継続的デリバリーによる高速化

Mark Mansour

継続的デリバリーによる高速化

継続的な改善およびソフトウェアの自動化

Amazon では、10 年以上前にチームがアイデアをいかに早く高品質のプロダクションシステムとして実現できるのかを知るために、あるプロジェクトを実施しました。これにより、ソフトウェアのスループットを測定し、実行速度を向上させることができました。コードチェックインから実稼働までに、平均で 16 日間かかっていることがわかりました。Amazon では、チームはアイデアから始めて、そのアイデアを実現するコードを書くのに、通常 1 日半かかりました。新しいコードの構築デプロイには、1 時間もかかりませんでした。残りの約 14 日間は、チームメンバーがビルドを開始し、デプロイを行い、テストを実施するのに費やしました。プロジェクトの最後に、チェックイン後のプロセスを自動化し、実行スピードを改善できないか提案しました。目標は、品質を維持あるいはさらに改善しながら、遅延をなくすことでした。

この助言の意図は、継続的にプログラムの改善を行い、実行速度を向上させることです。最高水準を追及するというリーダーシッププリンシプルに基づいて、実行速度の改善を決定しました。このプリンシプルは、絶えず高い基準を掲げ、継続的に水準を引き上げ、高品質の製品やサービス、プロセスを提供することです。リーダーシッププリンシプルは、Amazon がどのようにビジネスを行うか、リーダーがどのように皆を率いるか、そして私たちがどのように意思決定においてお客様を中心に置くかを説明しています。

Amazon は、ソフトウェアエンジニアの生産性を高めるためのソフトウェア開発者用ツールを既に構築しています。デプロイ可能なアーティファクトを生成するためにサーバー上で一連のコマンドを実行する、独自の集中型およびホスト型ビルドシステム Brazil を開発しました。この時点では、Brazil にソースコードの変更はありません。ビルドを開始しなければなりませんでした。独自のデプロイシステム Apollo もありましたが、デプロイを開始する際にビルドアーティファクトをアップロードする必要がありました。継続的デリバリーに対する業界の関心に触発され、Brazil と Apollo 間のソフトウェア配信プロセスを自動化する独自のシステムパイプラインを構築しました。

パイプライン: 継続的デプロイツール

少数のチームにおけるソフトウェア配信プロセスを自動化するパイロットプログラムを開始しました。それを終えるまでに、メインのパイロットチームは、チェックインから実稼働までにかかる全体の時間を 90% 削減することに成功しました。

このプロジェクトは、チームが顧客にソフトウェアをリリースするために必要なすべてのステップを明確にする方法としてのパイプラインの構想の正当性を確認しました。パイプラインの最初のステップは、アーティファクトを構築することです。それから、パイプラインは、すべてのお客様にアーティファクトがリリースされるまで、一連のステップを通じてそのビルドアーティファクトを実行します。コード変更による欠陥のリスクを軽減するために、パイプラインを使用します。パイプラインステップで、ビルドアーティファクトに不具合が含まれていないという信頼性を高めるはずです。欠陥が本番環境に悪影響を与えてしまった場合、できるだけ早く本番を健全な状態に戻したいと考えています。

パイプラインの立ち上げ時、アプリケーションごとに単一のリリースプロセスのみをモデル化することができました。この制限のおかげで、チームのリリースプロセスを、一貫性、標準化、簡素化といった点で改善できました。結果として、欠陥がほとんどなくなりました。パイプラインを使用するようになる前は、チームはバグ修正と主要な機能リリースに対して異なるリリースプロセスを取ることが一般的でした。試験的に自動配信を行ったチームの成功を見た他のチームは、一貫性を向上できるように、手動管理のリリースプロセスをパイプラインに移行し始めました。さまざまなリリースプロセスを使用していたチームに、ようやく、共通で使う標準化された 1 つのプロセスができたのです。さらに、リリースプロセスをツールに移行したとき、チームメンバーはしばしば自分たちのアプローチを再検討し、プロセスを簡素化する方法を見つけました。

パイプラインチームは、「魅力的な採用」を用いて使用量を増やすことを年間目標にしていました。 言い換えれば、彼らは皆が利用したがる良い製品を作る必要がありました。パイプラインを使用してソフトウェアを本番稼働環境にデプロイするチームの数を調べ、自動化のレベルによってパイプラインを分類しました。ソフトウェアをリリースし、完全に自動化されたリリースに移行するためにパイプラインを使うという目標を達成するチームがありました。しか

しながら、一部の組織では、品質を測定する方法によってチームがテストを一切行わずにリリースプロセスを自動化できてしまうことに気付きました。

「どれだけテストをすれば十分ですか？」という質問への答えは、個人的意見しかありません。チームは、彼らが活動している状況を理解する必要があります。この状況に対処するには、別のリーダーシッププリンシプルである、オーナーシップを使用しました。このプリンシプルは長期的な考え方です。短期的な結果のために、長期的な価値を犠牲にするものではありません。Amazon のソフトウェアチームは、テストに対して高い水準を持っており、それに多大な労力を費やします。製品を所有するということは、その製品の不具合の影響 (の責任) も所有することになるからです。問題がお客様に影響を与えかねない場合、その問題にリアルタイムで対処し修正するのは、小規模でシングルスレッドのなソフトウェアチームのメンバーです。実行速度の向上と本番稼働環境の問題への対応との間にある緊張や不安は、チームが適切にテストする動機になります。ただし、テストに過度に力をつぎ込むと、他社に遅れをとり成功できない可能性もあります。私たちは常に、ビジネスの邪魔者になることなくソフトウェアリリースプロセスを改善しようとしています。

もう 1 つの問題として、チームがお互いからソフトウェアリリースのベストプラクティスを学んでいないことが挙げられます。シングルスレッドのチームは、自律的に作業することが奨励されています。つまり、エンジニアはデプロイの問題を単独で解決していました。ソフトウェアリリースのニーズに対応するためのソリューションを見つけたときは、メーリングリストや運用会議、その他のコミュニケーションチャネルを通じて、他のエンジニアにテクニックを勧めようとします。このコミュニケーションのスタイルには、2 つの問題がありました。第一に、これらのチャネルはどれもベストエフォート型のコミュニケーションチャネルであり、すべての人が新しいテクニックについて学んだわけではありません。第二に、チームが新しいベストプラクティスを採用することを奨励したリーダーは、チームが実際にベストプラクティスを採用するために必要な作業を行ったかどうかがわかりませんでした。私たちが学んだベストプラクティスへのアクセスをすべてのエンジニアが持つための手助けの必要と、注意を必要とするパイプラインを特定する力をリーダーに与える必要があることに気付きました。

解決策として、ソフトウェアの構築とリリースに使用するツールに、ベストプラクティスのチェックを追加することによって学習を機械化しました。ある組織のベストプラクティスが別の組織のベストプラクティスではない可能性もあるため、これらのチェックを組織ごとに構成できるようにしました。組織レベルのベストプラクティスチェックにより、リーダーはビジネスのニーズに合わせてリリースプロセスを調整することができるようになりました。新しいベストプラクティスを奨励または実施したいと思っていたリーダーは、エンジニアが日常的に使用しているツール内で通知を送ることから始められました。ツールにメッセージを配置することで、チームメンバーがベストプラクティスについて学習し、そのベストプラクティスの効果が出る時期をほぼ保障しました。新しいベストプラクティスについて学び議論する時間をチームに与えることで、組織はベストプラクティスのチェックを繰り返して改善する機会を得ました。最終的な結果として、ベストプラクティスの品質が向上し、エンジニアリングコミュニティからもより多くの賛同が得られました。

適用すべきベストプラクティスを系統的に特定しました。最も経歴の長いエンジニアのグループが、リリースが機能しないよくある理由を一覧化しました。彼らは、リリースを機能させるためのステップを特定しました。次に、ベストプラクティスチェックのセットを作成するために、そのリストを使用しました。このプロセスを通じて、新しいソフトウェアの改訂版をすぐに、手間をかけずに、可用性を低下させることなく顧客に届けたいと考えていました。しかし、まずは可用性を一番に、次いで速度の向上、それからエンジニアにとっての使いやすさという優先順位をつけました。

欠陥によりお客様に影響するリスクの削減

すべてのエンジニアは、ある時点で、システムの 1 つに不具合を埋め込むことができると考えられます。パイプラインやデプロイシステムなどのリリースプロセスは、できるだけ早くそれらの不具合を特定し、お客様に影響を与えないように予防する必要があります。リリースプロセスが正しく構成されていることや、ビルドアーティファクトが意図したとおりに機能していることを、確認する必要があります。

デプロイメントハイジーン: 新しくデプロイされたアーティファクトが開始され作業に応答できることが、デプロイテストの最も基本的な形式によって保証されています。デプロイ後のワークフローの一環として、新しくデプロイされたアーティファクトが開始されトラフィックを処理していることを確認するためのクイックチェックを実行します。たとえば、AWS CodeDeploy AppSpec ファイルでライフサイクルイベントフックを使用して、デプロイを停止、開始、有効化するための簡単なスクリプトを動かします。お客様のトラフィックを処理するのに十分な容量があることも確認します。CodeDeploy に最低限の健全性を持つホストなどの技術を構築し、お客様にサービスを提供するのに常に十分な能力があることを確認しました。最後に、デプロイエンジンが障害を検出できる場合、変更をロールバックして、不具合がお客様の目に触れる時間を最小限に抑えなければなりません。

実稼働前のテスト: Amazon のベストプラクティスの 1 つは、単体テストや、結合テスト、実稼働前のテストを自動化し、これらのテストをパイプラインに追加することです。当社はこれらのテストをパイプラインに追加しながら、負荷テストやセキュリティテストを実行していることを断言します。単体テストとは、スタイルチェック、コードカバレッジ、コードの複雑性など、ビルドマシンで実行するすべてのテストを意味します。結合テストには、障害挿入テストや自動ブラウザテストなどの、すべてのオフボックステストが含まれると考えています。単体テストと結合テストに関しては、優れた記事が多くありますので、ここではこれ以上詳しく説明しません。

単体テストおよび結合テストの目的は、ビルドアーティファクトの動作が機能的に正しいことを確認することです。検証を行っただけ、不具合がお客様の目に触れるリスクが低くなります。お客様に製品をお届けするまでの時間を短縮するために、リリースプロセスのできるだけ早い段階で、不具合を検出するよう努めています。一般的には、テストが小規模で速く行える場合、変更による問題について、より迅速なフィードバックを受け取れるということです。

Amazon では、製品化前テストと呼ぶ手法も採用しています。製品化前環境は、製品への変更点をデプロイする直前にテストを行う最後のチャンスです。製品化前環境でのテストは、実稼働システムと同じ構成を使うので、製品版とまったく同じ動作をさせられます。このアプローチには 2 つのメリットがあります。その 1 つめとして、製品化前環境では製品の構成をテストするので、そのサービスが実稼働状態のデータストアを含むすべての製品リソースと適切に接

続することを確認できることが挙げられます。2 つめは、そのシステムが、動作に必要な実稼働サービスの API と正確にやり取りしているかを確認できることです。製品化前環境はそのサービスを所有するチームによってのみ利用されるもので、ユーザーからのトラフィックを受け取ることはありません。製品化前テストを行うと、同じコードと構成が実稼働においても機能するということの自信が持てるようになります。

実稼働での検証: お客様に対しコードをリリースする際も、一度にすべてを完了することはありません。製品を全ユーザーに対し一度にリリースすると、欠陥があった場合の影響範囲が大きくなりすぎるからです。そうする代りに、デプロイは、サービスの中で完全に独立したインスタンス、つまりセルごとに行います。最初のセルの中でユーザーの最初のグループに対し変更内容をデプロイするときは、最大限の注意を払います。新しい変更点に触れるのは少数のユーザーだけにして、この新しいコードが機能するかについてのフィードバックを集めます。Canary デプロイ後も、このサービスに発生するエラーの数をモニタリングし続けます。仮にエラー発生率が上昇するような場合は、この変更は自動的にロールバックされます。たとえば、デプロイを続行する前には、ネガティブなデータポイントが含まれない、3,000 のポジティブなデータポイントが確認できるまで待機するのです。

しかし、この自動テストがユースケースを逃すようなことがあると、複雑な事態になりえます。当社では、それが自動であれ手動であれテストを構造化し繰り返し行うことで、すべてのエラーを発見できるように努力しています。とはいえ、仮に最大限の努力をしていても、欠陥というものはそれをすり抜けることがあるのです。こういったテストのテストをする意味で、新しい変更内容を決まった期間だけ製品の中に放置し、チーム以外のメンバーが何かの問題を発見するか確認しています。当社では、変更は単純に製品内に置くようにすべきか、あるいは、Canary デプロイの後で残りのデプロイグループをリリースするまでどの位の時間待機すべきかについて、長い間にわたり議論を続けてきました。担当チームの多くは、ポジティブなデータポイントを収集する時間の後、デプロイサイクルを進める前に固定的な待機時間を追加する方法を取っています。パイプラインが待機する時間は、チームに大きく依存します。数時間待機するようなチームもあれば、それを数分で終わらせるチームもあります。問題の影響度が高く、その修正に時間を要すればするほど、リリースプロセスは遅くなっていきます。

最初のセルについての自信が得られたら、その変更された新しいコードの提供をより多くのユーザーに拡大し、リリースを完了します。Canary デプロイに対し行ったのと同様に、次のセルに取りかかる前に、最初のセルのデプロイで自信を得るまで時間が必要です。ビルドアーティファクトに対し、より強い自信が得られることで、コード変更の確認に費やす時間を削減できます。このことは、チェックインの段階から、最初の実稼働ユーザーにより可能な限り素早く受け入れられることをめざすパターンを生み出します。しかし、実稼働に入った後で新しいコードをゆっくりリリースしていても、残りのデプロイを徐々にスピードアップしており、さらに自信を強めるための努力も行っています。

実稼働のシステムがユーザーからのリクエストに継続的に応えていることを確認するためには、システムへの人工的なトラフィックを生成します。仮に、サービスが適切に動作していない場合は、最初にそれについてのフィードバックが必要ですが、当社ではそれを基に、最低で 1 分ごとの人工的トラフィックによるテストを行うのです。人工的トラフィックによるテストは、実行中のプロセスが正常であり、すべての依存関係がテスト済みであることが確認できるように設計されます。しばしばこういったテストは、公開されたすべての API のテストが含まれます。

リリース済みソフトウェアのコントロール:ソフトウェアリリースにおける安全性をコントロールするために、当社では、パイプライン上の動きを変化させるためにスピードをコントロールするメカニズムを構築しています。メトリクスやタイムウィンドウ、そして安全チェックなどにより、ソフトウェアのリリース時期をコントロールするのです。

メトリクスでの変化に基づきアラームが発令された際にデプロイを停止するように、パイプラインを設定することができます。メトリクスは広範囲で使用されていますし、アラームでは、システムやセル、アベイラビリティゾーンやリージョン、そして考え得るほぼすべてのものの正常性を監視できます。パイプラインは、重要なメトリクスがアラームをトリガーしたときデプロイを停止するように定義しています。しかし時折、担当のチームが確定版のデプロイをする必要があるとき、システムのアラームが発生することがあります。こういったシナリオに対処するため、変更内容がパイプライン内を移動するのを防ぐために、チームはアラームを上書きできるようになっています。

パイプラインでは、そこで変更内容が承認され、パイプラインを介して処理を進めるための時間ウィンドウを指定できます。各チームは、変更内容がユーザーに届くまでの時間を制御するために、独自の時間ウィンドウを使用できるのです。AWS チームでは、デプロイが原因で発生した問題に素早く対応できるエンジニアが多くいるタイミングで、ソフトウェアのリリースをしたいと希望しています。これを現実にするため、基本的に各チームでは、ビジネスの時間帯のみにデプロイするような時間ウィンドウ設定をしています。同時に Amazon には、ユーザーからのトラフィックが少ない時間でのソフトウェアリリースを希望するチームもあります。これらの時間ウィンドウは、必要に応じて上書きが可能になっています。

また、ビルドアーティファクトのコンテンツに基づき、パイプラインを停止する機能もあります。たとえば、問題がある既知のパッケージ、もしくは特定の Git リファレンスを含むビルドアーティファクトを、ブロックすることができます。この機能は、パッケージへの変更がパフォーマンスの低下原因につながると分かったときに利用されてきました。こういったパッケージをパッケージレポジトリーから単純に削除するだけでは、欠陥があるそのパッケージを既に取り込んでいるパイプラインは、ユーザーにとって外のあるデプロイをそのまま行ってしまいます。

実行速度へのアプローチ手法

各チームが自動化の恩恵に預かることを強く希望しているのは分かっています。当社には、お客様の生活を向上させる機能の構築とリリースに関する強い意欲がありますし、それは継続的な供給により持続していくと考えています。自動化が、エンジニア達のフラストレーションを無くし、間違いを起こしにくくし、手作業での重労働を減らすことで、彼らに時間のゆとりを与える様子も目にしてきました。そして、継続的なデプロイが、品質にポジティブな影響を与えることも示してきました。自動化により、各チームは頻繁なリリースを行え、変更点は 1 度に 1 つとすることで、性能低下の発見を容易にします。

システムが新しい時期では、通常、表面的な部分におけるテストはほとんどのチームメンバーにも理解しやすく、一部の手動テストの追跡も可能です。しかし、システムがより複雑さを増し、チームのメンバーにも入れ替わりがあったりすると、自動化の価値が高まってきます。当社では、システムの自動化を進めています。それにより、変更内容のユーザー提供を手動で処理することより、顧客価値を増加させることの方に注力できています。

Amazon では、ソフトウェアリリースとその安全性をお客様に届けるスピードを向上させるプログラムを、何年もの間、継続的に進めてきました。そのスタート段階では、この記事に書いたすべてのリスクチェックやテストがあった訳ではありません。リスクの特定と緩和手法は、これまで時間をかけて積み上げてきたものです。

この継続的な改善プログラムは、組織内での異なるレベルにいるビジネスリーダー達により実行されています。これにより、各ビジネスリーダーは、ビジネス上のリスクや影響に合わせて、ソフトウェアリリースの過程を調整できるのです。Amazon での継続的改善プログラムの一部は社内の大きなセクションで実施されますが、より小規模な組織のリーダー達が、独自のプログラムを実施することもあります。そして、ルールには常に例外があるということも認識しています。当社のシステムはオプトアウトメカニズムを採用していますから、恒久的あるいは一時的な除外を希望するチームの動きを邪魔することはありません。ソフトウェアの振る舞いは、最終的に担当チームが掌握しているものであり、彼らはソフトウェアリリースのプロセスに適切な投資を行うための責任も負っています。

まず、そのために問題となる部分はどこか調査し、その対策を繰り返し講じるのです。この仕事を持続していくためには、作業料を徐々に増やしつつ、改善点を実感しつづけることが必要です。Amazon でパイプラインを使い始めた時点では、この継続的なデプロイが良好に働くかについて、多くのチームは自信を持っていませんでした。当社では、そんな各チームを動かすために、その時点で彼らの下にあったリリースプロセスや手動ステップなどすべてを、パイプライン内にエンコードするよう後押ししました。そして多くのチームでは、そのパイプラインを、リリースプロセスを通じてのビルドアーティファクトの自動昇格を伴わない、リリースプロセスに関する 1 つの仮想的インターフェースとして機能させていました。そして、彼らの中の自信が育ってくると、パイプラインの異なるステージ上で自動化を起動しはじめ、パイプラインでのあらゆるステップにおいて手動トリガーの必要性をなくしていったのです。

現代に話を飛ばします。今、Amazon では、新しいコードを記述した各チームが、完全な自動化を好んで使用するような段階に来ています。当社にとって自動化とは、ビジネスを持続的に成長していける、唯一の道なのです。