スタートアップ起業家向け ソフトウェア デリバリー ガイド

すべての成長ステージで スピードと品質の基盤を固める

Rob Zuber (CircleCI CTO) [著]

circleci

2 名構成のチームによるソフトウェア デリバリーと、200 名構成のチームによるソフトウェア デリバリーは、まったくもって異なります。スタートアップ企業から成長を遂げるにつれて、選ぶプロセスやツールは自然に変化します。ただし、あまりに早い段階で最適化したり、目標も定めないまま野放図に変化を続けたりしてしまうと、後で時間やスピードが犠牲になりかねません。だからこそ、私はこのガイドを執筆することにしました。目的意識を持ってデリバリー プロセスを進化させるためにはどうすればいいか、これからご説明しましょう。

ソフトウェア デリバリーに最適なアプローチは、ソフトウェア アーキテクチャによって異なります。コンウェイの法則 (リンク先英語) に従えば、それはすなわち、貴社の組織構造に左右されるということです。この eBook では、企業の規模拡大時に上記の要素がどのような影響力を発揮して、企業の急成長を促すのか、あるいは障害が累積する原因となってしまうのかについて詳しく解説します。すべては、企業が転換点を迎えたときにどのような意思決定を下すかに左右されます。

率直に言って、私は継続的インテグレーション & デリバリー (CI/CD) の概念を強く支持しており、また CI/CD 企業の CTO でもあります。ですから、このガイドの中で私が CI や CD の利用を勧めたとしても、どうか驚かないでください。他にも戦略は数多く存在しますし、CI/CD が役立つ領域もスタートアップ企業の成長につれて移り変わります。それでも、今すぐ CI/CD を採用するべきなのはなぜなのかを、本書で詳しく説明します。

筆者の経歴と経験について

ソフトウェア業界に 20 年間にわたって関わり、スタートアップ企業を 4 回立ち上げ、CTOに3回就任した経験を持つ者として言わせていただければ、スタートアップ企業にとって最も価値あるリソースは "時間" です。早々に誤った選択をしてしまうと、回避できていたはずの問題を修正しなければならなくなり、貴重な時間が無駄になります。適切な選択をすることがいかに大きな利益につながるかを、私は実体験として知っています。また、あらゆる不測の事態に備えようとしてアーキテクチャを過剰に構築することのないように、特に慎重になるべきポイントも把握しています。過去 7 年間、私は CircleCI でエンジニアリング チームを統率しながら、当社のプラットフォームで何千、何万もの組織がソフトウェア デリバリーを合理化し、高速化させる様子を見てきました。疑いなく有効だったアプローチも、そうでなかったアプローチもあります。これからお話しするのは、企業の成長ステージも、さらには業種も問わずに共通するインサイトです。

創業ステージ

エンジニア数 1 〜 10 名

企業の発展過程の中でも、私が特に好きなステージです。士気が高く、アイデアにあふれています。ただ、ツールのことを考えるのは後回しになりがちです。いずれ、初期の環境に対し、20 回目の変更を手動でデプロイする瞬間がやってきます。そのとき、「こんなやり方ではだめだ」と思うかもしれません。その感覚は正解です。たとえそのような運用が可能だったにしても、この時点でこそ CI と CD を導入するべきです。こうした初期段階においては、まだまともにビジネスの舞台に立てていない企業すらあります (実際、まだプロダクトマーケットフィットも叶わないうちから大幅な方針転換を行う企業を私は見てきましたし、指導したこともあります)。しかし、迅速にデリバリーを行う能力と、確かな自信を持ってプロダクトをリリースできる能力があれば、貴社の目標の実現はぐんと近づきます。

試験的なフェーズにおいて求められるのは、シンプルさです。この企業規模であれば調整コストが低いので、それを強みとして生かしましょう。単一チームで、モノリシックなコードベースを構築し、基本的な (自動化された) デプロイを行うのです。

共同創立者の邪魔をして、「Capistrano のコマンドを忘れたから教えてくれる？」や、「今、君のプッシュを上書きでプッシュしてしまったかな？」とは、尋ねたくはないものです。

せっかく手元にノートパソコンがあるのに、だれかがコードをプッシュするためにいちいちあなたの元に来なければならないような "ビルド専用ノートパソコン" にしてしまうのも勿体なさすぎます。

ボタンをワンクリックするだけで、CI/CD は手に入ります。難しいことは何もありません。CircleCI では無料プランをご用意しています。まずは使ってみましょう。この行動は、現時点では貴社にとって "必要" とは思えないかもしれません (必要と感じていただけるように、今から話を進めます)。しかし、たったこれだけの行動で、いずれ他社と大きく差を付けることができるようになります。実際にビジネスを進める中で、予期せぬセキュリティの問題が発生したり、顧客ができて規模の問題に直面したりしたときも、シンプルに修正を配信するだけで済むからです。

「Rob がランチに行ってしまったからデプロイできない」状況よりも、はるかに良い方法だと言えるでしょう。

するべきこと

CI/CD を今すぐ導入する。

アプリケーションには多数の変更が加わるうえ、なるべく早く学びたい事柄がたくさんあります。安全なデプロイ方法を思い出すだけのことに時間を浪費してはいけません。システムの運用が始まり、コードを複雑化させる必要性に迫られるときが来たら (複雑さへの対処を可能な限り後回しにすべき理由は、後ほどご説明します)、その複雑なシステムを取り扱うための手法とツールを導入すればよいでしょう。

シンプルに保つ。

CI と CD の導入は実に簡単。Google Analytics を導入して、CircleCI をセットアップするだけです。このステージなら、これだけで十分でしょう。逆に、これらを準備できていなかったら大きな問題です。このような基本的なツールがなければ、競合他社に大幅に差を付けられてしまいます。起業とは、実行あるのみです。自信を持ってリリースするために必要なツールのみ入手し、他のツールのことは横に置いておくべきです。

起業経験者によるアドバイス

アーキテクチャとツールは可能な限りシンプルに。

複雑さは敵です。スピード、文化、プロダクト リリースのスピードなど、あらゆるものを阻害します。

Boldstart VC 社、プリンシパル、Shomik Ghosh 氏

> 「良質なビルド ツールがあると、チームのプロセスの自動化や、有用な予防策の実践に役立ちます。CircleCI に似たスケーリング可能なツールもあります。ただ、このステージにいるスタートアップ企業は、開発者のワークフローと簡単に連携できるとわかっている最高のツールを使用し、ビジネスをスムーズに進められるようにすることを重視するべきです。リリースと雇用のスピードは、早期段階での成功確度を測る良い目安になります。CircleCI は、開発ツール向けに非常に広く使用されている CI/CD プラットフォームです。使い方が理解しやすく、愛用されているため、このステージにおいて非常に役立ちます。リリースの速度も高速化できます。
>
> 中核的な構成要素は可能な限り自動化しましょう。独自の CI/CD ツールを運用することは、スタートアップ企業の目標と不可分です。それが、ビジネスロジックの基盤になります。 そのような業務には、自動化されたプロセスをできる限り適用してください。後からスケーリングするためには、チーム間での調整がスムーズになるように、あらゆる物事についての真のソースが必要になります。そのときにも CircleCI は役立つでしょう。そして、企業が後期ステージに差しかかったら、多種多様なツールと簡単に連携できて、新規開発者のオンボーディングもスムーズに済ませられるような何かが必要になります。その目的を達成できる製品なら、どれも有用です (ここでも、CircleCI の名前が挙がります)。」

ただ反応するのではなく、考えてみる。

将来的に何が自社の足かせになるもの、あるいはその可能性があるものを把握しておくというのは、その未来に備えて体制を強化せねばならないということではありません。強化にはコストがかかりますが、思考するだけなら無料です。規模拡大の意思決定をする前に、そのロードマップを考えることで、後回しにすることのトレードオフを意識的に考慮できるようになります。

プレースホルダーを使用した実装手法を構築する。

これは、基本的には、複雑なことはまだ行わずに、エンジニアリング部門が抱える複雑な要素に対処するプロセスだけ先に作っておくことを意味します。トレーサビリティ プロセスはその好例です。このプロセスをモノリスでセットアップするのは比較的簡単ですが、アプリケーションをスケーリングして各種サービスを包含した後からでは、実装は難しくなります。なぜ、わざわざトレーサビリティ プロセスを構築するのでしょうか。それは、トレーサビリティがあれば、ある種の動作や判断を行いやすくなるからです。どれだけシンプルな実装であっても、それが存在する以上は、いずれすべてのビルド対象の追跡を検討することになります。早めにこの手法を導入しておくと、コーディングの方法が変わります。コードベース内の非同期的なハンドオフが減る、設計が後から理解しやすいものなるなどの効果があります。

最初から運用性を優先する。

スタートアップ企業に創業段階から所属している開発者は、最初から責任を持ってソフトウェアの運用性を考慮しておく必要があります。これは、コードの 1 行目から明白にすべきです。たとえば、中央システムに出力をリダイレクト可能なロギング ライブラリを選択する行為に意味はありません。それよりも、ロギングを構造化し、ビルド時のデバッグ用に使用していたログ ポイントを削除または整理しましょう。

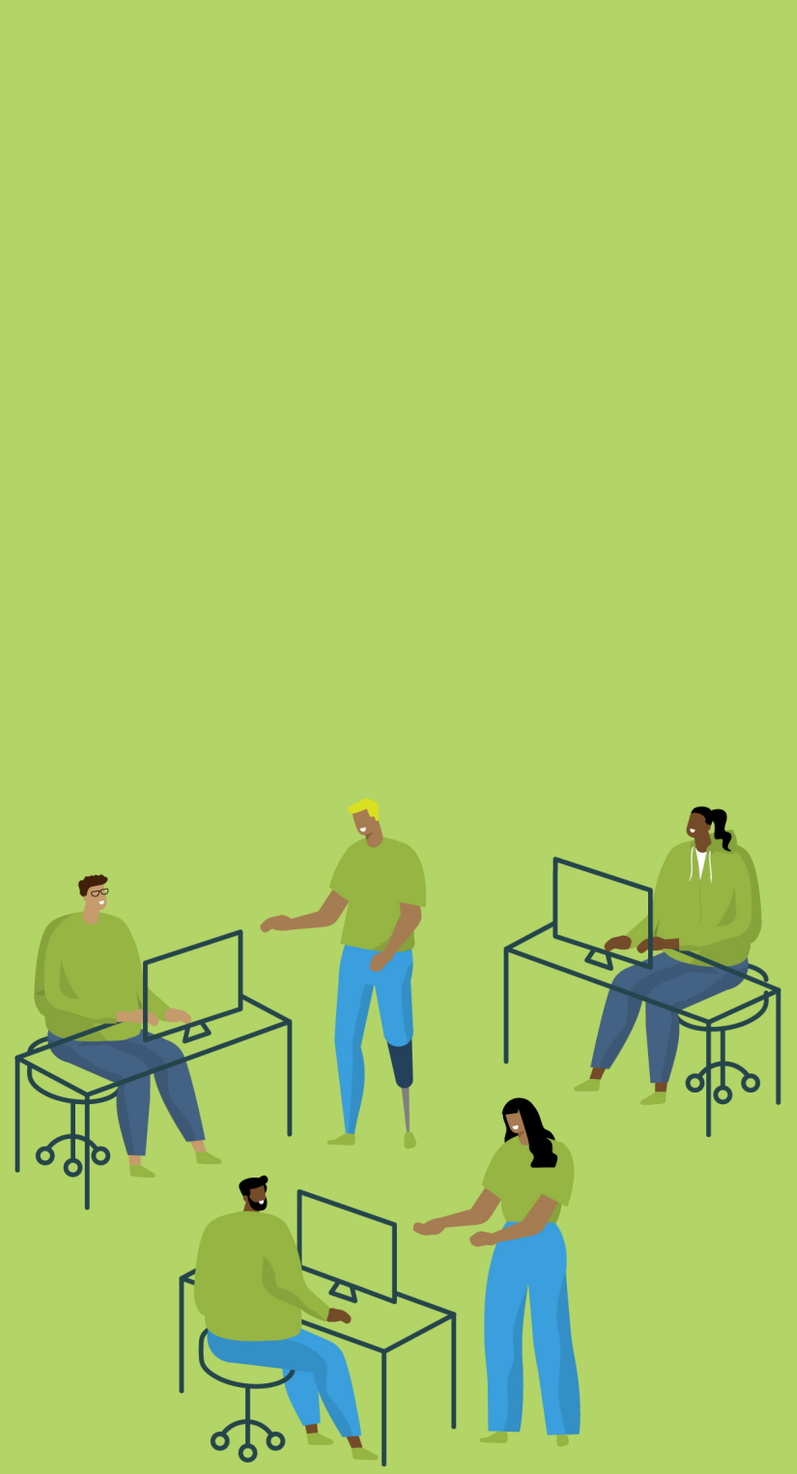
するべきではないこと

大手の手法を真似る。

スタートアップ企業が挑む問題は、Netflix や Google、Facebook とは違うものです。大手各社が力を入れているようなやり方は、貴社のコンテキストにはとても応用できません。そのまま真似るなどは、もっと非現実的です。可能な限り、シンプルさと柔軟性を保ちましょう。いつかは大手と同じ舞台に立つかもしれませんが、今はまだその時ではありません。

最初期ステージ

- エンジニア数 10 〜 20 名
- プロダクトマーケットフィットの発見



この段階の企業では、まだ開発効率や開発者用ツールを専門とする人員はいないでしょう。いたとしても、一般的なケースではありません。多くの場合、社内全員で問題の原因に対応しているはずです。この段階では、デベロッパー エクスペリエンスやツールの専門チームの編成を後回しにする方が、メリットがあります。

なぜなら、このステージの場合、専門チームの編成を後回しにしてこそ、そのマインドセットが企業文化として定着するようになるからです。だれか 1 人に頼りきるのではなく、各自が責任を負うようにした方が、企業の将来性は高まります。チーム間で共有するためのツールやプロセス、コードを作成するのはその後です。

エンジニア数 10 ～ 20 名の規模では、いくつかの手法を導入しつつソフトウェアのシンプルさを維持します。まずはプロダクトマーケットフィットを発見しましょう。それがベンチマークとなります。この段階から、投資を始めることができます。

プロダクトマーケットフィットを発見できるまでは、何をビルドしようと、それを破棄する可能性を解消できません。なるべくシンプルさを維持し、方針転換や破棄を手軽に行えるようにしておきましょう。フィットさえすれば、投資を始めることができるのですから。

このステージは、運用環境の不要な複雑化を防止しながらも、各自が独自に作業できる力を付けていくという、ややぎこちなさの漂う段階です。複数のチームに分割すべきでしょうか？単一のチームとしては人数が多すぎます。しかし、事業の優先順位はまだ流動的です。とはいえ、同じことに全員が取り組むのも煩雑に思えます。

そろそろ各種サービスを構築したくなるかもしれませんが、まだ時期尚早です。デプロイが不必要に複雑化するかもしれませんし、そうなったら後でツケを払うことになります。

それよりも、コンポーネントとライブラリの間に境界線を引き、安定性を高めましょう。ライブラリを使用する場合でも、あるいは適切に決めた境界線をモノリスに引く場合でも (モジュール型モノリスなど)、コードは引き続きまとめてモノリシックにデプロイできます。こうすることで、デプロイのシンプルさを保ちつつ運用の独立性を高めることができます。

コードベースをサービスごとに分割すると、また新しいタイプの意思決定が必要になります。つまり、どのようなサービス間通信を使用するかということです。どのようにサービス ディスカバリを行うべきか？リトライはどうするか？最初のサービスは、多大なコストを要し、複雑化してしまうものです。

ですから、そのことは一旦横に置いておきます。

継続的デリバリー (CD) プロセスはそのままに、継続的インテグレーション (CI) プロセスを変更しましょう。言い換えれば、デプロイ モデルを変えずにビルド モデルを変えるのです。

するべきこと

✔ デプロイのシンプルさを保つ。

ビルドとデプロイはできるだけシンプルに保ってください。サービスを作りたい欲求にはまだ耐えてください。過度な複雑化をもたらすような作業は、後回しにできるなら後回しにします。

✔ 効率性と生産性を維持する。

いわゆる "ピザ 2 枚分" の人数を超えて企業が成長する時期に、このことが重要になります。確固たるチームの定義がなくても個別の作業ストリームを構築できるような、より柔軟な方法を模索します。

起業経験者によるアドバイス

✔ 創業 1 日目から、包括的で思慮の行き届いたテスト カバレッジを確保する。

✔ 創業 1 日目から技術的負債の解消を始める。

Scale Venture Partners 社、パートナー、Andy Vitus 氏

するべきではないこと

✖ マイクロサービス。

コンポーネントやライブラリで代用可能なサービスを構築してしまうと、この規模のチームでは対応しなくてもよいはずの大量のオーバーヘッドが、デプロイにおいて発生してしまいます。後回しにしてください。

起業経験者によるアドバイス

✖ 売上総利益の悪化を気にせず、"後で修正しよう" とする。

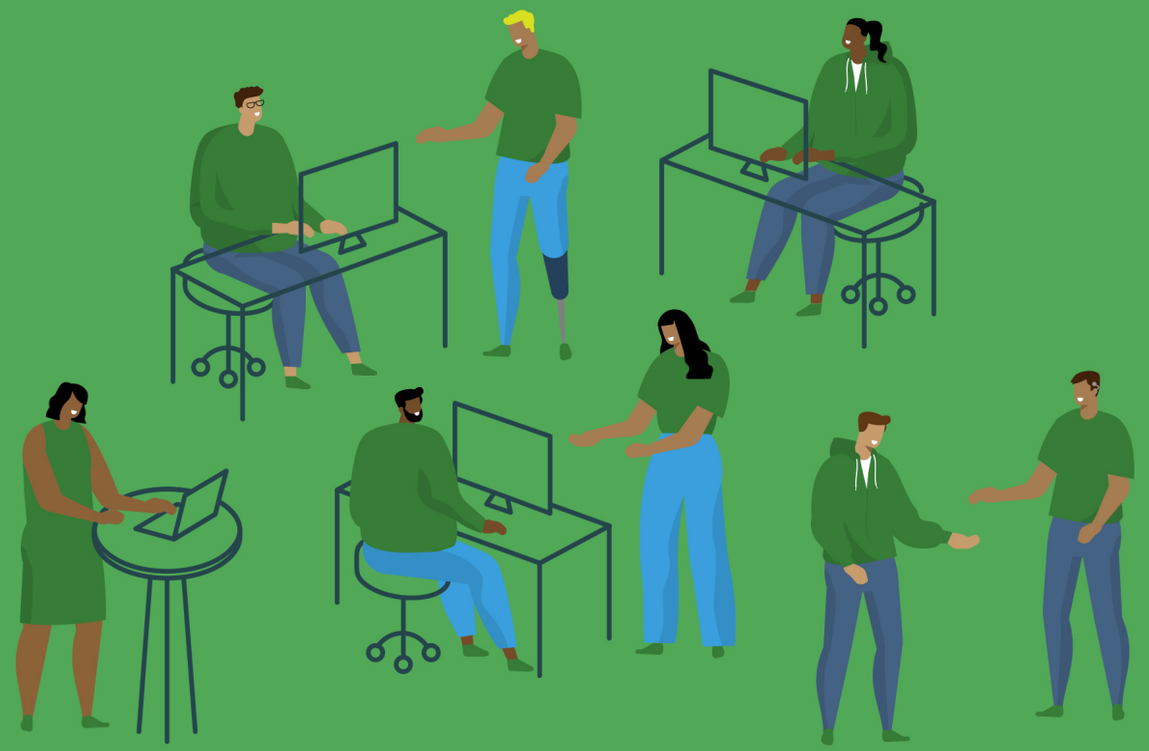
✖ 不必要に複雑化させる。

Scale Venture Partners 社、パートナー、Andy Vitus 氏

20 ～ 50 名

初期ステージ

エンジニア数 20 〜 50 名



この成長ステージでは、きっとプロダクト チームがいくつかできているでしょう。各チームが独立して作業できるように、プロダクトの分割も検討され始めます。この段階に到達すると、コンポーネントや手法をツール間で共有するために行ってきた投資の効果が表れてきます。チーム全体でパターンを共有することのメリットは、今のうちから実装時間を短縮できるということだけではありません。今後発生するメンテナンスの負荷も軽減できます (CircleCI Orb が開発された理由がこれです。チーム間で設定を共有できるという特徴は、このステージから今後にかけての非常に大きな価値となります)。

この時点から、エンジニアごとの多様な言語の好みに合わせたチームの分割を検討し始める方がいるでしょう。また、各開発者がそれぞれベストだと思う言語で作業させるとどんな支障が起こるのか、考えている方もいるかもしれません。

これは望ましくありません。

今は物事をすばやく進められそうな気がしたとしても、明日には失速することでしょう。スピードという概念は、エンジニアの頭の中では、コードの記述量と結びつきやすいものです。一見、すばやく作業できているように思えても (「Rust は知ってる！ばりばり書けるぞ！」)、あなたやエンジニアが書いているのがボイラープレート コードであれば、それはライブラリを使うべきだった証拠に他なりません。

この時点で最優先されるべきなのは、価値の提供であり、
大量のコードの羅列ではありません。

エンジニア数20〜50名のステージは、チームの分裂が始まる段階です。チームの分裂傾向を抑えるために投資を行いましょう。DevOps には、"舗装道路" と呼ばれる概念があります。山に登頂する場合、道を切り開きながら歩いてもよいでしょう。しかし、高速道路で目的地に向かうこともできます。エンジニアに提供するツール (CI/CD ツールを含む) は、良好に舗装された道路の方が選ばれやすくなるものであるべきです。もし組織のメンバーがそちらの道を選んでいないとすれば、目標設定が不適切であると考えられます。

[舗装道路]

事前プロビジョニング済みのツールや共有コンポーネントを提供して、エンジニアが適切な行動を取りやすいようにしましょう。

言い換えると、YouTube でかっこう良い動画を見たからといって Rust を使おうとすると、苦労するだろうということです。

どうやって道のりを"舗装"すればよいかというと、共有コンポーネントを構築してください。言語スタックなら、モノリス コードのままで作業します。CircleCI では、ビルドとテストの済んだ Docker コンテナをご用意しています。最新の CVE (共通脆弱性識別子) への対応が必要な方のために、CircleCI では Clojure パイプラインにこの要件を組み込んでいます。Clojure でのサービス構築に使えるバックエンドの共有構成も揃っています。プロジェクトの開始時点で一般的に行われる作業はすべて完了しています。ただ自分のするべきことを選んで、本番環境に組み入れるだけなのです。セキュリティの更新が発生しても、無料で入手できます。ただし、Go で開発する場合は、こうした要素はすべてゼロから構築することになります。

企業がするべき仕事は、顧客に対して価値を提供するということです。記述したコードの量とも、使った言語とも関係なく、これが常に最低条件となります。チームがこの目標を実現しやすくなるように、環境を整えてください。

するべきこと

✔ 規約とリンターを用意する。

コーディング規約は面白いものではありません。また、コードの記述法 (インデントの使い方やかっこなど) についての話し合いも、無駄な努力になりがちで何の価値も生みません。何らかの規約を選び、準拠を義務付けてください。そして、価値の提供に集中しましょう。

✔ 自社の文化を自動化する。

自社のソフトウェアの仕組みのうち、明確に説明できる要素はすべて CI/CD に組み込みましょう。作業にまつわる意思決定が作業フローに埋め込まれ、自動的に実行されるようになります。自社のルールにいつでも立ち返れるということこそが、創業直後からCI/CDを開始すべきだった理由です。

✔ 必要に応じてプレースホルダーを作成する。

たとえば、無料の脆弱性スキャナーがあるなら、使いましょう。コーディングの規約や手法を企業文化に含めましょう。そして、それらすべてを CI/CD に追加します。こうすれば、将来サービスをアップグレードするときも、有料版を追加するための接続口は既にできていることになります。このステージでなら簡単なことですが、後から追加するのは困難です。

起業経験者によるアドバイス

✔ エンジニアに顧客からのフィードバックを確認させる。

Scale Venture Partners 社、プリンシパル、Eric Anderson 氏

> 「エンジニアリング チームには、なぜ顧客がその機能を求めているのか、その機能が何を解決するのか、理解しておいてもらいたいものです。大規模なプロダクト管理チームであれば、情報を整理し、中核的な原則にチームの目を向けさせ、プロダクト ポートフォリオ全体の一貫性を保つための時間も余裕もあるのですが、市場参入戦略の初期段階ではまだ捻出できないでしょう。プロダクトマーケットフィットを狙いながら、小規模チームの強みを最大限に引き出すには、顧客とエンジニアの間の直接的なコミュニケーションを目指して最適化するのが近道です。緊密なフィードバック ループを維持してください。」

するべきではないこと

✖ ゼロから書き直す。

コードベースを初めて記述したときに比べて、チームとしての知見は増しています。その知見をコードベースに反映させてみたり、苦労して得た知識を元にゼロからコードベースを書き直したりしたくなるかもしれません。その衝動は抑えましょう。コードは複雑化しますが、前進を続けてください。ただ、ビジネスがペースに乗り始めただけです。複雑さを容認し、少しずつ解消していきましょう。絶対に、大幅な書き直しで進捗を止めてはなりません。

✖ コンテキストが共有されていると思いこむ。

人数が 20 名程度だった頃なら、チームは分割されていなかったため、各自がコンテキストを共有していると想定してかまいません (これは戦略ではなく、小ワザです)。これより規模が大きくなったら、全員が同じコンテキストを共有しているとは考えないようにしてください。コンテキストの共有に投資しましょう。これは、ツールで実現できます。たとえば、適切なコメントなしでは CI でのビルドが失敗するように設定してみましょう。なぜそういう設定になっているのか、必ず周知しましょう。こうすることで、だれも読まないドキュメントを際限なく作成するような事態を避けることができます。テストを共有コンテキストの例として使うなら、期待される動作をテスト自体が体現することになります。しかし、過去のテストをじっくりと読み込んでも、なぜそのような動作になったのかは把握できないでしょう。テストを分解しても、ドキュメントやエラー メッセージがなければ、その結果を信頼することすらできません。

中期ステージ

エンジニア数 50 〜 150 名

会社は大きくなりました。さすがにそろそろ、サービスをいくつか構築したい頃合いでしょう。

ソフトウェアはかなり複雑化し、デリバリー速度を向上させるためにパイプラインの数も増えました。作業単位を分割したくなっているかもしれません。慌てて取り掛かって事態をややこしくする前に、デリバリー パイプラインやアプローチに見られるパターンの抽出を行ってください。

サービスの構築に移行する際も、組織の分裂傾向は引き続き可能な限り管理してください。課題は 1 つずつ着手して、適切に対処します。1 つのサービスに集中し、ビルドとデプロイのパイプラインの仕組みを理解しきってください。一度に 10 個ものサービスを見ようとしてはいけません。

一貫性こそが勝負のカギです。多種多様な花を咲かせるべきときではありません。基準を作り、それを複製していきましょう。

パイプライン設計も、ソフトウェア設計と同じように考えます。一度構築してみて、試運転し、また構築しなおして、一部をコピー & ペーストします。どこで失敗しているのかを理解します。3 回ルール (Rule of Three) に則り、3 回使用されたコードは抽象化します。それまでは "完璧な" 抽象化をしようとしても時間の無駄です。なぜなら、まだ理解が不十分だからです。空回りして、間違った方向へ進んでしまうでしょう。

最初から完璧な抽象化をしようとしても、時間の無駄です。

もう 1 つ述べておきます。これまでは、ただ無作為な仲間どうしのやり取りで構造と一貫性は強化されていましたが、このステージからは失敗するようになってきます。他のチームの作業が見えなくなるので、ただチームの内側で効率性を追求するばかりになります。

[3 回ルール]

類似するコードが 2 回登場しただけなら、リファクタリングする必要はありません。3 回目になったら、そのコードを抽出して新たに書き直すべきです。

(Martin Fowler 氏の著書『Refactoring』(邦題: リファクタリング: 既存のコードを安全に改善する) によって広まった概念。発案者は Don Roberts 氏)

デベロッパー エクスペリエンスやツールの専門チームは、まだ編成が完了していないこともあるでしょう。その間には、何らかのソリューションを開発したチームがそれを全チームに広めて共有するようにすれば、ギャップを埋めるのに役立つ場合があります。このアプローチなら、デリバリーを他人事として捉える風潮の発生も回避できます。責任を共有することで、全チームが成果を意識するようになるからです。

するべきこと

デリバリー パイプライン全体の一貫性に投資する。

ここでのコツは、本当の意味での所有権を割り当てながらも、他のチーム メンバーにとっての他人事になってしまわないようなバランスを見つけることです。まず、だれもがソフトウェア デリバリーの効率化を自分の仕事の一部として捉える文化を醸成しましょう。そうしてから、だれかに所有権を与えて、効率化を実現します。そうでないと、だれもが自チーム内で発生した問題しか解決しないようになり、作業が重複し、教訓が共有されなくなってしまいます。

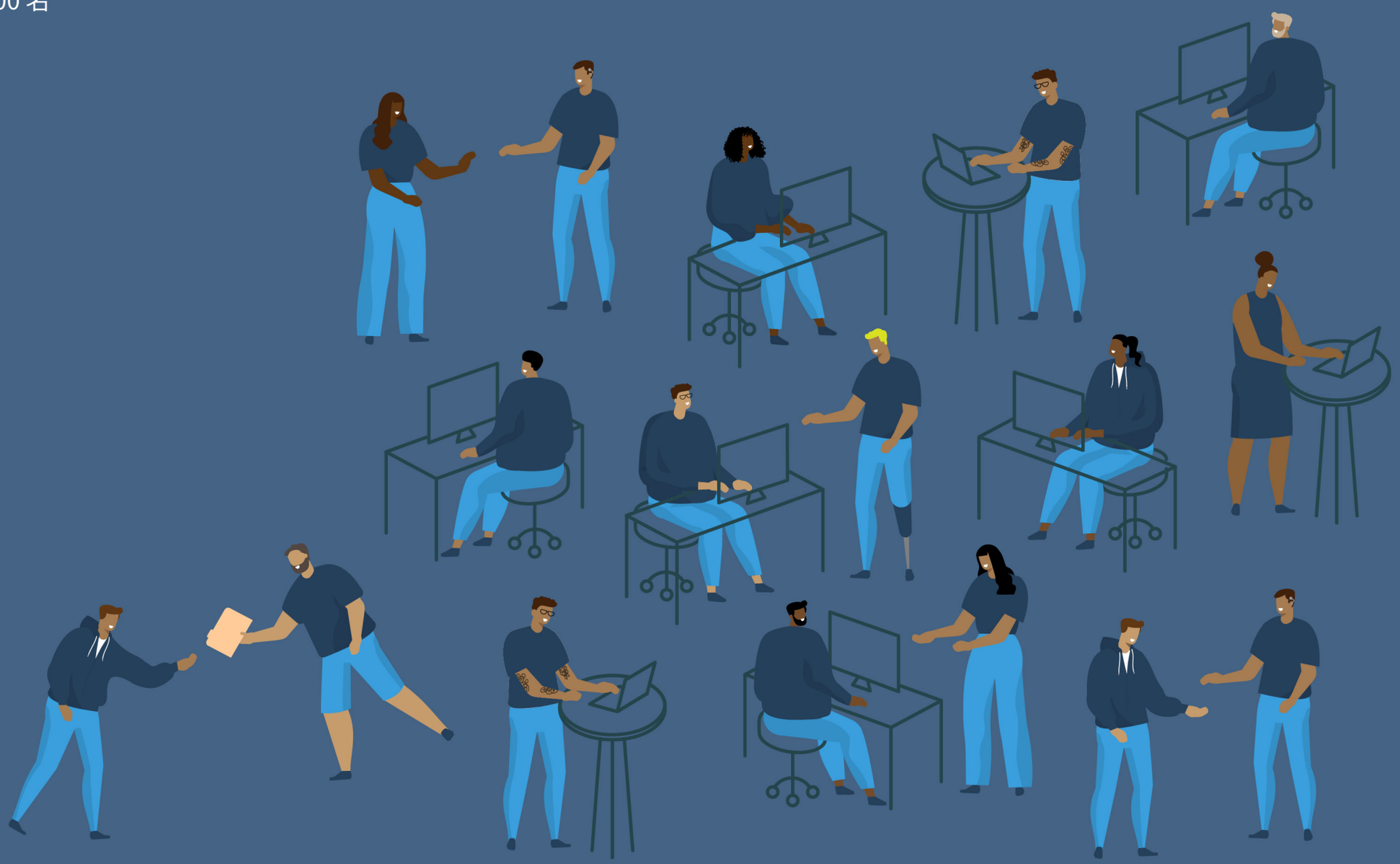
するべきではないこと

チームの自由裁量にまかせる。

全員が自由裁量で作業できるようにすると、スピードが上がったような気になりますが、その感覚はまやかしです。多数のチームが各自だけの効率性を上げようと取り組む事態に陥ると、長期的にはいずれ複雑化のツケを支払うことになります。

成長期ステージ

エンジニア数 150 〜 500 名



作成しておいたプレースホルダーや、プロセスを通じて醸成してきた文化が、実際に機能し、元が取れるようになるステージです。

企業は驚くべきスピードで成長を続けるので、これまでの判断のおかげでハードルの下がったあらゆる要素が非常に大きな強みとなります。最初は、何かをしたいという予感だけがあり、そのための余地を残していました。その後、その余地を無料ツールで置き換え、次に有料ツールへ、人員へ、そしてチーム全体へと移行が行われました。このようなことができたのも、すべてはそのための余地を思慮深く残しておいたからです。このステージでは既に、エンジニアリング組織向けツールの選定に責任を負う、複数のチームの一群が存在していることでしょう(SRE、デベロッパー エクスペリエンス、リリース エンジニアリングなどの担当チーム)。今こそ、真の転換点です。

> このとき、あまり独断的になりすぎないように注意してください。一貫性によって最大限に価値を高められる領域と、価値よりもオーバーヘッドの方が大きくなる領域を見極めましょう。一貫性の確保と、チーム メンバーの自由裁量のバランスを取ってください。

チームの中でも最も使用頻度の高いプロジェクトに、共有ツールのエキスパートを参加させましょう。誰かが同じことを2回繰り返すたびにこのエキスパートを配置する必要はありません。

いずれ、一貫性の価値よりもオーバーヘッドの方が上回る瞬間が来ます。どこでその状態を容認するか、それが大きな問題です。常にトレードオフを意識しましょう。

個人的な関係ではなく、システムとプロセスに投資する。

ただ目を通すだけで何が起こっているかを理解できた段階は過ぎ去りました。今や、人員ではなくシステムによって事業が回っています。

個人的な関係からシステムへの移行を進めながら、運用ツールがどう役立つのかを明確にしていきます。短期間で、何が起こっているのかについて、より適切なフィードバックが得られるようになるはずです。もちろん、そのツールを使うためのビジネス ケースの作成は簡単ではありません。しかし、この作業もこれまでと同じように、早期に対応しておけば非常に安く済みますが、後回しにするほどコストは高くなります。小さく始めて、毎日少しずつ作業を進めます。立ち止まらずに進んでください。

システムのどこで何が行われていようと、あるいはだれとやり取りしていようと、変わらずにサービスをデプロイできるよう、チームの境界を取り払うことが非常に重要です。自由裁量の本当の価値が生まれてくるのはこのときです。つまり、特定の人にやり方を相談しなくても、作業の完遂に向かって集中できるようになっています。

起業経験者によるアドバイス

技術的負債への対処と解消のための時間を作る。

Sapphire Ventures 社、社長兼マネージング ディレクター、Jai Das 氏

> 「技術的負債は避けようがありません。その負債に対応しないままでは、やがて負債どうしが絡み合い、もっと大きな問題になります。技術的負債の発生パターンは、通常、いくつか存在します。アーキテクチャや技術に関する負債は、どんなソフトウェアであっても作成から 4 ～ 5 年後には発生します。クラウドネイティブ テクノロジーの急速な変化により、レガシー テクノロジーを使用したソフトウェアにもアーキテクチャの不備が発生し、拡張やスケーリングが難しくなります。ソフトウェアには、時間の経過と共にコードの負債が溜まっていきます。開発者が、明確に定義されたインターフェイスや API を使い綺麗なコードを書いてくれるわけではないからです。また、ソフトウェアにはテストとドキュメントの負債も累積します。コードを適切にドキュメント化したり、十分なカバレッジのあるテストを書いたりできる開発者はごくわずかです。このように技術的負債は避けがたいものなので、能動的に修正に取り組む方が最適なアプローチだと言えます。ソフトウェア チームを率いる優秀なリーダーのほとんどが、4～5年ごとに6か月ほど時間を取って、一切の新機能の追加を停止し、技術的負債の解消に注力しています。」

するべきではないこと

✖ 功労者に褒賞を与える。

もう、問題が起きてもエキスパートに修正してもらうべきではありません。この時点で問題に対応するためのツールがないとしたら、問題です。問題に対処するための良質なドキュメント、使いやすいツール、明確なプロセスが既に存在するはずです。後手の対応はもうしてはなりません。堅牢なインフラストラクチャと運用ツールが必要なのです。チームで何かをするとき、個人的な関係性に頼らないようにしましょう。

✖ すべてを修正しようとする。

今や、1 つのシステムに 150 名以上のエンジニアがかかわっています。コードベースの中には、エンジニアがまだ 10 名規模だった頃から手が加えられていない部分もあります。一方で、毎日編集されている部分もあります。まず注意を払うべきはこちらの方です。レガシー コード (レジェンドとも呼んでもよい) は、改善するどころか壊してしまう可能性の方が高いからです。毎日手が加えられている箇所があるなら、そこを最初に修正しましょう。

最後に

成長中も革新をし続ける

エンジニア数が 500 名を超えたら、もはやスタートアップ企業ではありません。おめでとうございます！すばらしい発展を遂げていますね。ここまで到達できたのは、優秀なソフトウェア デリバリー組織になろうと研鑽してきたからこそです。その努力は続けていきましょう。

貴社より大きな他のソフトウェア組織も、例外なく貴社と同じ運用体制を目指しています。

貴社より大きな他のどのソフトウェア組織も、目指すところは貴社と同様の運用体制です。ですから、他社の鈍足なレガシー プロセスに巻き込まれないように注意し、他社のモデルであり続けてください。

企業が成長を続けるにつれ、実装は変化しますが、これまでに役立ったアプローチを忘れてはなりません。せっかく築いた CI/CD 文化をなくしてはなりません。たしかに、システムとプロセスは今後も増やすことになります。しかし、これまでも新しいツールやプロセスに対してそうしてきたように、貴社にはもう、鋭い目で導入効果を見極め、スピードを実現する力が付いています。

組織が拡大を続けるにつれ、監査人など、新たな関係者からの要望も発生するようになります。彼らがどんな結果を求めているのか理解する必要はありますが、その結果を得るための提案まで鵜呑みにする必要はありません。ソフトウェア開発の最先端に立っているのは、彼らではなく、これまでの成長過程をすべて経て成功したスタートアップ、つまり貴社だからです。貴社が優秀なソフトウェア チームであり続けられるようなツールを作成してください。大規模チームとしての運用に悩み、立ち止まってしまわないようにしましょう。

最後までお読みいただき、ありがとうございました。このガイドでご紹介したマインドセットの、たとえ一部でも実践できれば、世界トップクラスのソフトウェア デリバリー組織になれるはずです。そうなったあかつきには、ぜひ大きく胸を張っていただければと思います。