circleci

効果的な DevOps 文化に欠かせないもの

エンジニアリングの

スピードと効率を

向上させるうえで

重要な指標と

ヒント

本ガイドの目的

今日、エンジニアリングチームのリーダーは、次々とコードを配信しなければならないという重圧にさらされています。顧客の要求レベルは高く、企業はこれまでとは異なる新しい方法で進化、成長、適応していく必要があります。継続的インテグレーションおよび継続的デリバリー (CI/CD) などの DevOps の原則を取り入れると、信頼性の高いコードをスピーディに配信できるようになりますが、ツールに頼るだけではその目標を達成できません。CircleCI は何千というチームの皆様を拝見する中で、優れたチームには共通点があると気付きました。それは、DevOps の効果が発揮される文化が醸成されているという点です。このガイドでは、DevOps が息づく文化を育むために必要なことについてご説明します。

以下のトピックについて取り上げます。

- **成果を評価するうえで欠かせない指標**
- **運用の構造化**
- **優れたチームに学ぶベストプラクティス**

> 継続的インテグレーション (CI) と継続的デリバリー (CD) は、
> 文化、運用原則、手法を体系化したものであり、
> アプリケーション開発チームが行うコード変更の
> デリバリー頻度や信頼性を向上させるのに役立ちます。
> これを実践したものを CI/CD パイプラインと呼びます。”

出典: InfoWorld (英語)

エンジニアリングチームはほぼ例外なく、厳しい納期やプレッシャーを経験しているでしょう。タスクのバックログはいつになっても減らず、仕事が山積みで、すべてをやり遂げるにはあまりにも時間が足りません。

そうした理由から、近年では多くのエンジニアリングチームが CI/CD モデルに移行しています。このモデルでは、少ない回数でまとめて更新をローンチするのではなく、継続的に更新をデプロイします。つまり、企画から構築、展開までのプロセスを短縮できるわけです。チームに勢いが付くと、こなせる作業も増えます。世界最大のハンドメイド EC サイトを展開する Etsy の例を見てみましょう。同社のエンジニアリングブログ (英語) によると、コードの開発担当者が配信も担当しているようです。社内で「プッシュトレイン (英語)」と呼ばれるエンジニアたちが連携して更新をリリースしており、その回数は 1日最大 50回にも及びます。

同社で 5年以上にわたりエンジニアを務めている Sasha Friedenberg 氏はこう記しています。「この戦略の勝因はいくつか考えられますが、特に重要なのは、どのデプロイもその変更内容を一番よく知る担当者が行っている点でしょう。最も的確に不具合を見つけ、修正できるのは、コードを書いた本人です。だからこそ、開発者が必要に応じてコードをデプロイし、ロールアウトにも密接にかかわるべきなのです」

そのためには、適切なワークフローとプロセスを整備する必要があります。しかし、そう簡単ではありません。試行錯誤を繰り返す時間もおそらくないでしょう。DevOps とは、開発と運用の統合を意味する比較的新しい手法であり、それに沿った技術ワークフローを確立している企業は必ずしも多くありません。

そこで CircleCI では、エンジニアリングチームのリーダーが試行錯誤しなくてもプロセスをまとめることができるよう、このガイドをご用意しました。第 I 部ではチームにとって有効な指標を解説し、第 II 部では CircleCI のお客様から寄せられたヒントやベストプラクティスを紹介します。

第 I 部

チームの連携状況を測定する方法

CircleCI では、お客様が採用されている開発手法を常に評価・分析し、成功を収めているチームのエンジニアリング文化から共通点を見いだそうと努めています。ある調査では、GitHub と Bitbucket 上のプロジェクト (いずれも CircleCI のクラウドプラットフォームで構築されたもの) をサンプルデータとして、Alexa Internet の世界ランキングと突き合わせて分析を行いました。収集したデータに ClearBit からの情報を加え、Alexa Internet の上位 10 % にランクインした企業とデータ全体を照合して、上位企業に見られる特徴を調べました。その結果、上位企業の対応スピードの速さを浮き彫りにする 3つの指標が明らかになりました。

では、その速度に関する指標を詳しく見ていきましょう。

エンジニアリングの

スピードに関する

指標

メインラインブランチの安定性

定義

メインラインブランチは、開発者が各機能ブランチを作成するときの原型となるものです。CircleCIではプロジェクトのデフォルトブランチがエラー状態であった実測時間を測定し、安定していた割合を評価します。

有効性

メインラインブランチの安定性は、どれだけデプロイ可能な状態であったかを示します。メインラインブランチが安定していなければ、コードを稼働させることはできません。

評価基準

安定性の中央値は 98.5 %、最高で 99.9 % です。

調査対象企業の 8割は、master ブランチを安定させていた時間が 90 % にのぼりました。

デプロイ時間

定義

コードの作成、レビュー、テストを終えても、その後さらにユーザーにデリバリーしなければなりません。コードをメインラインブランチから本番環境へと移す作業は、数分で済む場合もあれば、数時間かかることもあります。

CircleCI では、ビルドがキューに置かれてからデプロイが完了するまでの実測時間 (分単位) でデプロイ時間を測定します。

有効性

デプロイ時間からは、デプロイのコストを評価できます。デプロイ時間が短いほど、製品の変更にかかるコストは抑えられます。

エンジニアにとってはデプロイを待つ無駄な時間が減り、次の作業にすばやく取り掛かれるようになります。プロダクトオーナーはより多くのテストを行い、より多くのプロトタイプを作成できます。

その結果、ユーザーがアップデートを手にするまでの時間が短縮され、バグの発見から数分でパッチが適用されます。

評価基準

80.2 % の企業が 15分以内にデプロイを行っていました。デプロイ時間が短い企業 (上位 5 %) は 2.7分以内、中央値は 7.6分です。

一方、下位 5 % の企業では、デプロイ時間が 30分に到達しています。トップクラスの業績を収めている企業 (Alexa Internet ランキングの上位 10 % の企業) を見てみると、そのうちの 80 % は 17分未満、上位 5 % は 2.6分以内でデプロイしています。

こうした企業におけるデプロイ時間の中央値は 7.9分、下位 5 % のデプロイ時間は 36.1分でした。

デプロイ頻度

定義

デプロイのスピードを把握するための指標です。CircleCI では、1週間のうちにプラットフォーム上でデフォルトブランチのビルドを実行してデプロイまで至った回数の中央値をデプロイ頻度としています。

有効性

この指標によって、どれくらいの速度でリリースを市場に投入している (問題が解決されている) のかがわかります。

評価基準

最も積極的にデプロイしているプロジェクトのデプロイ頻度は、75 % の企業で 1週間あたり 13 回未満でした。業績がトップクラスの企業 (上位 5 %) では、1週間あたりのメインラインブランチのデプロイ回数が 32回にのぼります。これは中央値の 5倍以上、下位 5 % の企業の 24倍近い頻度です。

スピードを追求することも大切ですが、それはまだ全体の半分にすぎません。生み出した価値についても、信頼できる指標によって測定する必要があります。貴社のチームは市場のニーズに確実に対応できているでしょうか。努力が成果につながっているでしょうか。エンジニアリングのスピードに関する指標では、運用プロセスが適切に機能しているかを評価できるのに対し、効率に関する指標では、正しい方向に進んでいるかを確認できます。

エンジニアリングの

効率に関する

指標

コミットからデプロイまでの時間 (CDT)

定義

コードをコミットしてからデプロイするまでにかかる時間を指します。企業によっては、この間にテスト、QA、ステージングを行う場合もあるでしょう。

有効性

継続的インテグレーション (CI) のベストプラクティスを実践していて、自動テストで十分なカバレッジが確保されているという理想的な状況であれば、コミットからデプロイの準備が完了するまでわずか数分、マイクロサービスなら数秒しかかかりません。

主に手動の QA プロセスを採用しているなら、CDT は長引きやすく、改善の余地があります。

評価基準

市場の変化にスピーディに対応している企業は、1日に何百回もデプロイを行っています。

ペースの緩やかなチームでのデプロイ頻度は、1日または1週間に 1回です。

この値は、ビジネスモデルやチーム構成によって大きく異なります。

ビルド時間

定義

テストが完了するまでの間、エンジニアと開発者がただ座って待っている――こんな時間の浪費は、最も避けなくてはなりません。テストの規模が大きく包括的であるほど、時間が長引く傾向にあります。

有効性

2名の開発者がテストの完了を待っていて、どちらにも時給 50ドル (年間 10万ドル弱) を支払っているとしたら、10分のビルド時間で約 17ドル相当の生産性が失われる計算です。

この金額は、2名ともが同様のテストを 1日に 5回実行すれば、1週間で 833ドル、年間 43,000ドルに膨れ上がります。

評価基準

ビルド時間は、テスト内容やチーム構成、企業規模などに左右されます。貴重な時間を有効に使えるよう、この数値はできるだけ低く抑えましょう。

キュー時間

定義

ビルド時間ほどの長さではありませんが、ビルド実行の前にもエンジニアの待ち時間が発生します。キュー時間が長いとコストの増加に直結します。

有効性

その間、エンジニアは別のプロジェクトに取り組むこともできますが、作成したばかりの機能に関する貴重なコンテキストが失われてしまうかもしれません。次のプロジェクトに集中するよりも、変更がテストされるのを待機する方がよいでしょう。

評価基準

キュー時間は、企業の規模と、同時に開発している機能の数によって大きく左右されます。

master のダウンタイム

定義

master でジョブが失敗するたびに累積タイマーを起動します。そして「測定した累積時間 ÷ (年初からその時点までの時間 − 測定した累積時間)」を計算すると、master にエラーがある状態だった時間の比率が算出できます。もっと詳しく把握したいときには、月単位や日単位で計算しましょう。

また、失敗の原因を修正するまでにかかった平均時間を計算する方法もあります。この値が、1時間を超えるようでは問題です。

有効性

継続的デリバリーの原則では、ソフトウェアを常にデプロイ可能な状態に保つことが重視されます。デプロイできない状態なら、そのまま放っておかず、すぐに修正すべきです。

評価基準

master にエラーがある状態が続くと、コミットのボトルネックとなり、修正作業が長期化し、開発の遅れにつながります。

エンジニアリングの間接コスト

定義

エンジニアリングの間接コストには、人件費、ライセンス費用、AWS の料金だけでなく、ツールのメンテナンス費用も含まれます。

有効性

多くの CEO は、シートあたりのツールのコストは確認しているものの、ツールの設定、メンテナンス、モニタリングにかかる時間までは気に留めていません。

評価基準

ツールを利用するために恒常的に多くの時間と労力が費やされているなら、そのツールの価値を見直した方がよいでしょう。ツールそのものに時間が取られると、その分開発作業の時間が奪われてしまいます。

スピードと効率に関する両方の指標を測定すると、エンジニアリングチームがどれだけ効果的に動けているかを評価できます。このとき目標とされるのは、間接費を削減すること、ダウンタイムをなくすこと、できるだけスムーズに作業をこなすことです。ご紹介してきた指標から、開発作業に必要なものがメンバーに十分に提供されているかどうかが見て取れます。

効果的な DevOps 文化とは、エンジニアが最高のパフォーマンスを発揮するために必要なリソース、サポート、ツールが整備されている環境と言えるでしょう。

第 II 部

DevOps のボトルネックを解消し

スピードと効率を高める 18 のアイデア

高い成果を収めている複数のエンジニアリングチームに成功の秘訣を伺ったところ、どの答えにも共通するいくつかのベストプラクティスが見つかりました。そのどれもが、これまでに説明してきたスピードまたは効率に関する指標に直接的に影響するものです。

ただ、DevOps の文化は企業ごとに異なります。新しいプロセスを採用するときには、事前に条件設定を変えながら試してみることが大切です。工夫を凝らして、ボトルネックや課題に取り組みましょう。

これからご紹介するアイデアが、きっと皆様のお役に立つはずです。

1 **あらゆるものをドキュメント化する。**間違いをなくすことはできません。エラーが発生したら理由を解明し、改善可能な点を突き止め、少しずつ改めます。学習と進化を継続しましょう。あらゆるものをドキュメント化すると、気付いたことをミーティングやメールでチームと共有できます。

2 **一度にまとめるのではなく、少しずつ何度も変更する。**確率論で考えてみてください。更新を複数回に分けてリリースすれば、大失敗する確率は下がります。「1つの籠にすべての卵を盛るな」という格言は、デプロイにも当てはまります。

3 **ピアレビューのプロセスを導入する。**開発者同士でペアを組んでエラーを回避します。コードをステージング環境に移す前に、互いの作業をレビューしましょう。

4 **チームの士気に気を配る。**議論がいらない程度の機能のリリースであっても、最新状況を共有しておくことは大切です。何かを成し遂げると有能感を覚えるものです。進んで多くのことに取り組むよう、全員の意欲を高めましょう。

5 **サービスの閑散時にリリースする。**トラブル回避のために、トラフィックが少ない時間帯や、夜間や週末にデプロイすることを検討します。フィーチャーフラグを使用しない運用が必要になることもあります。場合によっては、プレスリリースと実際の提供状況が食い違うかもしれません。しかし、フィーチャーフラグを使用したからと言って、パフォーマンス低下やトラブル発生を避けられるわけではありません。いったん機能が人の目に触れたら、取り消すことはできません。閑散時にリリースするか、トラフィックの多い時間帯にリリースするかは、安全性に関する許容度に応じて判断します。ミスを起こさないチームなどありません。どれだけ才能のある、経験を積んだ開発者でもそれは同じです。

6 **本番環境でテストする。**世界は目まぐるしく変化しています。つまり、ローンチ前にテストを実行するのが必ずしも適切であるとは限りません。本番データをステージング環境に取り込もうと考えている場合にはなおさらです。あえて一部の顧客を対象に機能をローンチ、テストするという方法もあります。精度の高い結果が得やすくなり、問題が生じたときには機能を取り除くことができます。

7 **コードレビューでは高いレベルで議論する。**テストが成功したら、問題が生じるリスクは低減されています。コードレビューをより有意義なものにするために、変更について高いレベルで理解することに時間を割きましょう。アーキテクチャーやコードベースの長期的な方向性について話し合う機会を設けます。そうすれば、メンバー全員が効率、スピード、戦略を重視して取り組むことができます。

8 **簡単なデモを行う。**チームメンバーを集めて新機能をレビューするときには特に、議論の的を絞って短時間で行います。5分以内にするとよいでしょう。複数人で確認すれば、潜んでいるエラーをすばやく発見できます。

9 **ペアで開発する。**本ガイド作成のための調査の対象となった企業の一部では、エンジニアがペアでコーディングをしていました。この手法だと、コードレビューも省略できます。ペアで作業を進めながら、その場で互いにレビューすることで、ミスを防止できます。レビュープロセスをなくすのが不安であれば、必要に応じて妥当性テストを実施することで品質を管理できます。

10 **自己完結したチームを作る。**スムーズに独自のアイデアを試すのに適した人数は 6 ～ 7名です。社内全体で新しい取り組みを実施する前段階として、この人数のチームを作り、明確な方針の範囲内で自由に独自のプロセスを試してみましょう。

11 **定期的に振り返る。**月に 1回 (または特定の間隔で)、失敗したことと成功したことをきちんと評価します。このようなミーティングを行うと、チームメンバーが学び合い、急速な変化に適応し、プロセスを改善しやすくなります。また、ボトルネックを軽減しながらも、勢いを失わずに前進することができます。チームの貢献を称えることも大切です。

12 **責任を押し付け合わない。**DevOps環境ではプレッシャーが大きく、ペースが速くなりがちで、どうしても間違いが起きてしまいます。うまくいかなかったときは、積極的に話し合いましょう。責任の所在を暴き立てるのではなく、問題から学ぶことを重視します。

13 **ユーザーを保護する。**エラーによってユーザーの生活に影響が及ぶことがあります。銀行口座や医療文書を扱うコードをデプロイする場合は、問題の発生に備えて、フェイルセーフな仕組みや手動のリカバリー計画を準備しておく必要があります。これには、バックアップと復元を含めることもあります。

14 **「ボウリングレーンのバンパー」を作る。**会社がトップダウンで指揮をとるのは極端なやり方です。その真逆にしても、メンバー同士の連携が行われずに間違いが起きる可能性があります。それらのバランスをとったものが、共通のワークフローを中心にしてベストプラクティスを浸透させる方法です (master ブランチを常にデプロイ可能な状態にしておく、バグデータベースで既知の問題を管理する、一般向けの Web サイトには SSL 証明書を置く、など)。許容範囲を決めておくことで、開発者は自由にビルド効率を高められ、ボトルネックの発生も回避できます。

15 **ミーティングは少人数で行う。**ミーティングを開くときは、参加するエンジニアの数を、実りある話し合いに十分で、かつ状況を把握できる程度にしましょう。そうすれば、メインラインブランチのエラー状態の原因となる、作業の重複や競合を回避できます。

16 **コードの競合についてコミュニケーションを欠かさない。**技術スタックの各パーツ (バックエンド、フロントエンド、運用など) に応じてチームを構成するやり方は理にかなっているように思えますが、実は欠点があります。別のチームの作業が見えづらく、自分の書いたコードが他の部分にどう影響するかを理解しにくくなるのです。逆に、エンジニアを小規模な機能ごとのグループにまとめれば、チームが無理なく毎日ミーティングを行うことができます (デイリースタンドアップなど)。

17 **エンジニアリングの目標を全社的な優先事項に絡める。**CI/CD だけの範囲で考えるのではなく、エンジニアリング作業を会社全体の目標に関連付けて、機能のリリースやデプロイの優先度を決めます。四半期または月に 1回のペースで、部署をまたがった計画セッションを実施しましょう。なすべき事柄を洗い出し、優先順位を付け、週単位に分けて取り組みます。

18 **ユーザー中心の目標に向けて連携する。**ほんの少しのコード変更も、企業としての大きな目標を達成するための 1つのステップです。たとえば、あるエンジニアリングチームがフードスタンプアプリケーションを担当しているとしましょう。このときの目標は、「人々に食料を届ける」というシンプルなものかもしれません。それが理解できれば、チームは毎日の時間の使い方に優先順位を付けやすくなります。

絶えず

文化を

見直す

DevOps の文化自体も、皆様が作り上げる成果物の 1つです。

自分のチームにどのようなプロセスが適しているかは徐々にわかってくるでしょう。しかしスピードと効率は継続的に追求していかなければなりません。アイデアを提案する機会や、成果につながったもの、つながらなかったものを報告する機会をチームメンバーの全員に提供することが重要です。特にチームが少人数から大所帯へと成長したときには、チーム内で小さなグループを作ると、だれもがプロセスを検証し改善しやすくなります。定期的にミーティングを開くのは有益ですが、ささいなことに気を取られてしまっては台無しです。ユーザーのニーズを常に優先し、より高次元のビジネス目標を念頭に置いて、チームを運営していきましょう。