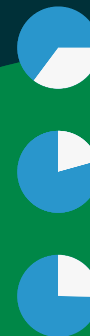
2020 年版
ソフトウェアデリバリーに関する現状調査

2020

データに基づく
エンジニアリングチーム向けベンチマーク

By Ron Powell and Michael Stahnke



circleci

ソフトウェア デリバリーで高いパフォーマンスを発揮できるのは、どのようなチームでしょうか。

自分たちのチームが順調かどうかはどう判断すればよいのでしょうか。

エンジニアリング リーダーは、どうすれば意義のある目標を設定できるでしょうか。

重要なメトリクスはどれでしょうか。

何が業界のベンチマークとなるのでしょうか。

エグゼクティブ サマリー

CircleCI には、世界のテクノロジー デリバリー チームがどのように業務を行っているかに関して、独自のデータが豊富にあります。このレポートを作成するにあたって、44,000 以上の組織の 16 万以上のプロジェクトから得られた、5,500 万以上のデータポイントを分析しました。

デリバリーの成功を示すメトリクスは、必ずしも 1 つではありません。それぞれのチームごとに評価指標は異なります。一方で、CircleCI のプラットフォーム上で見られたソフトウェア デリバリーのパターンには、大きな類似点がいくつかありました。特に上位のデリバリー チームのデータポイントにおいて顕著なこれらの類似点は、価値あるベンチマークとなり得るもので、チームの目標として使用できます。CircleCI の総合的なデータから特定できた、エンジニアリング チームのパフォーマンスに関するベンチマークは、以下の4つです。

- **スループット:** ほとんどの場合 (あるいは常に)、ワークフローの実行回数よりもデプロイ可能な状態を維持することの方が重要です。
- **実行時間:** ワークフローの実行時間は5～10分に収めるのが理想です。
- **平均復旧時間:** 実行が失敗したときは、それがいかなる原因であれ、修正またはロールバックにより 1 時間未満で復旧させることが目標です。
- **成功率:** アプリケーションのデフォルトブランチでは90%を超える成功率を基準とする必要があります。

企業固有の理由で、別のメトリクスを目標として選んでいるチームもあります。しかし、エンジニアリングの生産性を高めるためのどのような取り組みやプロセスであっても、その成功は、自社の基準メトリクスの測定能力と、たゆまぬ改善能力にかかっています。

あえて言うまでもないほど、2020 年は例外的な年でした。このレポートでは、世界経済の見通しが最も不確かで、だれもが不安のピークにあった時期のチームパフォーマンスを詳しく分析し、チームの成功のためにテクニカルリーダーが考慮すべき推奨事項をご紹介しています。一言で言えば、リーダーは困難に負けないチームの構築に集中し、個々人の "燃え尽き" を防止することに力を注ぐのが望ましいでしょう。これを実現する方法の 1 つは、より大きなチームを作ることです。チームの規模が大きければ、柔軟性が増し、新機能の開発にも取り組めます。保守の正常性も維持され、緊急の問題が発生しても十分な余裕を持って対処できるでしょう。**意欲のあるチームを作るには、ツールと人的プロセスの両方に、スケーラビリティが必要です。**

はじめに

波乱万丈の 2020 年は、ソフトウェアデリバリー チームにとってスムーズにサイクルを回せる体制が競争上の差別化要因となることが明らかになった年でもありました。すべての組織が COVID-19 (新型コロナウイルス感染症) に見舞われ、リモート中心どころかリモートのみで業務を行わねばならなくなったとき、数多くのチームが、数多くの手動プロセスを見直す必要に迫られました。だれかのデスクにビルド用マシンがあるという前提や、マシンに問題が発生したら再起動すればよいという前提が突然崩れたのです。まったく唐突に、すべてを自動化しなければなりませんでした。

この自動化という概念、つまり業務の高速化と信頼性向上の取り組みは、"あったら便利"程度にはとどまらず、ソフトウェアデリバリーチームの中核的な責務になりました。

2019 年版のレポートでは、継続的インテグレーション (CI) を少しでも取り入れているチームの方が、採用していないチームよりもはるかにパフォーマンスが高いことを示しました。ところが、今や CI と DevOps の取り組みは爆発的に広まったと言っても過言ではありません。みなさまの多くの競合企業もまた、それらの手法を使用するようになっています。

もし、みなさまの競争上の優位性がデリバリー能力の高さにあるとしたら (実際にそうだと思いますが)、この変動の時代にも引き続き優位性を保つにはどうすればよいのでしょうか。**そこで、本年度版では、CI/CD を実施しているチーム向けのまったく新しいベンチマークを設定しました。**パフォーマンスの高いエンジニアリング チームを量的に評価したときに、どのような特徴が見られるのかが浮き彫りになっています。

継続的インテグレーションの歴史の概観

ソフトウェア開発のアジャイル手法は、ソフトウェア チームが高品質なプロダクトをより効率的にデリバリーできるように作成されたもので、変更に対してオープンに構え、すぐに対応し、短い作業期間（スプリント）とデリバリーサイクルのために最適化を進めるというものでした。この開発理念により、失敗や方針転換は問題視されなくなり、むしろクールなこととして見られるようになりました。Google、Etsy、Thoughtworks などの企業は、この考え方を実践に移し、それによって世界でも最大手の、影響力のある企業に成長しました。こうしてアジャイルは、ソフトウェア開発の標準的な手法となりました。

アジャイルの他にも、組織のデリバリー能力を向上させた開発理念があります。アジャイルの導入で、開発チームの作業は非常に高速かつ反復的になり、変更を頻繁に行うことへの抵抗感は薄れました。しかし、市場投入の部分にはボトルネックがまだあり、同じような高速化は阻まれていました。そのボトルネックとは、QA （品質保証）とデプロイメント タスクを抱えた運用チームです。そこで導入されたのが DevOps 手法でした。開発チームに適用されたのと同様の反復の原則が、運用チームでも使われ始めます。コードの更新を行い、そのコードを顧客の元にある製品にデリバリーするために、パイプラインが作成されました。DevOps を導入できた企業は、その変革スピードで競合他社との差別化を果たしました。今日では、DevOpsはほとんどの企業で採用されるようになっています。

こうして、継続的インテグレーションという概念が生まれました。実践的なアジャイル開発の体現です。ビルド パイプラインにインテグレーションという概念が加わったことで、開発チームと運用チームは、コードベースへのすべての変更に対してテスト スイートを実行できるようになりました。テストが自動化されたことで、反復的なコードの刷新と迅速な更新が実現しました。ただし、このような高速なデリバリーでは、網羅的なテスト カバレッジが重要になります。すなわち、テストを実施することが、コードの信頼性を大幅に高めることを意味します。CI を通じて作業をすれば、十分なテスト カバレッジが達成されるため、チームは思いのままにデプロイができるという自信を付けて、パイプラインに自動デプロイメントのシナリオを含められるようになります。以上が、私たちがたどってきた歴史の概要です。

本レポートでは、CI パイプラインから意義のあるシグナルを得るための方法をご紹介しています。意義のあるシグナルを得るには、コードベースに対する網羅的なテスト カバレッジが必要です。カバレッジが完全でないと、たとえテストにパスしても、バグがないとは言い切れなくなり、テスト カバレッジの不足 (あるいはテストの欠陥や非決定性) が露呈するだけの結果になります。

完全なテスト カバレッジなしでは、実行の成功をコードの信頼性を保証するものと解釈することはできません。CircleCI をはじめとする CI ツールから価値を引き出すために、網羅的なテスト カバレッジは最も欠かせないものと言えるでしょう。『2020 年版 State of DevOps レポート』では、調査対象の全グループで、変更管理を自動化するうえで最大の課題が不完全なテスト カバレッジであったことが判明しました。このトピックの詳細については、『Software Testing for DevOps-Driven Teams (DevOps 駆動型チームによるソフトウェア テスト)』を参照してください。

競争力を付けるには

信頼性の確保と迅速な行動を両立させる継続的インテグレーション&継続的デリバリー(CI/CD)パイプラインは、"あったら便利"のレベルを超えました。もしみなさまがソフトウェアのデリバリーを手掛ける企業であるなら (おそらくそうでしょうが)、速さと規模を兼ね備えた高品質なデリバリーこそが、市場での価値と競争力を維持する鍵です。とはいえ、継続的インテグレーションを実践しているチームにとっての標準とは何でしょうか。デリバリー能力をさらに磨くためには、みなさまの CI の取り組みをどのように測定し、改善すればよいのでしょうか。

CircleCI は世界最大規模の独立した CI プロバイダーです。数万に上るチームを対象に、コミット内容を独自に詳しく調査して、ソフトウェアデリバリーを量的に評価することができます。今回は、当社のプラットフォーム上にある 1,100 万のワークフローから取得したデータを分析して、さまざまなチームが実際にどのようにソフトウェアをビルドおよびデプロイしているかを調査しました。

実際のところ、パフォーマンスの高いチームはどのように作業しているのか。調査では、この疑問に対する明確な答えを探りました。

調査結果のレポートの前に、パフォーマンスの測定に役立ついくつかのキーワードをご説明します。

- **継続的インテグレーション(CI):** ソフトウェアデリバリーの基盤です。CI によって、ソフトウェアのビルド、テスト、デプロイの自動化されたステップが定義されます。
- **実行時間:** ワークフローの実行にかかる時間です。
- **スループット:** ワークフローの1日あたりの平均実行回数です。
- **平均復旧時間:** 失敗が起きてから、次に成功するまでにかかる時間の平均です。
- **成功率:** 一定期間内に成功した実行の回数を合計実行回数で割った値です。

これらのデータはCircleCIから収集されていますので、当社のプラットフォームがどのように処理しているかを理解するのに役立ついくつかの用語もご紹介しておきます。

- **ジョブ:** 順番に実行されるコマンドの集合です。
- **ワークフロー:** ジョブの集まりです。ワークフローにより、一連のジョブの実行順が定義されます。
- **パイプライン:** 1 つ以上のワークフローで構成されます。パイプラインにより、すべてのアクティビティのオーケストレーションが行われます。これにより、コードのコミットからデプロイまでが完了します。

実行時間、スループット、平均復旧時間、成功率を最適化すれば、DevOps 成熟度の低い企業に対して圧倒的優位に立つことができます。このことはデータがはっきりと示しています。

それでは、各メトリクスについて見ていきましょう。

実行時間

実行時間は、1 つのワークフローの実行にかかる時間の長さとして定義されます。これは最も重要なメトリクスです。なぜなら、高速なフィードバック サイクル (スループットや平均復旧時間を含む) を作り出せるかどうかは実行時間にかかっているからです。言い換えれば、ワークフローの実行が終わらない限り、どれだけ必要な修正であってもプッシュすることができません。実行時間は、開発者が意味のあるシグナル (「ワークフローの実行は成功したか失敗したか」など)を得られるまでの速さも表しています。実行時間を短縮するには、最適化されたワークフローが必要です。

すべてのワークフローが同じ終了状態をもたらすわけではありません。たとえば、アプリケーション コードベースの変更箇所によっては、特定のテストのためだけに実行されるワークフローもあります。そのため、実行時間は、本番環境へのデプロイ時間の明確な測定結果ではありません。ワークフローが完了するまでの時間を示しているにすぎません。

CI の最終的な目標は迅速なフィードバックです。認知していないものを修正することはできないため、ビルドの失敗を示すシグナルはなるべく速やかに開発者に届ける必要があります。しかし、認知するだけでは足りません。開発者には、失敗したビルドの情報も必要です。適切な情報を得るためには、厳格なソフトウェア テストを記述する必要があります。

ここで、速度だけがゴールではない点を理解することが重要です。テストのないワークフローはすばやく実行でき、成功のシグナルが返ってきますが、それはだれの役にも立ちません。チームは失敗からなるべく多くの情報を取得して、できる限り迅速に対応できるようになる必要があります。品質の良いテスト スイートがなければ、実行時間の短いワークフローはフィードバック サイクルに何の価値ある情報ももたらしません。つまり、目標とするべきなのは、豊富な情報量と短い実行時間の両立です。

テストはしばしば、ビルドが成功か失敗か、グリーンかレッドかの二元論で語られます。しかし、3 種類の結果を想定する方が役立ちます。第 3 の状態は「エラー」です。具体的には、次のようになります。

- ビルドは成功し、実験も成功した。
- ビルドは失敗し、実験も失敗した。
- エラー:実験の完了に失敗した。

大半の時間を注ぎ込むべきなのは最初の 2 つのシナリオ、つまりビルドの成功と失敗であることに注意が必要です。最終的な成果が得られ、開発者が意義のあるシグナルを得られるのは、この 2 つです。言い換えると、開発者がエラーに注ぎ込んだ時間は、得られるはずのないシグナルを待っている時間です。エラーの原因がインフラストラクチャの欠陥や容量の不足であることがあり、その場合すぐに事態は複雑になってしまいます。もし既存のCIパイプラインで頻繁にエラーが発生しているようなら、ツールを見直す時期かもしれません。

平均復旧時間

平均復旧時間は、失敗が起きてから、次に成功するまでにかかる時間の平均として定義されます。これは２番目に重要なメトリクスです。失敗のシグナルが発生したときにチームが迅速に問題に対処できるかどうかは、計り知れないほど重要です。平均復旧時間はテストカバレッジの包括性が上がるにつれて短縮されるため、このメトリクスを見れば、アプリケーションがどれほど入念にテストされているかを測ることができます。

ビルドの失敗、価値あるシグナルの認知、迅速な修正、そしてビルドの成功。継続的インテグレーションによって、この高速なフィードバックループが実現します。シグナルの高速化により、チームは新しい挑戦をしながら、あらゆる事態に即座に対応できるようになります。同様に、十分なテストカバレッジによって、壊れたコードを本番環境のコードベースに導入する心配が低減され、開発を担当するエンジニアリングチームに創造性と敏捷性を高めるよう促すことが可能になります。

スループット

スループットは、ワークフローの１日あたりの実行回数の平均として定義されます。ワークフローは、開発者が共有リポジトリにあるコードベースを更新したときにトリガーされます。お使いのバージョン管理システム(VCS)へのプッシュが行われると、ワークフローを含むCIパイプラインがトリガーされます。

ワークフローの実行回数は、アプリケーション開発パイプラインを通過する個別の作業単位の数を示しています。コミットのサイズも、スループットに影響する要素の１つです。プッシュされたのが多数の小さい変更か少数の大きな変更かによってスループットが変わります。チームによって作業単位の適切なサイズは異なりますが、目標は、すばやく簡単にデバッグができる程度にサイズを小さくしつつも、意義のある変更をデプロイできる程度の大きさを確保することです。

お勧めなのは、設定した目標に対するスループットの達成率を監視することです。事象の発生頻度を監視することは重要です。スループットはコミットの頻度を直接測定するものです。同じタスクに２人の開発者が取り組み、プッシュされるコミット数も少なくなるオンボーディングのような状況では、スループットは変動する可能性があります。組織のベースラインとなるメトリクスを確立しておけば、このような予測可能な状況でのエンジニアリングの生産性を前もって考慮し、影響に対処することができます。未知の状況に直面したときも、ベースラインが定まっていれば、未完了作業の量を判断するのに役立ちます。

スループットとデプロイ頻度は異なるものですが、どちらも生産性の目安として使われてきました。しかし、このことは誤判断の原因になることがあります。たとえば、スループットでは明示的に測定できない事柄として、デプロイの発生の有無があります。１日あたりのデプロイ ワークフローの数が多かったからといって、意義のある作業が行われたとは限りません。重要性の低い小さな変更を複数プッシュしただけでも、この数は増加します。

たいていの場合、向上させたいのは量よりも品質です。ですが、この違いをわかっていれば、スループットとデプロイ率はどちらも価値あるメトリクスとなり、デリバリー能力の記録、トラブルシューティング、微調整に役立ちます。

新たな変更をすべて継続的にバリデーションできていれば、アプリケーションはよくテストされた状態に保たれ、いつでもデプロイできるようになります。完全に自動化されたソフトウェア デリバリー パイプラインがないと、チームがデプロイするのは緊急時か、トラブルに備えての訓練のときだけになり、しかも間の悪いタイミング (金曜日の夜など) になりがちです。完全に自動化されたソフトウェア デリバリー パイプラインがあれば、思いどおりの頻度 (とタイミング) でエンド ユーザーに更新をデリバリーできます。即時のホットフィックス、開発したばかりの機能のアップグレード、ビジネスニーズに基づいて設定された予定表の大規模変更などが思いのままです。**1日あたりの特定のデプロイ数を達成することが目標ではありません。パイプラインを通じて行う、コードベースの継続的バリデーションこそが目標です。**

成功率

成功率は、一定期間内に成功した実行の回数を合計実行回数で割った値として定義されます。デフォルト ブランチではなくトピックブランチで開発を行うgit-flowモデルを採用していれば、デフォルトブランチの成功の状態を維持することができます。重要な注意点の1つとして、ワークフローがデフォルトブランチとトピックブランチのどちらで実行されているかによって、成功率は大きく変動することが予想されます。多くのgit-flowモデルでは、ほとんどの作業がトピックブランチで行われるので、成功や失敗のシグナルが現れる実験のほとんどもこちらで実行されます。トピック ブランチでの機能開発まで範囲に含められるようにすることで、意図的な実験 (ビルドの失敗が価値あるものとして扱われ、想定されている) と安定性に関する問題(ビルドの失敗は望まれていない)を区別することができます。**デフォルトブランチでの成功率は、トピックブランチでの成功率よりも意義のあるメトリクスです。**

新機能やバグ修正の開発とテストをデフォルト ブランチ以外で行うことで、チームはトピック ブランチ上で十分にテストされたコードだけをデフォルト ブランチにマージできるようになります。こうすることで、最速でシグナルを受け取れる場所がトピック ブランチになります。ここでなら、エンド ユーザーにネガティブな影響を与えることなく安全に失敗できます。しかも、チームのメンバーにもほとんど影響はありません。影響が及ぶのは、同じブランチで作業している開発者だけです。トピック ブランチを 1 つのプロジェクトに複数作り、並行して開発することもできるようになります。

本レポートでは、本番環境へのデプロイの準備ができたアプリケーション ブランチを "デフォルトブランチ" と呼びます。このブランチには一般的に、"トランク (trunk)"、"マスター (master)"、"メイン (main)" などの名前が付けられていることもあります。ブランチの名前付けと、その歴史的背景に関しては、後ほどレポート内で詳しく解説します。

データからわかること

以降のセクションでは、2020年8月1日から30日までの間にCircleCIで確認された1,100万以上のワークフローから得られたデータを詳しく見ていきます。

データソースの内訳は以下のとおりです。

- 1日あたりに実行されるジョブは200万件
- 44,000以上の組織
- 16万以上のプロジェクト

2019年は、90日の間に発生した3,000万のワークフローを分析しました。今年は集計期間内で起こる変動の影響を抑えるために、特定の1か月に注目します。

本レポートでは、CircleCI の顧客データ セット全体を対象としてワークフローを総合的に分析しますが、CircleCI Insights ダッシュボードをお使いいただくと、ワークフロー単位でメトリクスを分析することもできます。

実行時間

実行時間のデータについて解説するにあたって、注意事項が 1 つあります。**ワークフローの実行時間をできるだけ短くすることがゴールではありません。** たとえば、ワークフローの実行時間データうち、最短から 10 パーセンタイルは 30 秒未満です。シナリオによっては、この実行時間でも十分なカバレッジを確保できるかもしれませんが、ほとんどのワークフローでは 30 秒では堅牢なテストを実行できません。テストは、失敗したワークフローに対する理解を深めてくれます。そのため、テストカバレッジ(そしてそこから得られる意義のあるシグナル)を最優先の目標に据えるべきです。それさえ確立できれば、実行時間を最適化して短縮できます。

また、CircleCI では実行時間を最も重要なメトリクスと考えていますが、注目しているのは速度だけではありません。目標は、品質の確保です。CircleCIでは、堅牢なテストによって品質を確保しています。

測定された1,100万のワークフローの半数が、4分未満で完了しています。また、ワークフローの4分の3は11分未満で完了しています。さらに、測定対象の全ワークフローのうち、95%が35分未満で完了しています。

チームとして、どの程度の実行時間を目標にするべきでしょうか。CircleCIの経験からは、実行時間が5～10分であれば、迅速に情報を得るのに十分な短さと、ビルドの失敗時に価値あるインサイトを得るのに十分な長さが両立していると言えます。

5～10分よりも短いと、フィードバックが完了しない可能性があるので、復旧時間はかえって長くなってしまいます。逆に、これよりも長いと、開発者の注意力がそがれてしまいます。長時間かかるテストを待っているうちに、別の作業を始めてしまうからです。そのようなシナリオでは、開発者はテストの終了後、発生した問題の修正に必要なコンテキストを復元する必要があるため、失敗した状態で過ごす時間が長くなります。

平均復旧時間

最速の平均復旧時間は、数分レベルでした。繰り返しになりますが、必ずしも、平均復旧時間の絶対的な短さだけが目標基準になるのではありません。そのような認識を変えるには、たとえば、できの悪いコードを送信したと気付いてからすぐに、元のワークフローがまだ実行されている間に修正をプッシュした開発者がいたらと考えてみてください。あるいは、2 人の開発者が数分間だけ間を空けて、同じメイン ブランチに複数のコミットをプッシュし、片方が失敗して片方が成功した状況を想像してみてもよいでしょう。どちらも、普通ならありえなさそうなシナリオです。平均復旧時間は2番目に重要なメトリクスなので、目標にはもっと現実的な事象を反映する必要があります。

CircleCIのデータセットでは、ワークフローの50%が平均して約55分で復旧しています。つまり、500万以上のワークフローが平均1時間未満で復旧しているのです。これが現実の分析結果です。75パーセンタイルでは、平均復旧時間が最長9時間半に及ぶワークフローがデータセットに含まれるようになり、急激な長時間化が見て取れます。

チームにとって適切な目標を設定するには、さまざまな要素を考慮する必要があります。たとえば、組織の開発者が全員 1 つの街にいて、通常の営業時間に業務にあたっているなら、失敗したビルドを退勤前に解決することは難しいでしょう。一方、グローバルな開発チームのある組織なら、日々の作業が滞らないように、チーム間で作業を受け渡しできるようなプロセスが採用されている可能性が高いでしょう。最も重要なのは、現在の能力を測定する力です。このインサイトを活用することにより、反復的に改善を進め、時間短縮を目指すことができます。

> 最終的に、失敗したビルドを 1 時間未満で解決できるようになることが、迅速な問題解決のためには最も望ましいと言えます。

もちろん、さらに高いチーム目標を設定してもかまわないでしょう。

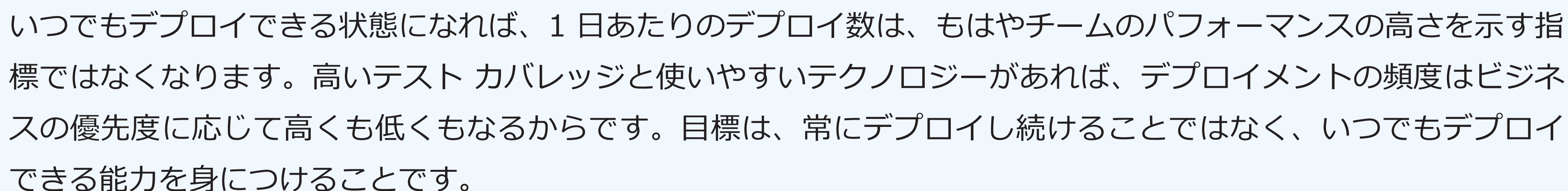
スループット

年を追うごとに、この業界では、パフォーマンスの高いチームは 1 日あたり数十回もデプロイをしているという話を聞くことが増えています。カンファレンスのトピックにすれば盛り上がりそうな統計情報ですが、CircleCI のデータからはその情報の裏付けは得られませんでした。多数のチームがそれほどの頻度でワークフローを実行している様子はありません。

このデータセットには 8 月に CircleCI で実行されたすべてのワークフローが含まれているということを強調しておきます。たまにしか実行されないワークフロー、プロジェクトでよく実行されるワークフロー、1 回だけ実行された後に削除はされないまでも破棄されるワークフローなど、さまざまなワークフローがほとんどのチームで存在しています。このレポートでは、ワークフローの正規化は行っていません。使用状態がアクティブでないものも含まれています。

CircleCIのデータセットにあるワークフローの50%は、1日あたりの実行回数が1回未満でした(平均で1日あたり0.7回です)。ただし、95パーセンタイルでは、1日あたり35回を超えて実行されているワークフロー、または24時間を通して45分ごとに実行されているワークフローが含まれていました。

みなさまのチームではどうでしょうか。ソフトウェアデリバリーの頻度は、チームが構築しているソフトウェアやビジネスニーズに合わせて決めるべきです。特定のスループット値の達成を追求したり、他社と自社の統計情報を比較したりするのではなく、ベースラインとなるスループットを測定してから変動を監視するようにすれば、開発パイプラインの正常性についてより深く理解できます。



成功率

成功率の重要性は、チームの構造や規模、作業内容によって変動します。1 人が 1 つのトピックや機能に取り組んでいて他のだれも関係していないブランチであれば、ビルドの失敗は影響が小さいでしょう。一方で、メイン ブランチが成功状態になるまでエンジニアリング部門全体が作業できないという状況であれば、ビルドの失敗を防ぐことが何よりも重要になります。当社のデータには、これら両方のシナリオ、およびその中間のあらゆるワークフローが含まれています。

1か月間、ビルドの失敗が一切なかったサンプルもあれば、多くの失敗が見られたサンプルもあります。成功率の中央値は61%でした。デフォルトブランチで実行されているワークフローとそうでないワークフローを分けて見てみると、さらなるインサイトが得られます。別々に集計すると、デフォルト ブランチの成功率の中央値は 80% で、それ以外のブランチの成功率の中央値は58%でした。このことは、多くのチームがアプリケーション開発でgit-flowモデルを使用していることを示しています。このモデルでは、開発者はトピックブランチまたはデフォルト以外のブランチで実験を行っており、ビルドの失敗もこれらのブランチで起こるよう想定されています。このような作業をトピックブランチだけに限定することにより、デフォルトブランチでのエラーの発生が防止されます。

ワークフローの成功率は、開発に選んだgit-flowモデルによって異なります。トピックブランチで開発することを選んだチームの場合、デフォルトブランチでの成功率の目標は高く設定する必要がある一方、デフォルト以外のブランチには目標値を設定しません。

> デフォルトブランチでの成功率の現実的な目標は90%以上です。

多くのチームが、デフォルトブランチで実行されるワークフローでこの目標を達成しています。いずれにしても、チームの目標を確立するために重要なのは、現在のワークフローの成功率を測定する能力です。**改めて言いますが、ビルドの失敗は必ずしも悪いことではありません。特に、早く失敗のシグナルを得てチームが迅速に問題を解決できる状況ならなおさらです。**

チームの目標を設定するには

すべてのチームが追求するべき唯一の基準は存在しませんが、CircleCI のデータと、CircleCI プラットフォームで観測されたソフトウェア デリバリーのパターンを見てみれば、チームの目標として合理的なベンチマークはいくつか存在しているのがわかります。最終的に価値があるのは、"理想の" 数値を達成することではなく、これらのメトリクスに基づいて自社のベースラインを測定し、徐々に改善していくことです。自社チームのメトリクス情報や目標を達成する方法については、CircleCI のInsightsダッシュボードをご覧ください。

	CircleCIユーザーの中央値	推奨ベンチマーク
スループット ワークフローの1日あたりの実行回数の平均	0.7回/日	すべてのプルリクエストがマージできる状態であること
実行時間 ワークフローの実行にかかる時間の長さ	4分未満	5～10分*
平均復旧時間 失敗が起きてから次に成功するまでにかかる時間の平均	56分未満	1時間未満
成功率 一定期間内に成功した実行の回数を合計実行回数で割った値	デフォルトブランチで80%	デフォルトブランチで90%以上

* ここに記載した長さがみなさまのプロジェクトに当てはまるかどうかは、ワークフローの内容しだいです。
特定の数値の達成を目指すより、ベースラインの実行時間を把握し、それを改善しようと取り組むことの方が重要です。

製造に関しては、マインドセット、ツール、プロセスに関するエンジニアリングの成熟度は長い道のりを歩んできました。組織は提供中のサービスについてより深く把握していて、製造効率や障害原因の根本的な解決がかつてない速さでできるようになりました。同じような厳しさがソフトウェア開発の分野にも向けられていることを見るのは喜ばしいことです。コントリビューターの人数とコードベースの複雑さが増すにつれ、CIをボトルネックにしないことがこれまでにないほど重要になっています。

CircleCI でも同様の傾向が見られます。数々の組織においてデフォルトブランチでの失敗を開発の停止として扱い、最大の緊急事態として対処しています。開発の停止は、ビジネスに重大な影響を及ぼします。関連コストは増加し、生産性は失われます。CI の可視性が高まったことにより、エンジニアリングリーダーが直感的に知っていたことが、ようやく定量的に提示されるようになりました。

パフォーマンスの高いエンジニアリング組織なら、次の2つの質問に答えることができるはずです。

- デフォルトブランチで失敗が発生したとき、修正するのにどれくらいの時間がかかるか。
- デフォルトブランチではどれくらいの頻度で失敗が発生するか。

製造の場合と同様、平均復旧時間 (MTTR) と平均故障間隔 (MTBF) を把握しておけば、改善もしやすくなります。適切なツールを提供できれば、エンジニアがデフォルトブランチの成功状態を維持できるようになります。

BRYAN LEE 氏

DataDog 社、プロダクト マネージャー

Webサイトをデジタルプロダクトやソフトウェアのように扱えるよう支援するWebOpsプラットフォームとして、我々は四半期(以上) のリリース サイクルを週レベルに短縮できるよう支援することがよくあります。これはきわめて大きな変革です。設計と機能をこのペースで進化させられるようになれば、"ビッグバン"リリースにすべてを賭けている現状から脱却できます。クラウドベースの環境とCIを利用したオンライン開発は、このような状況で大きな助けになります。なぜなら、シリコンバレーの決まり文句である"すばやく行動し破壊せよ"とは対照的に、ほとんどのチームはWebサイトを破壊することが許されていないからです。

我々が把握している別の変化には、振る舞いや機能の"アウトサイドイン(outside-in)"テストのさらなる自動化があります。手動で行われていたいわゆる"スモークテスト"QAを、スクリプト制御のヘッドレスブラウザーとビジュアルリグレッションテストを使用して自動化することで、あらかじめ決まった領域しか見ない人間なら見落としてしまうようなエッジケースを検出できるようになります。Google社のBrettSlatkin氏が2013年にConsumerSurveyのプロダクト開発でこの手法を使用していると話すのを聞いたことがあります。非常に興味深いことに、今ではこれが主流になりつつあります。

JOSH KOENIG 氏

Pantheon 社、製品責任者

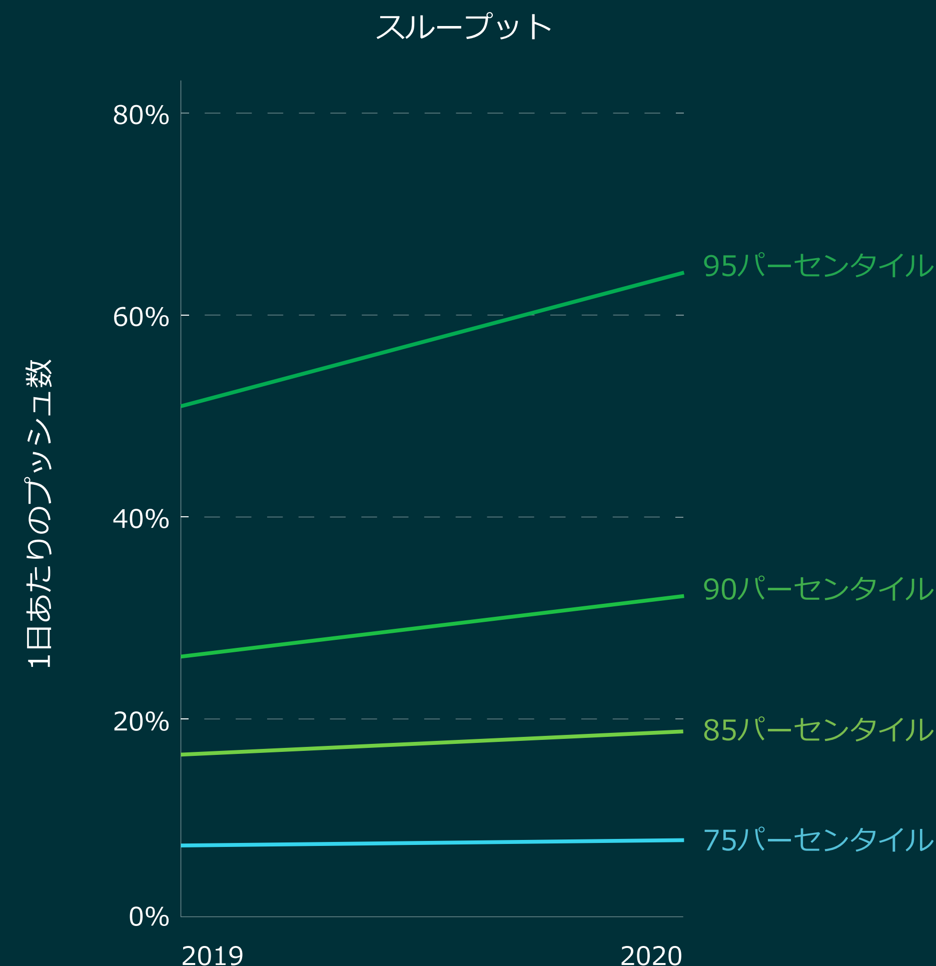
2020年と2019年のデータを比較してわかること

今回のレポートは、CircleCI の年間メトリクス レポートとして２年目にあたります。２年分のデータと2つのデータセット(2019年と2020年のそれぞれ8月1日から30日に実行されたワークフローを含む)が揃ったので、過去12か月でソフトウェアデリバリーがどのように進化したのかを確かめてみました。

その結果判明したのは次のようなことです。

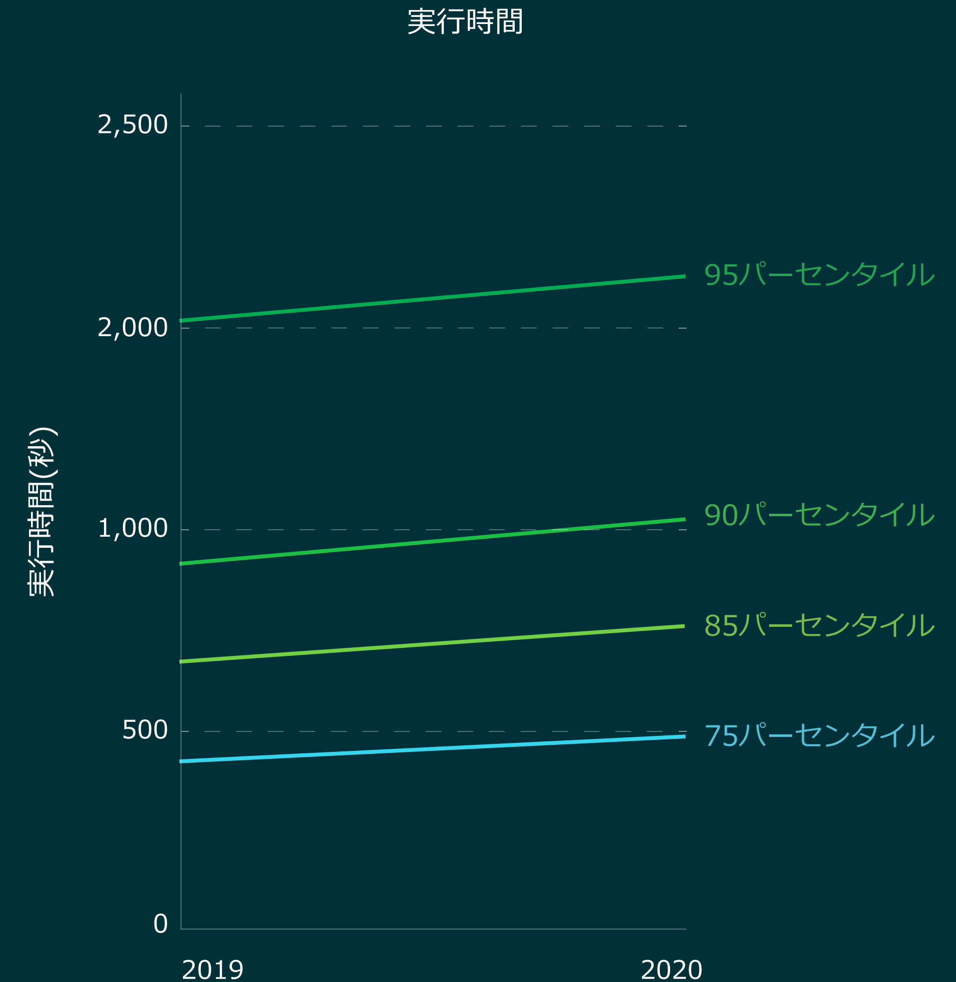
最高レベルのパフォーマンスを達成しているチームのスループットのしきい値が、2020年は高くなっている

75パーセンタイル以上のワークフローのスループットの値が高くなっていました。1日あたり１回未満しか実行されないワークフローはまだ数多くありますが、きわめて実行回数の多いワークフローの数が増加していました。



ワークフローの実行時間は長くなっており、テスト回数の増加が原因である可能性が高い。

すべてのワークフローの実行時間が長くなっていました。変化が最も小さかったのは、最も実行時間の短いワークフローです。しかし 1 分未満で終了するワークフローを分析しても、CI フィードバック サイクルの効率性向上に役立つような価値ある情報は得られなさそうです。最も実行時間の長いワークフローを見ても、去年と比べて大幅に長時間化しているわけではありません。四分位範囲、つまり25パーセンタイルと75パーセンタイルの間に位置するワークフローの実行時間を見ると、2019年と比べて約15%長くなっていました。テストの実行回数の増加が、このような実行時間の増加につながったのだとCircleCIでは推測しています。75パーセンタイルには実行時間が最大11分のワークフローが含まれていることに注意してください。これはワークフローの実行時間としては理想に近いものと言えそうです。テストを入念に行うとワークフローの実行時間が長くなるかもしれませんが、それによって復旧時間が短くなるなら、この時間も価値ある投資です。来年も実行時間の長時間化傾向が引き続き見られるかどうか、ぜひ注目してください。

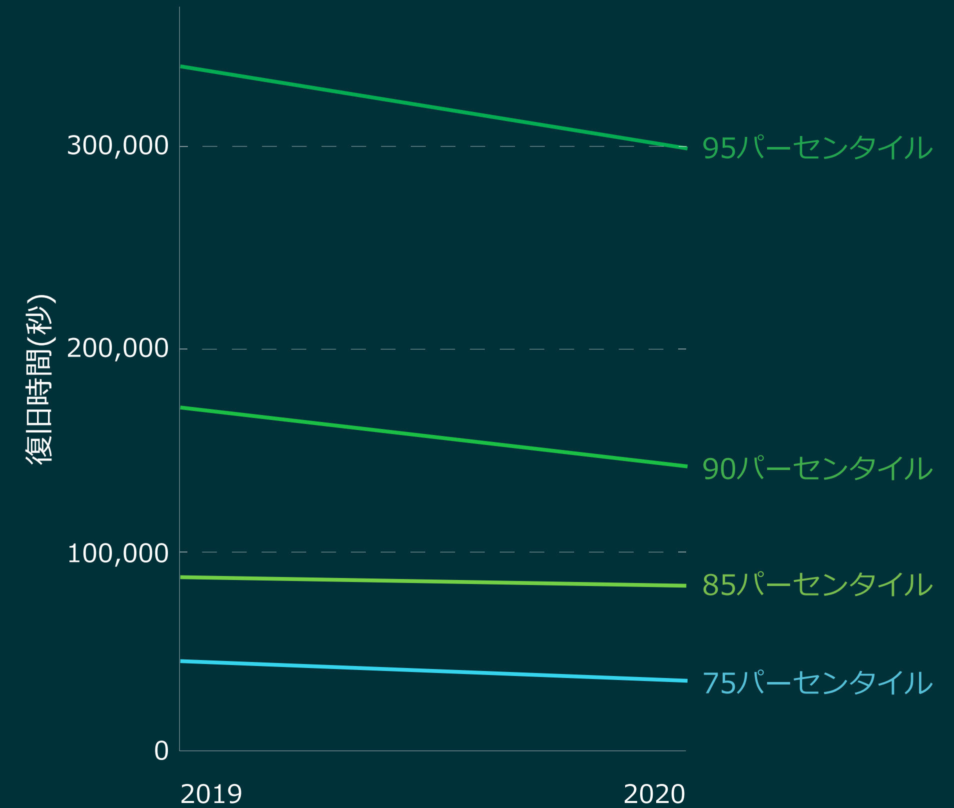


2020年は平均復旧時間が短縮された。

2020年に75パーセンタイル以上で記録された最長の復旧時間は、2019年の75パーセンタイル以上での記録よりも短いものでした。ワークフローの実行時間が明らかに長くなっていることと合わせて考えると、これは実行時間の長いワークフローが入念にテストされており、そのおかげで復旧時間が短縮されていることを示しています。

成功率は2019から2020年にかけてほとんど変化がない。

2019年8月の成功率の中央値は60%でした。この値は、2020年8月には61%に向上しました。既に説明したとおり、git-flowモデルを使用しているチームではデフォルトブランチで高い成功率を達成する(100%に近づける)ことが期待されます。失敗は、開発プロセスの一貫として、トピックブランチで発生するものと捉える必要があります。



50パーセンタイルの成功率:

60%
2019

61%
2020

2020年:世界的な混沌の中、エンジニアリングチームは何に取り組んだか

2019年から2020年にかけて、CircleCIの4つの主要なメトリクスに大きな変化は見られませんでしたが(変化が起こると予想されていたわけでもありませんが)、グローバル経済の不確かさや蔓延する不安、ソフトウェア開発業界で起こったかつてない変化から受ける影響を確認することは重要だと感じられました。

COVID-19はチームの生産性にどのような影響を及ぼしたでしょうか。

BlackLivesMatter運動を受けて、名前付けの慣例を変更することはあったでしょうか。

困難に負けないチームを作るため、今年の出来事からリーダーが学ぶべき教訓とは何でしょうか。

COVID-19の影響の表れ

ここまでは、2020年8月と2019年8月のデータを比較してきましたが、今年のレポートではもう一歩踏み込んで、COVID-19の世界的パンデミックによる影響が最も大きかったと思われる期間のデータを分析しました。2020年3月、4月、5月のデータを確認し、これらの月のデータを年間レポートの8月のデータと比較しました。

その結果、次のようなことが判明しました。

先行きが不確かな状況下で、開発者は自分がコントロールできるシステムを安定させるために、自動化に関心を向けていました。

今年のスループットと成功率のピークは 4 月でした。世界中で外出禁止令が発令された際、多くのチームができる限り多くの手動タスクを急いで自動化しようとしたというのが、CircleCIの仮説です。4月のスループットと成功率の向上は、このような事情を反映したものでしょう。

デフォルトブランチとトピックブランチの成功率の違いについて、レポートの前半で述べました。もしブランチ全体で成功率が向上していたなら、それはチームがトピックブランチのイノベーションから一歩後退し、ビジネスクリティカルなシステムの補強に注力したことを示しているかもしれません。

可能性は他にもあり、考察すれば興味深い発見があるでしょう。CircleCI の経験上、開発者は、問題の解決や、その方法の発見を非常に重視します。この時期、エンジニアの人々が、世界的なトレンドに乗ってオンラインでの業務参加を積極的に実施し、チケット対応、テスト、コードベースの保守に取り組んだ可能性は高いでしょう。これは問題の解決策や軽減策であると同時に、不確かさに対する不安を和らげるための方策でもあったはずです。

さらに、4 月には対面式会議の実施も停止され、遠方への移動も仕事か私用かを問わず中止されました。企業にとっては、(ほぼ) すべての従業員にかかわる規模の稀有な体験となりました。従業員が力をフルに発揮したのですから、生産性の向上は当然のことだったと言えるでしょう。スループットは4月に最も高く、5月から8月にかけて下がっていきました。

4月は週末にすら、スループットの向上が見られました。自宅で過ごす時間が増えたので、開発者が業務時間外でも作業しやすくなっていたのかもしれません。だとすれば、8月までにスループットが落ちていたことも納得できます。きっと燃え尽きてしまったのでしょう。

BlackLivesMatter運動を受けて、ブランチの名前付けを変更することはあったでしょうか。

ソフトウェアエンジニアリングにおける "マスター(master)" と "スレイブ(slave)" という用語の使用をめぐる議論は特に新しいものではなく、多くの人々が何年も取り組んできた深い問題でした。議論自体は以前からあったものの、もっと大規模な社会的運動を反映し、この問題は2020年にさらなる注目を集めました。2020年5月、ミネソタ州ミネアポリスで起こったGeorgeFloydの死により、構造的な人種的不平等に新たな注目が集まり、問題提起がなされました。ソフトウェア業界では、しばしばプロジェクトのデフォルトブランチの名前を"マスター"から"メイン"など他の名前に変更せよという議論が行われていました。

もし多数のチームがデフォルト ブランチの名前を変更していたら、CircleCI のデータにも反映されているはずです。CircleCI では当初、多数の組織がブランチ名を変更したと予想していました。しかし、データを見てもそのような裏付けは見当たりませんでした。

一体なぜなのでしょうか。多くのチームがこのような変更をしているという、事実と違う情報が、ソーシャル メディアによって増幅された可能性はもちろんあります。一方で、組織のリポジトリのデフォルト ブランチの名前を変更することは影響が非常に大きく、互換性を損なう変更を避けるために多くのエンジニアリング作業が必要になるという点は考慮に値します。エンジニアリング上の課題があったために、チームが変更を実装できなかったのではないでしょうか。そのような中、デフォルト ブランチの名前の変更に興味のある組織向けに、**GitHubでベストプラクティスガイドがロールアウトされました**。既にGitHub上の新しいプロジェクトでは、すべてのデフォルトブランチの名前が"メイン"になっています。この名前付けのトレンドに関するデータは引き続き追跡し、来年のレポートで評価する予定です。

先行きが不確かな今、困難に負けないチームを作るには

CircleCIの主要メトリクス４つのうち３つは、チームの規模の拡大とエンジニアリングパフォーマンスの向上を反映します。チームの規模(特に、特定のプロジェクトのコントリビューター数)は重要です。先行きが不確かな今、リーダーはチームの規模を拡大して衝撃吸収力を高め、変化に対応する必要があります。

実行時間は、チーム人数が 1 人のときに最も長く、チームの規模が大きくなるほど短くなります。大きな組織はしばしば、作業を分割して小規模チームに割り当てます。その小規模チームが責任を負っている範囲のコードに小さな増分変更を行い、関連性の高いコードだけテストすることで、ワークフローが短縮されているのです。平均復旧時間も、チームの規模が大きくなるにつれて短くなります。より多くの人が失敗の優先順位付けに参加すれば、解決にかかる時間も短くなることは納得できます。大規模チームのスループットが高いことにも、これで説明が付きます。

データによれば、チームの人数を増やすにしたがって、システムを通じてプッシュできる作業の量が多くなり (スループット)、平均復旧時間も短縮されます。しかし、理想的なチームの規模というものはあるのでしょうか。理想的なチームの規模は、経験、全体的な責任範囲、オンコール業務の量などにより左右されますが、CircleCI のデータを見ると、5人から20人のコードコントリビューターが適切だと言えます。

チームは大きいほど良いという考え方も、チームパフォーマンスのエクスペリエンスに関するデータから間接的に判断したものです。2020年に明らかになったことの1つは、どれだけ綿密に計画を立てていても、人生は思いどおりにはいかないということです。チームが小さすぎると、だれかが長期休暇を取ったり、緊急事態に陥ったり、あるいは単に休息が必要になったりしたときに、燃え尽きてしまう危険性が高くなります。理想のチームの規模とは、人生の中で起こる不測の事態を吸収できるほどの人手がある規模のことです。

プラットフォームやサービス、テクノロジーの耐久性は、たしかにCI/CD手法、監視、テストなどのDevOpsプロセスの質によって左右されます。しかし、重要なのは人です。

> パフォーマンスの低下は、しばしば担当者にとって大きな負担となります。
>
> システムとチームの強さは、それらを実際に動かしている人々をどれだけ優先できるかによって決まります。

困難に強いチームを作るにはどうすればよいのでしょうか。完全に分散化された組織では、日常業務もこなしつつイノベーションもできるほど十分に大きなチームを持つことが不可欠です。このことは、ユーザーの期待と要求が劇的に大きくなったこの時代に特に重要になっています。継続的に増える保守やエスカレーションの負担に対処するため、チームには十分な人数のコントリビューターが必要です。チームが小さすぎると、イノベーションのペースも下がります。

ホースの圧力を保つこと、つまり前進する意欲を保つことが、チームにも顧客にも重要です。確実に前進しているという手応えを持って顧客に与えるためには、一定のリズムでイノベーションを続ける必要があります。

困難に強いソフトウェアデリバリーチームは、次の3つの優先事項のバランスが取れています。

- ユーザー本位の機能の構築:プロダクト開発の目的は、ユーザーの生活をより良くすることにあります。
- 技術的投資:システムの保守をするには何をする必要があるのか。つまずくとしたらどこか。リファクタリングや技術的負債の解消は、チームの成功に不可欠なタスクです。
- エスカレーション時や不具合時の可用性:期待どおりにシステムが動かなくなっていませんか。システムが壊れたことで、ユーザーが目的を達成できなくなっていませんか。

これらの3つの優先事項への時間の割り振り方は、プロダクトの成熟度や対象となるユーザー数、修正のしやすさなど、ビジネス固有の要素によって異なります。

システムが大きくなるにつれ、保守に割かれる時間も当然増加します。顧客を重視した作業の割合を維持するには、チームの規模を大きくする必要があります。しばらくしてチームが大きくなりすぎたら、そのときは小さなグループに分割します。いわゆる、Amazonの「チームのメンバー数はピザ2枚分」ルールです。

もちろん、コミュニケーションや調整のコストがかさむため規模の拡大にも限界がありますが、それでもチームはサービスの保守とイノベーションの継続を両立できる人数の開発者を確保することに尽力するべきです。

困難への耐性を付けるには、プロジェクトに友人を招き、コントリビューターのプールを拡大しましょう。ソフトウェアの作成はコラボレーションによるチーム スポーツであることをCircleCIのデータが証明しています。

2020 年を振り返ると、特に重要だったのは、開発チームの燃え尽きに配慮したり共感したりすることでした。他のすべてを犠牲にしてスピードばかりを追い求めることは、この業界ではとても簡単なことです。2020 年にわかったことがあるとすれば、人々が燃え尽きを避けるには、無理のない期待とサポートのもと快適な方法で作業できる必要があるということです。このレポートの分析結果の非常に役立つ点は、メトリクスとパフォーマンスに関する無理のないベンチマークの設定に焦点が当てられていることです。

CircleCI は、1 日あたりの開発件数はパフォーマンスに優れ困難に強いチームを測るメトリクスとして適切ではないと明言しています。もちろん、本番環境へのデプロイが 1 日あたり数百回に上る組織もあります(Netflix が好例です。と言っても、それ自体で 1 つの業界として憧憬の目で見るばかりです)。しかし、ほとんどの組織にとって、これは最重要のメトリクスではありません。

これらのワークフローの実行時間について、よく考えられた期待値を設定することには、大きな価値があります。同僚が今何をしているのか、CircleCI はテレメトリに基づいて判断します。システムの実際の利用状況について、データに基づく視点から考えることは非常に有用です。

このレポートが、開発者や運用担当者への共感という観点からデータの検証を試みている点を非常に好ましいと感じます。2020 年は厳しい年でしたが、人にもっと気を配るという学びは、きっと今後も引き継がれると思います。

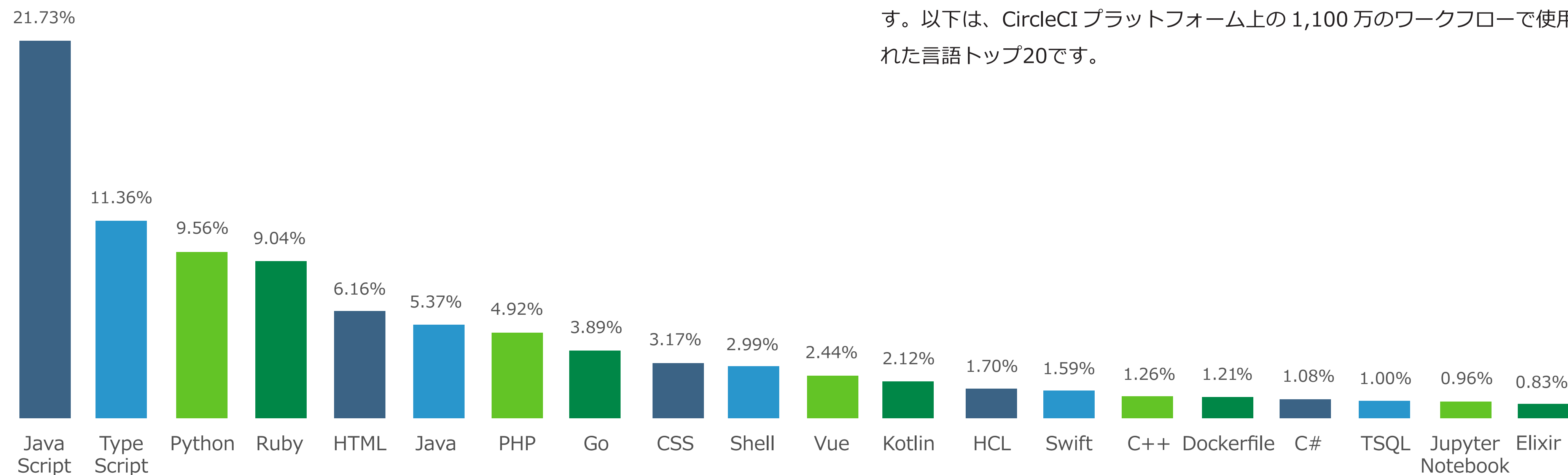
JAMES GOVERNOR 氏

RedMonk 社、アナリスト兼共同創業者

言語の選択に関するこぼれ話

CircleCI 上で最も使用された言語トップ 20

開発チームの目標を設定するうえで重要ではありませんが、言語ごとにワークフローを分けてメトリクスのパフォーマンスの変化を見るのは楽しいものです。以下は、CircleCI プラットフォーム上の 1,100 万のワークフローで使用された言語トップ20です。



メトリクスごとの言語トップ 20

言語のスループット ランキング
50 パーセンタイル

1. Ruby
2. TypeScript
3. Go
4. Python
5. Kotlin
6. Elixir
7. Swift
8. HCL
9. JavaScript
10. TSQL
11. PHP
12. Java
13. C#
14. Jupyter Notebook
15. Shell
16. Vue
17. C++
18. HTML
19. CSS
20. Dockerfile

スループットではRuby とTypeScriptが上位です。開発者が小さな作業単位でCIを頻繁に活用していることがわかります。

言語の成功率ランキング
50 パーセンタイル

1. Vue
2. CSS
3. Shell
4. Dockerfile
5. TSQL
6. HTML
7. HCL
8. Go
9. TypeScript
10. JavaScript
11. Elixir
12. PHP
13. Jupyter Notebook
14. Python
15. Ruby
16. Java
17. Kotlin
18. C#
19. C++
20. Swift

この一覧の上位にある言語での成功率の高さは、おそらくテストが頻繁には実行されていないことを示しています。なぜなら、上位に着けた言語のほとんどはテストの堅牢さで知られているわけではないからです。これらの言語は、大きなプロジェクトの中でアーティファクトや出力を生成する手段になっているようです。続いて Go が高い成功率を収め、その後にほとんどの動的言語がランクインしています。下位にはコンパイラ言語がランクインしました。おそらく、これらの言語には固有のビルドステップとテストステップがあることが原因でしょう。Goはこの点に関しては例外と言えます。

復旧時間の短さランキング
50 パーセンタイル

1. Go
2. JavaScript
3. Elixir
4. HCL
5. Shell
6. Python
7. TypeScript
8. CSS
9. C#
10. HTML
11. Vue
12. Jupyter Notebook
13. Kotlin
14. Java
15. Scala
16. Ruby
17. PHP
18. TSQL
19. Swift
20. C++

Go の開発者が、他の言語の開発者よりもパイプラインをよく監視しています。この分野でも、続いて動的言語のグループが登場し、次にコンパイラ言語がランクインしました。これらの言語は、他より実行時間が長かったりワークフローが複雑だったりするために、復旧時間が長くなる場合があります。

実行時間の短さランキング
50 パーセンタイル

1. Shell
2. HCL
3. CSS
4. HTML
5. Gherkin
6. JavaScript
7. Vue
8. Go
9. Jupyter Notebook
10. Python
11. PHP
12. TypeScript
13. Java
14. Elixir
15. TSQL
16. Kotlin
17. Scala
18. Ruby
19. C++
20. Swift

ここではビルド ステップが少ない言語が最初に実行を終えています。多くのシェル スクリプトは、ただ実行して完了するだけで、堅牢なテストは行いません。アーティファクトをアップロードするだけの場合もあります。JavaScriptとGoは、実行時間の分野でも好成績を収めています。

成果を改善するなら CI がスタート地点

今やすべての企業がテック カンパニーであり、その多くで既に継続的インテグレーションが実施されています。はたして、うまく行っているのでしょうか。CircleCIのデータから言えるのは、開発チームにとって最も重要なメトリクスを向上させるにあたって、CIのエキスパートである必要はないということです。平均的なCircleCIユーザーでも、業界内ではトップクラスのパフォーマンスを発揮するチームになるでしょう。

では、チームに競争力を付けるためにはどうすればよいでしょうか。CircleCIを平均的な使い方で利用するだけでパフォーマンスはトップクラスになりますが、レポートでは、そのような中央値を大きく上回るワークフローも測定し、どのようにその域へ到達したのかを考察しました。

- チームの状況を把握していなければ、現実的な目標を設定することは不可能です。競争力の維持のためには、エンジニアリングの生産性を測定してベースラインを設定できることが絶対に必要です。現在 CircleCI をお使いなら、Insights ダッシュボードにアクセスすることで、すべてのワークフローについて本レポートで挙げた各メトリクスの測定結果を確認できます。
- CircleCI では幅広いマシン タイプやクラス サイズが提供されています。より大きな規模のマシンを選んでワークフローを実行すると、ワークフローの実行時間が短縮される場合があります。
- 速度:CircleCIでは、多彩なコンビニエンスイメージをユーザーに提供しています。これらのイメージはCI向けに最適化されているため、決定論的に優れ、読み込みも迅速です。

- インテリジェントなテスト分割と並列処理のオプションを利用して、堅牢なテスト スイートを大幅な短時間で実行できます。これにより、テストを増やしてもワークフローの実行時間が比例的に増えるようなことはなくなります。

- プラットフォームで高度なキャッシュ オプションを使用すると、ワークフローを最適化したときの実行時間を大幅に削減できます。アプリケーションのビルドに使用する多くのパッケージをキャッシュして再利用できるので、毎回のパッケージのダウンロード時間を削減できます。CircleCIのプレミアム機能であるDockerレイヤーキャッシュを利用すれば、さらにワークフローの実行時間を短縮できます。

- 失敗したビルドのデバッグは、ワークフローが失敗したマシンにアクセスして行うのが一番です。CircleCI には、失敗したワークフローを再実行する機能と、失敗が発生するマシンに SSH を使用してアクセスする機能が備わっています。迅速にシグナルを得られるという特長は、CIフィードバックループの一側面に過ぎません。迅速に復旧できる点もメリットの1つです。

- CircleCIでは、Orbという再利用可能な設定パッケージが用意されています。CI設定ファイルのコードのレイヤーを、コミュニティが作成したオープン ソースのバリデーション済みコンポーネントとして抽象化することで、失敗リスクのないサービスの追加、置き換えを実現しています。よくテストされた設定コンポーネントを使用することで、エラーの原因を減らすことができます。また、Orbを使用してテスト スイートをCI パイプラインに組み込むことで、より多くの情報を実行から得られるようになります。Orb の使用が増えるほど(0個から1個、1個から複数個)、ワークフローの復旧時間は短縮されるという分析結果が出ています。

- プレミアムサポートには、CircleCIのDevOpsエキスパートによる設定レビューのオプションが含まれています。これらのレビューを利用すると、最適化の機会を見つけ出し、ワークフローの実行時間を大幅に短縮したり、他の設定のボトルネックを解消したりできる可能性があります。

> 継続的インテグレーションは新しい概念ではありませんが、テクノロジー中心の組織以外では思うほど広く採用されていません。最新の CI 環境がないチームでは、最新のソフトウェアプロセスやツールの採用に苦労しがちですが、これらの多くはパイプラインにスムーズに適合します。その一例がセキュリティです。セキュリティ テストを CI パイプラインに徐々に移行すれば、開発者は安全なソフトウェアをより迅速に提供することができます。メトリクス ベースのアプローチが有用である理由もここにあります。組織でのCIの採用や利用の拡大に向けたビジネスケースを作成する際に大いに役立つのです。

GARETH RUSHGROVE 氏

Snyk 社、製品管理ディレクター

調査手法

本レポートのデータは、2019 年 8 月、2020 年 3 月、2020 年 4 月、2020 年 5 月、2020 年 8 月の各 1 日を初日とした 30 日間に収集されたものです。VCS として GitHub を使用しているプロジェクトのワークフローのみが反映されるようフィルタリングを施しています。44,000 以上の組織の 16 万以上のプロジェクトから得られた 5,500 万以上のデータ ポイントの分析結果を表しています。

レポートの著者

- Ron Powell (@whyD0My3y3sHurt)
- Michael Stahnke (@stahnma)

レポートの編者

- Emma Webb
- Gillian Jakob Kieser

寄稿者

- RedMonk 社、James Governor 氏
- Pantheon 社、Josh Koenig 氏
- Snyk 社、Gareth Rushgrove 氏
- DataDog 社、Bryan Lee 氏
- The New Stack 社、Jen Riggins 氏

レポートのデザイナー

- Alex Moran

謝辞

- Dawit Gebregziabher
- Melissa Santos