデプロイ可能なビルドアーティファクトを常に手元に持つことになります。

継続的デプロイを使用すると、デベロッパーからの明示的な承認なしでリビジョンが本稼働環境にデプロイされ、ソフトウェアリリースプロセス全体が自動化されます。これにより、製品ライフサイクルの早い段階で継続的なフィードバックループを実現することができます。

継続的デリバリーは継続的デプロイとは異なる

継続的デリバリーに関する誤解の 1 つとして、コミットされたすべての変更が自動テストをパスした直後に本稼働環境に適用されるということがあります。しかし、継続的デリバリーの主眼は、すべての変更を即座に本稼働環境に適用することではなく、すべての変更を本稼働環境に適用する準備が整った状態にすることです。

本稼働環境に変更をデプロイする前に、本番デプロイを承認し、監査することを保証する意思決定プロセスを実装します。この決定は人が行い、その後ツールを使用して実行することができます。

継続的デリバリーを使用することで、稼働の決定はビジネス上の決定となり、技術的な決定ではなくなります。技術的な検証はコミットごとに行われます。

本稼働環境への変更のロールアウトは、破壊的なイベントではありません。デプロイは、技術チームが次の一連の変更作業を中断することなく、プロジェクト計画、引き渡しドキュメント、またはメンテナンスウィンドウも必要としません。デプロイは、テスト環境で複数回実行され、検証される反復可能なプロセスになります。

継続的デリバリーの利点

CD は、プロセスの自動化、デベロッパーの生産性の向上、コード品質の向上、および顧客へのアップデートデリバリーの迅速化など、多数の利点を提供します。

ソフトウェアリリースプロセスの自動化

CD は、自動的に構築し、テストし、本稼働リリース向け準備したコードをチェックインする方法をチームに提供します。これにより、ソフトウェアデリバリーが効率的で回復性があり、迅速かつ安全なものになります。

デベロッパーの生産性の向上

CD のプラクティスは、デベロッパーをマニュアル作業から解放し、複雑な依存関係を解消し、焦点をソフトウェアの新機能の提供へと戻すことで、チームの生産性を向上させます。デベロッパーは、コードをビジネスの他のパートと統合し、どのようにコードをプラットフォームにデプロイするかに時間を費やすのではなく、必要な機能を提供するロジックのコーディングに集中することができます。

コード品質の向上

CD は、バグが後になって大きな問題に発展する前に、デリバリープロセスの早い段階でバグを発見して対処するのに役立ちます。プロセス全体が自動化されているので、チームは追加的なコードテストを簡単に実行できます。より多くのテストをより頻繁に行う手法により、チームは変更の影響に関する即時のフィードバックを得て迅速に反復することができます。これにより、チームは高い安定性とセキュリティを確保してコードの品質を向上することができます。デベロッパーは、即座のフィードバックを通じて、新しいコードが動作するかどうか、破壊的な変更やバグがもたらされていないかを知ることができます。開発プロセスの早い段階で発見されたミスは、修正が最も容易です。

更新のデリバリーの迅速化

CD は、チームが迅速かつ頻繁に更新を顧客にデリバリーするのに役立ちます。CI/CD を実装すると、機能やバグ修正のリリースを含むチーム全体の速度が向上します。エンタープライズは、市場の変化、セキュリティの課題、顧客のニーズ、およびコストのプレッシャーに対してより迅速に対応することができます。例えば、新しいセキュリティ機能が必要な場合、テストチームは、自動テス

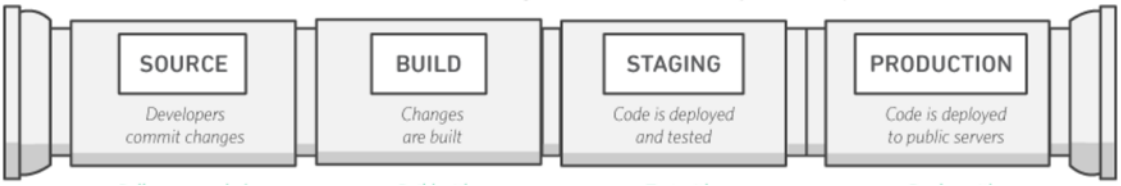
トを備えた CI/CD を実装し、高い信頼性を持って修正を迅速かつ確実に本稼働システムに導入できます。これまで数週間から数ヶ月かかっていたことが、数日または数時間で実行できるようになります。

継続的インテグレーションと継続的デリバリーの実装

このセクションでは、お客様の組織で CI/CD モデルの実装を開始する方法について説明します。このホワイトペーパーでは、成熟した DevOps およびクラウド移行モデルを使用している組織が CI/CD パイプラインを構築して使用する方法については説明していません。DevOps の道のりを支援するために、AWS にはリソースとツールを提供できる数多くの認定 DevOps パートナーがいます。AWS クラウドへの移行の準備に関する詳細については、『クラウド運用モデルの構築』を参照してください。

継続的インテグレーションと継続的デリバリーへの道筋

CI/CD は、パイプラインとして描くことができます (下の図を参照)。ここでは、一方の端で新しいコードが引き渡され、一連のステージ (ソース、ビルド、ステージング、本稼働) でテストされた後、本稼働環境で使用できるコードとして公開されます。組織が初めて CI/CD を使用する場合、このパイプラインに反復的な方法でアプローチできます。つまり、小さく始めて各ステージで反復することです。これにより、組織の成長に役立つ方法でコードを理解し開発できるようになります。



CI/CD パイプライン

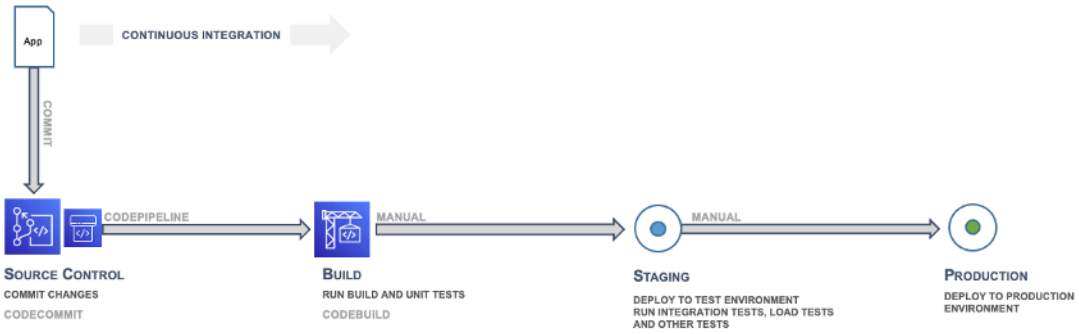
CI/CD パイプラインの各ステージは、デリバリープロセスの論理ユニットとして構造化されています。さらに、各ステージは、コードの特定の側面を吟味するゲートとして機能します。コードがパイプラインを進むにつれて、より多くの側面が検証され続けるため、後のステージではコードの品質が高くなると想定されます。早いステージで問題が明らかになると、そのコードはパイプライン上の進行が停止します。テストの結果は即座にチームに送信され、ソフトウェアがステージをパスしなければ、それ以降のビルドやリリースはすべて停止されます。

これらのステージは提案です。ビジネスニーズに基づいてステージを適応させることができます。一部のステージは、複数のタイプのテスト、セキュリティ、およびパフォーマンスのために繰り返すことができます。プロジェクトの複雑さやチームの構造によって、一部のステージを異なるレベルで複数回繰り返すことができます。例えば、あるチームの最終成果物に、次のチームのプロジェクトが依存する場合があります。つまり、最初のチームの最終製品は、続いて次のチームのプロジェクトでアーティファクトとしてステージングされます。

CI/CD パイプラインの存在は、組織能力の成熟に大きな影響を与えます。組織は、開始時点から複数の環境、多くのテストフェーズ、およびすべてのステージでのオートメーションを備えた完全に成熟したパイプラインを構築しようとするのではなく、小さなステップから開始する必要があります。高度に成熟した CI/CD 環境を持つ組織であっても、パイプラインを継続的に改善する必要があることに留意してください。

CI/CD に対応した組織の構築は長い道のりであり、その道には多くの目的地があります。次のセクションでは、継続的インテグレーションから始まり継続的デリバリーのレベルに至るまで、組織が取り得る道筋について説明します。

継続的インテグレーション



継続的インテグレーション — ソースとビルド

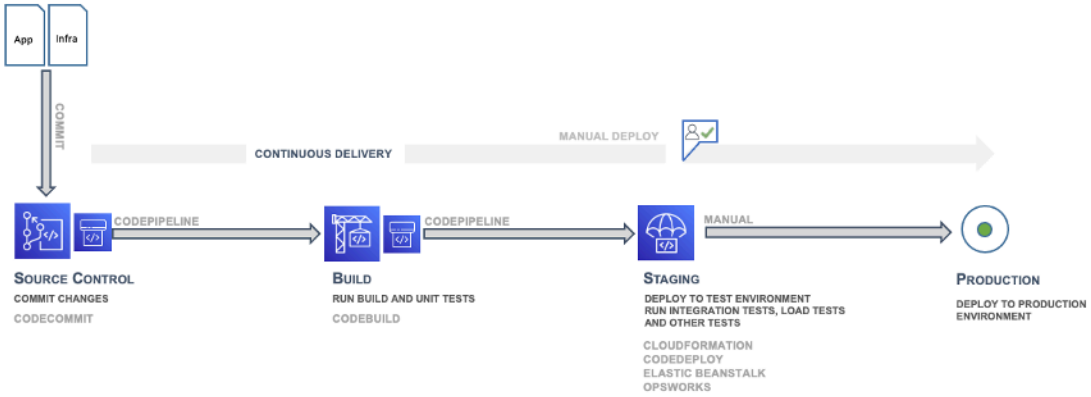
CI/CD の 道のりにおける最初のフェーズは、継続的インテグレーションでの成熟度を高めることです。すべてのデベロッパーが定期的にコードをセントラルリポジトリ (CodeCommit や GitHub にホストされているもの) にコミットし、すべての変更をアプリケーションのリリースブランチにマージしていることを確認する必要があります。デベロッパーは、単独でコードを保持するべきではありません。一定期間だけ機能ブランチが必要な場合、可能な限り頻繁に上流からマージして最新の状態に維持する必要があります。完全な作業単位での頻繁なコミットとマージは、チームが規律を確立するために推奨され、プロセスによって促進されます。コードを早期かつ頻繁にマージすれば、後に統合の問題が発生する可能性が少なくなります。

また、アプリケーションのテストユニットをできる限り早く作成し、コードをセントラルリポジトリにプッシュする前にこれらのテストを実行するようデベロッパーを促す必要があります。ソフトウェア開発プロセスの早い段階で発見されたエラーは、修正が最も低コストで容易です。

コードがソースコードリポジトリのブランチにプッシュされると、そのブランチを監視するワークフローエンジンがビルダーツールにコマンドを送信し、コードを構築し、制御された環境でユニットテストを実行します。ビルドプロセスは、迅速なフィードバックのために、コミットステージで発生す

る可能性があるプッシュやテストを含む、すべてのアクティビティを処理するために適切なサイズにする必要があります。ユニットテストカバレッジ、スタイルチェック、静的分析などのその他の品質チェックもこのステージで実行できます。最後に、ビルダーツールは、1 つまたは複数のバイナリビルドおよびアプリケーション用のイメージ、スタイルシート、ドキュメントなどの他のアーティファクトを作成します。

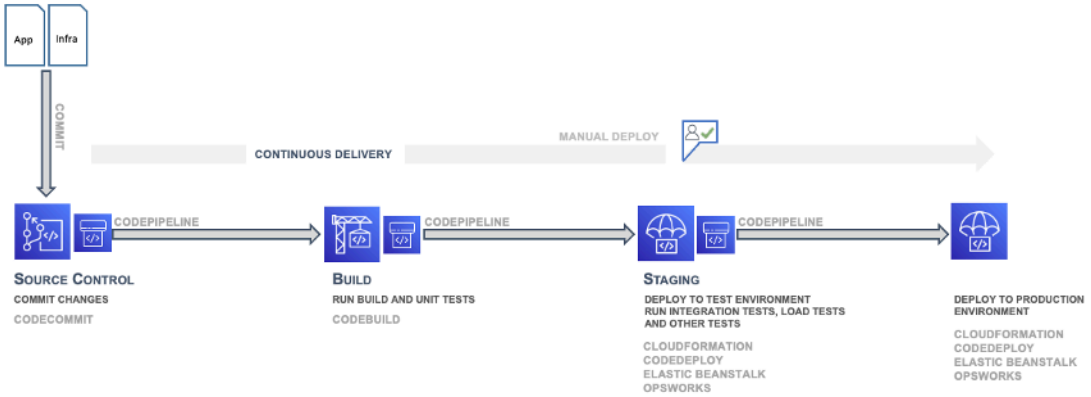
継続的デリバリー: ステージング環境の作成



継続的デリバリー — ステージング

継続的デリバリー (CD) は、次のフェーズであり、ステージング環境へのアプリケーションコードのデプロイを伴います。ステージング環境は、本稼働スタックのレプリカであり、より機能的なテストを実行します。ステージング環境は、テスト用に事前準備した静的環境を使用するか、アプリケーションコードのテストおよびデプロイ用にコミットされたインフラストラクチャと設定コードを使用して動的な環境をプロビジョンし設定することもできます。

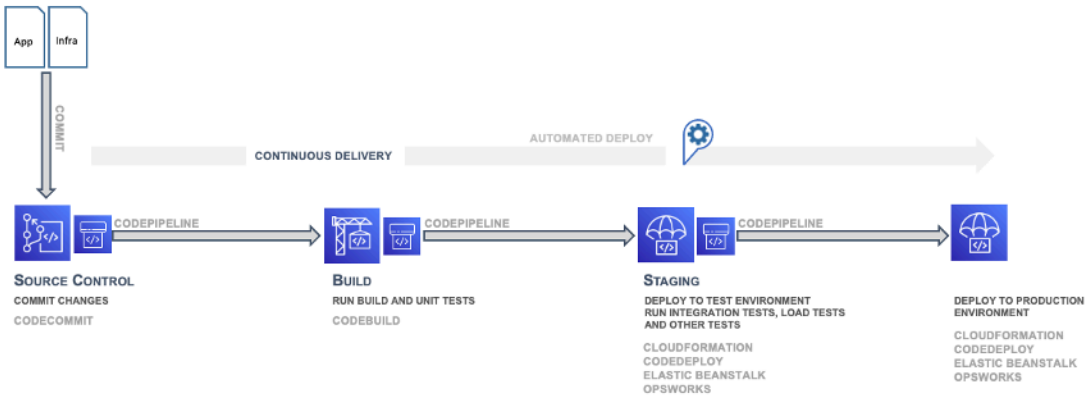
継続的デリバリー : 本稼働環境の作成



継続的デリバリー — 本稼働

デプロイ/デリバリーのパイプラインシーケンスでは、ステージング環境の後に本稼働環境があり、これも Infrastructure as Code (IaC) を使用して構築されます。

継続的デプロイ



継続的デプロイ

CI/CD デプロイパイプラインの最後のフェーズは、継続的デプロイです。これは、本稼働環境へのデプロイなどのソフトウェアリリースプロセス全体の完全なオートメーションを含む場合があります。完全に成熟した CI/CD 環境では、本稼働環境への道筋は完全に自動化され、コードを高い信頼性でデプロイすることが可能になります。

成熟とその先

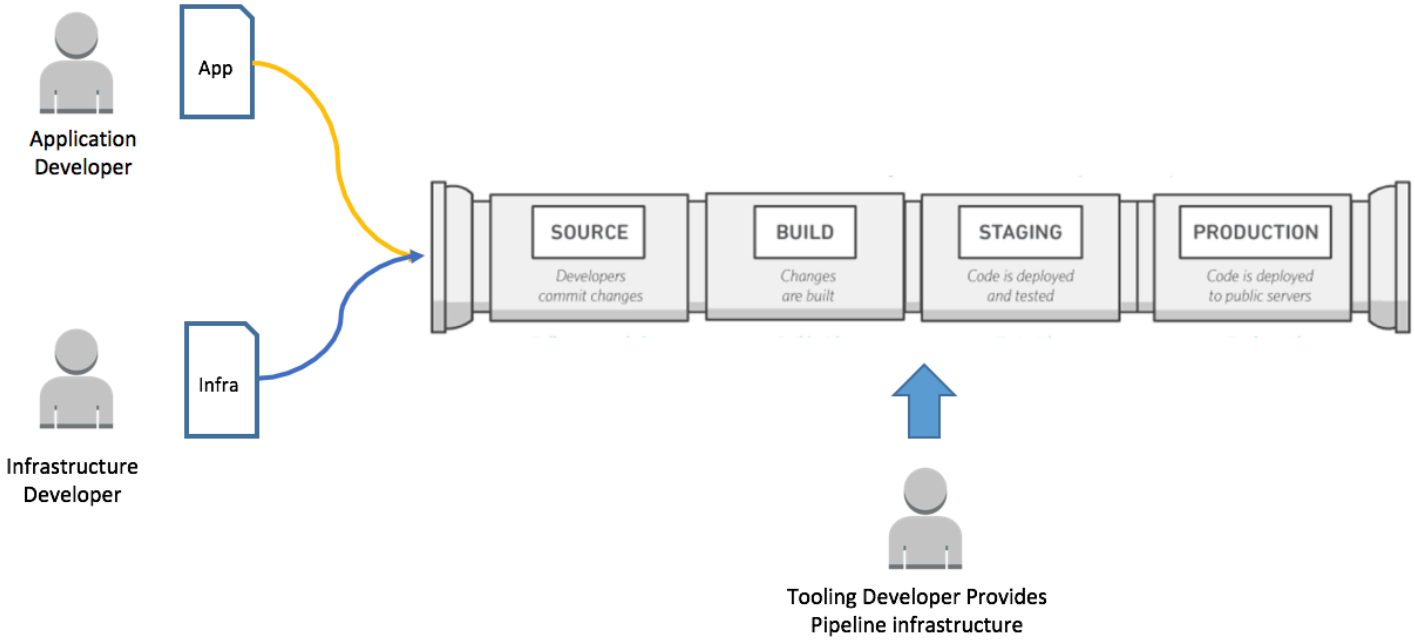
組織が成熟すると、CI/CD モデルの発展が継続し、以下の改善点のより多くが盛り込まれます。

- 特定のパフォーマンス、コンプライアンス、セキュリティ、およびユーザーインターフェイス (UI) テストのためのステージング環境の追加
- アプリケーションコードに加え、インフラストラクチャおよび設定コードのユニットテスト
- コードレビュー、問題の追跡、イベント通知などの他のシステムやプロセスとの統合
- データベーススキーマの移行との統合 (該当する場合)
- 監査およびビジネス上の承認のための追加ステップ

複雑な複数環境の CI/CD パイプラインを持つ最も成熟した組織であっても、改善を求め続けています。DevOps は目的地ではなく、道のりです。パイプラインに関するフィードバックは継続的に収集され、スピード、スケール、セキュリティ、信頼性の改善は、開発チームのさまざまな部分の間のコラボレーションとして実現されます。

チーム

AWS では、CI/CD 環境を実装するために、アプリケーションチーム、インフラストラクチャチーム、ツールチームの 3 つのデベロッパーチームを編成することを推奨しています (下の図を参照)。この組織は、急速に変化するスタートアップ、大規模なエンタープライズ組織、および Amazon 自身で構築され、適用されてきた一連のベストプラクティスを示しています。チームは、2 枚のピザが十分な食事となるグループ、すなわち約 10 ~ 12 人よりも大きくするべきではありません。これは、グループサイズが拡大し、コミュニケーション経路が増加するにつれて有意義な会話が限界に達するというコミュニケーションのルールに従っています。



アプリケーション、インフラストラクチャ、およびツールチーム

アプリケーションチーム

アプリケーションチームは、アプリケーションを作成します。アプリケーションデベロッパーは、バックログ、ストーリー、およびユニットテストを担当し、指定されたアプリケーションターゲットに基づいて機能を開発します。このチームの組織上の目標は、デベロッパーがコア以外のアプリケーションタスクに費やす時間を最小限に抑えることです。

アプリケーションチームは、アプリケーション言語における機能的なプログラミングスキルを持つだけでなく、プラットフォームスキルとシステム構成の理解を持ち合わせる必要があります。これにより、チームは機能の開発およびアプリケーションの強化のみに集中することができます。

インフラストラクチャチーム

インフラストラクチャチームは、アプリケーションの実行に必要なインフラストラクチャを作成および構成するコードを記述します。このチームは、AWS CloudFormation などの ネイティブ AWS ツール、または Chef、Puppet、Ansible などの汎用ツールを使用することがあります。インフラストラクチャチームは、必要なリソースの指定を担当し、アプリケーションチームと密接に連携します。インフラストラクチャチームは、小規模なアプリケーションでは 1 人または 2 人 のみで構成されることがあります。

チームには、AWS CloudFormation や HashiCorp Terraform などのインフラストラクチャプロビジョニング手法に関するスキルが必要です。また、チームは、Chef、Ansible、Puppet、Salt などのツールを使用した構成オートメーションのスキルを高める必要もあります。

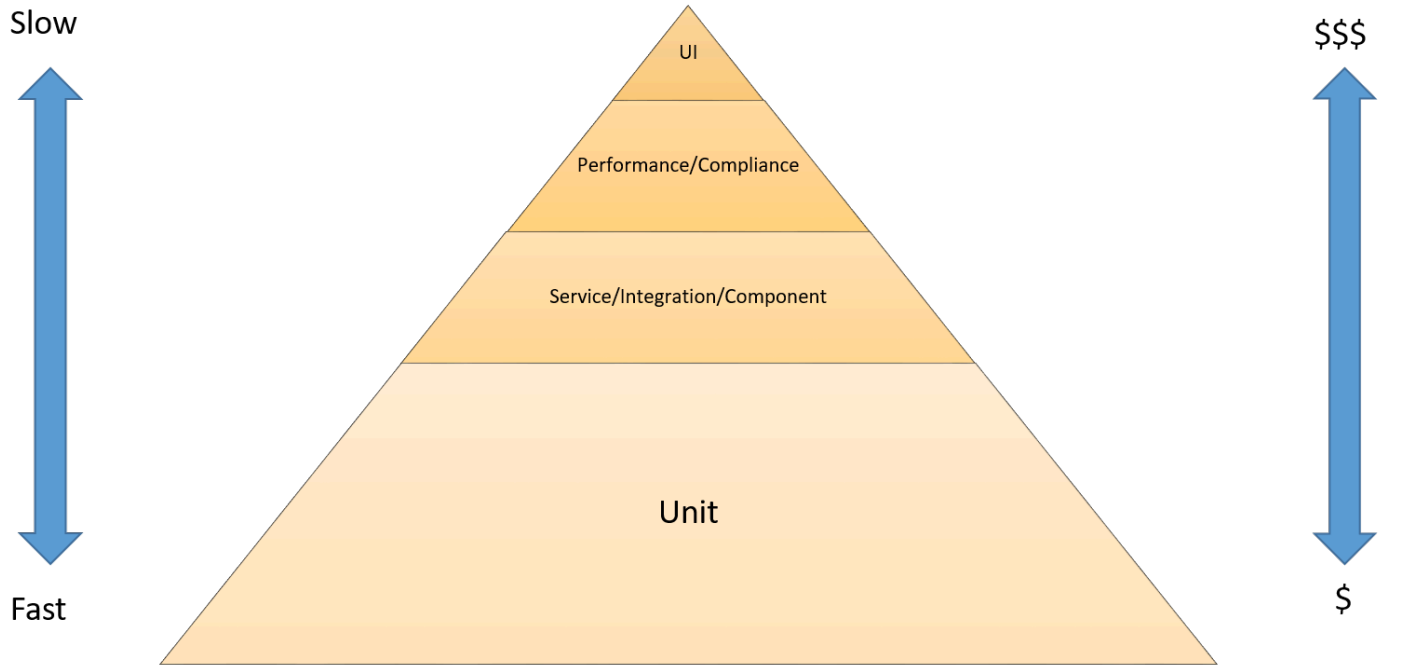
ツールチーム

ツールチームは、CI/CD パイプラインを構築および管理します。このチームは、パイプラインを構成するインフラストラクチャとツールを担当します。このチームは「2 枚のピザ」チームの一部ではありませんが、組織内のアプリケーションおよびインフラストラクチャチームによって使用されるツールを作成します。組織は、ツールチームがアプリケーションチームおよびインフラストラクチャチームの成熟の一歩先を行くように、ツールチームを継続的に成熟させる必要があります。

ツールチームは、CI/CD パイプラインのすべてのパートを構築して統合するためのスキルを備えなければなりません。これには、ソース管理リポジトリ、ワークフローエンジン、ビルド環境、テストフレームワーク、およびアーティファクトリポジトリの構築が含まれます。このチームは、AWS CodeStar、AWS CodePipeline、AWS CodeCommit、AWS CodeDeploy、AWS CodeBuild、AWS CodeArtifact などのソフトウェアに加え、Jenkins、GitHub、Artifactory、TeamCity およびその他の類似ツールを実装することも選択できます。これを DevOps チームと呼ぶ組織もありますが、AWS はこれを推奨せず、DevOps をソフトウェアデリバリーにおける人材、プロセス、ツールの総体として考えるよう推奨しています。

継続的インテグレーションと継続的デリバリーにおけるテストステージ

3 つの CI/CD チームは、CI/CD パイプラインの様々なステージでソフトウェア開発ライフサイクルにテストを組み込む必要があります。全体として、テストはできるだけ早期に開始する必要があります。以下に示すテストのピラミッドは、Succeeding with Agile で Mike Cohn により提示されている概念です。これは、テストのコストとスピードとの関係において、さまざまなソフトウェアテストを示しています。



CI/CD テストのピラミッド

ユニットテストは、ピラミッドの底にあります。これらは、実行が最も高速で、最も低コストです。したがって、ユニットテストはテスト戦略の大半を占める必要があります。目安としては、約 70% です。このフェーズで発見されたバグは迅速かつ安価に修正できるため、ユニットテストにはほぼ完全なコードカバレッジが必要です。

サービス、コンポーネント、および統合テストは、ピラミッドでユニットテストの上に位置します。これらのテストには詳細な環境が必要なため、インフラストラクチャ要件においてコストが高くなり、実行が遅くなります。次のレベルは、パフォーマンスとコンプライアンスのテストです。これらは、本稼働品質の環境を必要とし、さらにコストが高くなります。UI およびユーザー受け入れテストは、ピラミッドの最上部にあり、同様に本稼働品質の環境を必要とします。

これらのすべてのテストは、高品質なソフトウェアを保証するための完全な戦略の一部です。しかし、開発スピードのためには、ピラミッドの下半分のテストの数とカバレッジが重要視されます。

以下のセクションでは、CI/CD のステージについて説明します。

ソースのセットアップ

プロジェクトの開始時には、未処理のコードと設定、およびスキーマの変更を保存できるソースをセットアップすることが不可欠です。ソースステージでは、GitHub または AWS CodeCommit などでホストされているソースコードリポジトリを選択します。

ビルドの設定と実行

ビルドのオートメーションは、CI プロセスにとって不可欠です。ビルドのオートメーションを設定する際、最初のタスクは適切なビルドツールを選択することです。次のようなビルドツールが多数あります。

- Ant for Java、Maven for Java、Gradle for Java
- Make for C/C++
- Grunt for JavaScript
- Rake for Ruby

お客様にとって最適なビルドツールは、プロジェクトのプログラミング言語やチームのスキルセットによって異なります。ビルドツールを選択したら、すべての依存関係をビルドスクリプトおよびビルドステップに明確に定義する必要があります。また、最終的なビルドアーティファクトのバージョンを作成することもベストプラクティスです。これにより、デプロイおよび問題の追跡が容易になります。

構築

ビルドステージでは、ビルドツールはソースコードリポジトリへの変更を入力として受け取り、ソフトウェアをビルドし、次のタイプのテストを実行します。

ユニットテスト – コードの特定のセクションをテストして、期待されていることをコードが実行することを確認します。ユニットテストは、開発フェーズでソフトウェアデベロッパーが実行します。このステージでは、静的コード分析、データフロー分析、コードカバレッジ、およびその他のソフトウェア検証プロセスを適用することができます。

静的コード分析 – このテストは、ビルドおよびユニットテストの後にアプリケーションを実際に実行することなく実施されます。この分析は、コーディングエラーおよびセキュリティホールを見つけるのに役立ち、コーディングガイドラインへの準拠を確認することもできます。

ステージング

ステージングフェーズでは、最終的な本稼働環境を反映した完全な環境が作成されます。以下のテストが実行されます:

統合テスト – ソフトウェア設計に対してコンポーネント間のインターフェイスを検証します。統合テストは、反復的なプロセスであり、堅牢なインターフェイスおよびシステムの整合性の構築を容易にします。

コンポーネントテスト – さまざまなコンポーネント間のメッセージの受け渡しとその結果をテストします。このテストの主な目的は、コンポーネントテストにおける冪等性を確保することである場合があります。テストには、非常に大きなデータボリューム、または境界条件や異常な入力が含まれる可能性があります。

システムテスト – システムテストをエンドツーエンドでテストし、ソフトウェアがビジネス要件を満たしているかどうかを確認します。このテストには、ユーザーインターフェイス (UI)、API、バックエンドロジック、および終了状態のテストが含まれる場合があります。

パフォーマンステスト – 特定のワークロードで実行する際のシステムの応答性と安定性を判定します。パフォーマンステストは、スケーラビリティ、信頼性、リソース使用状況など、システムのその他の品質属性を調査、測定、または検証するためにも使用されます。パフォーマンステストのタイプには、ロードテスト、ストレステスト、スパイクテストなどが含まれます。パフォーマンステストは、事前定義された基準に対するベンチマークに使用されます。

コンプライアンステスト – コード変更が機能以外の仕様や規定の要件に準拠しているかどうかを確認します。これにより、定義済みの基準を実装し満たしているかどうかを判定します。

ユーザー受け入れテスト – エンドツーエンドのビジネスフローを検証します。このテストは、ステージング環境でエンドユーザーにより実行され、システムが要件仕様書の要件を満たしているかどうかを確認します。通常、お客様はこのステージにアルファテストおよびベータテストの方法論を採用しています。

本稼働

最後に、これまでのテストに合格した後に、本稼働環境でステージングフェーズが繰り返されます。このフェーズでは、コードを本稼働環境全体にデプロイする前に、新しいコードをサーバーの小さなサブセット、あるいは 1 つのサーバーや 1 つの AWS リージョン のみにデプロイすることで最終的な canary テストを完了することができます。本稼働環境に安全にデプロイする方法の詳細については、「デプロイ方法」のセクションで説明しています。

次のセクションでは、これらのステージとテストを組み込むためにパイプラインを構築する方法について説明します。

パイプラインの構築

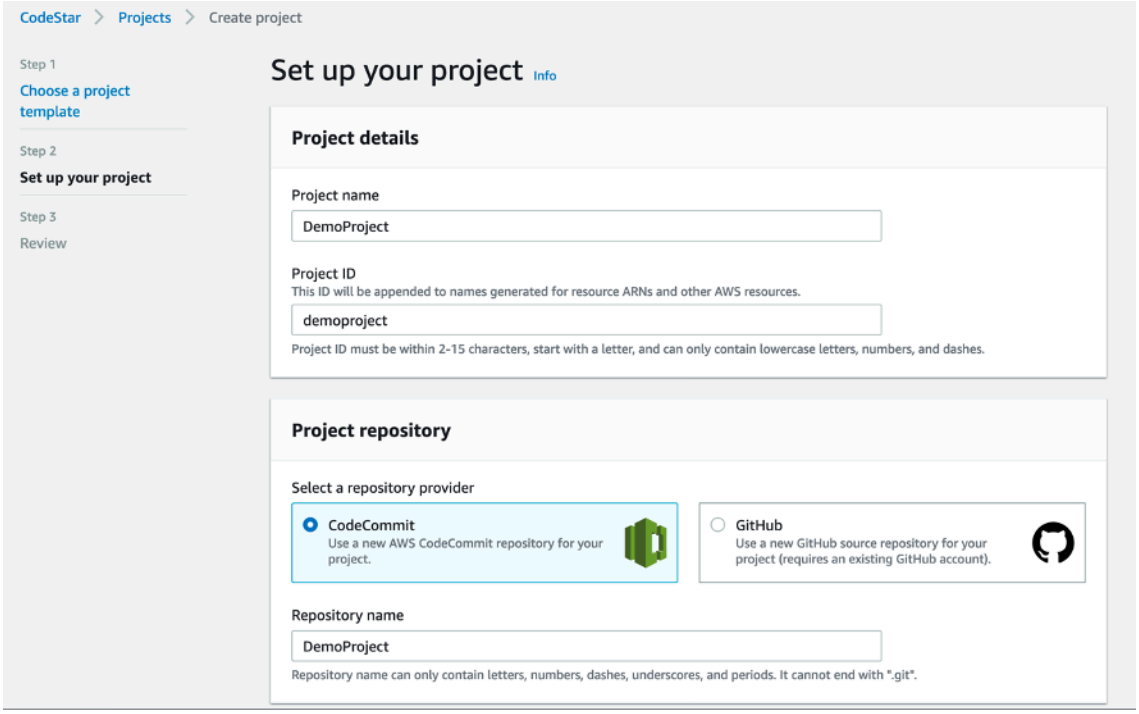
このセクションでは、パイプラインの構築について説明します。まず、CI に必要なコンポーネントだけでパイプラインを確立し、その後より多くのコンポーネントとステージを使用した継続的デリバリーパイプラインに移行します。また、このセクションでは、大規模なプロジェクト向けに AWS

Lambda 関数と手動承認を使用する方法、複数のチーム、ブランチ、および AWS リージョン の計画方法についても説明します。

継続的インテグレーションのための実用最小限のパイプラインから開始する

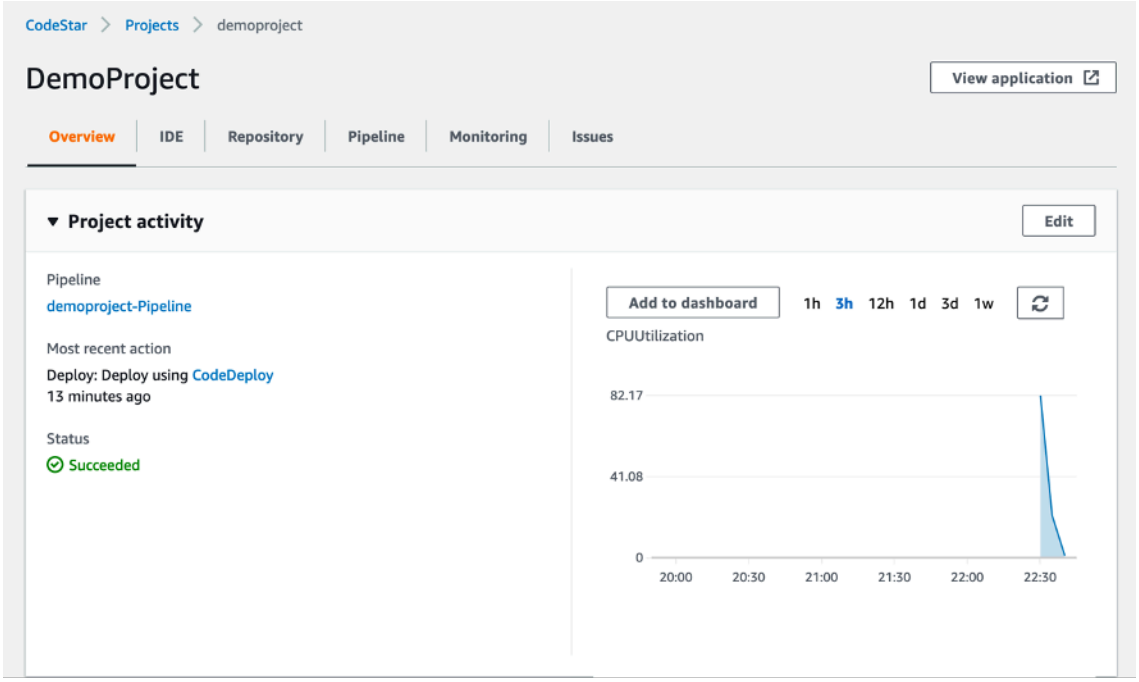
継続的デリバリーへ向けた組織の歩みは、実用最小限のパイプライン (MVP) から始まります。「継続的インテグレーションと継続的デリバリーの実装」で説明したように、チームは、コードスタイルのチェックやデプロイのない単一のユニットテストを実行するパイプラインの実装などの非常にシンプルなプロセスから開始することができます。

主なコンポーネントは、継続的デリバリーのオーケストレーションツールです。このパイプラインの構築を支援するために、Amazon は AWS CodeStar を開発しました。



AWS CodeStar 設定ページ

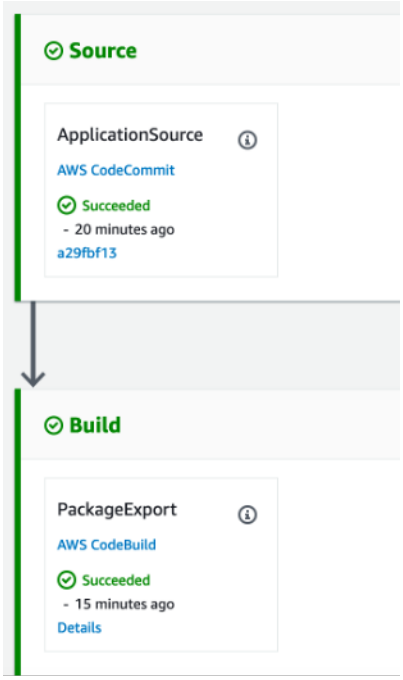
AWS CodeStar では、AWS CodePipeline、AWS CodeBuild、AWS CodeCommit、AWS CodeDeploy を使用して、セットアッププロセス、ツール、テンプレート、ダッシュボードが統合されています。AWS CodeStar は、AWS でアプリケーションを迅速に開発、構築、およびデプロイするために必要なすべてを提供します。これにより、お客様のコードのリリースの高速化が可能になります。既に AWS マネジメントコンソールに精通し、より高いレベルの制御を求めるお客様は、最適なデベロッパーツールを手動で設定し、必要に応じて個々の AWS サービスをプロビジョンできます。



AWS CodeStar ダッシュボード

AWS CodePipeline は、迅速かつ信頼性の高いアプリケーションおよびインフラストラクチャの更新のために、AWS CodeStar または AWS マネジメントコンソールを通じて使用できる CI/CD サービスです。AWS CodePipeline は、お客様の定義するリリースプロセスモデルに基づいて、コードの変更があるたびに、コードの構築、テスト、デプロイを実施します。これにより、機能と更新をすばやく、信頼性の高い方法でデリバリーできます。GitHub などの広く利用されているサードパーティーサービス用の構築済みプラグインを使用したり、独自のカスタムプラグインをリリースプロセスの任意のステージに統合したりすることで、エンドツーエンドソリューションを簡単に構築できます。AWS CodePipeline では、実際に使用した分に対してのみお支払いいただきます。前払い金や長期契約はありません。

AWS CodeStar と AWS CodePipeline のステップは、ソース、ビルド、ステージング、および本稼働の CI/CD ステージに直接マッピングされます。継続的デリバリーが望ましいですが、ソースレポジトリをチェックしてビルドアクションを実行する単純な 2 ステップパイプラインから開始することができます。

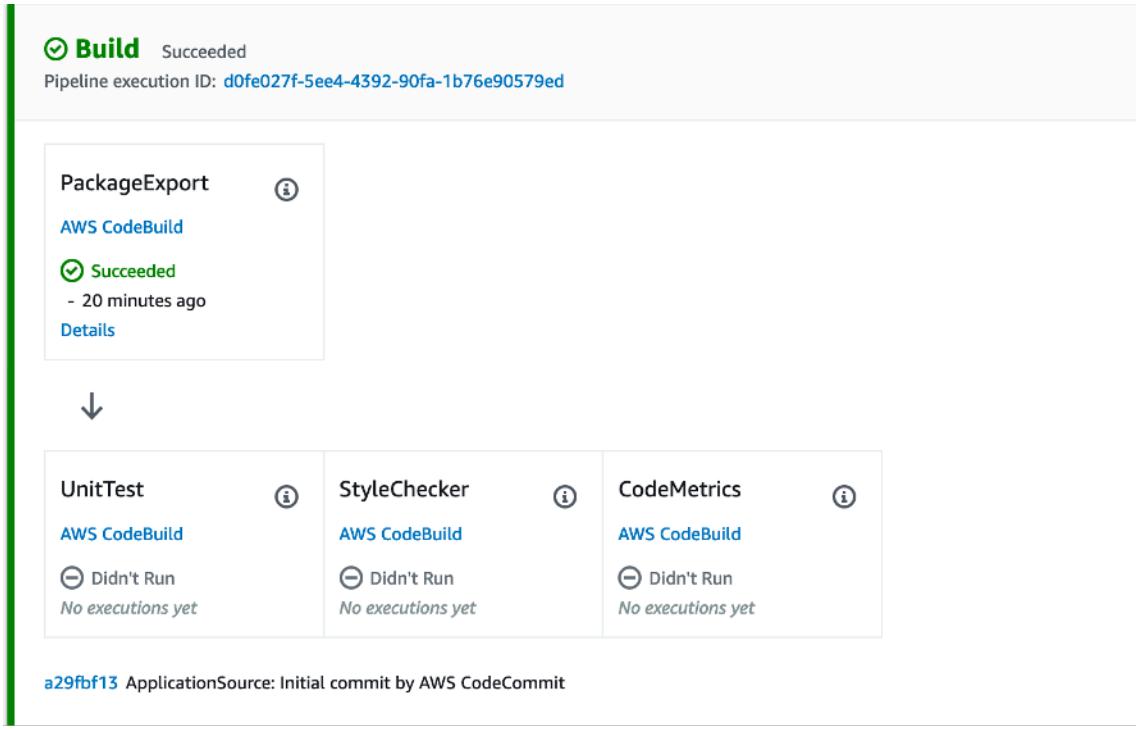


AWS CodePipeline — ソースステージとビルドステージ

AWS CodePipeline では、ソースステージは GitHub、AWS CodeCommit、および Amazon Simple Storage Service (Amazon S3) からの入力を受け入れることができます。ビルドプロセスの自動化は、継続的デリバリーを実装して継続的デプロイに移行するための重要な最初のステップです。ビルドアーティファクト作成への人的関与を排除することにより、チームの負担を軽減し、手作業のパッケージングによるエラーを最小限に抑え、消費可能なアーティファクトのより頻繁なパッケージ化を開始できます。

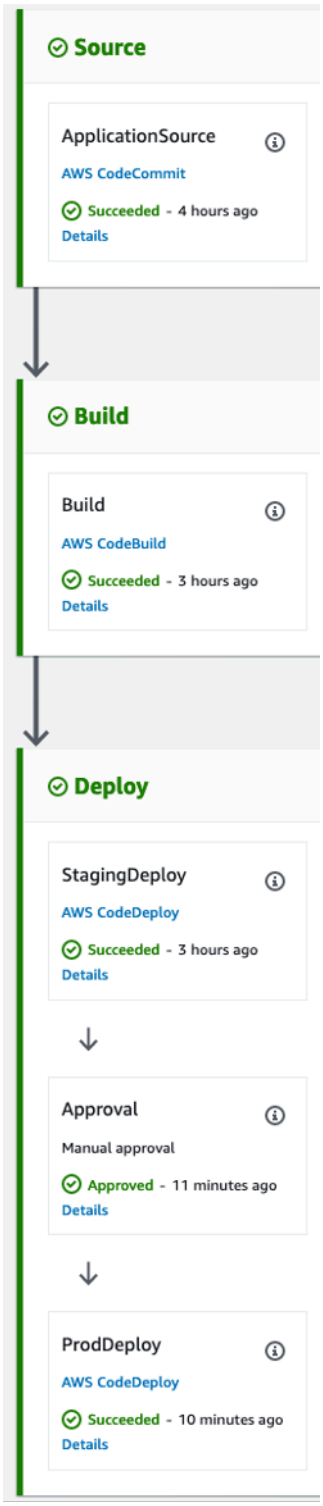
AWS CodePipeline は、フルマネージド型ビルドサービスである AWS CodeBuild とシームレスに連携するため、コードをパッケージ化しユニットテストを実行するパイプラインに容易にビルドステップを設定できます。AWS CodeBuild により、ビルドサーバーのプロビジョン、管理、またはスケールが不要になります。AWS CodeBuild では、スケールが継続的に行われ、複数のビルドが同時に実行されます。ビルドの実行までキューを待つ必要はありません。AWS CodePipeline は、Jenkins、Solano CI、TeamCity などのビルドサーバーとも統合されています。

例えば、以下のビルドステージでは、3 つのアクション (ユニットテスト、コードスタイルチェック、およびコードメトリクス収集) が並行して実行されます。AWS CodeBuild を使用することで、ロードを処理するためにビルドサーバーを構築またはインストールする必要なく、これらのステップを新しいプロジェクトとして追加できます。



AWS CodePipeline — ビルド機能

図 AWS CodePipeline — ソースおよびビルドステージに示されているソースおよびビルドステージは、サポートプロセスとオートメーションとともに、継続的インテグレーションへのチーム移行を支援します。この成熟度では、デベロッパーはビルドおよびテスト結果に定期的に注意を払う必要があります。また、デベロッパーは健全なユニットテスト基盤を成長させ、保守する必要もあります。これにより、チーム全体の CI/CD パイプラインへの信頼が増し、さらなる導入が支持されるようになります。



AWS CodePipeline ステージ

継続的デリバリーパイプライン

継続的インテグレーションパイプラインを実装し、サポートプロセスを確立した後、チームは継続的デリバリーパイプラインへの移行を開始することができます。この移行には、チームがアプリケーションの構築およびデプロイの両方を自動化することが必要になります。

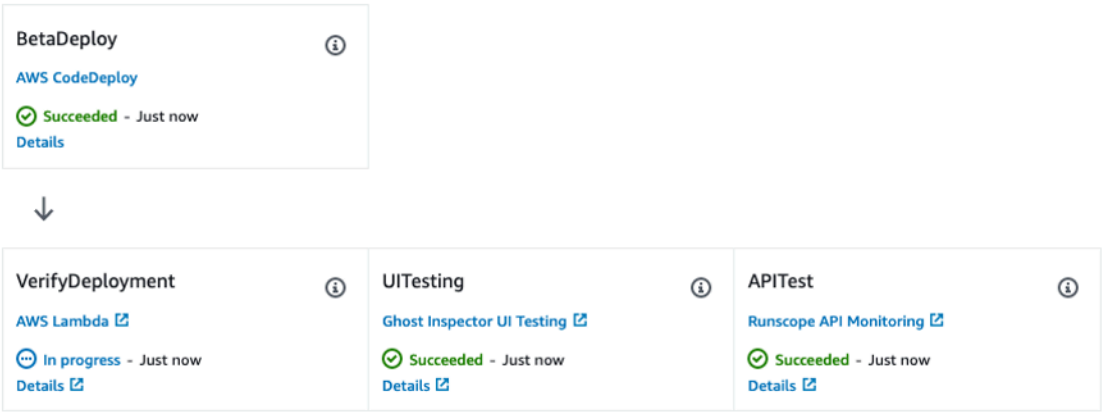
継続的デリバリーパイプラインは、ステージングおよび本稼働ステップが存在することを特徴とします。この本稼働ステップは、手動承認後に実行されます。

継続的インテグレーションパイプラインを構築したのと同様、チームは、デプロイスクリプトを記述することで継続的デリバリーパイプラインの構築を段階的に開始することができます。

アプリケーションのニーズに応じて、いくつかのデプロイステップを既存の AWS のサービスを使用して抽象化することができます。例えば、AWS CodePipeline は、Amazon EC2 インスタンスおよびオンプレミスで実行中のインスタンスへのコードデプロイを自動化するサービスである AWS CodeDeploy、Chef を使用してアプリケーションを操作しやすくする設定管理サービスである AWS OpsWorks、およびウェブアプリケーションおよびサービスのデプロイとスケーリングのためのサービスである AWS Elastic Beanstalk と直接統合されています。

AWS は、AWS CodeDeploy をお客様のインフラストラクチャおよびパイプラインに実装し統合する方法に関する詳細なドキュメントを提供しています。

チームがアプリケーションのデプロイの自動化に成功した後、様々なテストでデプロイステージを拡張できます。例えば、Ghost Inspector、Runscope などのサービスを使用して、すぐに使用開始可能なその他の統合を追加することができます。



AWS CodePipeline — デプロイステージでのコードテスト

Lambda アクションの追加

AWS CodeStar と AWS CodePipeline は、AWS Lambda との統合をサポートしています。この統合により、環境でのカスタムリソースの作成、サードパーティーシステム (Slack など) との統合、および新しくデプロイされた環境のチェックの実行など、幅広いタスクを実装できます。

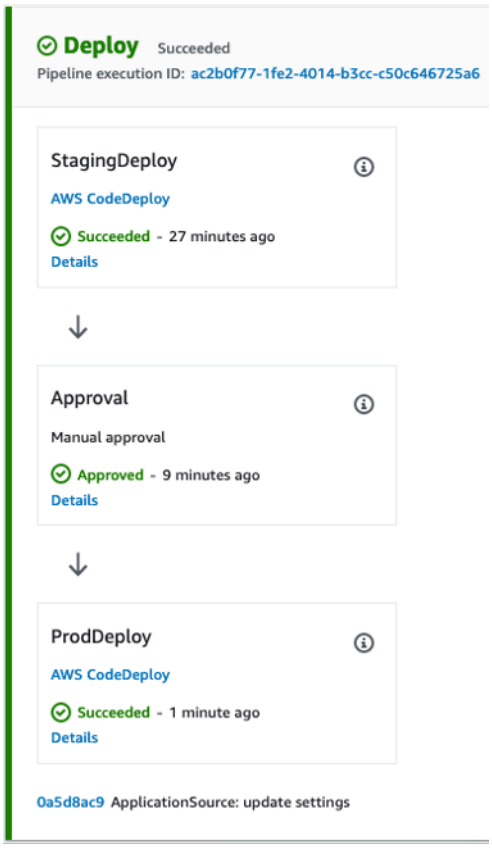
Lambda 関数は、以下のタスクを実行するために CI/CD パイプラインで使用できます。

- AWS CloudFormation テンプレートを適用または更新することで、環境に対する変更を展開する。
- AWS CloudFormation を使ってパイプラインの 1 つのステージでリソースをオンデマンドで作成し、別のステージで削除する。
- 正規名レコード (CNAME) 値をスワップする Lambda 関数 を使用して、アプリケーションバージョンをゼロダウンタイムで AWS Elastic Beanstalk にデプロイする。
- Amazon Elastic Container Service (ECS) の Docker インスタンスにデプロイする。
- AMI スナップショットを作成し、構築またはデプロイの前にリソースをバックアップする。
- インターネットリレーチャット (IRC) クライアントにメッセージを投稿する等、サードパーティー製品によってパイプラインに統合を追加する。

手動承認

必要な AWS Identity and Access Management (IAM) 権限を持つユーザーがアクションを承認または拒否できるように、パイプラインの処理を停止するところで承認アクションをパイプラインのステージに追加することができます。

アクションが承認されている場合、パイプラインの処理は再開されます。アクションが拒否された場合、またはパイプラインがそのアクションに到達して停止してから 7 日以内に誰もアクションを承認または拒否しない場合、結果はアクションの失敗と同じとなり、パイプラインの処理は続行しません。



AWS CodeDeploy — 手動承認

CI/CD パイプラインにおけるインフラストラクチャのコード変更のデプロイ

AWS CodePipeline ではパイプラインのどのステージにおいても AWS CloudFormation をデプロイアクションとして選択することができます。その後、スタックの作成や削除、変更セットの作成や実行など AWS CloudFormation で実施したい特定のアクションを選択します。スタックは、AWS CloudFormation の概念であり、関連する AWS リソースのグループを表します。Infrastructure as Code をプロビジョニングするためのさまざまな方法がありますが、AWS CloudFormation は、最も包括的な AWS リソースのセットをコードとして記述できる、スケーラブルで完全なソリューションとして AWS が推奨する包括的なツールです。AWS は、AWS CodePipeline プロジェクトで AWS CloudFormation を使用して、インフラストラクチャの変更とテストを追跡することを推奨しています。

サーバーレスアプリケーションの CI/CD

AWS CodeStar、AWS CodePipeline、AWS CodeBuild、および AWS CloudFormation を使用して、サーバーレスアプリケーション用の CI/CD パイプラインを構築することもできます。サーバーレ

スアプリケーションは、Amazon Cognito、Amazon S3、Amazon DynamoDB などのマネージドサービスをイベント駆動型サービスと統合し、AWS Lambda はサーバーの管理を必要としない方法でアプリケーションをデプロイします。サーバーレスアプリケーションのデベロッパーは、AWS CodePipeline、AWS CodeBuild、および AWS CloudFormation の組み合わせを使用して、AWS Serverless Application Model を使用して構築されたテンプレートで表現されるサーバーレスアプリケーションの構築、テスト、およびデプロイを自動化できます。詳細については、AWS Lambda のドキュメント、Lambda ベースのアプリケーションのデプロイを自動化するを参照してください。

また、AWS Serverless Application Model パイプライン (AWS SAM パイプライン) を使用して、組織のベストプラクティスに従った安全な CI/CD パイプラインを作成することもできます。AWS SAM パイプラインは、AWS SAM CLI の新機能で、デプロイ頻度の加速、変更のリードタイムの短縮、デプロイエラーの低減などの CI/CD の利点を数分でご利用いただけるようになります。AWS SAM パイプラインには、AWS CodeBuild/CodePipeline 向けに、AWS のデプロイのベストプラクティスに従ったデフォルトのパイプラインテンプレートが用意されています。詳細およびチュートリアルの表示については、ブログ Introducing AWS SAM Pipelines を参照してください。

複数のチーム、ブランチ、AWS リージョン用のパイプライン

大規模なプロジェクトの場合、複数のプロジェクトチームでさまざまなコンポーネントに取り組むことがよくあります。複数のチームが 1 つのコードリポジトリを使用する場合、各チームが独自のブランチを持つようにマップできます。プロジェクトの最終マージには、統合ブランチまたはリリースブランチも必要です。サーバー指向アーキテクチャまたはマイクロサービスアーキテクチャを使用する場合、各チームは独自のコードリポジトリを持つことができます。

最初のシナリオで、単一のパイプラインを使用した場合、1 つのチームがパイプラインをブロックすることで他のチームの進捗に影響が及ぶ可能性があります。チームブランチ用に特定のパイプラインを作成し、最終製品のデリバリー用に別のリリースパイプラインを作成することをお勧めします。

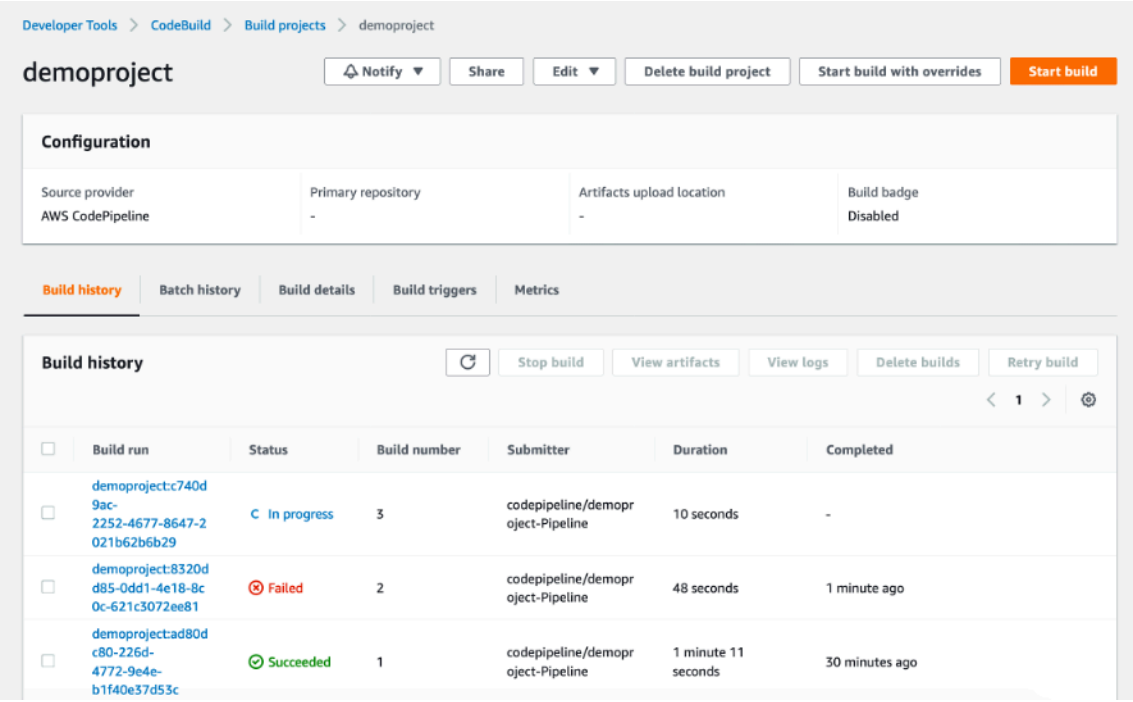
AWS CodeBuild によるパイプライン統合

AWS CodeBuild は、組織がほぼ無制限の規模で可用性の高いビルドプロセスを構築できるよう設計されています。AWS CodeBuild は、多くの一般的な言語向けのクイックスタート環境に加えて、指定した Docker コンテナを実行する機能を提供します。

AWS CodeCommit、AWS CodePipeline、AWS CodeDeploy に加え、Git および CodePipeline の Lambda アクションとの緊密な統合の利点を持ち、CodeBuild ツールは非常に柔軟性があります。

ソフトウェアは、ビルド前後のアクションを含む各ビルドステップを指定する buildspec.yml ファイルや CodeBuild ツールを通して指定されたアクションを組み込んで構築することができます。

CodeBuild ダッシュボードを使用して、各ビルドの詳細な履歴を表示できます。イベントは、Amazon CloudWatch Logs ログファイルとして保存されます。



AWS CodeBuild の CloudWatch Logs ログファイル

Jenkins によるパイプライン統合

Jenkins ビルドツールを使用して、デリバリーパイプラインを作成することができます。これらのパイプラインは、継続的デリバリーのステージを実装するためのステップを定義する標準的なジョブを使用します。しかし、このアプローチは、大規模なプロジェクトには最適ではない可能性があります。これは、Jenkins の再起動時にパイプラインの現在の状態が維持されず、手動承認の実装が容易ではなく、複雑なパイプラインの状態を追跡することが困難になる可能性があるためです。

代わりに、AWS は、AWS CodePipeline Plugin を使用して Jenkins による継続的デリバリーを実装することをお勧めします。このプラグインにより、Groovy のようなドメイン固有言語を使用して複雑なワークフローを記述することができ、複雑なパイプラインをオーケストレートするために使用できます。AWS CodePipeline Plugin の機能は、パイプラインで定義された現在の進行状況を視覚化する Pipeline Stage View Plugin、異なるブランチのビルドをグループ化する Pipeline Multibranch Plugin などのサテライトプラグインを使用することで拡張できます。

AWS は、パイプラインの設定を Jenkinsfile に保存し、ソースコードリポジトリにチェックインしておくことをお勧めします。これにより、パイプラインコードの変更を追跡することができます。また、Pipeline Multibranch Plugin を使用する際にこれはさらに重要になります。また、AWS はパイプラインをステージに分けることをお勧めします。これにより、パイプラインステップが論理的にグループ化され、Pipeline Stage View Plugin でパイプラインの現在の状態を視覚化できるようになります。

以下の図は、Pipeline Stage View Plugin で視覚化された 4 つの定義済みステージを備えた Jenkins パイプラインのサンプルを示しています。

Pipeline Stage View Plugin で視覚化された Jenkins パイプラインの定義済みステージ

デプロイ方法

継続的デリバリープロセスでは、新しいソフトウェアバージョンのロールアウトについて、複数のデプロイ戦略とバリエーションを検討できます。このセクションでは、一括デプロイ (インプレースデプロイ)、ローリング、イミュータブル、およびブルー/グリーンという最も一般的なデプロイ方法について説明します。AWS は、AWS CodeDeploy と AWS Elastic Beanstalk がどの方法をサポートしているかを示します。

次の表は、各デプロイ方法の特徴をまとめたものです。

方法	デプロイ失敗の影響	デプロイ所要時間	ゼロダウンタイム	DNS の変更なし	ロールバックプロセス	コードのデプロイ先
一括デプロイ	ダウンタイム	🕒	×	✓	再デプロイ	既存のインスタンス
ローリング	単一のバッチがサービス停止状態。新しいアプリケーションバージョンを実行している、失敗前のすべてのバッチ。	🕒 🕒 †	✓	✓	再デプロイ	既存のインスタンス
追加バッチを使用したローリング (beanstalk)	最初のバッチが失敗した場合、影響は最小限。それ以外の場合はローリングと同様。	🕒 🕒 🕒 †	✓	✓	再デプロイ	新規および既存のインスタンス

方法	デプロイ失敗の影響	デプロイ所要時間	ゼロダウンタイム	DNS の変更なし	ロールバックプロセス	コードのデプロイ先
イミュータブル	最小限	🕒🕒🕒🕒	✓	✓	再デプロイ	新規のインスタンス
Traffic splitting (トラフィック分割)	最小限	🕒🕒🕒🕒	✓	✓	トラフィックの再ルーティングと新しいインスタンスの終了	新規のインスタンス
ブルー/グリーン	最小限	🕒🕒🕒🕒	✓	×	古い環境への切り替え	新規のインスタンス

一括デプロイ (インプレースデプロイ)

一括デプロイ (インプレースデプロイ) は、既存のサーバーフリートに新しいアプリケーションコードをロールアウトするために使用できる方法です。この方法では、1 つのデプロイアクションですべてのコードを置き換えます。フリートのすべてのサーバーを一度に更新するため、ダウンタイムが必要となります。既存の DNS レコードを更新する必要はありません。デプロイが失敗した場合、操作を復元する唯一の方法は、すべてのサーバーにコードを再デプロイすることです。

AWS Elastic Beanstalk では、このデプロイは一括デプロイと呼ばれ、単一およびロードバランサーを備えたアプリケーション向けに利用できます。AWS CodeDeploy では、このデプロイ方法はインプレースデプロイと呼ばれ、デプロイ設定 `AllAtOnce` があります。

ローリングデプロイ

ローリングデプロイを使用すると、フリートは複数の部分に分割されるため、フリート全体が一度にアップグレードされません。デプロイプロセス中は、新旧の 2 つのソフトウェアバージョンが同じフリートで実行されています。この方法では、ゼロダウンタイムの更新が可能になります。デプロイが失敗した場合、フリートの更新された部分のみが影響を受けます。

canary リリースと呼ばれるローリングデプロイ方法のバリエーションでは、最初にごく一部のサーバーに新しいソフトウェアバージョンをデプロイすることが含まれます。このようにして、破壊的変更の影響を最小限に抑えながら、ソフトウェアが本稼働環境でどのように動作するかを少数のサーバーで確認することができます。canary デプロイメントでエラー率が上昇した場合、ソフトウェアはロールバックされます。それ以外の場合は、新しいバージョンを使用するサーバーの割合を徐々に増やしていきます。

AWS Elastic Beanstalk は、ローリングおよび追加のバッチでのローリングの 2 つのデプロイパターンを使用し、ローリングデプロイのパターンに従っています。これらのオプションを使用すると、サーバーを停止する前にアプリケーションをスケールアップし、デプロイ中も完全な機能を維持することができます。AWS CodeDeploy は、このパターンを OneAtATime および HalfAtATime のようなパターンを使用したインプレースデプロイのバリエーションとして実現しています。

イミュータブルおよびブルー/グリーンデプロイ

イミュータブルパターンは、新しい構成またはバージョンのアプリケーションコードを使用した全く新しいサーバーセットを起動することで、アプリケーションコードのデプロイを指定します。このパターンは、シンプルな API コールで新しいサーバーリソースを作成するクラウドの機能を活用します。

ブルー/グリーンデプロイ戦略は、別の環境の作成を必要とするイミュータブルデプロイの一種です。新しい環境を起動しすべてのテストが合格すると、この新しいデプロイにトラフィックが移行します。重要なのは、古い環境、つまり「ブルー」環境は、ロールバックが必要となる場合に備えてアイドル状態に保たれることです。

AWS Elastic Beanstalk は、イミュータブルおよびブルー/グリーンのデプロイパターンをサポートしています。また、AWS CodeDeploy もブルー/グリーンパターンをサポートしています。AWS のサービスがこれらのイミュータブルパターンを実現する方法の詳細については、ホワイトペーパー『AWS でのブルー/グリーンデプロイ』を参照してください。

データベーススキーマの変更

最新のソフトウェアは、データベースレイヤーを備えていることが一般的です。通常、データとデータの構造の両方を保存するリレーショナルデータベースが使用されます。継続的デリバリープロセスでは、そのデータベースの変更が必要となることがよくあります。リレーショナルデータベースの変更の処理には特別な考慮が必要であり、アプリケーションバイナリをデプロイする際とは異なる課題が発生します。通常、アプリケーションバイナリをアップグレードする場合、アプリケーションを停止し、アップグレードを実行し、その後再起動します。アプリケーションの外部で処理されるアプリケーションの状態を気にする必要はありません。

データベースをアップグレードする場合は、データベースには多くの状態が含まれながら比較的少ないロジックと構造が含まれているため、状態を考慮する必要があります。

変更が適用される前後のデータベーススキーマは、異なるバージョンのデータベースとして見なす必要があります。Liquibase や Flyway などのツールを使用してバージョンを管理することができます。

一般的に、これらのツールは、以下の方法に類似したものを採用しています。

- データベースのバージョンを保存するテーブルをデータベースに追加します。
- データベースの変更コマンドを追跡し、バージョン管理された変更セットにそれらを束ねます。Liquibase の場合、これらの変更は XML ファイルに保存されます。Flyway は、わずかに異なる方法を採用し、変更セットは独立した SQL ファイルとして扱われるか、より複雑な移行のためには独立した Java クラスとして扱われる場合もあります。
- Liquibase は、データベースのアップグレードを要求されると、メタデータテーブルをチェックし、最新のバージョンを使用してデータベースを最新化するために実行する変更セットを決定します。

ベストプラクティスのまとめ

以下は、CI/CD のためにすべきこと、してはいけないことのベストプラクティスの一部を挙げています。

すべきこと:

- Infrastructure as Code を処理する。
  - インフラストラクチャコードにバージョン管理を使用する。
  - バグトラッキング/チケット発行システムを利用する。
  - 変更を適用する前にピアレビューを実施する。
  - インフラストラクチャコードのパターン/設計を確立する。
  - コード変更などのインフラストラクチャの変更をテストする。
- デベロッパーを 12 人以下の自立したメンバーで構成されるチームに統合する。
- 長く続く機能ブランチを持たずに、すべてのデベロッパーにメイントランクにコードを頻繁にコミットさせる。
- 組織全体で Maven や Gradle などのビルドシステムを継続的に採用し、ビルドを標準化する。
- コードベースの 100% のカバレッジを目標としてデベロッパーにユニットテストを構築させる。
- ユニットテストが期間、数、およびスコープにおいてテスト全体の 70% を占めていることを確認する。
- ユニットテストが最新のものであり、無視した部分がないことを確認する。ユニットテストの失敗は、バイパスするのではなく修正する必要がある。
- 継続的デリバリーの設定をコードとして扱う。
- ロールベースのセキュリティ制御 (つまり、誰が何をいつできるか) を確立する。
  - 可能な限りすべてのリソースを監視/追跡する。
  - サービス、可用性、および応答時間に警戒する。
  - ものごとを把握し、学習し、改善する。
  - チームの全員とアクセスを共有する。
  - メトリクスとモニタリングをライフサイクルの計画に組み込む。
- 基本的なメトリクスを保存し、追跡する。
  - ビルドの数。
  - デプロイの数。

  - 変更が本稼働環境に到達するまでの平均時間。
  - 最初のパイプラインステージから各ステージまでの平均時間。
  - 本稼働環境に到達する変更の数。
  - 平均ビルド時間。
- 各ブランチとチームに区別された複数のパイプラインを使用する。

してはいけないこと:

- 大規模で複雑なマージを伴う長期的なブランチを持つ。
- 手動テストを行う。
- 手動の承認プロセス、ゲート、コードレビュー、セキュリティレビューを持つ。

まとめ

継続的インテグレーションと継続的デリバリーは、組織のアプリケーションチームにとって理想的なシナリオを提供します。デベロッパーは、コードをリポジトリにプッシュするだけです。このコードは、インテグレーション、テスト、デプロイ、および再テストが行われ、インフラストラクチャとマージされ、セキュリティと品質のレビューを経て、非常に高い信頼性でデプロイする準備が整った状態になります。

CI/CD を使用すると、コードの品質が向上し、破壊的変更がないことに高い確信を持ってソフトウェアの更新を迅速にデリバリーできます。リリースの影響は、本稼働およびオペレーションのデータと関連付けることができます。これは、次のサイクルの計画にも役立てることができます。このことは、組織のクラウド移行において不可欠な DevOps プラクティスの 1 つです。

その他の資料

このホワイトペーパーで説明されているトピックの詳細については、次の AWS ホワイトペーパーを参照してください。

- AWS でのデプロイオプションの概要
- AWS でのブルー/グリーンデプロイ
- Setting up CI/CD pipeline by integrating Jenkins with AWS CodeBuild and AWS CodeDeploy
- AWS におけるマイクロサービス
- AWS での Docker: クラウドでコンテナを実行する

寄稿者

本ドキュメントは、次の人物および組織が寄稿しました。

- AWS、プリンシパルソリューションアーキテクト、Amrish Thakkar
- David Stacy、DevOps シニアコンサルタント、AWS Professional Services
- Asif Khan、ソリューションアーキテクト、AWS
- Xiang Shen、シニアソリューションアーキテクト、AWS

改訂履歴

このホワイトペーパーの更新に関する通知を受け取るには、RSS フィードをサブスクライブしてください。

update-history-change	update-history-description	update-history-date
初版発行	ホワイトペーパーの初回発行	2021 年 10 月 27 日
初版発行	ホワイトペーパーの初回発行	2017 年 6 月 1 日

注意

お客様は、この文書に記載されている情報を独自に評価する責任を負うものとします。本書は、(a) 情報提供のみを目的とし、(b) AWS の現行製品と慣行について説明しており、これらは予告なしに変更されることがあり、(c) AWS およびその関連会社、サプライヤーまたはライセンサーからの契約上の義務や保証をもたらすものではありません。AWS の製品やサービスは、明示または暗示を問わず、一切の保証、表明、条件なしに「現状のまま」提供されます。お客様に対する AWS の責任は、AWS 契約により規定されます。本書は、AWS とお客様の間で締結されるいかなる契約の一部でもなく、その内容を修正するものでもありません。