DevOps の定義

GitHub で DevOps プラクティスを構築する

目次

DevOps の概要

記事の見出しから求人票の職務欄まで、「DevOps」はここ 10 年いたるところで見聞きするホットなバズワードです。そして、それには相応の理由があります。

多くの場合、組織が DevOps の導入に成功すると、その組織ではソフトウェア開発速度の大幅な向上、信頼性の向上、製品のイテレーションの高速化、サービスの拡張を簡単に行えるようになります。

しかし、ルーツはソフトウェア開発であるにもかかわらず、DevOps は総合的なビジネス プラクティスであり、人、プロセス、文化的慣習、テクノロジを組み合わせて、それまではサイロ化していたチームを 1 つにまとめ、スピード、価値、品質を**ソフトウェア開発ライフ サイクル** (**SDLC**) 全体にもたらします。

DevOps は、人、プロセス、文化的慣習、テクノロジを組み合わせた包括的なビジネス慣行です。

これはつまり、多くの場合、汎用的なアプローチは存在しないという意味です。ただし、成功する DevOps の実装には共通のプラクティスと原則があります。この eBook では、DevOps の中核となる基本原則のほか、組織内で DevOps パイプラインを実装してお客様により高い価値を提供する方法について概説します。その後、組織の DevOps プラクティスを最もよく補完するツールに関するガイダンスが提供されます。

GitHub では DevOps を、開発、IT 運用、セキュリティの各チームが連携し、SDLC を通じて構築、テスト、イテレーション、定期的なフィードバックを行うための、哲学と一連のプラクティスとして捉えています。

DevOps の基礎

DevOps はコードの記述、テスト、デプロイの間の摩擦を減らすことで、チームが高品質な製品をより迅速に出荷するのを支援します。

GitHub は、組織が DevOps の導入を成功させ、継続的なソフトウェアの出荷と改善を容易にするのを支援する総合的なプラットフォームを提供します。

DevOps モデルとは何かとよく聞かれますが、この質問は DevOps の論点から外れています。DevOps は開発ライフ サイクル全体に関わるソフトウェア構築のアプローチです。これは、エンド ユーザーに継続的に価値を提供することを目的としたプラクティス、文化、テクノロジの組み合わせです。

つまり、DevOps には、全てのケースに適合する 1 つのアプローチはありません。実装は組織ごとに異なります。それでも、DevOps には全組織がさまざまな形態で活用するプラクティスのフレームワークがあります。

DevOps の中核には、製品を担当する全員が統一されたチームとして協力すべきであるという考え方があります。開発、品質保証、セキュリティ、運用の各サイロで別々に作業するのではなく、DevOps は、ソフトウェアの計画、構築、提供、改善に向けてエンドツーエンドの責任を担うユーザーを連携させます。

プログラミング チームがコードを記述し、テスト チームがバグを見つけて、運用チームがインフラを担当するという従来の開発方法と比べると、DevOps は抜本的な変化のように感じるかもしれません。プラクティスとして、DevOps は基本的に、SDLC のあらゆる部分で従来はサイロ化されていたチームをまとめることによって組織の変革を目指しています。

組織ごとに DevOps 導入ジャーニーは異なりますが、これらは成功を示す重要な原則です。次の項目を行っている場合、DevOps を適切に行っていることになりますが、業界によっては、DevOps プラクティスに必要な特有の項目があります。

DevOps の基本原則の定義

DevOps の実装は組織によってさまざまです。GitHub では、DevOps は、ソフトウェアを通じてどのように価値を提供するかを検討するためのフレームワークとして理解するのがよいと考えています。DevOps は、単なる 1 つの手法でもプロセスの寄せ集めでもありません。DevOps は基本的に基本原則のセットであり、文化的および技術的な基本原則の両方が含まれています。では、さらに掘り下げてみましょう。

- **コラボレーション**: DevOps を成功させるためには、これまでサイロで互いに隔てられてきた 2 つのチーム、つまり運用と開発が緊密に連携しなければなりません。DevOps モデルのもとでこれらのチームを緊密に連携させることで、チーム間のコミュニケーションと協力を促し、アプリケーションやソフトウェアのスタックの開発、テスト、運用、デプロイ、監視、イテレーションの能力を高めることを目指します。

- **バージョン管理**: バージョン管理は DevOps にとっても、今日の多くのソフトウェア開発にとっても不可欠な要素です。バージョン管理システムは、ファイルの変更を自動的に記録し、以前のファイル バージョンの記録を保持するためのものです。

- **自動化**: DevOps における自動化とは、一般的に、テクノロジーやスクリプトを活用し、基になるインフラのメンテナンスやスケーリングを担当する者と中核的なソフトウェアの構築を担当する者との間にフィードバック ループを作成することです。環境のスケーリングの支援からソフトウェア ビルドの作成やテストのオーケストレーションまで、DevOps における自動化はさまざまな形態を取ることができます。

- **インクリメンタル リリース**: インクリメンタル リリースは DevOps プラクティスが成功するための柱であり、それまでの機能に基づき小さな変更や更新を迅速にリリースすることが特徴です。インクリメンタル リリースでは、アプリケーション全体を更新するのではなく、開発チームが小さな変更をメインのブランチにすばやく統合し、品質とセキュリティをテストして、エンドユーザーにリリースします。

- **オーケストレーション**: オーケストレーションとは自動化されたタスクのセットを単一のワークフローに組み込んだものを指し、コンテナの管理、新しいウェブサーバーの立ち上げ、データベース項目の変更、ウェブ アプリケーションのインテグレーションなど一連の機能を処理します。簡単に言えば、オーケストレーションは、アプリケーションを効率的に実行するために必要となるインフラ要件の構成、管理、調整を支援します。

- **パイプライン**: DevOps に関する会話では、おそらく「パイプライン」という言葉をかなり頻繁に耳にするでしょう。分かりやすく言えば、DevOps パイプラインとは、自動化やさまざまなツールを活用し、開発者がコードをテスト環境に迅速にデプロイできるようにするプロセスです。運用チームと開発チームはその後、セキュリティ問題やバグなどがないかコードをテストしてから、本番環境にデプロイします。

- **フィードバック共有 (またはフィードバック ループ)**: フィードバック共有、またはフィードバックループは Gene Kim 氏の画期的な著書「The Phoenix Project (The DevOps 逆転だ !)」で最初に定義された、一般的な DevOps 用語です。Kim 氏は次のように述べています。「ほぼ全てのプロセス改善の取り組みの目標は、フィードバック ループを短縮、増幅して必要な修正を継続的に行えるようにすることである」。簡単に言えば、フィードバック ループとは、問題やバグが潜在していないかアプリケーションやインフラのパフォーマンスを監視し、アプリケーション内でのエンドユーザーの活動を追跡するためのプロセスです。

これらの DevOps の基本原則は、成功する DevOps プラクティスを構築するために最も重要です。成功する高機能 DevOps プラクティスには、次のような特徴と利点があります。

- より迅速な配信とリリース サイクル
- 自動化の促進と生産性の向上
- コラボレーションによる品質の向上
- より拡張性の高い製品
- 継続的な測定が継続的な改善につながる、プロセスの拡張性の向上
- アプリケーション、インフラ、チームの回復力の向上

これらの利点は、チーム間のコラボレーションだけでなく、お客様により高い価値をもたらし、生産性を向上するという DevOps の最終目標につながります。

DevOps 導入ジャーニーの主要なステージ

組織の DevOps プラクティスを構築する際は、それが目的地ではなく旅であることを理解してください。各ジャーニーは、組織が何を必要としているかによって異なります。多くの場合、組織は DevOps プラクティスの小規模な実装から始め、時間をかけて進化し、習熟度を高めていきます。以下の表の各ステージは、少数のチームのみが DevOps を実践する実験的 DevOps から、組織全体で DevOps が採用され、サイロが分解されてチーム間の情報の流れが容易になるネイティブ DevOps まで、組織の典型的な進化を示しています。

私たちの哲学は、将来の企業のために自動化と優れた DevOps を構築することです。」

Adobe、シニア SCM エンジニア、**Todd O'Connor 氏**

実験的な DevOps	**学習済み DevOps**	**事前対応型 DevOps**	**ネイティブ DevOps**
1～2 チームが DevOps を検討しています。 役割ベースのサイロがまだ大部分残っています。 自動化で一部実験していますが各ステップで人間の介入が必要です。 正式なプロセスはありません。	組織の一部でコラボレーションする製品チームを採用しています。 これらのチームは DevOps ツールを効果的に使用していますが各チーム独自のアプローチがあります。 プロセスは形成されており大部分は他の組織から学んだものです。	全ての新製品が DevOps モデルから始まっています。 プロセスの有効性を監視し、改善につなげるための測定が実施されています。 プロセスが組織のニーズに適応し始めています。 組織の大半が DevOps ツールを使用しています。	DevOps は組織全体で導入されています。 DevOps プロセスは組織のニーズに合わせて正確に調整され、状況の変化に応じて定期的に更新されています。 テスト、ビルド、デプロイは DevOps ツールを使って自動化されています。 すべてのチームが製品に特化し、組織全体にわたりコミュニケーションとコラボレーションの流れを容易にしています。

DevOps のプラクティスが潜在能力をフルに発揮するには、

組織全体の賛同を得る必要があります。ただし、個々の製品チーム内では、変更が小規模な影響を受ける可能性があります。これは、実験的 DevOps ステージに反映されます。多くの場合、組織の他の領域は、DevOps を実践しているチームの成功に気づき、彼らがしていることを再現したいと考えます。これは、学習済み DevOps ステージに反映されます。その後、組織ではすべての新製品開発が DevOps モデルに従うようになるかもしれません。このステージでは、開発チームと運用チームが協調して、製品にフォーカスしたチームを形成します。これは、事前対応型 DevOps ステージに反映されます。ネイティブ DevOps ステージは目標です。ここでは、プロセスが明確に定義され、コミュニケーションとコラボレーションが組織全体で容易に流れます。ネイティブ DevOps ステージでは、組織内の全員が協力して、お客様により高い価値を提供します。

基礎となるプラクティス

DevOps の仕組みは企業ごとに異なります。ただし、DevOps の導入に成功している組織には、3 つの核となるテーマがあります。

品質について全員が責任を負う

DevOps はソフトウェア開発チームに存在する分野間の障壁を排除します。実行者はサイロ化された段階的なプロジェクトを完了するのではなく、エンドツーエンドの製品の構築に集中する傾向があります。これは、製品の計画から構築、テスト、デプロイまで、SDLC 全体にわたって同じ個人が協力することを意味します。

準備が整い次第コードを出荷する

従来のソフトウェア開発プラクティスでは、多くの変更を大規模なリリースにバンドルすることがよくあります。つまり、顧客は通常、ソフトウェアのアップデートをより長い期間待つことになります。また、大規模なコード変更がもたらす影響を予測することが難しくなり、開発チームや運用チームに大きなプレッシャーをかけることになります。一方、DevOps では、構築とテストが容易で、準備が整い次第すぐに出荷できる、段階的なコード変更が好まれます。開発者がコード変更をプロジェクトにコミットすると、**継続的インテグレーション**と**継続的デプロイ (CI/CD)** ツールにより、自動テスト、アプリケーションのビルド、コード インテグレーションや問題の報告が促進されます。多くの DevOps 実行者は継続的な改善の概念を各自の作業にも適用し、時間をかけてプロセスを測定および調整します。

自動化によって品質と予測可能性が向上する

成功する DevOps プラクティスでは、自動化できるものは全て自動化します。これにより、人為的なミスのリスクが低減し、製品を拡張しやすくなります。ツールはインフラの構成とデプロイを自動化するために使用され、静的分析ツールはセキュリティの脆弱性を見つけて示します。DevOps 実行者はあらゆるステージで反復タスクの自動化に努めています。

DevOps の成熟度

組織に DevOps を導入することは、製品デリバリーのさまざまなステージにわたり、さまざまなレベルで導入を行う継続的なジャーニーです。DevOps はビジネス ニーズと同様に動的であり、その実装は組織ごとに異なります。

つまり、定義された DevOps 成熟度モデルは 1 つではないということです。GitHub では DevOps 成熟度モデルについてあまり説明しません。どの組織にも「DevOps」の実現に使用できるチェック リストが存在することを暗に示してしまうためです。これは正しくありません。基本的に DevOps は継続的プラクティスです。しかし、次の図に示すように、企業が目指すことができる共通のステップと成功のマーカーがあります。

図 1: DevOps 成熟度モデル

この DevOps 成熟度モデル図は、組織の現在の状況と、次のレベルに到達するために必要なものを大まかにプロットするために使用します。現在の組織が成熟度モデルのどのステージにあるとしても、重要な概念は、コラボレーション、継続的な学習、反復的な改善を促進する環境を推進することです。

次のセクションでは、DevOps パイプラインの考え方、および各ステージを構成するツールとプラクティスを紹介します。ソフトウェア開発においては、人、プロセス、文化的慣行、テクノロジを結びつける DevOps の定義を念頭に置く必要があります。テクノロジは最後に言及されており、全体像のほんの一部にすぎないことに注意することが重要です。テクノロジは確かに DevOps のプラクティスに影響を及ぼし、最適化するのに役立ちますが、中心にあるのは人と文化です。組織は最先端の DevOps ツールを持つこともできますが、DevOps のメリットを十分に引き出すには、協調的な文化が必要です。

「私たちのチームは、自分たち自身を自動化してより良い仕事ができるように常に模索しています。」

Buzzfeed エンジニアリング ディレクター、Andrew Mulholland **氏**

DevOps パイプライン

DevOps パイプラインは、開発とエンド ユーザーへのソフトウェアのデプロイを容易にするために、SDLC 全体にわたる自動化、ツール導入、プラクティスを組み合わせたものです。

重要なことは、DevOps パイプラインを構築するための万能なアプローチはなく、多くの場合、組織ごとに設計と実装が異なるということです。ただし、ほとんどの DevOps パイプラインには、自動化、CI/CD、自動テスト、レポート、監視が含まれます。

DevOps パイプラインを成功させるための重要な概念は、反復可能で継続的であり、常時稼働している、ということです。DevOps パイプラインには孤立したイベントがあってはならず、代わりに、各ステップがそれ自体の再現性によって定義される、より大規模なシステムを構成する必要があります。

重要なことは、DevOps パイプラインの構築は、多くの場合、DevOps の導入を検討している組織にとって最も具体的な要素の 1 つであり、これは、自動化や特定のツール導入によって定義されるように、密接なコラボレーションを好む文化的側面によっても定義されます。

適切なテクノロジと、人材とプロセスへの投資があれば、最初はシンプルなものだとしても、どのような組織でも常時稼働の DevOps パイプラインを構築できます。ただし、段階的な開発作業と SDLC 全体にわたる緊密な部門横断的コラボレーションを優先する DevOps 文化を完全に採用しなければ、組織は DevOps の価値を十分に実現できない可能性があります。

> 私たちのチームにはこんなスローガンがあります。「人間に機械の仕事をさせるな。そのためには GitHub を活用せよ」。

Blue Yonder チーフソフトウェア エンジニア、
Gabriel Kohen 氏

DevOps パイプラインのステージ

DevOps を説明する時によく引き合いに出されるのが組み立てラインとの比較です。SDLC の各部分を分析して、自動プロセスと手動プロセスの一貫したセットを確立します。その結果、全体的な出力の効率と一貫性が向上します。

しかし、組み立てラインと違い、DevOps は明確な開始と終了を伴うエンドツーエンドのプロセスではありません。DevOps は継続的な改善のサイクルであり、ソフトウェアが出荷された後も改善が継続されます。

実際、新しいソフトウェア機能が開発ステージを経て直線的な経路をたどるとしても、システム全体 (およびその機能) は継続的な反復サイクルをたどることになります。

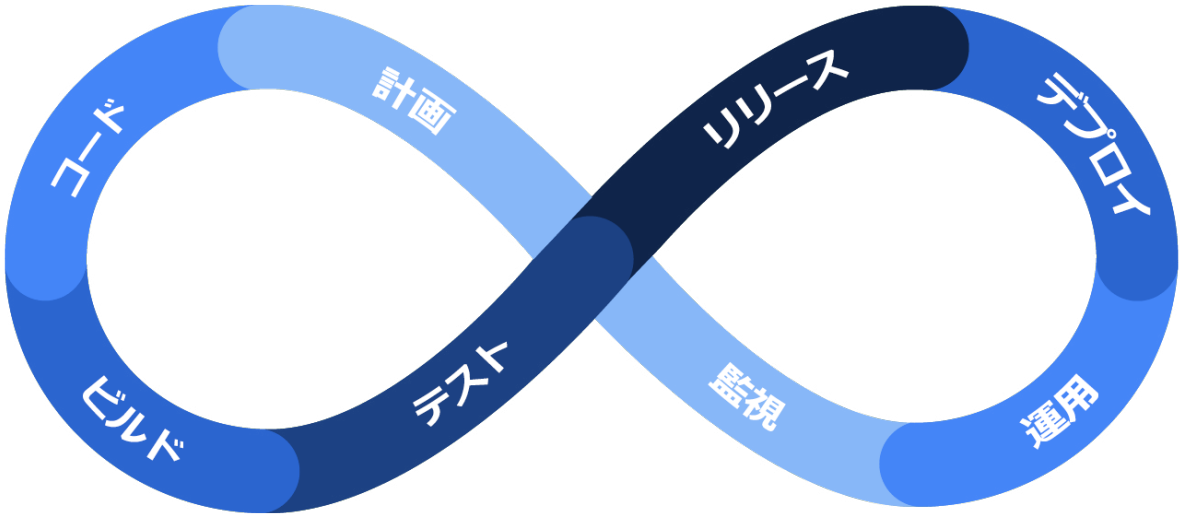

図 2: DevOps パイプライン

この仕組みを理解するため、DevOps パイプラインのステージと、ステージ間のフィードバックを細かく見ていきましょう。

1. **計画**：どの DevOps パイプラインもスタート地点は、新しい機能や修正が導入され、スケジュールが行われる計画ステージです。計画ステージでの主な目標は、大規模な DevOps プラクティスの中でさまざまな役割を担う人々が最初からコラボレーションできるようにすることです。つまり、協力してユーザーのニーズを理解し、ソリューションを設計し、変更の影響を理解し、変更点が既存のシステムにスムーズに適合するようにします。GitHub Issues を使用して、プロジェクトの計画に GitHub を使用する方法をご覧ください。

2. **コード**：コーディング ステージでは、組織は計画に従ってコードを書き始め、Git などのバージョン管理システムを介して作業を追跡します。DevOps パイプラインのコーディング ステージでは、開発者は開発環境でさまざまなツールを使用して、コード スタイルに一貫性を取り入れたり、潜在的なセキュリティ上の欠陥を特定したりできます。これには、クラウド ホスト**統合開発環境** (**IDE**) などのツールの利用が含まれる場合があります。これらのツールは、開発者ワークフロー全体で一貫性を取り入れ、コーディング環境を高速化するためによく使用されます。GitHub でコーディングする方法をご確認ください。

3. **ビルド：** ビルド ステージでは、DevOps パイプラインが完全に始動しています。開発者が共有リポジトリにコード変更をコミットするとビルド ステージが始まります。この時点で、開発者はプルリクエストを送信して、コード変更をコードベースにマージすることができます。これにより、チームの他のユーザーにマージを承認する前にコードを確認するよう警告が表示されます。同時に、一般的な DevOps パイプラインは、コード ベースをマージし、一連のインテグレーション テストとユニット テストを開始する自動構築プロセスを開始します。これらのテストやビルド自体が失敗した場合、プルリクエストも失敗し、開発者にその問題が通知されます。DevOps パイプラインにおけるこのレベルのワークフロー オートメーションは、組織が潜在的なビルド インテグレーションの問題を軽減し、SDLC の早い時点でバグやセキュリティ問題を特定するのに役立ちます。GitHub でコード インテグレーションとアプリケーション構築の継続的インテグレーション / 継続的デプロイを実現する方法をご覧ください。

4. **テスト**：ビルドが承認されると、DevOps パイプラインのテスト ステージが開始され、ビルドは本番環境に近いテスト環境にデプロイされます。組織によっては、ステージング用のテスト環境のプロビジョニングを自動化するために、DevOps パイプラインに **Infrastructure as Code** (**IaC**) を採用する場合があります。また、どのような新しいビルドにも対応できる

ように、専用のビルド済みテスト環境を用意している場合もあります。こうした選択は、組織のニーズとコンピューティング リソースによって大きく異なります。ビルドがテスト環境にデプロイされると、さまざまな自動テストと手動テストが実施されます。これには、脆弱性やリスク領域を特定するための、**動的アプリケーション セキュリティ テスト** (**DAST**) や**対話型アプリケーション セキュリティ テスト** (**IAST**) などの自動セキュリティ テストが含まれる場合があります。また、チーム メンバーがアプリケーションを使い、顧客が遭遇する可能性のある問題やバグを記録する手動**ユーザー受け入れテスト** (**UAT**) を含めることもできます。組織はその DevOps パイプラインのテスト ステージにおいて、それぞれが独自の自動および手動のテスト スイートと戦略を持っています。そして、重要なことに、このステージでは開発者ワークフローを中断することなく、組織がテストを適用できる場所が提供されます。GitHub を使用して継続的テストのパイプラインを構築する方法をご覧ください。

5. **リリース**：リリース ステージは、DevOps パイプラインの中で新しいビルドが完全にテストされ、デプロイの準備が整うポイントを示しています。コード自体のテストに加え、運用パフォーマンスのテストも完了しているため、組織は未発見のバグや問題の影響を受けることなく本番環境で正常にコードを実行できることを確信できます。このステージでは、いわゆる継続的デプロイと呼ばれるプラクティスで、このステージに達したコードを自動的にデプロイすることを選択する組織もあります。そうすることで 1 日に複数回、コード変更をデプロイするソフトウェア チームもいます。代わりに、新しいビルドを手動で本番環境にリリースし、最終承認段階を含めることを決定するチームもあれば、特定の日または特定の時間に自動リリースが行われるようにスケジュールするチームもあります。CI/CD プラットフォームやその他の DevOps ツールを使用すると、組織は最適なリリース サイクルを構築し、DevOps パイプラインのリリース ステージ全体に自動化を適用できます。GitHub でソフトウェアをリリースする方法をご確認ください。

6. **デプロイ**：ビルドはリリースが終わると、本番環境にデプロイできるようになります。DevOps パイプラインのデプロイ ステージでは、組織は IaC を介して新しい本番環境をプロビジョニングしたり、ブルーグリーン デプロイを調整したりすることで、さまざまなツールを活用してデプロイ プロセスを自動化します (この場合、新しい環境の一部のユーザーにコード変更を徐々にロールアウトすると同時に、別の環境の他のユーザーが引き続き古いコード ベースを利用できるようにします)。また、ブルーグリーン デプロイ戦略により、組織は、問題が発生した場合に、ユーザーを以前のビルドにすばやく戻すことができます。GitHub を使用してコードをデプロイするする方法をご覧ください。

7. **運用** : DevOps パイプラインはアプリケーションがデプロイされたら終わりではありません。デプロイ後は運用ステージが始まり、組織は全てがスムーズに実行されていることを確認する必要があります。運用ステージには、リアルタイムの需要に合わせてリソースを自動的にスケーリングするルールを適用するインフラ オーケストレーションと構成設定が含まれます。また、多くの場合、行動ログや顧客フィードバック フォームなど、アプリケーション内のユーザー アクティビティをキャプチャするメカニズムも含まれます。運用ステージの目標は、ステージの名称そのものが示唆するとおり、アプリケーションと基になるインフラを適切に運用し、ソフトウェア自体の運用プロファイルを改善する方法を模索することです。GitHub がソフトウェア運用を実現するしくみをご覧ください。

8. **監視** : DevOps パイプラインの運用ステージを足場として、組織は自動監視ツールをセットアップして、潜在的なパフォーマンスのボトルネック、アプリケーションの問題、ユーザー動作を特定します。このステージでは、アプリケーションとインフラのパフォーマンスに関するデータを収集するツール導入を実装し、実行可能な項目を製品チームに渡し、未解決の問題を解決するか、アプリケーションの既存のユーザー動作をサポートする新機能を開発する必要があります。監視は DevOps パイプラインのいわば「最後」のステージですが、プロセス自体は継続的であることを理解することが重要です。つまり、監視ツールは組織が、DevOps パイプラインを通じてフィードバックする追加の計画とイテレーションが必要な領域を特定するのに役立ちます。各ステージは、無限ループで次のステージにフィードされます。GitHub が組織に高度な監視機能を提供するしくみをご確認ください。

これらの各ステージが、DevOps の全体像の一部を構成します。図が示すように、これらの各ステージは無限ループの中で次のステージへと流れます。組織が順調に動作する DevOps マシンへと変容するには、各ステージで入念な努力が必要です。DevOps パイプラインの各ステージが組織にとってより自然なものになるにつれ、品質を向上し、お客様に大きな価値をより迅速にもたらすことができるようになります。

DevOps では、コードから運用までのプロセスを可能な限り自動化することが基本です。次のセクションでは、CI と CD の概念と、それが DevOps パイプラインにどのように適合するかについて説明します。

継続的インテグレーションと継続的デプロイの概念

継続的インテグレーション (CI) は DevOps の基礎となるプラクティスであり、開発チームは複数のコントリビュータからのコード変更を共有リポジトリに統合します。継続的デプロイ (CD) とは自動化されたソフトウェア リリース プラクティスで、コード変更は、定義済みテストに合格すると同時に各種ステージにデプロイされます。

CI を使用すると、組織は欠陥を迅速に特定し、高品質のソフトウェアをより迅速に出荷できます。CD の目標は、自動化を使用してリリースを高速化し、デプロイメント プロセス中に人間の介入を可能な限り排除することです。これらをまとめて CI/CD と呼ぶこともあります。

次の複数のセクションでは、CI/CD の概念と、最良の結果を得るために組織が従うべきガイドラインについて説明します。CI と CD は、包括的な DevOps パイプラインの大部分を占めており、最初に作成、構築、テストされた時点からリリースまでのコードを網羅しています。さらに、DevOps (DevSecOps) におけるコンテナ化とセキュリティのトピックを紹介し、DevOps 組織でこれらがどのように重要な役割を果たすことができるかを見ていきます。

GitHub Actions を使用した CI/CD により、GitHub から直接ビルド、テスト、デプロイすることができます。ビルド時間を 80 分から 10 分に短縮できました。」

Pinterest、エンジニアリング アーキテクト

継続的インテグレーション

継続的インテグレーションは、ソフトウェア開発における重要な問題 (複数のコントリビュータとの共有リポジトリにおけるコード インテグレーションの課題の管理) を解決することで、開発サイクルの高速化と効率化を促進しようとしています。

開発者は、ソフトウェアの更新やバグの修正に取り掛かるときに、作業用のコード ベースのコピーを作成します。これは、開発者がコード ベースのコピーを作成 (つまり「フォーク」) できる Git などのバージョン管理システムによって行われます。

より多くの開発者がコード ベースのコピーを作成するようになると、複数のコントリビュータからの変更を統合することが困難になります。特に、ある開発者が作業を開始したコード ベースが古くなり、メイン リポジトリと一致しなくなった場合は困難になります。

最悪のシナリオでは、一致していないコードをそれぞれの開発者が解消しようとするため、コード変更をうまく統合するのに、各自で変更するよりも時間がかかる可能性があります。開発者はよくこれを「インテグレーション地獄」と呼びます。

継続的インテグレーションは、開発者が変更を加えるたびにインテグレーションするよう促すことで、これを防止しようとしています。また、継続的インテグレーションは自動化を活用してコードの統合とテストを高速化し、追加の変更が必要ないことを確認し、開発者の負担を軽減します。より頻繁なコード インテグレーションに自動構築とテストを組み合わせることで、ソフトウェア開発プロセスのスピードアップに役立ちます。

各企業は、独自のニーズに応じて継続的インテグレーション プラクティスを定義します。企業によっては、より厳格な自動セキュリティ テストを導入する場合があります。迅速なコードのマージを優先し、より時間のかかる自動テストを SDLC の後半に予約する企業もあります。

それにもかかわらず、効果的な CI パイプラインは、一連の共通ツールとベスト プラクティスを共有しています。これには以下が含まれます。

- **共有コード リポジトリ**: バージョン管理システムの共有コード リポジトリは、効果的な継続的インテグレーション プラクティスを作成するための基礎となります。バージョン管理システムは、コード、スクリプト、自動テストなどを保存する場所として機能するだけでなく、開発者が作業の拠点となる複数のブランチを作成することもできます。

- **定期的なコード コミット**: 自動化、テスト、ツールは、効果的なパイプラインを構築するために重要ですが、頻繁なコード変更のコミットを優先するチーム文化の転換がなければ、大きな成果を上げることはできません。開発者がコードをコミットする頻度について厳密なルールはありません。ただし経験則として、個人が変更をコミットする頻度が高いほど、開発環境の生産性は高くなります。

- **ビルドの自動化**: ビルドの自動化は CI パイプラインの重要なコンポーネントであり、チームがソフトウェア ビルドを標準化できるようになります。典型的な構築プロセスには、ソース コードのコンパイル、ソフトウェア インストーラの生成、デプロイを成功させるために必要なアイテムが全て揃っていることの確認が含まれます。継続的インテグレーション プラクティスでは、このプロセスは自動化され、段階的なコード コミットをコード ベースに統合するのに役立ちます。

- **自動テスト**: 大量のコード コミットを行い、完全に自動化された構築プロセスを実行できます。しかし、プログラムが実行されたからといって、正しく実行されているとは限りません。そこでテストの出番です。自動テストは、CI パイプラインの重要な部分です。コミットごとに、バグ、セキュリティ上の欠陥、コミットの問題を特定するための一連のテストをトリガーします。これらのテストは、メイン コード ブランチを稼働状態、つまり「グリーン」に保ち、コード変更の有効性について開発者に迅速なフィードバックを提供することを目的としています。似ている CI モデルは 2 つとありません。どの組織も、独自のニーズとチームの要件に従って継続的インテグレーションを実装します。

しかし、継続的インテグレーションを正常に実装するには、すべての組織が実行する必要がある共通のステップがいくつかあります。これらは 5 つのプラクティスに分類されます。

1. **テスト戦略を策定する**: すべての CI プラクティスは、説得力のある明確なテスト戦略から始まります。どのような種類のテストを実行するのか、自動テスト シーケンスの構築にどのようなトリガーを使用するのか、開発チームが作業する各コーディング ブランチにどのテストを適用するのかを検討する必要があります。

2. **CI ツールを選択する**: CI プラットフォームの選択は、どのような組織でも CI モデルを構築する上で重要な部分です。これは、自動化されたビルド、テスト、パッケージ、リリースをトリガーするツールです。

CI プラットフォームを選択する際には、いくつかの質問をする必要があります。これには以下が含まれます。

現在のテクノロジ スタックとどの程度うまく統合されていますか？

プログラミング言語から、バージョン管理システム、サードパーティ ツールに至るまで、CI プラットフォームはスタック内の全てと簡単に統合できる必要があります。また、将来導入する可能性のある技術を考慮し、拡張可能なプラットフォームを探すことも価値があります。

コンテナをネイティブにサポートしていますか？

コンテナは DevOps と継続的インテグレーション プラクティスの重要な部分であり、CI プラットフォームが Docker などのコンテナ アプリケーションをネイティブにサポートしていることを確認することが重要です。現在はコンテナを活用しないかもしれませんが、DevOps プラクティスが拡大するにつれて、何らかの形でコンテナを使用するようになる可能性は十分にあります。

マトリックス ビルドのテスト機能が有効になっていますか？

マトリックス ビルドを使用すると、複数のオペレーティング システムやランタイム バージョンにまたがるビルドを同時にテストできます。マトリックス ビルドをネイティブにサポートしている CI ツールを探してください。マトリックス ビルドは、テストを効率化し、アプリケーションが全てのエンドユーザーで動作することを保証します。

組み込みのコード カバレッジとテストの視覚化を提供しますか？

コード カバレッジとテストの視覚化により、現在テストされているコード ベースの量、現在のテストがリアルタイムで実行されている様子、過去のテストが実行された様子を簡単に確認できます。

セキュリティ要件にどのように対応しますか？

セキュリティは、どのような技術投資においても重要な考慮事項です。特に、その投資がコード ベースやコア サービスと深く統合される場合は重要です。

3. **できるだけ早くコードを統合する**：CI の導入を成功させるには、開発者ができるだけ早くコードを共有リポジトリに統合することから始まります。

これには 2 つのメリットがあります。

- 古いブランチをメイン リポジトリにマージし直すときに発生する可能性のある、大規模なインテグレーションの衝突を回避できます。

- 小さなコード変更を定期的に統合することになるため、チーム間のナレッジの伝達に役立ち、テスト計画が簡素化されきます。

また、既存のソフトウェア開発ライフ サイクルが現在どのようになっているか、CI パイプラインを実装する際に、今後手続き上どのような変更を加える必要があるかを考慮する必要があります。これは、機能ブランチとトランクベースの開発のどちらが優れているかという話ではありません。代わりに、コーディング、テスト、マージ、レビューの安定した流れを促進する組織的な開発ワークフローがあることを確認してください。

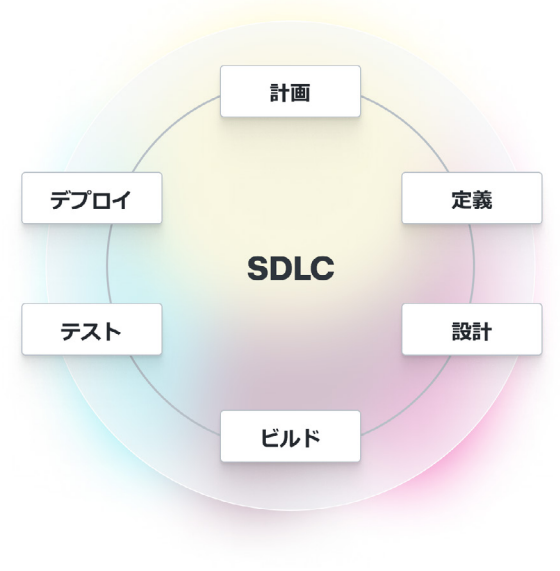

図 3: SDLC のステージ

4. **メイン ブランチが壊れたらすぐに修正する**: 経験則として、メイン ブランチが壊れたらすぐに修正する必要があります。CI モデルでは、開発者はコード変更をできるだけ早く統合する必要があります。これはよいことです。しかし、コード変更によってメイン ブランチが壊れ、開発者がさらに変更を追加し続けた場合、最初の失敗の原因を特定することが困難になります。そのため、1 つのコード変更によってメイン ブランチが壊れたときに、開発者に直ちに通知するテストを作成します。これは、DevOps の重要なプラクティスであるフィードバック ループの作成に役立ちます。ビルドをグリーンに保つ、つまり稼働状態を保つためには、テストの速度とテスト カバレッジのバランスを取るようにしてください。テストの実行に時間がかかりすぎると、どのコード変更が失敗の原因になったかを特定するのが難しくなります。最適なテスト スイートは、ビルド テストやインテグレーション テストなどの簡単なテストから開始して、より時間のかかるテストに進みます。

5. **新しい機能を導入するたびに新しいテストを作成する**: CI モデルでは、テスト スイートはソフトウェアやアプリケーションに応じて拡張する必要があります。つまり、新しい機能を構築し、大規模なアップデートを準備する際には、これらの機能を検証するためのテストも作成する必要があります。新しい機能を構築したりバグを修正したりする際には、テストを作成することを検討してください。これには時間がかかるように感じるかもしれませんが、後からテストを作成する方が、コードを構築しながらテストを作成するよりも確実に時間がかかります。

CI の背後にある考え方は、コードをできるだけ頻繁に小さな部分に統合するというものです。開発者にとっては、数週間分の作業が必要な複数のブランチをマージするよりも、単一の新機能または修正プログラムを含むコード ブランチをマージする方が容易です。CI の概念が守られていれば、チームはどのコードがビルドを壊したのか、あるいはテストが失敗したのかを特定しやすくなります。これにより、開発者はビルド エラーのトラブルシューティングではなく、価値の創造により多くの時間を費やすことができます。

CI では、複数の開発者が同じコードベースで作業する際に発生するマージの競合を最小限に抑えるため、コードが早期かつ頻繁に統合されます。次のセクションでは、プロセスの次の手順である CD を紹介します。CI と CD を連携させることで、組織は自信を持ってコードを頻繁にビルド、テスト、デプロイできます。

継続的デプロイ

継続的デプロイは、DevOps プラクティスの中でも最も高度な自動化例の 1 つです。これには、アプリケーションの設計と開発プロセス全体での厳格なテスト、チーム間の密接なコラボレーション、高度なツール導入、およびワークフロー プロセスの組み合わせが必要になります。

継続的デプロイは、正常に実装されるときに能力を発揮します。継続的デプロイを導入する DevOps 組織は、コードをより迅速にリリースし、他の企業よりもパフォーマンスが 4 ～ 5 倍高いことがわかっています。

DevOps は、SDLC の各ステージに自動化を適用することによるイノベーションと価値提供の速度向上を目指します。この観点から考えると、CD は DevOps の最終的な目的となります。一連の定義済みテストに合格する場合にすべてのコード変更が本番環境にプッシュされる、完全に自動化された SDLC です。

図 4: 実際の継続的デプロイ

自動化されたパイプラインの構築は、ある意味で、CD モデルの導入において最も簡単な部分の 1 つだと言えます。しかしながら、DevOps 組織へのジャーニーを始めるにあたって、まず CD プラクティスを構築することから始める組織はほとんどありません。というのも、継続的デプロイが象徴する文化的変化と、必要とされるテスト スイートにある程度の成熟度レベルが必要とされるからです。

図 5: CI/CD での自動化への手順

その観点から、完全に機能する CD プラクティスを達成するプロセスとジャーニーを理解することが一番です。

図 5 は、組織が SDLC の自動化について考え始める一般的な過程を示す、おおまかなジャーニー マップです。

まず、組織は CI プラクティスを構築する必要があります。強固な継続的インテグレーション プラクティスの基礎的要素 (定期的なコード コミット、バージョン管理ツール、および CI プラットフォーム) は、組織が CD プラクティスの開発を開始する土台となります。

基本的に、CD は自動化されたビルド、テスト、およびデプロイを単一のリリース ワークフローにまとめます。目的は、ソフトウェア ビルドの本番環境へのデプロイを自動化することです。各企業は、テスト スイートを構成するユニット テスト、機能テスト、およびストレス テストの適切な組み合わせを特定する必要があります。また、ビルドとリリース候補の効果的なステージングとテストのためにも、本番環境における負荷を本番前環境に反映することが重要です。

すべてを適切に行うことは、より迅速でより安定したリリースという大きなメリットにつながります。これは、組織が完全に自動化された CI/CD パイプラインで CD を実現する態勢も整えます。

テスト計画、自動化トリガー、ワークフロー構成、および CI/CD プラットフォーム全体で DevOps プラクティスが細かく調整されていき、自然と CD につながることが理想的です。その結果、ソフトウェア リリースのオーケストレーションにおける人的介入の必要性が、時間とともに消失します。

しかし、実際のところは、耐久性に優れたスケーラブルな CD モデルを実現するには、エンジニアリング リソースとツール導入に多額の投資を行う必要があります。CI/CD プラットフォームと関連するツール導入は CD プラクティスを維持するうえで大きな役割を果たしますが、チーム間のコラボレーションと、定期的なコード コミットを強調する文化的な変化も極めて重要です。

CD パイプラインのステージ

CD には単一の「モデル」はありません。組織それぞれが、組織のニーズ、ソフトウェア開発プラクティス、およびお客様の需要に固有の CD パイプラインを構築することになります。

しかし、CD パイプラインには、すべての組織がエンジニアリング計画に組み込む必要がある、一般的に認められた 4 つのステージがあります。これには以下が含まれます。

1. **検証**: CD は CI に基づいて構築され、この段階で CI が停止し、CD が開始されます。新しいコードがコミットされ、コード ベースに統合されると、リリース候補ビルドで一連のテストを実行する、自動化された検証プロセスがトリガーされます。これには、リリース候補がデプロイ後も機能することを確実にするための、機能、インテグレーション、セキュリティ、および本番環境レベルのテストが含まれます。

2. **デプロイメント**: テストを通じてコードが検証されると、自動デプロイ プロセスが開始されます。通常、より高度な実装では、コミットされると同時にコードをデプロイに移行するオートメーション ワークフローを作成します (もちろん、コードが継続的インテグレーション ステージのすべての定義済みテストに合格することを前提としています)。

3. **監視**: 継続的監視は、CD をサポートするために組織が投資する必要がある重要な要素です。監視は、SDLC 全体で実施される必要がありますが、デプロイの前、途中、およびその後で、何が機能しており、何が機能していないかを確認し、リアルタイムのアラートを受け取る能力が重要になります。チームがパフォーマンス メトリクスを可視化し、システムの負担を表示できるようにするツール導入も、役に立つ投資のひとつです。

4. **対応**: 本番環境レベルのシステム エラーに対応したり、セキュリティ インシデントを特定したり、今後開発の余地がある潜在的な新機能を特定しているかにかかわらず、イベントに対応できることは、継続的デプロイ パイプラインの重要な要素です。CD のメリットは、コードが本番環境に即時的にリリースされることです。これは、組織がデプロイ後に発生するあらゆる問題に対応して対処する準備を整えておく必要があることも意味します。対応時間を評価するために使用される一般的なメトリクスには、**平均修復時間 (MTTR)** があります。これは、経時的な改善を評価するために組織が追跡するメトリクスです。

ソフトウェアが主にフロッピー ディスクまたは CD で配布されていた時代は終わりました。クラウドベースのホスティング ソリューションとテクノロジの出現により、CD の概念を実践することで、組織はソフトウェアの更新を通じて顧客に積極的に価値を提供できます。これは、必ずしもすべての DevOps 組織が 1 日に数十回コードを本番環境にデプロイする必要があるという意味ではありません。そうではなく、コードの準備ができたらすぐに組織がデプロイできるという考え方です。本番環境へのデプロイメントは、DevOps を実践している組織ではイベントではありません。展開されるコードは機能が比較的小さく、既に厳密にテストされており、展開の自動化によって展開プロセスに人為的ミスが生じる余地がありません。

コンテナ化

問題が発生する可能性があるのはコードだけではありません。ある人のマシンで機能しても、同僚のノート PC や、さらに悪い場合は、本番環境では異なる動作をする可能性があります。コンテナ化は DevOps プラクティスで使用される技術の一種であり、開発、テスト、本番運用において、マシン間で一貫したソフトウェア環境を確保します。

コンテナ化とは、ソフトウェア コードを依存関係やオペレーティング システムとともに、他のコンピューター上で実行可能なスタンドアロン アプリケーションの形式でパッケージ化することです。これらの仮想化環境は設計上軽量で、必要なコンピューティング パワーは比較的少なくて済みます。また、基になるインフラを問わず実行でき、移植が可能で、あらゆるプラットフォームで一貫して実行できます。

アプリケーション コード、構成ファイル、オペレーティング システム ライブラリ、全ての依存関係を一緒にバンドルすることで、コンテナはソフトウェア開発でよく見られる問題の解決を支援します。よくあるのは、ある環境で開発されたコードを別の環境に移植すると、バグやエラー メッセージが発生するという問題です。たとえば、開発者が Linux 環境でコードをビルドし、それを仮想マシンや Windows コンピューターに移植すると、コードが想定どおりに動作しなくなることがあります。これに対して、コンテナはホスト インフラから独立しており、一貫した開発環境を提供します。

しかし、コンテナが特に優れているのは、簡単に共有できるという点です。コンテナ イメージはコンテナのコード、構成、その他のデータのスナップショットとして機能するファイルであり、これを使用することで SDLC の各ステージで一貫性した環境を迅速に立ち上げることができます。これにより、組織は開発からテスト、運用に至るまで、迅速かつ簡単に作業できる再現性のある環境を作成できるようになります。

コンテナについての基本的な理解が得られたので、次のセクションでは、コンテナ化が DevOps にもたらす利点に焦点を当てます。その後、コンテナが最も効果を発揮する CI/CD の具体的な側面について説明します。

DevOps におけるコンテナ化の利点

DevOps の中心になるのは、ソフトウェア開発プロセスを自動化する軽量で反復可能なプロセスです。しかし、最新のアプリケーションはますます複雑になり、特にさまざまなサービスを備えるようになっています。

コンテナは優れた標準化と反復性により複雑さを軽減し、より速く、高品質で、効率的な SDLC を実現します。GitHub は、企業が DevOps プラクティスにおいてコンテナを導入し管理するのを支援するツールを提供しています。その経験を通して、GitHub では、組織がコンテナを SDLC に正常に統合するために考慮する必要のある重要な領域を特定しました。

コンテナ化には次のような利点があります。

- **移植性**：基になる環境の一見小さな違いでも、コードの動作に影響を与える可能性があります。「私のマシンでは動作する」というよくある表現が大抵は無意味で、冗談のようになりがちなのはこのためです。また、「一度書けば、どこでも実行できる」という言葉が、ソフトウェア開発のプラクティスの向上を目指す人々に繰り返し目標とされてきた理由でもあります。コンテナはアプリケーションに必要な全てのものを一貫した移植可能な環境にまとめ、アプリケーションのパフォーマンスの標準化を容易にして、組織がこれを実現するのを支援します。

- **スケーラビリティ**：Kubernetes などのオーケストレーション管理ツールを使用することで、コンテナ同士が大規模なシステム アーキテクチャで連携して動作するようにデプロイ、構成できます。これらのツールを使用すると、新しいコンテナ化された環境のプロビジョニングを自動化し、リアルタイムの需要に合わせてスケールすることもできます。つまり、コンテナ化された環境を適切に構成すれば、ほとんど人手を介さずに迅速にスケール アップやスケール ダウンできます。

- **クラウドに依存しない**：コンテナは移植可能に構成すれば、ノート PC、ベア メタル サーバー、クラウド プロバイダーのプラットフォームなど、あらゆる場所で実行できます。また、コンテナは基盤となるプラットフォームの違いを抽象化することで、プラットフォームがロックインするリスクを軽減します。また、コンテナを使用すると、複数のクラウド プラットフォーム上でアプリケーションを実行し、プロバイダー間で切り替えることができます。

- **DevOps パイプラインへの統合**：コンテナ化プラットフォームは、多くの場合、大規模な自動化ワークフローに挿入されるように設計されています。そのため、CI/CD ツールでテストや本番環境へのデプロイなどのタスク用にコンテナを自動で作成や破棄する DevOps に最適です。

- **システム リソースの効率的な使用**：仮想マシンとは異なり、コンテナは多くの場合、より効率的で、必要なオーバーヘッドも少なくなります。通常、コンテナにネイティブなハイパーバイザーや追加のオペレーティング システムはありません。代わりにコンテナ ツールは、各コンテナを可能な限りホスト システムの共有リソースを利用するスタンドアロン環境にするのに十分な構成を提供し、これには基になるオペレーティング システムも含まれます。

- **ソフトウェア リリースの加速**：コンテナを使用すると、基になるコード ベースを連携して動作する小さなランタイム プロセスに分割することで、大規模で複雑なアプリケーションを簡素化できます。これにより、実行者は広範なコード ベース全体を扱うのではなく、アプリケーションの特定の部分に集中できるため、組織は SDLC の各ステップを加速させることができます。

- **柔軟性**：コンテナは、組織がより多くのコンピューティング リソースをすばやくプロビジョニングしてリアルタイムの需要を満たすことを可能にし、SDLC に固有の柔軟性をもたらします。また、コンテナを使用すると多くの場合、冗長性を実現し、アプリケーションの信頼性とアップタイムを向上させることができます。

- **アプリケーションの信頼性とセキュリティの向上**：アプリケーション環境を DevOps パイプラインの一部にすることで、コンテナはアプリケーションの残りの部分と同じ品質を保証されます。また、コンテナは連携して動作しますが、コンテナが提供する分離環境により、アプリケーションの一部分の問題が広範なシステムに影響を与えるのを防ぐのを容易にします。

コンテナはソフトウェアを効率的で大規模に開発するための最新の方法を提供します。基本的な DevOps プラクティスと相性の良い柔軟性と再現性を提供します。次のセクションでは、コンテナによって CI/CD プロセスを強化する方法について説明します。

CI/CD におけるコンテナの仕組み

CI/CD パイプラインは DevOps ワークフローを動かすベルトコンベヤーと考えることができます。CI/CD パイプラインが効率的であるためには、速度と完全性のバランスが取れていなければなりません。速度が遅ければ、CI/CD フローでパイプラインを通過するよりも早くコミットが発生するため、バックログが発生するリスクがあります。完全性に欠ければ、本番環境で問題が発生し、CI/CD パイプラインに対する信頼が失われます。

コンテナ化により、CI/CD の両側面が主要ステージでどのように強化されるかを以下で説明します。

1. **統合**：コンテナを使用することで、コードの変更を大規模なコード ベースに統合するときにゼロから始める必要がなくなります。アプリケーションの依存関係をあらかじめ保持している基になるコンテナを作成し、インテグレーション フェーズでそれを変更します。
2. **テスト**：コンテナは必要に応じて迅速にプロビジョニングや廃止できます。明示的なテスト環境を手動で維持したり、構成スクリプトが環境を構築するのを待ったりする必要がなく、コンテナをプロビジョニングして大規模に自動でデプロイできます。そのため、テストをより迅速に実行でき、人手を介してテスト環境を構築する必要も少なくて済みます。
3. **リリース**：全てのテストに合格すると、CI/CD パイプラインのビルド フェーズでコンテナ イメージが生成され、コンテナ レジストリに保存されます。通常、そのイメージが存在していれば、リリースとデプロイのフェーズで行われる作業の多くはすでに完了しています。Kubernetes などのオーケストレーション ツールは、コンテナのデプロイ先やコンテナ同士がどのようにやり取りするかを管理します。

DevOps CI/CD ワークフローにおけるコンテナの予測可能性とスケーラビリティは、DevOps の基本的な原則を反映しています。コンテナの使用は DevOps の絶対要件ではありませんが、コンテナを効果的に使用している組織は、DevOps 成熟度モデルでより高い機能を示す傾向があります。コンテナは、一貫性と再現性の向上を可能にし、最新のアプリケーション開発の摩擦を軽減するのに自然に適合すると考えられています。DevOps の実践を成功させるには、この一貫性と再現性、そして信頼性が不可欠です。

DevOps のセキュリティ (DevSecOps)

DevOps は、多くの組織がソフトウェアを構築してリリースする方法に変革をもたらしました。しかし最近まで、SDLC の 1 つの側面 (セキュリティ) は、DevOps の対象外でした。DevSecOps は、DevOps がソフトウェア開発の全ステージを通じて品質、速度、緊密なコラボレーションを優先するのと同じ方法で、セキュリティを SDLC に組み込むことで、これを修正しようとしています。

DevSecOps は、SDLC の全てのステップにセキュリティを組み込もうとします。これは理想的に、コーディングからブランチのマージ、ビルド、デプロイ、そして本番ソフトウェアの運用に至るまで、各ステージでセキュリティ関連のテスト (自動化されているかどうかにかかわらず) が行われることを意味します。さらに、DevSecOps は、製品に携わる全員がそのセキュリティに責任を負うという考え方を推進します。これにより、チームは本番環境に移行する前に脆弱性を発見できるようになり、ソフトウェア リリースを遅らせる可能性のある後半ステージの手動によるセキュリティ レビューの必要性を軽減します。

DevSecOps を導入する組織には通常、以下のようなメリットがあります。

- **データ侵害のリスクの軽減** : DevSecOps は、設計によってコードを安全にすることを目指しています。SDLC 全体でセキュア コーディングの文化的プラクティス、セキュアな開発者環境、自動セキュリティ テストを組み合わせることで、セキュリティの脆弱性や欠陥が本番ソフトウェアに混入する可能性を軽減するのに役立ちます。
- **コンプライアンスの向上** : DevSecOps の実行者は、多くの場合、自動化を使用してコード コンプライアンスを適用し、CI/CD パイプラインに直接ポリシー適用ツールを統合します。
- **依存関係の信頼性向上** : 最新のテクノロジ スタックは、多くの場合、パブリック パッケージ リポジトリからのサードパーティ コードに大きく依存しています。DevSecOps の実行者は、ソフトウェア リリース前に潜在的な問題を特定するために、ツールや自動テストを頻繁に活用しています。
- **エンド ユーザーへの迅速な価値提供** : セキュリティ優先の文化を生み出し、自動チェックを適用することで、DevSecOps は、コードのデプロイを遅らせる個別のセキュリティ レビューの必要性を軽減します。

DevSecOps プラクティスを成功させるには、SDLC の全てのステージにセキュリティを組み込む必要があります。これは組織によって異なり、業界によって従わなければならない規制が異なることも多いものです。DevSecOps では、SDLC の各フェーズにセキュリティを組み込むことは、ソフトウェア開発を遅らせる可能性のある煩わしい管理を実装することを意味しません。まったく逆です。効果的な DevSecOps プラクティスのセキュリティはリリース自体の一部になり、より迅速でセキュアなデプロイメントにつながります。

ベストプラクティス

DevSecOps は、SDLC 全体にセキュリティを組み込む必要があると主張しています。組織が既に DevOps を実践している場合でも、DevOps 文化を導入する方法を検討している場合でも、ここでは、DevSecOps プラクティスを確立するために必要な基礎となるベストプラクティスを紹介します。

- **DevSecOps 文化を構築する**: DevSecOps の成功は、全員がセキュリティの責任を負うことに依存します。つまり、SDLC の各担当者が、アプリケーションとインフラの設定を防御的にコーディング、ビルド、テスト、構成します。DevOps と同様に、DevSecOps は、各個人が協力して可能な限り最もセキュアな製品を構築するオープンな文化の中で成功します。

- **製品にセキュリティを設計する**: DevSecOps は、最初の計画ステージから本番環境レベルのコードのデプロイまで、製品にセキュリティを設計することを目指しています。つまり、セキュリティ作業は機能作業と並行して計画され、実行者には開発作業の各ステージを通じてセキュリティのナレッジとテストが提供されます。目標は、セキュリティをチームの日常業務の一部にすることです。

- **脅威モデリングのプラクティスを構築する**: セキュリティの脆弱性の種は、多くの場合、コード行が記述される前に蒔かれます。計画フェーズで潜在的な脅威をモデル化し、そのような問題を軽減するようにインフラとアプリケーションのアーキテクチャを設計します。また、信頼できる人物がシステムへの侵入を試みる侵入テストを定期的に実施することで、脅威モデルで見逃している可能性のある弱点を明らかにすることができます。

- **速度とセキュリティを自動化する**: 自動テストは SDLC 全体で使用され、適切なセキュリティ チェックが適切なタイミングで行われます。これによって、セキュリティ要件が満たされていることを確認しながら、コア製品の構築に集中する時間を増やすことができます。

- **製品開発におけるセキュリティ チェックポイントを計画する**: SDLC でリスクプロファイルが変化する移行ポイントを特定しますこれは、開発者が自分のコードをメイン ブランチにマージした時点である可能性があり、そのコードがチーム メンバーのマシンで実行され、最終的に本番環境に到達する可能性が高まる可能性があります。その場合、プル リクエストを開始することは、適切な手動エスカレーションとともに、自動セキュリティ チェックの適切なトリガー イベントとなる可能性があります。

- **セキュリティの失敗を学習の機会としてとらえる**: DevOps の継続的な改善の文化に基づいて、DevSecOps プラクティスを成功させるには、セキュリティ インシデントを学習の機会に変えるように努めます。これは、監査ログを活用し、インシデント レポートを作成し、悪意のある行為をモデル化して、ツール、テスト、プロセスを改善し、アプリケーションとシステムのセキュリティを強化することで実現できます。

- **依存関係を常に把握する**：依存関係から生じる潜在的な脅威を理解し、軽減することは、製品のセキュリティにとって非常に重要です。社内コードと同じ脅威モデリングと自動テストを依存関係に適用します。GitHub では、オープン ソース ソフトウェアに存在する何千万もの脅威を特定し、その詳細を共有することで、組織や開発者が脆弱性を認識し、回避できるよう支援しています。

- **分析機能とレポート機能を構築する**：継続的監視は DevSecOps プラクティスの重要な要素であり、これにはリアルタイムのアラート、システム分析、プロアクティブな脅威監視が含まれます。アプリケーションと DevSecOps パイプラインのあらゆる側面を測定することで、アプリケーション ヘルスを理解するための共通点を作成できます。ダッシュボードとアラートのレポートにより、問題が早期に浮き彫りになります。問題が発生した場合、設定したテレメトリー (アプリケーション レベルのログ収集など) により、インシデントの解決と根本原因の分析のためのインサイトが得られます。

DevSecOps 文化の構築は、セキュリティを全員の責任とすることから始まります。これは、多くの組織にとって大きな変化となる可能性があります。従来、セキュリティは開発者がセキュリティ専門家の手に委ねるものでした。エンジニアリング チームは、セキュリティ プラクティスを、ソフトウェアを迅速にリリースするための障害とみなしていたため、しばしば摩擦が発生していました。さらに極端なケースでは、セキュリティはその過程での単なる承認印にすぎませんでした。

DevSecOps は、コードの作成、テストの実行、およびサービスの構成の際に、セキュリティを SDLC の中核とすることで、この認識を根本的に変えようとしています。DevOps モデルが開発者と運用チームをまとめたのと同じように、DevSecOps はセキュリティを最前線に押し上げます。新しい機能や修正はそれぞれ、セキュリティへの影響を考慮することから始まります。セキュリティとコンプライアンスのポリシーは、自動テストを通じて適用されます。最新の組織では、DevSecOps は単なる「DevOps」となり、セキュリティは SDLC ワークフローのあらゆる側面に組み込まれています。

DevOps の計画、ツール、機能

自動化、継続的な監視、継続的なフィードバックは、DevOps モデルの重要な部分です。

包括的な用語として、DevOps ツールには、SDLC 内のプロセス自動化、組織のコラボレーション改善、監視とアラートの実装を行うあらゆるアプリケーションが含まれます。組織は SDLC の各ステージに対応するために「DevOps ツールチェイン」(DevOps プラクティスで使用するツールのコレクション) の構築に投資することがよくあります。

DevOps ツールチェインはあらゆる DevOps プラクティスのコア テナントであり、組織が SDLC に自動化を適用し、より高品質なソフトウェアを迅速にリリースするのを支援します。DevOps ツールチェインは、DevOps の具体的側面の 1 つでもあります。

DevOps ツールチェインを構築するために「オールインワン」プラットフォームに投資する組織もあります。さまざまな最善の組み合わせで ソリューションを統合してツールチェインを構築する組織もあります。しかし、DevOps または DevOps ツールチェインを構築する万能なアプローチはありません。

> 私たちの哲学は、将来の企業のために自動化と優れた DevOps を構築することです。」
>
> Adobe、シニア SCM エンジニア、**Todd O'Connor 氏**

ツールのためのガイド

DevOps パイプラインの各ステージにはユニークな考慮事項があり、1つ、または数多くのツールがその解決に役立ちます。次のいくつかのセクションでは、ツールを選択する際に留意すべきさまざまな考慮事項とともに、DevOps パイプラインの 8 つのステージすべてを再度紹介します。1 つのツールで DevOps パイプラインの複数のステージに対応する場合もあります。逆に、DevOps パイプラインの 1 つのステージを複数のツールで表すこともできます。使用するツールにかかわらず、最も成功を収めている組織では、DevOps パイプラインの各ステージ間でまとまりのあるフローと統合を持つことになります。

このセクションでは、DevOps ツールが将来の戦略計画、コミュニケーション、ロードマップをどのように形作ることができるかを見ていきます。

DevOps 計画およびコラボレーションツール

DevOps は主に、以前はサイロ化されていたチームを SDLC の全ステージにわたって統括することを目的としており、それは計画ステージから始まります。チャット アプリケーションからプロジェクト管理ツールまで、組織が DevOps ツールチェインに導入し、計画ステージで組織内の連携を強化し、コラボレーションを促進するツールは多数あります。

DevOps 計画およびコラボレーション ツールは通常、次の 2 つに分類されます。

- **製品とロードマップ計画**: 作業を計画、追跡、管理する一元化された場所を設けることは、現在の開発チームや DevOps 組織にとって基礎的な機能です。優れたツールは、組織が計画、スプリント、ロードマップを作成し、初期計画から配信された最終製品まで作業を割り当て、追跡可能にするのに役立ちます。例については、当社の公開された製品のロードマップ計画を参照してください。これは、GitHub のプロジェクトを使用して作成しています。

- **チームのコミュニケーション**: 計画プロセス全体にわたりコミュニケーションを維持することは、コラボレーションを促進するうえで重要です。特定の決定に至った会話の記録を保存しておくことも非常に有用です。ここでは、GitHub Discussions、チャット アプリケーション、問題トラッカーなど、チームの会話に活用できるツールが重要になります。GitHub にはチームが Slack または Microsoft Teams と統合できるようにするアプリが用意されています。優れたツールはプロジェクト計画とも統合されます。つまり、ディスカッションを実行可能な作業に変えることも、作業を始める前にさらに会話が必要な場合はアイディアをディスカッションに変えることもできます。

DevOps ビルド ツール

開発者がコード変更を中央リポジトリにコミットすると、構築ステージが開始され、バージョン管理を使用した共有リポジトリの作成、開発環境のプロビジョニング、コードの統合などが行われます。

このステージで、組織は通常、次の DevOps ツールを活用することができます。

- **バージョンとソースの管理**: バージョン管理システムは、ファイルの変更を自動的に記録し、以前のファイル バージョンの記録を保持するように設計されており、コードのロールバック、履歴の参照、複数のコード ブランチを実行できるため、開発者は共同でコーディングし、並行して作業できるようになります。GitHub などのプラットフォームには、プル リクエストなどの機能を使ったバージョン管理とソース管理が提供されており、メインコード ブランチに統合される前に個々の開発者は提案されたコード変更についてレビューを行うことができます。優れたバージョン管理とソース管理プラットフォームは広範な DevOps ツールチェインと統合され、製品チームは SDLC 全体でコラボレーションできます。

- **本番運用前開発環境**：DevOps プラクティスでは、開発者は運用環境にできるだけ近い仮想環境を活用する必要があります。これらの環境は互いに同一であり、簡単にプロビジョニングできるため、どの開発者も一貫した環境でコード変更をすばやく構築とテストができます。多くの場合、組織はコンテナ化されたプラットフォームや GitHub Packages などのレジストリを活用して、開発チーム向けに標準化された本番運用前環境をビルドします。理想的には、これらのプラットフォームをソース管理ソリューションと統合して、チーム メンバーが新しいコードをコミットすると、本番運用前環境の自動化されたプロビジョニングがトリガーされるようにする必要があります。

- **クラウドベースの IDE**: クラウドベースの IDE では、事前に構成されて迅速にプロビジョニングできる包括的な開発環境が提供されます。これらのツールは、マシン間のセキュリティ設定などの開発者環境を標準化するのに役立つため、DevSecOps（および、より広範な開発サークル）でますます一般的に使われています。クラウドベースの IDE は一元管理されており、個々の開発者のコンピューターにコードを残さないため、開発全体のセキュリティが強化されます。GitHub Codespaces などのツールもコア DevOps プラットフォームに密接に統合されます。これにより、開発者の環境を立ち上げるのに要する時間が短縮され、ローカルで構築とテストを実行するまでの待機時間を減らすことで、開発速度を迅速化できます。

- **IaC**: クラウド インフラ、または **Infrastructure as a Service (IaaS)** の台頭により、リアルタイムの需要に合わせてリソースをよりシンプルかつ迅速にプロビジョニングできるようになりました。また、組織の間では、複雑なクラウドベースのインフラを大規模に管理する必要性が生じています。IaC は DevOps のベスト プラクティスを活用し、GitHub などのバージョン管理システムから YAML ファイルを介してクラウド インフラ リソースをプロビジョニングおよび管理します。これらのファイルはプル リクエスト、コード コミット、コード マージなどのイベントによってトリガーされる CI/CD ワークフロー オートメーションを指定します。このイベントが発生すると、ワークフローによってクラウド インフラ リソースのプロビジョニングと管理が自動化されます。GitHub Actions などのツールによってこのタイプの統合が提供されるため、CI/CD を使用してリポジトリから直接インフラを簡単に管理できます。

DevOps CI ツール

CI はあらゆる DevOps プラクティスの中心であり、頻繁なコード コミットの文化的プラクティスと、そのコードを正常に統合してビルドを作成するための自動化を組み合わせます。

- **CI**: プラクティスとして、多くの場合、CI は 1 日に複数のコード変更を共有リポジトリにコミットし、自動化を使用してこれらの変更を統合し、マージされたコード ベースに一連の自動テストを適用して安定性を確保し、デプロイするためのコードベースを準備します。このレベル

の自動化にはバージョン管理ソリューションと大きな CI/CD プラットフォーム間の密接な統合が必要になります。これにより、DevOps 組織はコード コミットによってトリガーされる CI/CD パイプラインをビルドできるようになります。優れた CI ソリューションを探すとき、バージョン管理ソリューションと簡単に統合できることを確認する必要があります。この統合は、開発チームがコード変更をコミットするとすぐに開始される自動化パイプラインをビルドできるかどうかを確認するうえでの鍵となります。この統合レベルの良い例は GitHub プラットフォームに含まれており、GitHub Actions を介したプラットフォーム ネイティブの CI/CD や、多数のサードパーティ CI/CD サービスの事前構築済み統合を特徴としています。選択する CI/CD プラットフォームが SDLC の全ステージで自動的にテストを適用でき、コンテナ化プラットフォームのネイティブ サポートが含まれていることを確認する必要もあります。

- **自動テスト**: 自動テスト ツールは DevOps ツールチェインの中核となります。ほとんどのプラットフォームは、パイプラインの重要な部分、例えばコード変更が main ブランチにマージされた後などに自動テストを簡単に組み込むことができる機能を提供しています。目標は、SDLC の重要な時点で適用される基本的なユニット テスト、統合テスト、受入テストを備えた包括的なテスト戦略を策定することです。優れたテスト ツールは CI/CD プラットフォームにシームレスに、またはその一部として統合され、コード カバレッジとテストの視覚化機能が組み込まれています。マトリックスのビルド テスト機能を実行でき、複数のオペレーティング システムとランタイム バージョンにまたがるビルドを同時にテストできるテスト プラットフォームを探す必要もあります。使いたい自動テスト ソリューションに、お好みのチャット アプリケーションと統合される監視とアラートが備わっていることを確認することもお勧めします。それにより、何かが壊れた場合はすぐに通知を受け取り、発生した根本的な問題の修正に取り組むことができます。GitHub Actions などのツールを使用して、たとえばテストに失敗するとすぐに修復できるように、チャット アプリケーションにアラートを送信できます。

- **パッケージ化**: コード変更が CI/CD パイプラインの全てのテストに合格すると、個別のコード ユニットにパッケージ化され、デプロイの準備が整います。DevOps 組織は通常、GitHub Packages などのパッケージ マネージャーを活用して、リリースに備えてソフトウェア パッケージを共有リポジトリに配信するのを促進します。パッケージ マネージャーは手動でインストールする必要性を取り除き、特定のプロジェクト内のコード依存関係をバンドルするのに役立ちます。コードライブラリごとにパッケージ マネージャーは異なりますが、バージョン管理システムおよび CI/CD プラットフォームと統合できるソリューションを見つけることをお勧めします。

DevOps CD ツール

CD はソフトウェアのリリース時に人が介入する必要性を取り除くことによって CI/CD に構築されます。CD プラクティスは、SDLC の全ステージに自動化を適用します。つまり、コード変更が全ての自動テストに合格した場合、運用環境にデプロイされます。次のツールは CD をサポートしています。

- **自動デプロイメント**: 自動デプロイは CD の中核であり、自動デプロイをサポートするツールチェインが備わっています。これらの機能はほとんどの CI/CD プラットフォームに通常含まれます。ただし、CD パイプラインをビルドする万能なアプローチはありません。これは、全てのアプリケーションや環境で動作するわけではありません。CD への投資を決めたら、複数の環境の開発と管理をすぐにサポートできるプラットフォームを探します。重要なのは、「サーバー ドリフト」、つまり、開発環境、運用前環境、運用環境間の違いを起こさないようにするのに役立つソリューションが必要であるということです。ブルーグリーン デプロイをサポートするプラットフォームを検討する必要もあります。これにより、トラフィックをアプリケーションの旧バージョンから新しいリリースに徐々に移行できるため、運用環境での安定性が確保されます。GitHub では、ネイティブ CI/CD ツールである GitHub Actions の一部として、デプロイ ダッシュボードと CI/CD 視覚化を提供し、CD ツールチェインにとって、これらは中核的な機能だと考えています。これは、さまざまなコード ブランチ、自動テスト結果、監査ログ、進行中のデプロイの完全な視覚化を、発生と同時に DevOps 組織に提供することを意図しています。
- **構成管理**: 構成管理は、製品のライフサイクルを通じてコア インフラとアプリケーション システムに必要なさまざまな環境構成を技術チームが管理するためのプロセスです。これは、自動化を通じて CI/CD およびバージョン管理と頻繁に組み合わされるプロセスでもあります。CI/CD パイプラインが自動化を SDLC 全体に適用するように、構成管理ツールでも構成の変更はトリガーベースのイベントに応じて自動的に適用されます。これらの自動ワークフローはプラットフォームによるコンテナ クラスタのオーケストレーションおよび管理に使用できます。GitHub リポジトリと問題により、IT プロフェッショナルは IaC と **Configuration as Code** (**CaC**) の両方に対してテキストベースの構成ファイルを生成するシステムを簡単に操作できるようになります。

継続的テスト ツール

DevOps プラクティスでは、テストは CI/CD にとどまらず、SDLC 全体にわたり進行していくプラクティスです。さらに重要なことに、DevOps は、サイロ化された品質管理チームを SDLC 全体にわたり自動化と総合的なテスト戦略を活用する継続的テスト プラクティスで置き換えることを目指しています。

各 DevOps 組織は必要性に従って独自の継続的テスト戦略を設計します。GitHub Actions はテストに関連するワークフロー オートメーションを提供し、オープン ソースと商用のテストツールの豊富なセットをサポートしています。全ての継続的テスト戦略で、SDLC 全体にわたり以下のテスト タイプの組み合わせが活用されます。

- **ユニット テスト**: ユニット テストはコードの小さなユニットをテストして、それらが分離されたコンポーネントで正しく構成されていることを検証する方法です。継続的テストのプラクティスにおいて、最も簡単にビルドでき、最も速く実行できる、自動化するための基礎となるテストでもあります。

- **統合テスト**: リポジトリへのコード変更のコミットを行うと、統合テストにより、ビルドの安定性と、コード ベースが継続して正常に動作することが確認されます。これらのテストはさまざまなアプリケーションプロセスやコード ユニットがまとめてマージされる場合に発生する欠陥の特定に使用されます。統合テストは一般的に、コード変更がコード ベースにコミットされるとすぐに開始され、アプリケーションの複数の部分の相互作用をテストするように自動化されます。

- **エンドツーエンド テストと回帰テスト**: エンドツーエンド テストと回帰テストは、統合テストを基に、コード ベースがパッケージ化されて、運用前環境にステージングされた後に適用されます。これらのテストはコードの変更によって古い欠陥、バグ、問題が再び発生しないかどうかをチェックするために使用されます。回帰テストは一般的に、アプリケーションが想定どおりに動作し、以前に特定された問題が含まれていないことを確認するために、デプロイの前後に使用されます。

- **本番運用テスト**: アプリケーションがデプロイされた後、本番運用レベルのテストはアプリケーションヘルスと安定性を監視し、エンドユーザー対して問題が発生する前に問題を特定します。重要なのは、これらのテストは、運用前の環境では完全に再現できないライブ ユーザーのトラフィックを伴う本番環境の潜在的な問題を、組織が特定するのに役立つ点です。

DevOps 運用ツールと継続的監視ツール

成功する DevOps プラクティスは SDLC の全てのステージに影響します。これには運用レベルのソフトウェアも含まれます。つまり、企業はアプリケーションとインフラのパフォーマンスを評価するためのコア操作と継続的監視ツールに投資する必要があります。正しく使用されれば、これらのツールは継続的に SDLC 全体にわたる潜在的な問題を特定します。

- **アプリケーションとインフラの監視**: アプリケーションとインフラの監視は正常な継続的監視プラクティスの中心的なコンポーネントです。優れたツールはアプリケーションとインフラの健全性を年中無休で自動監視し、問題が発生すると DevOps 実行者にアラートを送信して、根本的な問題が何であるかを可視化します。理想的には、運用前環境と運用環境でアプリケーションヘルスを監視し、プロセスの問題や全体的なパフォーマンスを向上できる領域を追跡する必要があります。これは、インフラにも当てはまります。つまり、監視によって、IaC や構成管理ポリシーの改善方法に関するインサイトを得ることができます。バージョン管理ツールやチャット アプリケーションと統合できるツールを探して、適切な担当者に即座にアラートを送信し、問題を作成してソリューションの作業範囲を概説できるようにします。

- **監査ログ**：監査は効果的な運用と継続的監視プラクティスの中心であり、何かが発生した場合にあらゆるインシデントを解決します。DevOps 実行者は何がどこでいつ発生したのかを記録でき、問題につながる行動モデルが形成されたことに批判的になることができ、アプリケーションとインフラの健全性を改善するうえで重要な役割を果たします。コア サービスとアプリケーションのパフォーマンスを改善するために必要な情報をチームに提供する、ライブ ログと監査保持期間を使用する DevOps ツールを探します。

- **インシデントと変更の追跡**：DevOps の主な目的は、密接なコラボレーションと自動化を通じて組織が高品質のソフトウェアをより迅速に出荷するのを支援することです。インシデントや変更を発生時に追跡して適切な担当者と共有することが極めて重要です。成功する DevOps ツールチェインをビルドするには、コア DevOps プラットフォームと共有リポジトリ上でインシデントと変更を表示するツールを組み込む必要があります。インシデントと変更に関する全てのレポートを一元管理することが推奨されます。目標は、問題の特定と修正を容易にする信頼できる唯一の情報源を作成することです。

- **継続的フィードバック**：DevOps の中核的原則である継続的フィードバックは、ユーザー動作やコア製品に関する顧客からのフィードバックを追跡し、新機能やシステム更新への今後の投資に役立つ実用的なデータの構築に重点を置いたプラクティスです。これにはユーザーが製品をどのように評価しているかに関する NPS アンケート データを含めることができます。製品自体にユーザー動作の追跡とモデル化を含めることもできます。継続的なフィードバック プラクティスを構築するためには、製品内のコア エリアを特定し、ソーシャル メディアやレビューのような製品外の場所でも、予期せぬユーザー動作や顧客の痛点を特定できる必要があります。ユーザー動作をモデル化して分析できるツールを探します。ソーシャル メディアやレビュー サイトの履歴パターンを追跡するために使用できるソーシャル リスニング ツールを検討することもできます。

セキュリティ ツールと DevSecOps ツール

DevOps がプラクティスとして進化するにつれて、コア SDLC からサイロ化されることが多いセキュリティに対して、より伝統的なアプローチから移行する必要性が強調されています。高品質なコードを確実に出荷するには、セキュリティを DevOps のプラクティスの中核とすることが重要です。このプラクティスは一般に DevSecOps と呼ばれ、セキュリティを SDLC の全ステージに統合し、CI/CD パイプラインの中核にすることを目的としています。

DevOps に投資する企業は、多くの場合、ソフトウェア セキュリティを確保するために DevSecOps プラクティスの構築にも投資する必要性を認識しています。これには通常、組織が潜在的な脅威をモデル化し、SDLC の主要ステージで自動セキュリティ テストを適用するのに役立ついくつかのツールが含まれます。組織は個々のツールを取得してソリューションを作成しようとしますが、GitHub Advanced Security などの統合製品は DevSecOps をチームに導入する際の摩擦を軽減できます。DevOps ツールチェインを DevSecOps ツールで補完することで、多くの場合、企業は以下のようなソリューションを探します。

- **脅威モデリング**：セキュリティの脆弱性や潜在的な弱点を見つけるのは、実際、ソフトウェアをリリースした後ではなく、開発中の方がはるかに簡単です。脅威モデリングは DevSecOps 実行者が SDLC の初期の計画ステージから取り組み、あらゆる問題を予測して解決するための計画を策定するプラクティスです。現在の DevSecOps 組織は脅威と軽減策を予防的に特定するために自動化と監視を活用する脅威モデリング ツールにも投資しています。優れたツールはアプリケーションとインフラの脅威を調べて、基になるコード ベースとインフラ アーキテクチャの変更を自動的に追跡します。チームの関係者に更新を提供し、SDLC 全体にわたりリスク評価スコアを示すために、コア DevOps ツールチェインと統合できるソリューションを探します。

- **セキュリティ ダッシュボード**：潜在的なリスク、テスト カバレッジ、アラートなどを含むセキュリティ プロファイルを一元表示することは、どの DevSecOps プラクティスにとっても不可欠です。セキュリティ ダッシュボードは関連する全てのセキュリティ情報を照合して分類し、問題のトリアージとタスクの割り当てを迅速に行う方法としてよく使用されます。GitHub では、プロジェクトとリポジトリにまたがるリスク カテゴリーやアラートの詳細を示すのに役立つ GitHub Advanced Security を使ったセキュリティの概要ページを提供しています。理想的には、広範な DevSecOps セキュリティ ツールチェインと統合してセキュリティ プロファイルを一元表示できるツールを探す必要があります。

- **静的アプリケーション セキュリティ テスト (SAST)**: SAST ツールは、実行する前にコードを評価し、潜在的なセキュリティ リスクや脆弱性を特定するために使用されます。重要なのは、これらのツールは実行に稼働中のシステムを必要とせず、静的なコード ベースに対して実行できる点です。優れたツールは共有リポジトリに直接統合され、セキュリティの脆弱性を検出し、依存関係のレビューを実施し、機密のパスワードやシークレットをスキャンし、運用環境に移行する前にコーディング エラーを特定します。これらのツールは、コード ベースに存在するあらゆる問題を検出、トリアージし、修正の優先順位付けを簡単に実行します。理想的には、リポジトリと統合し、分析に基づいて問題を構築するように自動化できるソリューションを探す必要があります。GitHub では、たとえば Dependabot という SAST ツールがあます。これは、既知のセキュリティの脆弱性に対する全ての依存関係を分析し、プラットフォーム上のあらゆるリポジトリと直接統合されます。

- **動的アプリケーション セキュリティ テスト (DAST)**: DAST はアプリケーションに対する悪意のある攻撃を再現し、実際のセキュリティを危険にさらす可能性のある潜在的な脆弱性を見つけるために使用されます。DAST ツールは通常、運用前環境のアプリケーションを分析し、運用環境に移行する前に DevSecOps 実行者がセキュリティ上の欠陥を特定しやすくします。これらの欠陥には通常、攻撃者が SQL インジェクション攻撃や**クロスサイト スクリプティング** (**XSS**) 攻撃などを実行するために悪用できる根本的な問題が含まれています。優れた DAST ツールは、広範な SDLC 内でデプロイを自動化できるよう、選択した CI/CD プラットフォームと統合されます。

- **対話型アプリケーション セキュリティ テスト (IAST)**: IAST ソリューションは実行中のアプリケーションのリスクと脆弱性を特定してプロファイリングを行うために、多くの場合、SDLC の初期ステージでリリース前に使用されます。これらのソリューションはソフトウェア インストルメンテーションを活用し、手動テストと自動テストを通じて運用前環境の情報を監視して収集します。優れた IAST ソリューションにはオープン ソース コンポーネントの脆弱性を特定する**ソフトウェア構成分析** (**SCA**) ツールが含まれます。

- **コンテナ イメージ スキャン**: コンテナのアーキテクチャは軽量であるため、DevOps 組織はアプリケーションのビルド、テスト、デプロイ、更新を高速かつ柔軟に行うことができます。ただし、大規模なコンテナ環境は、外部からのアクセス数や脆弱性の可能性から、セキュリティ リスクをもたらします。あらゆるリスクを軽減するために、DevSecOps 実行者はコンテナ スキャン ツールを活用してコンテナレジストリの問題を特定し、実行時にコンテナ クラスタをスキャンして、脆弱性が運用環境に入り込むのを防止します。CI/CD パイプラインに統合でき、デプロイ前のビルド、統合、パッケージ化ステージなど、SDLC の特定の時点で実行されるように自動化できるツールを探します。

監視ツール

監視は効果的な DevOps プラクティスの核となる要素であり、潜在的な問題が本番環境に持ち込まれる前に検出して把握し、本番環境で発生する可能性のある問題を明らかにするために重要な手段です。

継続的監視

少し前まで、監視はコストがかさむものでした。ツールは貴重なシステム リソースを消費し、人間の介入も必要でした。さらに、多くの場合、これらのツールが提供するデータを解析してアクションにつなげるには時間がかかっていました。結果的に、組織は通常コーディングの問題や本番環境レベルのパフォーマンスといったミッション クリティカルなプロセスのみを監視していました。

現在では、より高度なツールの導入に伴いデータの収集ははるかに簡単になりましたが、データの量も大幅に増加しています。つまり、現在の組織ははるかに膨大なデータを適切に管理して解釈し、それをアクションにつなげる最適な方法を見極める必要があるということです。

継続的監視は、SDLC の各所に監視を取り入れてこの問題を解決しようとするプラクティスです。主な目標は、潜在的な問題を迅速に検出してリアルタイムのフィードバックを提供できるようにすることです。

継続的監視のプラクティスでは、一連のツールや自動テストを活用して、アプリケーションとその基になるインフラの新しいコードと本番運用のパフォーマンスを評価します。主な目標は、全てのシステムの自動化された包括的なビューを提供し、適切な人がいつどこに介入すべきかを把握できるようにすることです。

多くの場合、継続的監視のベスト プラクティスではできるだけ多くのデータを収集し、システム全体を監査して、潜在的な運用上の問題やコンプライアンスおよびセキュリティのリスクを分析することを優先します。

DevOps プラクティスに監視ツールを導入する

DevOps そのものへの移行と同様に、効果的な DevOps 監視戦略を策定するには文化やプロセス、ツールを融合させる必要があります。他の組織が監視を管理する事例からヒントを得ることはできるものの、組織が採用する厳密なモデルは、組織とその SDLC 独自のニーズによって決まります。

どのようなデータを収集すべきかについては、ガイダンスとなるフレームワークが多数存在します。一方、どこに監視を導入すべきかというのは、最適化の問題です。どのような質問に答える必要があるでしょうか。その答えを得るにはどのようなデータが必要でしょうか。そのデータに基づいてどのようなアクションを実行するでしょうか。そこに誰が関与すべきでしょうか。

DevOps 監視ツールに求める機能

DevOps プラクティスに監視を組み込むのに役立つツールには豊富な選択肢があります。採用する厳密な製品は、組織の SDLC の状態やアプリケーションのインフラによって決まります。ただし、監視ツールを評価する際に確認すべき核となる初期段階の質問が 2 つあります。

そのツールは実用的でしょうか。そのツールを DevOps パイプラインやお使いのその他のツールに統合し、そのデータに基づいてアクションやアラートを自動化できるでしょうか。

そのツールから新しい情報を得られるでしょうか。生成するデータを増やすのは簡単ですが、データが増えればストレージがいっぱいになり、注意を払って管理することが求められます。差益が得られるツールではなく、監視の新たな手段を得られるツールを選びましょう。

DevOps プラクティスにおいて監視を実施すべき領域はいくつかあり、他の領域に比べてわかりやすい領域も存在します。インフラとネットワークを監視するツールを使用して、メモリや CPU などの制約がアプリケーションにどのような影響を与えているかを把握します。アプリケーション層では、**アプリケーション パフォーマンス監視** (**APM**) ツールを使用して、アプリケーションのパフォーマンスに関するシグナルを表示します。これらのツールは、アプリケーションをより最適化する方法に関するインサイトを提供します。GitHub には、GitHub Actions で CI/CD ワークフローを監視し、GitHub Advanced Security でセキュリティ調査結果を集約して、組織のインサイトを使用して、開発者のベロシティ メトリクスを提供する機能があります。

ツール導入は、多くの場合、DevOps プラクティスの最も可視的な側面です。これは、SDLC のすべてのステージに影響を与える DevOps の文化とプロセスの実用的な現れです。DevOps では、多くの場合ツールを使用して可能な限り自動化を採用し、フィードバック ループを作成して、組織のリソースを解放します。自動化された監視とレポート作成ツールでフィードバック ループを促進することで、DevOps はチームがより堅牢なソフトウェアを構築するのを支援します。問題が発生した場合、DevOps の自動化とツールは、従来のソフトウェア開発プラクティスよりも迅速に本番環境に修正をデプロイするのに役立ちます。

DevOps はツール一式を購入するだけで実装できるわけではありませんが、オープンで協調的な性質を持つツールによって DevOps の原則を育むことができます。最先端の DevOps ツールは、高機能な DevOps の文化とともに、組織に計り知れない競争上の優位性をもたらします。

結論：価値を提供するためのフレームワークとしての DevOps

10 人の人に DevOps とは何かと尋ねれば、異なる答えが少なくとも 5 つは返ってくるでしょう。

DevOps の実践的な実装である継続的インテグレーション / 継続的デプロイやテストの自動化などを根拠に、DevOps はプロセスだと言う人もいるかも知れません。DevOps は、首尾一貫した原理に基づき連携する一連のプロセスを伴う手法だと言う人もいるかも知れません。しかし、どちらの定義も大事なポイントを見落としています。DevOps は、導入する企業に応じて調整できる一連のプラクティスで構成されているという点です。

GitHub では、DevOps は、ソフトウェアを通じてどのように価値を提供するかを検討するためのフレームワークとして理解するのがよいと考えています。DevOps は、単なる 1 つの手法でもプロセスの寄せ集めでもありません。DevOps は基本的にプラクティスのセットであり、文化的なプラクティスと技術的なプラクティスの両方が含まれています。

多くの場合、ツール導入は DevOps の最も目立つ側面ですが、DevOps は単一のツールではありません。DevOps 変革は、専門知識と責任についての考え方を変えることを意図した文化的な転換から始まります。情報の流れが DevOps パイプラインの通貨です。重要な考慮事項は、その情報がステージ間を自由に流れることができるかであり、それはテクノロジと同様に文化やプロセスによっても促進されます。

高機能な DevOps 文化の基盤が整うと、ツールが組織における DevOps プラクティスの全体的な成功に絶対的な影響を与える可能性があります。継続的な学習フィードバック ループを伴う最高の DevOps 文化には、DevOps の 1 つのステージから次のステージへのまとまりのあるフローを可能にするツールが必要です。人、プロセス、文化的実践、テクノロジのすべてが一体となって機能することが、DevOps プラクティスの成功の指標となります。それぞれの柱が一体となって機能してこそ、DevOps のメリットである納期の短縮、品質の向上、拡張性の高い製品を実現することができます。

リソース

- DevOps モデルとは ?DevOps の基礎となるプラクティスを探求する
- DevOps の基本 : DevOps の原則を定義する
- DevOps は「手法」か ?
- DevOps パイプラインとは完全ガイド
- DevOps における継続的インテグレーションの基礎
- DevOps における継続的デプロイの基礎
- コンテナ化とは
- DevSecOps の説明
- DevOps ツールと DevOps 自動化ツールチェインへのガイド
- DevOps 監視ツール : DevOps 監視プロセスを自動化する

GitHub で DevOps プラクティスを構築する

GitHub は統合型のプラットフォームで、アイディアから計画、運用まで、焦点を絞ったデベロッパーエクスペリエンスと、強力なフルマネージド型の開発、自動化、テスト インフラを組み合わせて企業を支援します。

GitHub の包括的な一連のスイートは、DevOps パイプライン全体を単一のツールセットにまとめます。計画を支援するために、GitHub Issues と GitHub Projects は開発者ファーストの革新的な作業管理アプローチを提供します。アイディアが計画されると、開発者は GitHub Codespaces を使って、チーム メンバーと同じ分離された開発コンテナでコード作業を開始できます。機能をレビューする準備ができたら、GitHub 内でプルリクエストを行うことで、開発者は協力してリアルタイムのフィードバックを受け取ることができます。GitHub Actions は、CI、CD、その間のあらゆるものの自動化に使用される自動化プラットフォームです。GitHub Packages は、ソフトウェア パッケージの保存、管理、配布に使用されます。開発者のフローを中断することなくコードの安全性とソース管理のシークレットを保持するには、GitHub Advanced Security ツールセットを活用します。GitHub とその機能を使用することで、DevOps パイプラインの全てのステージを強化できます。

世界最大かつ最先端の開発プラットフォームとして、GitHub は何百万人もの開発者や企業の迅速なコラボレーション、ビルド、リリースを支援しています。何千もの DevOps 統合を活用して、初日から既知のツールでビルドしたり、新しいツールを発見したりできます。今すぐ完全な開発者プラットフォームを入手し、GitHub でソフトウェアを開発している 8,300 万人以上の開発者と 400 万を超える組織に参加しましょう。

GitHub Enterprise に関する質問については

私たちに
お任せください。

GitHub Enterprise ページをご覧になるか、
セールスチームにお問い合わせください。