本番環境でテストする

ための合理的ガイド

Rob Zuber 著

本番環境でテストを行うことは、以前なら笑い話でしたが、ソフトウェア配信におけるリスク管理が複雑になるにつれ、バリデーション作業の重要な手段として認識されるようになりました。そのため、本番環境のテストを適切に行えるよう、その意義と想定されるコストについて把握しておくことは、顧客に良質なサービスを提供するという観点だけでなく、ビジネス運営の観点からも、企業の果たすべき義務と言えます。

本書では CircleCI CTO の Rob Zuber が、今日のソフトウェア開発における徹底したリスク アセスメントとはどのようなものかを解説しながら、あらゆる規模や業界の企業がテスト手法について慎重かつ合理的な選択を行えるよう、役に立つフレームワークやツールを紹介します。

変化とは、サプライチェーンにおける大きな課題であり、今やすべての企業にとって避けては通れないものとなっています。

過去 10 年の間に、IT 業界は増え続ける変化の要因に対処する手立てとして、次々に新しいテスト手法を採用してきました。本番前の環境でも、それ以降の環境でも、ソフトウェアの正確性を評価するための選択肢がかなり広がったため、それぞれのテスト手法がコストとリスクの観点からどのような特徴を有しているのかをきちんと理解する必要があります。この種の評価は、本番環境でのテストについて検討する際には特に重要です。というのも、ソフトウェアの複雑化と運用ツールの大幅な改善とが相まって、本番環境でのテストがビジネス上の正しい判断とされるケースも出てきているからです。

本番環境でのテストは、従来のテストとは異なるやり方で行われることが多いため、表現としては紛らわしいかもしれません。本番環境でのバリデーションは、実際にはバリデーション作業の延長線上にあるものとして捉えるべきです。サードパーティ製サービスの利用が拡大し、データセットがますます巨大化するにつれて、本番環境への配信前にコードを完全に検証することが難しくなっているからこそ、本番環境でのバリデーションと他のテスト手法と組み合わせることでリスクは軽減します。このような状況では、「継続的バリデーション」が効果を発揮します。すべての変更や開発者が前提としている依存関係が正常に機能しているかどうか、足元で生じた変化が見過ごされていないかどうかを継続的に検証することが重要です。本番環境でのテストをバリデーション戦略の代替要素ではなく、1 つの追加要素とする考え方は、チームの自信向上や、市場投入の迅速化に役立ちます。もちろん、本番環境でのテストにはコストとリスクが伴います。そこで、この手法の有効性が最大となる条件と、安全に実施する方法を探っていきたいと思います。

コストとリスク

今日、ソフトウェア エンジニアとして活躍している人の多くは、コンピューター サイエンスのスキルを身に着けています。学習したことをエンジニアリングの現場で実践することは、獲得した理論的知識を駆使し、制約のある現実世界でいかに応用するかというノウハウを磨くことにほかなりません。理論を越えた実世界のプロジェクトでは考慮すべきことがいくつもありますが、その 1 つが、プロジェクトがビジネスにもたらすコストとリスクです。

> 「プランニングに注力すればするほど、
> リスクは減らずにむしろ増える一方である
> ことに、私たちは時が経ち気づいたのです」

コストの比較

エンジニアリングの世界では、コストの比較により、時間とリソースに対して最大のリターンを得られる方法を探るだけでなく、ある一定のリターンを得るためのコストに調整の余地があるかどうかの検討も行われます。これを土木工学の例で考えてみましょう。高層ビルの柱に使われているコンクリートは、歩道に使われているコンクリートの約 5 倍の強度がありますが、それには理由があります。高層ビルの柱にはより大きな荷重がかかるだけでなく、柱が折れた場合の影響が歩道にひびが入った場合よりも桁違いに大きいからです。

しかし、より確実な結果を求めるほど、必要なコストも高くなるものです。

この例では材料のコストについてしか触れていないように見えますが、さらに掘り下げると、より高い強度のコンクリートを製造するためには、原材料の品質に関する要件が厳しくなり、廃棄物が増え、基準が満たされていることを確認するためにより高度なテストが必要になるなど、追加の作業が発生してコストが上乗せされます。

建設資材の欠陥を減らそうとすると、コストは増加します。そして、ソフトウェアについてもこれと同じことが言えるのです。

ソフトウェア配信のリスク要因とは

ここでは、エンド ユーザーの期待どおりに動作しないソフトウェアを配信することのリスクに限定して議論します。この種のリスクは、主にソフトウェア自体に変更を加えることで増大します。

ソフトウェア エンジニアリングの黎明期には、リスクは十分な努力をすれば排除できるという、根本的に破たんした捉え方が蔓延していました。あるいは、リスクを完全に排除できない場合は、事前分析によって最小限に抑えようと努力されてきました。事前分析の費用は増加する一方でしたが、それは先行投資であり、リスク軽減という利益が後から返ってくるはずだと期待されていました。

しかし、プランニングに注力すればするほど、リスクは減らずにむしろ増える一方であることに、私たちは時が経ち気づいたのです。それは、準備や計画が大規模になるにつれ

て、SaaS としてデプロイするにしろ、App Store に登録するにしろ、他のいかなる方法にしろ、ユーザーへの配信が最大のヤマ場となってしまったからです。配信のたびに、前回よりも多くのコードとリード タイムが必要になりました。結果、リリースの際に問題が表面化しうる範囲が広くなり、リスクもコストも増えてしまいました。

どのような変更もリスクの原因となるため、従来の考え方では、変更を少なくして、入念に検証された設計、多くの承認、大規模な QA サイクルによって、それぞれの変更を十分に検証することが重視されていました。しかし、ひとたびバッチ サイズが大きくなってしまうと、結果的にリスクは増大してしまいます。また、従来のやり方では配信が遅くなり、コンテキストが失われて問題解決に時間がかかるようになるというデメリットもありました。

リスクの予測、抑制、容認

業界では最終的に、変更をできるだけ小さく分割して少しずつ加えていくことで、1 つの変更に伴うリスクを軽減できるのではないかという発想に至りました。

コードベースに対して小規模な変更を頻繁に加えていくモデルが採用されるようになり、障害や想定外の事象にすばやく対応することが可能になりました。近年耳にする TDD、アジャイル、そして継続的デプロイメントといった手法は、不確実性に向き合い対処するうえでの助けとなっています。こうした手法を活用することで、本番環境に予期せぬバグが流出し、エンド ユーザーに影響を及ぼしてしまうリスクを大幅に軽減できました。

変更を小さな増分に分割すると、1 日に複数の変更を本番環境のコードベースにデプロイすることになるケースがよくあります。そこで、CI/CD などのツールを、包括的なテストと組み合わせることで、テストとバリデーションが行われるまでは本番環境には何もデプロイされないことを保証でき、自信を持って迅速に行動できます。本番環境に到達する前にコードを徹底的にテストすることによって、計画サイクルの負担軽減、リアルタイムのユーザー フィードバックによるフィードバック ループの短縮といったメリットを享受できるようになります。いずれのメリットも、コードの信頼性を高め、リスクを低減させました。業界にとって喜ばしいことでした。

しかし、テクノロジーは進化し続けているため、私たちがリスクを最小限に抑えることに注力している間に、状況は根底から変わり、これまでには存在しなかった新たなリスクが出現しました (これらの新しい展開については、次の章で詳しく説明します)。ここ数年の間に、ソフトウェア開発の複雑さが増してきたことで、本番前の包括的なテストが複雑化し、場合によってはきわめて困難であったり、多大なコストがかかったりするようになりました。その結果として「本番環境でのテスト」が注目されるようになったのです。かつては、悲惨な結果をもたらす怠慢な行いをイメージさせるような言葉でしたが、近年では、最新のソフトウェア配信のための手法として意図的に採用されるように変わってきました。そして、場合によっては、これが最も現実的な選択となります。

このように、本番環境でのテストが広く浸透する素地は十分整っていますが、これに勝るとも劣らず重要なのが、特定のビジネスやリスクについて分析したうえで、テストに関する意思決定によって生じるリスクを合理的に評価することです。そうすることによって、テストの適切な段取りを確定できます。もちろん、状況はますます複雑になっています。それでも、コンクリートの品質について検討するときと同じように、ソフトウェアのデプロイメントについて検討するときに

> 「しかし、テクノロジーは進化し続けているため、
> リスクを最小限に抑えることに注力していた間に、
> 状況は根底から変わり、これまでには
> 存在しなかった新たなリスクが出現しました」

も、実際のコストとリスクを考慮しなければなりません。私たちは今後、常に何かしらを本番で検証する世界に生きていくことになりますが、成功するか否かは、これまで以上に、リスク、変更のコスト、それらの選択肢の潜在的なメリットなどを評価するための優れた枠組みがあるかどうかにかかっています。

本番前テストのコスト (とリスク) の上昇

変化するソフトウェア開発の世界

IT 業界だけでなく、小売、医療、金融といった伝統的な業界も含め、さまざまな企業にとってソフトウェアが競争上の差別化要因となるにつれ、ソフトウェア開発チームは、より迅速な配信を効果的に行うための最適化に取り組んでいます。それに伴い、いくつかの傾向が見られるようになりました。

- サードパーティ製サービスやツールの利用拡大
- マイクロサービス アーキテクチャの普及
- データセットの大規模化

これらは、ソフトウェア開発の洗練化と最適化に役立ってはいるものの、一方で、本番前の環境で完全な検証が行えているという確信が揺らぐ要因にもなっているため、これらについてここで整理しておきたいと思います。別の言い方をすれば、こうした傾向のいずれか (またはすべて) のせいで、配信前にこれまでどおりコードの信頼性を確保するためのコストが上昇しているということです。ではなぜ、コードの徹底的なテストが困難になっているのでしょうか。これらの要因を 1 つずつ見ていきましょう。

サードパーティ製サービスやツールの利用拡大

多くのサードパーティ製サービスが導入されるようになったことで、コードベースのすべての変更を確実に把握するのが (または認識することさえ) 難しくなっています。大規模なフレームワーク上で開発を行っている場合、独自に構築したものは実はコードベースのごく一部で、残りは共有サービスやライブラリであるということも珍しくありません。そうした便利なリソースは開発の面では効率的ですが、チームが作成したコードよりも挙動を予測するのが難しく、サードパーティ製サービス (支払いプラットフォームや分析プラットフォームなど) の場合、こちら側が変更を行わなくても動作が変わる可能性があります。こうした理由から、ユーザーに影響を与える可能性のある変更をすべて確実に把握することが非常に困難になっています。

マイクロサービス アーキテクチャの普及

マイクロサービスの登場により、単一のモノリシックなものだったソフトウェアから複雑な部分を取り除けるようになりました。ソフトウェアを細かく分割することは、開発者にとって有益ではありますが、複雑さが消えてなくなったわけではありません。ソフトウェアのパーツ間のやり取りの中に紛れ込んだだけです。

適切に構築すれば、マイクロサービスは実質的に独立したプロセスとして動作します。開発者は、使用するサービスの依存関係を把握していなくても、自分が開発しているサービスは管理できます。しかし、徹底的にテストするなら、それらの依存関係も考慮しなければなりません。テストの結果や実際のパフォーマンスの信頼性を十分に確保するには、依存関係にあるすべてのサービスを含めた完全な環境を用意して、変更によって他のサービスのいずれにも不具合が発生しないようにする必要があります。しかし、それには莫大なコストがかかり、変更をプッシュするときの独立性や俊敏性が損なわれてしまいます。

データセットの大規模化

業界内では、データ駆動型のアプリケーションがもてはやされるようになりました。アプリケーションの機能を改善するためにできるだけ多くのデータを収集するようになったために、開発者自身も、開発環境においても、これまで以上

に大量のデータを処理することを余儀なくされています。今やストレージは安価になり、コンピューティングも高速になったとはいえ、データセットが大規模になることで管理しなければならないテスト データセットの規模も大きくなることには変わりありません。データの移動、保存、アーカイブ化に伴うコストも無視できません。また、本番前環境での徹底的なテストに関するコストと実行可能性にもかかわってきます。たとえば、CircleCI では本番環境でテラバイトのデータを管理していますが、テスト実行のたびにその量のデータを取り込むのは大仕事です。

大規模なデータセットに関しては、さらに考慮すべきことがあります。それは、CI の実行時には効果的に動作するように見えても、本番環境では異なる結果になることがあるということです。

データの多様度の指標である「カーディナリティ」を例に考えてみましょう。カーディナリティは、通常意図して設定できるものではありません。たとえば、データベースに名前の「姓」の列があるとします。この列の値の多様度は国によって大きく異なるでしょう。カーディナリティが低いと、姓の列にインデックスを作成しても、高速な検索は行えません。テスト データではなく実際のデータで試せれば、姓ではなく名にインデックスを作成するなど、パフォーマンスの観点から別の選択肢も生まれたはずです。本番環境のデータをテスト環境に取り込むことで解決を図る場合もありますが、プライバシーとセキュリティ上の懸念から匿名化が必要となり、結局は実際のデータを検証できなくなってしまいます。あらゆるエッジ ケースを想定し、適切なテスト データセットを作成したうえで動作を確認するのは大変な作業です。本番環境の本物のデータでテストすれば、サービスが実際に機能するかどうかを確認しつつ、ミスの影響範囲を最小限に抑えることができます。

これら 3 つの傾向それぞれが、本番前環境でのソフトウェア検証に対して十分な確信を持つことを難しくしています。そしてこれらが合わさると、もはや考えられる問題を 1 つずつ挙げることは不可能です。このようなリスクに、複雑化するテスト プラットフォームで対処しようとすると、コストが上昇し続け、最終的には持続不可能となってしまいます。本番環境でのテストにもそれなりのリスクがありますが、新たな手段として一連の手法に慎重に取り入れることで、コスト効率良く実世界のデータで不具合やエッジ ケースを特定できるようになります。

転換点

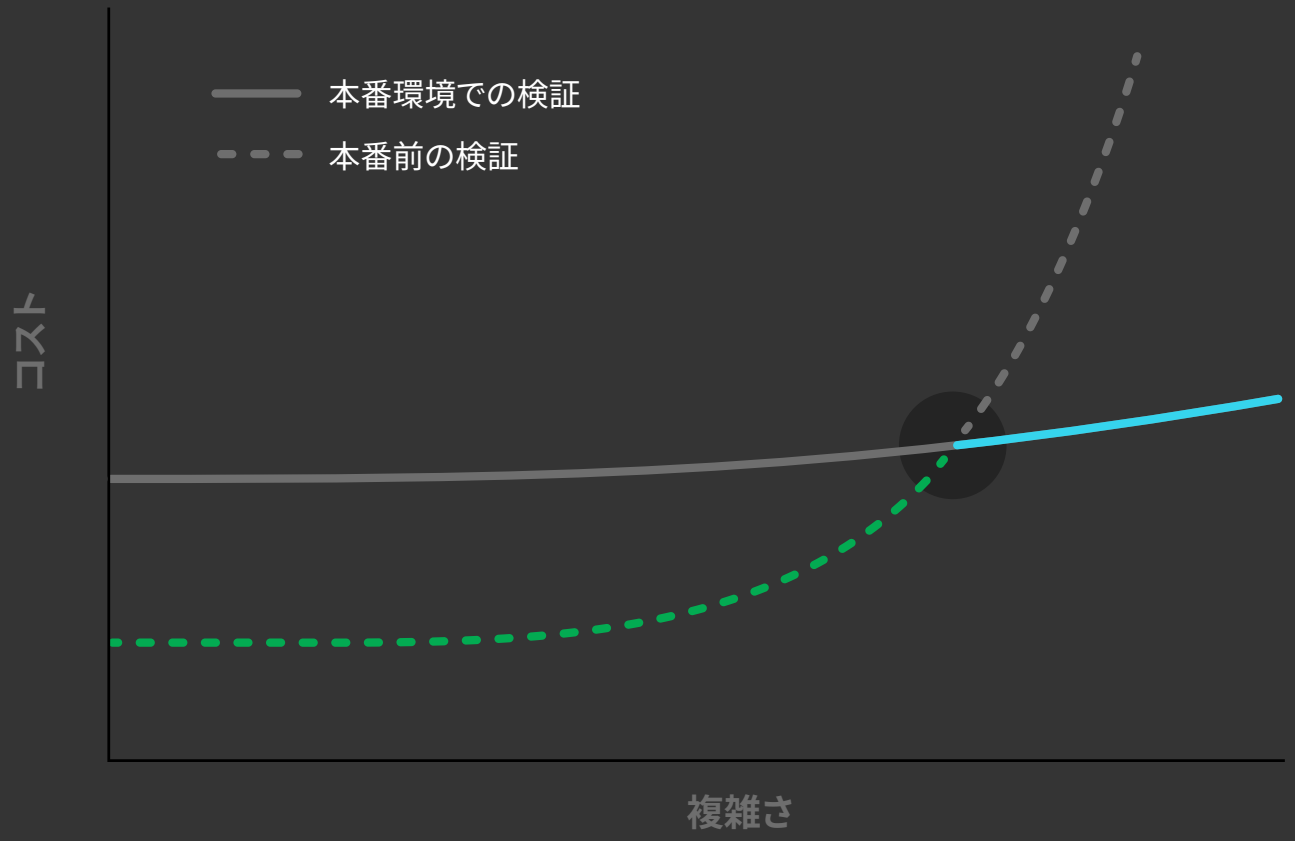
ここまで、本番前の環境でいくつかの問題を把握することがより難しくなり、その結果コストが増大している背景について見てきました。こうしたコストは、より積極的にリスクを取って本番環境で問題を発見し次第できるだけ迅速に修正する方がよい、といえるレベルまで必然的に増えていきます。こうして、本番環境でのテストは時を経て単なるミームから現実的な手法になっていきました。

> 「この転換点、つまり本番前の検証コストが本番環境での検証コストを上回るポイントは、静的なものではなく、常に変動しています」

一部のバグや問題が本番前のテストでの発見が非常に高コスト (あるいは不可能) であり、かつ (意図したかどうかにかかわらず) それらを本番環境でテストすることになると、影響範囲を限定し、本番環境内のバグの修正を急ぐことに注力しようという考えに至るのは自然なことです。

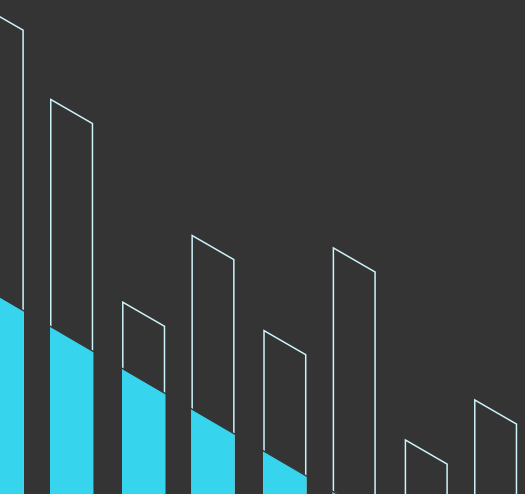
そして、この転換点、つまり本番前の検証コストが本番環境での検証コストを上回るポイントは、静的なものではなく、常に変動しています。本番環境でのテストのコストはある程度は固定されていますが、本番前に徹底的にテストするコストは、開発プロセスの複雑化に伴って増加しているため、どのタイミングが変換点となるかは臨機応変に評価する必要があります。

本番前の徹底した検証のコストと
本番環境での検証のコストの変化



リスク管理の複雑化が進む中、本番環境でのテストは今や重要な手段となりつつあります。
では、この手段をうまく利用するために、その意義とコストについて見ていきましょう。

本番環境でのテストのコスト削減



コストの見積もり

本番環境でテストをするコストは主にユーザーへの影響に結びついており、次の 4 つの軸で見積もることができます。

- 影響を受けるユーザーの数
- 影響を受けるユーザーのカテゴリ
- 影響の重大性
- 影響の持続時間

影響を受けるユーザーの数

これはかなり単純な指標ですが、ユーザー数の生データから障害の影響を推測しようとすると、サービスを利用しているユーザーの総数を把握する必要が生じるため、代わりにユーザーの数を割合で考えるとよいでしょう。影響を受けるユーザーの数を最小化するには、カナリア デプロイメントや機能フラグといった手法 (詳しくは後述) を用いて、特定のユーザーにだけ新コードを提供して、旧コードを利用するユーザーと分けられるような仕組みを用意することが重要です。

影響を受けるユーザーのカテゴリ

システムを利用する可能性のあるユーザーは、無料/試用ユーザー、中小企業/大企業、さらに理想的にはベータ版や早期リリースへのアクセス プログラムに参加しているか否かで分類することが可能です。影響が生じたときに最もコストがかかるカテゴリは、ビジネス モデルに依存し変わってきますが、共通して言えるのはそのカテゴリがどれなのかを把握し、ソフトウェアの変更の配信先を適切にコントロールすることは、とても効果的なリスク軽減策となります。

影響の重大性

ユーザーへの影響の重大性を判断するには、ユーザーがシステム上で使用する最も重要な機能に焦点を当てる必要があります。たとえば CircleCI の場合は、ユーザーがコードをビルドできないことは、ビルドの設定を変更できないことよりもはるかに深刻な問題です。

影響の持続時間

これは、不具合の影響がユーザーに認識できる形で見える期間を指します。

コスト削減を可能にするツール

本番環境でのテストが定着するにつれ、多くのツールが各ベンダーやプロバイダーから提供され、影響範囲の抑制やリスク軽減に活用できるようになっています。そうした選択肢をいくつか紹介し、本番環境で問題を検出する際のコストの削減にどう役立てられるかを探っていきたいと思います。ここでは、モニタリング ツールとデプロイメント/リリース ツールの 2 つのカテゴリに分けて説明します。

モニタリング ツール

モニタリングは、一般的にログ、メトリクス、トレースを組み合わせたツールが属するカテゴリです。それぞれの機能でできることや、環境に合わせた利用方法には共通する部分もありますが、まずはそれぞれ基本的なところから見ていきましょう。

もう 1 つ注意点があります。本番環境でのテストを実施するかどうかにかかわらず、本番環境でのデバッグが容易であることはとても重要です。デバッグ用の基盤が用意されているなら、それは既にモニタリングに大きな投資がされていることと等価です。万全を期せば、本番環境でのテストのためにも活用することができ、投資分はいずれどこかで回収できることになります。したがってデバッグの容易性を向上させることは賢明な投資と言えるでしょう。

ログ

ログはデバッグの手掛かりとなり、影響時間を短縮するのに役立ちます。この手法は古くから活用されてきましたが、現在ではツールによりログを賢く集約することができます。ただし、多くのログは構造化されていません。それらは開発者がコードを記述しているときの懸念事項を表しているにすぎず、障害発生時のデバッグ作業を支援することを意図していません。異常検知を活用すれば、ログの中から重要な情報をもっとすばやく特定できます。ログを書き出すときは、構造化に配慮し、あらかじめ運用を意識しておくとよいでしょう。

メトリクス

モニタリングに関連したメトリクスと言えば、システムから出力される統計情報やカウンターを基にしたダッシュボードが思い浮かぶでしょう。ダッシュボードを利用すると、開発者はわずかな労力でシステムの動作の変化を把握できます。ログに比べればデバッグ作業が意識されている場合が

多いですが、実際の稼働中に問題になっていることではなく、開発プロセスで問題になりそうだと想定されていたことだけに焦点が当たりがちであることに変わりありません。ログと同じく異常検知に使用すると有効ですが、絶対的な確信を得るには、異常検知についてだけでなく、メトリクスでカバーされる範囲についても信頼性を確保する必要があります。また、メトリクスは問題発生を知らせてはくれますが、その因果関係をすぐに見いだせなければ、問題の原因はつかめません。完璧なメトリクスの純粋な維持コストは、規模の拡大に応じて高くなる傾向がありますが、メトリクスは本番環境でのテスト以外にも有意義に活用できるため、投資価値は十分にあります。

トレース

トレースとは、システムを介したリクエストの流れをキャプチャして視覚化するために使用されるツールのことです。トレーシング ツールは、モノリスの場合でも便利ですが (コードベースを通じてリクエストをログに記録できるため)、1 人のユーザーからのリクエストが複数のバックエンド サービスに送信される場合や、同じサービスに対して何度もやり取りする必要がある場合に、真価を発揮します。

トレーシングはログやメトリクスほど一般的ではありませんが、分散システムの構築が広く行われるようになって、基本的な運用に必須のものとなっています。まずは時間をかけてトレーシングを実装するべきでしょう。そうすれば本番環境でテストする場合のさまざまなシナリオに対処できるようになります。トレーシングの実装コストは他のモニタリング ツールと同等であり、他のモニタリング ツールと同様に、デバッグが強化されることで影響の持続時間の短縮につながります。注目に値するのは、どのモニタリング ツールでも、スケーリングなどから生じる一部の影響を回避できるという点です。モニタリング ツールを利用していると、システムの使いやすさが低下する前に、一定の限界に達したことがわかるからです。

デプロイメントとリリース

このカテゴリのツールは、ユーザーが新しいコードにアクセスする方法に関係します。

ブルー/グリーン デプロイメント

ブルー/グリーン デプロイメント (「レッド/ブラック」デプロイメントと呼ばれることも) は、サービスやモノリスのようなシステムの完全なコピーが実行されている状態で、そのシステムの新しい完全なデプロイメントに対してトラフィックを切り替える手法です。こうすることで、開発者はパフォーマンスを監視して、問題が発生したらすぐに切り戻すことができます。ブルー/グリーン デプロイメントは、セットアップが複雑で、システムに大きく依存する傾向があります。サーバーを立てるのが当たり前だったころはコストが高くついていましたが、エラスティック コンピューティングやクラウドの登場で、格段に低コストで済むようになりました。このデプロイメント方式は、切り替えてからの影響をリアルタイムに判断できる場合にのみ有用です (モニタリングのセクションを参照)。それが可能なら、影響の持続時間を大幅に短縮できます。

カナリア デプロイメント

これは、コードベースのすべてのインスタンス (モノリス、サービスなど) を置き換えるのではなく、機能を検証する目的でインスタンスのごく一部分を置き換えるという、ソフトウェア アップデートのデプロイメント方法です。対象の「カナリア」にルーティングするトラフィックの分け方には、特定の種類の動作やユーザー、地域、時間など、無限の選択肢があります。カナリアの動作をモニタリングし、意図した効果が証明され、なおかつ想定外の悪影響がないと証明されたら、残りのインスタンスも置き換えを行います。このデプロイメント手法の全体的なタイムラインは、ビジネスのニーズに応じて自由に決められます。ブルー/グリーン デプロイメントと同様に、置き換えてからの影響をリアルタイムに判断できる場合にのみ有用です (モニタリングのセクションを参照)。カナリアへのトラフィックのルーティングをすばやく簡単に切り替えられるようにすると、有用性がさらに高まります。エラー状態が検知されたときに自動で切り替えられると理想的です。カナリア デプロイメントは、影響を受けるユーザーの数、影響を受けるユーザーのカテゴリ、影響の持続時間を制御するのに役立ちます。

機能フラグ

機能フラグとは、システムの中で問題となるセクションやリスクが高いセクションのコードレベルのラッパーです。たいていのシステムでは、地域、ユーザーの種類、組織の種類などでユーザーを絞り込み、そのユーザー グループに新機能を公開するように設定することができます。機能フラグの主な利点は、コードにバグがあったとき、フラグをオフにするだけでそのコードを本番環境からすぐに削除できることです。機能フラグのコストは、実装時には低くて済みますが、時間の経過と共に大きくなる傾向があります。コード パスの急増、理解不足、半端なテストが絡み合った技術的な負債の主因ともなるでしょう。機能フラグは、モニタリング ツールと併用すると便利ですが、膨大な管理要件を伴う場合も少なくありません。一般的には、影響を受けるユーザーの数、影響を受けるユーザーのカテゴリ、影響の持続時間を制御するのに役立ちます。

復帰とクイック修正

デプロイメントを効果的に自動化できると、その成果として、復帰とクイック修正が可能になります。すばやく確信を持ってデプロイできるということは、修正や既知の正常状態への復帰をプッシュして、発生した問題に対処できるということです。継続的デプロイメントに取り組んでいなくても、適切なツールを導入していれば、障害が発生しても直ちに対応できると確信が持てるため、迷いなくデプロイできるでしょう。

復帰とクイック修正は、影響の持続時間の短縮につながります。昨今、そのためのツールが業界全体で多数利用される中で、数え切れないほどの活用方法が編み出されてきました。ツールをバリデーション サイクルの一部として扱うアプローチと賢く組み合わせることで、危機管理のツールとしてだけでなく、テスト スイートの一要素と見なすことが可能になります。

危機管理のためであってもそうしたツールを現時点で使用しているなら、復帰とクイック修正に向けた投資を既に行っていることになります。その投資を有効に活用するための極意は、これまでテスト環境で検出しようと考えていた問題を、ツールによって本番環境で検出した方が理にかなっていそうなシナリオを描き、それを想定した計画を立てることです。

ツールとリスクについてのまとめ

本番環境でのリスクとユーザーへの影響を制御することに対する各ツールの有用性を次の表にまとめました。注目すべき点は、影響の重大性についてはこれらのツールで対処するのは難しいということです。影響を受けるユーザーが少なく、持続時間が短くても、製品の重要なユース ケースに支障をきたす場合も考えられます。そのため、製品のコア機能にとって絶対不可欠なフローや、一時的に利用できなくても不便なだけで済むフローを、きちんと把握しておくとよいでしょう。そうすれば、さまざまな領域のリスクについて許容可能なレベルを明確に定義したり、本番環境で検出されても許容できるエラーを見極めたりできるようになります。


ユーザーの数
ユーザーの
カテゴリ
影響の
重大性
影響の
持続時間
ログ
メトリクス
トレーシング
モニタリング ツール
ブルー/グリーン デプロイメント
カナリア デプロイメント
機能フラグ
復帰とクイック修正
デプロイメント/
リリース ツール

実際の例を踏まえた考察

多様なツールが利用できることがわかったので、ここからは現実世界の例を見ていきましょう。ROI を評価しながら、本番前ではなく本番の環境でどれだけのエラーケースを検出できるかを探っていきます。ここでは説明のために、大規模なデータセットの例に焦点を当てますが、本番環境でのテストを試みるあらゆる場合に応用できるはずです。

10 TB 程度のデータ サイズで本番環境の OLTP データストアが運用されているのは、特に珍しいことではありません。CircleCI でもこの規模のデータを運用しています。データ量は、CircleCI にとっていくつかの理由から重要な要素です。第一に、9 年の運用期間を経て、データセットには想定外のエントリが存在する可能性があります。そのエントリは、これまでとは違う発想でシステムの機能に変更を加えている新任の開発者には考えもつかないようなものかもしれません。第二に、クエリ実行計画を備えたデータベース エンジンの場合、データ

セットのサイズが大きく異なると、クエリの実行計画が変わる可能性があります。小規模なテスト環境で選択されたインデックスが、より大規模な本番環境のデータセットでは選択されないこともあります。

少なくともテストの一部分を本番環境規模のデータセットで行うと決めたら、主に次の 3 つことについて検討する必要があります。

- テスト環境への本番データの取り込み
- 複数のユーザーによるテスト、または並列テスト システムでのテストにおけるデータストア容量の確保
- 本番データのプライバシーの確保 (匿名化など)

まず、本番データをテストに利用できるようにする最も確実な方法は、個々のテスト実行用にデータをロードするか、テスト実行用のデータストアを管理するかのどちらかです。

動的なテスト環境にデータをロードする場合、ディスクレベルのイメージを転送すると、データベース エンジンを介した書き込みに伴う時間と労力をすべて回避できるため、最もオーバーヘッドが軽減されます。10 TB のデータ サイズなら、完璧なギガビット イーサネットがあったとしても (非現実的な仮定ですが)、データ転送だけで約 22 時間もかかります。これでは CI の実行に適しているとは言えません。

このサイズなら、データベースのマネージド インスタンスを使用する方が現実的でしょう。パフォーマンスに関する目標が設定されている場合はなおさらです。テストを分割して並列に実行したいという要望や、常時の CI 実行のボリュームなど、最初の時点でいくつかのハードルが考えられますが、master へのマージのみをテストする (つまり、本番環境にデプロイする直前にテストする) ことを想定して、まずは最小限のコストから検討していきましょう。考えられる困難なシナリオがカバーされるテストを何らかの形で特定したとして、ワークロードのサブセットのレベルでシリアル化し、単一の共有 DB インスタンスで実行できるようにします。

このテストでは高可用性や回復性の要件は考慮せず、中程度のパフォーマンスが出せればよいと仮定すると、AWS の EC2 (月額約 150 ドル) の基本的な xlarge インスタンスで十分でしょう。データを保持するために gp2 EBS ストレージを追加すると、月額 250 ドル、年額 3,000 ドルになります。なかなか悪くない金額です。ただし、本番データのコピーをメンテナンスする必要があり、それがコア アプリケーションのデプロイ パス上に存在するため、そのエンジニアリング時間が必要になります。ETL ツールが使用できて、セキュリティや運用保守の作業を行う前提で、エンジニアの時間の 50% が占有される (あるいは、これを複数のエンジニアで少しずつ分担する) と見積もってみましょう。

データセットには個人情報 (PII) が含まれているため、エンジニアはデータを匿名化する必要もあります。この作業は、先ほどの 50% の時間配分の中で行えることにします。

理論的には、これで本番環境さながらのデータセットが用意できましたが、データを匿名化したことで、エラー検知に最も役立つ 2 つの要素が失われてしまいました。想定外の状態になっていそうな古いデータにはランダムな文字列が埋め込まれ、他のレコードと区別がつかなくなっています。また、データのカーディナリティが変わったことで、インデックスの有効性が予測できなくなりました。

エンジニアリング時間のコストは、確かに一定しないものですが、現在は年間 5 万ドル以上費やしているとしましょう。デリバリー パイプラインには単一障害点があり、本番環境ではないことが明らかになるようにデータを操作し、テストのサブセットを実行することでフローが維持されるようにしました。この作業の有用性もかなり変動しますが、CircleCI の推定では、本番環境で発見されるエッジ ケースの約 5 〜 10% が検出されると見込んでいます。

テスト カバレッジを増やすには、データベースのインスタンス数を増やして並列に実行するか、すべてのデリバリーを遅らせてでも 1 つのインスタンスをより多くのテストで共有するようにする必要があります。ここでは前者を選択し、データベース インスタンスに年間 3 万ドルを費やして、増強したデータベースで 10 倍の並列ビルドが実行できるようにします。おそらく、これで 25% のエラー ケースを検出できるでしょう。

このモデル全体では、テスト サイクル中にクエリへの応答の遅さを把握できるような形ですべてのテストを記述することを前提としています。私の経験からすると、非常に珍しいやり方です。パフォーマンス テストに長けているチームでも、どのコード パスをテストするかについては意識的に決定しています。アプリケーション全体で応答時間を広範に検証するためのツールを構築、管理する作業が必要になるため、少なくともエンジニアの時間のもう 50% に相当すると想定しましょう。

すべて合わせて、年間 15 万ドル近くになります。これをまったく行わないとどうなるでしょうか。

本番環境で問題が見つかった場合のコストは、ユーザーへの影響という観点からほぼ推定できます。それには、2 つの値についてある程度合理的な仮定が必要です。1 つは、開発者による検出や CI 環境での検出が難しいバグが発生する可能性です。データの規模に関連しますが、バグが本番環境に入り込むとユーザーに深刻な影響が出ます。個人的な経験から、このインシデントは年に 1 回は発生すると予想されます。そしてもう 1 つ仮定する必要があるのは、このインシデントによる財務状況への影響です。15 万ドルのコストをかけなかったことに対して、年に 1 回代償を払うことになります。

基本的なモニタリング ツールを使用して、オールオアナッシングのデプロイメント戦略を採れば、このインシデントにすぐに気付けるでしょう。CircleCI のお客様を基準にすると、変更を元に戻せる能力のあるチームは、約 15 分でバグを特定して修正できます。アプリケーションの機能低下が 15 分間にわたって発生した場合、はたしてその損害は 15 万ドルで済むのでしょうか。皆さんならおわかりだと思います。

たとえば、Knight Capital は 2012 年 8 月 1 日、45 分間で 4 億 4,000 万ドルの損失を出しました。15 分で実に 1 億 5,000 万ドル近くになります。この問題を年間 15 万ドルで回避できていたとしたら (実際にはもっと少ない金額に抑えられます)、被った損失に相当する金額で、1000 年分の支出がまかなえたはずです。ただし、顧客への SLA が用意されていないような平均的なブログ エンジンの場合は、この

レベルの投資を行ったとしてもユーザーが得られるメリットはほとんどないでしょう。

ここまでは、影響を評価するにあたって、すべてのユーザーが影響を受けることを前提としてきました。カナリア デプロイメントのような優れた運用手法を導入すれば、更新したソフトウェアを本番環境に投入しても、それに対してルーティングするトラフィックを 2% に抑えることが可能です。使用率が十分に高ければ、同じようにすぐに問題が発生するはずですが、影響する範囲はトラフィック全体の 2% に絞られます。大規模なデータセットに対するコード変更の影響を阻止するには、カナリアへのルーティングを取り除くだけです。ほぼ瞬時に済みますが、長めに見積もって、回復するまでに 5 分かかるとしましょう。

先ほどの時間の 3 分の 1、影響の及ぶトラフィックは 50 分の 1 です。ここで、トラフィックの 2% (おそらくユーザーの 2% 相当) に対する 5 分間の機能低下を回避するために、15 万ドルのコストをかける価値があると判断できなくてはいけません。全ユーザーに対する 15 分間の機能低下に換算して考えるなら、自社にとって 2,250 万ドルの費用を投じるに値するかどうかということになります。

ここで強調しておきたいのは、これらの数字はどれも、皆さんの会社の実情に照らし合わせたものではないということです。チーム、製品、ユーザーに応じて詳細は異なります。上記の説明では、あらゆるケースを 1 つひとつ詳しく考察することが目的ではないため、シンプルさと正確さを優先しました。実際のコストを見積もるのは概算、つまり**フェルミ推定**のようなものです。主な手掛かりとなる数値が揃えば、すぐに答えを導き出せます。投入できる額には限度があり、だれもがそこから最大限の利益を引き出したいと考えているはずです。本番環境での検証を取り入れることに視野を広げ、そのためのトレードオフを理解しておけば、投資対効果を高めることができるでしょう。

進化する検証ツールキット

テクノロジーのトレンドやコスト モデルの変化からも、本番環境でのテストが今後も採用されていくことは明らかです。これは喜ばしいことでしょう。本番環境でのテストは、一連のバリデーション作業に新たに加わった手法であり、そのツールと戦略はこの 20 年間で成長と進化を遂げてきました。

歴史が示すように、物事に不具合が生じるのを防ぐことはできませんが、世界はリスクを受け入れる身軽さも十分に培ってきました。それでも、不具合が発生したら、それを修復する必要があります。本番環境でのテストを行うにあたっては、これまで以上に、本番環境を迅速かつ確実にコントロールできることが重要になります。そのためには、信頼性の高い CI/CD パイプライン、堅実なデプロイメント方法、すべてのステップでコードを継続的に検証するパイプラインが必要です。そして、本番環境でのテストを通じて機能していない部分を見つけたら、その不具合はそもそもどのようにして起こったのか、修正して元に戻すにはどうすればよいかを見極めなければなりません。それができないなら、本番環境でのテストは余計なリスクを生むだけです。

別の見方をすると、本番環境でのテストは、論理的にはデリバリー プロセスの延長線上に位置付けられます。高速な CI と本番環境でのテストの両方の戦略を取り入れれば、緊密なフィードバック ループを確保して、開発の流れをスムーズにできます。

> 「本番環境でのテストを通じて機能していない部分を見つけたら、その不具合はそもそもどのようにして起こったのか、修正して元に戻すにはどうすればよいかを見極めなければなりません」

問題が検出されたときには、すぐに信頼性の高い修正をプッシュする必要がありますが、これらの戦略を組み合わせることで、問題を速やかに見つけられるようになり、高スループットの CI/CD パイプラインの価値がさらに高まります。

最後にまとめると、本番環境でのテストには、以前なら高いコストがかかっていましたが、今では大幅にコスト面の負担が軽くなりました。ただし、実際に取り入れただけでは、コスト最適化に向けた最初の一歩を踏み出したにすぎません。仮に、皆さんの会社のシステムにおいて、本番環境でのテストが、コスト効率の観点から有効な選択肢となる転換点を迎えたとしましょう。そして、バリデーション プロセスに組み込み、複雑なリスクに対応できるようになったとします。そうなっても「なぜ本番環境でしかテストできないような複雑なものを構築してしまったのか。もっと効率的に構築する方法や、もっと慎重に設計する方法はないのだろうか」と自問することが大切です。本番環境でのテストがどれだけ安価になっても、より優れた設計を実現することこそが、サービスのテストにかかるコストを確実に削減できる道なのです。

CircleCI は、コードの信頼性を高めること、お客様がすばやく行動できるように支援することを目的に設立されました。

我々は、Rails から iOS や Docker へのお客様の移行、さらにはサービスの爆発的な増加など、現在に至るまでソフトウェア開発の世界が今までにないペースで進化を遂げていることを目の当たりにしてきました。

CircleCI の最大のミッションは、コードのバリデーションを通じて開発者の皆さんの自信を高め続けることです。本番前のテストも本番環境でのテストも、そのためのプロセスに含まれます。CI、本番環境でのテスト、そしてすべてのバリデーション戦略はどれも、コードの信頼性向上と市場投入までの時間短縮という同じ目的を支えるものであるため、1 つのサイクルのコンポーネントとしてより分析的かつ総合的に考えることをお勧めします。複雑なバリデーション作業の個々のピースを開発者と運用者の問題に分けるのではなく、包括的に取り組みましょう。そうすることで、コストとリスクについて適切に判断し、高品質のコードを配信し、優れた製品を作り出せるはずです。