Don’t build it.

つくってはいけない

シビックテックの実践者／エンジニアのためのガイド

このガイドは、プロジェクトの失敗を回避し、適切なチームを構成して、より早くプロジェクトを公開して改善を続け、長く成長させることを目的としています。

Luke Jordan / Grassroot・MITガバナンスラボ / **2021年**

GRASSROOT

MITGOV/LAB

引用文献
Luke Jordan。2021年。「Don’t Build It -- A Guide For Practitioners In Civic Tech / Tech For Development」Grassroot（南アフリカ）、MITガバナンスラボ（アメリカ）。

About

Luke Jordanについて
南アフリカを拠点とするシビックテックの組織Grassrootの創設者兼CEO/CTOであり、MITガバナンスラボの2021年度プラクティショナー・イン・レジデンスでもあります。
連絡先: luke.jordan@grassroot.org.za

Grassroot

Grassrootは、低帯域幅、低速度環境向けに構築された技術プラットフォームであり、テキストメッセージによるスマートメッセージングを可能にしています。（www.grassroot.org.za）

MIT Gov/Lab

MITガバナンスラボ（MIT GOV/LAB）は、市民参加と政府の対応の革新に焦点を当てた政治学者の団体です。（www.mitgovlab.org）

Thanks

Katlego Mohlabane、Busani Ndlovu、Mbalenhle Nkosi、March Ratsela、Zinhle Miya、Paballo Ditshegoをはじめとするグラスルートのチームの皆さん、ガイドの作成にご協力いただきありがとうございました。また、Tiago Peixoto、Arjun Khoosal、Lily L. Tsai、Alisa Zomer、Maggie Biroscakには、このガイドへ様々なご意見をいただき感謝いたします。Susy Tort、Gabriela Reyagadas、イラストとグラフィックデザインの制作ありがとうございました。

序文

このガイドは、人々がより効果的に政府に関わり、テクノロジーの検討や構築に携わる「シビックテック」のチームやマネージャーのためのものです。南アフリカのシビックテック・プラットフォームGrassrootの構築に4年間携わった私の経験を凝縮しました。

このガイドは実用性を重視しています。長年にわたって多くの人からアドバイスを求められたこと、始めたときに知っていればよかったこと、初期のアドバイスで最も価値があったこと、そして途中でうまくいかなかったことなどを振り返ってテーマを選びました。

ソフトウェア開発には、つくってはいけないものを作らないようにするためのルールはありません。組織やリーダーシップチームが独自の「つくらない」予防策を策定する必要があります。
しかし、自分の周りで人々が“魅力的で新しいアプリ”を作っているふりをしていたり「社会をよくするテクノロジーだ」という記事が次々と発表されていると、「つくらない」という判断をすることはとても難しいことです。「アプリを作ろうよ！」と善意はあるがテクノロジーを活用した厳密ではないプロジェクトに対して、限られた効果があったと指摘した研究報告がこの5年間で次々と発表されてきました。それでも、今も新しいアプリは作られ続けています[1]。

そこで、私がこのガイドを書いた理由を説明させてください。私は、テクノロジーによって一般の人々が力をつけ、国がもっと市民に説明責任を果たし、レスポンスを高めることができると信じています。適切なタイミングで適切な問題に取り組めば、テクノロジーが大きな変化をもたらすことができるのだと確信しています。ただ現代の思想、資金、地位などの力によって、構築するすべを知らないチームが、構築すべきでないものを構築しようとするところを間近で見てきました。読者の皆さんは、このガイドが典型的な学術の手引きのようではないことに気づくと思います。—私のアプローチは、率直かつ現実的にプロセスを説明することで、「アプリを作る」ということが何を必要としているのか、どうすればうまくできるのか、あるいは、そもそも作らないで済むのか、ということをより理解してもらうことを目的としています。

1　数ある中でもこの研究報告が参考になります。https://opendocs.ids.ac.uk/opendocs/bitstream/handle/20.500.12413/13452/RR_Synth_Online_final.pdf.

Don’t build it

このガイドでは、プロジェクトの選択、なぜプロジェクトを選択しないことが最善なのか、から始まります。次に、チームの構成、フルタイムの上級テックリード（または、優れたエンジニアリングマネージャーとして理解されている最高技術責任者（CTO））が極めて重要であることを説明します。
次にプロジェクトの工程表について説明し、早期にプロダクトをローンチし成熟するまでには大きな忍耐が必要であることを強調し、特にプロダクトの適合性を見極めること、バニティメトリクス（訳注：計測しても業務の改善に繋げられない指標。「虚栄の指標」ともいう。）を避けることについて述べます。さらに、CTO、上級エンジニア、デザイナー、若手エンジニアなどの採用についても詳しく説明します。

このガイドの中で、最も長いセクションは「採用」です。「採用」は、創業者や投資家、経営者など、あらゆる分野のあらゆる教えの中で重要視されているものです。またこの分野は、私が詳しいと自負している分野でもあり、読者の役に立つと信じています。

要約

この部分だけでも読んでください。

作らないで済むなら作らない方がいいです。どうしても作らなければならない場合は、CTOを雇い、早くローンチして長く成長させましょう。そしてそれができない場合あるいはできるとしても、信頼できるスタッフを集め、無駄なく迅速に作り、できるだけ早くユーザーに提供し、ユーザーと一緒に作り上げることです。

Don’t build it

IF YOU HAVE TO:

- *外注は戦術としては素晴らしいですが、戦略としては最悪です。*
- *フルタイム採用は、彼らをチームに留めておくことができるので、時間をかけて成長するスキルに投資できますが、コストレベルで考えて慎重に追加します。*
- *専任のコミュニティ・エンゲージメント・チームが、ユーザーとの距離を迅速に縮めます。*
- *適応性と学習の速さは、どの立場においても中核となる基準です。*
- *予算を設定することで、迅速なスタートを切ることができ、かつ時間をかけて開発サイクルを繰り返していくことができます。*

この種のガイドは一般的に、人々が自分の首を絞めるような具体的な内容は言いません。ソフトウェア開発のルールは「必要なだけ時間をかける」ことだと言われています。私はこの2つのルールを破り、3〜4人の優秀なチームがベストプラクティスに従えば、中程度の複雑さの製品を約3カ月でローンチし、12カ月以内にある程度の安定性と成熟度を得ることができるはずだと思っています。もちろん、その後もユーザーがいればリスタートができます。ユーザーがいない場合は、6ヶ月で止めるか、より良いのは4ヶ月でやめることです。そして最善な方法は、**つくらないこと**です。

技術的な原則：オープンソースにこだわり、人気のある言語を選びましょう。

早期の学習ほど重要なものはありませんが、バニティメトリクスはそのような学習を阻害します。

このプロジェクトは本当に必要ですか？

↳ たぶん、いらないです

ソフトウェアの問題の中心は、何でも作れるということです。

ソフトウェアの問題の中心は、何でも作れる、ということです。物理的な構造物は、自然や物理学がアイデアの空間にある程度の制約を与えます。ソフトウェアの場合は、プロジェクトが完全に完成して導入されてから、根本的な欠陥があることが明らかになります。そうなると私たちは皆、できの悪い技術に慣れてしまい、誰も気づくことができません。一方、人里離れた土地に**巨大なビルを建設する**場合は、その酷評は映画化されるかもしれません。誰も使わない無意味なアプリを作っても、資金提供者向けのレポートに何かの指標を引用して誤魔化せば、誰も気にしないでしょう。逆にセンスの良い場所に良い建物を建てても、誰も注目してくれないということです。プレスリリースで話題になるくらいの「まずまずな」アプリを作って、少しでも長く人々に使ってもらえれば、すぐに「Xアンダーx」リストにノミネートされるでしょう。

ですから、圧倒的に良い方法は、シンプルな原則を採用することです。

Don’t build it.

誰かが

「そのために作るべきだ」と言っても、

ただ「ノー」と言ってください。

投資家や寄付者が

「なぜ作らないのか」と言っても、

ただ「ノー」と言ってください。

誰かが自分のアプリで何かを達成したかのように見せかけて、検証されていない無意味な数字を引用している記事やTEDxトークを目にした時、

心の声が「作ってみようかな…」とささやいても、

ただ「ノー」と言ってください。

ということは、このガイドの残りの部分は無意味でしょうか？

そうあってほしいですね。しかし実際には、あるアイデアが勢いを増したり、説得力を増して「つくらない」という倫理観が揺らぎ始めることがあります。その時は、次の質問をしてみてください。

その技術でできるはずのことを、すでに誰かがやろうとしていませんか？

もし、誰かが作っている場合、彼らは今どのようにやろうとしていますか？そしてなぜそれがうまくいかないのでしょうか？既存の挫折している試みに、なぜテクノロジーが変化をもたらすと考えるのでしょうか？

もし、誰も作っていない場合、なぜテクノロジーがあることで違いが生まれるのでしょうか？Xをしたくなかった人が、Xをするための技術があるからといって、なぜXをしたくなるのでしょうか？

もちろん、このような質問に答えて、何かを作ることを正当化するような嘘の答えを出すことは簡単ですし、たいていはそうなるでしょう。しかし、誰かが嘘をつかずに正直に答えたとして、人々がたとえばその技術を使って何かをしようとしても、根本的な問題のためにうまくいきません。

すぐに技術を使って、より深い問題に取り組もうとする強い誘惑に駆られるでしょう。しかし、繰り返しになりますが、**「つくらない」**でください！まず、あなたが見つけた根本的な問題について、上記の質問をしてみてください。たとえば、ある現象（暴力やサービスの停止）を政府に知らせるためのソフトウェアを作る前に、このことを伝えようとしている人がすでにいるのでは？もし、政府に無視されているのなら、なぜ無視されているのでしょうか？もしそれが権力の不均衡によるものだとしたら、あなたの善意の警告アプリは本当に何かをしてくれるのでしょうか？あるいは、権力者が自分自身や他のステークホルダーに対して何かをしているように見せかけることで、無意味な、あるいは問題を悪化させることにならないでしょうか？誰も政府の会議に出席しないのは、会議そのものを知らないからなのか、それとも出席しても誰も聞いてくれないからでしょうか？このような例は数え上げればきりがありません。

Don’t build it

このように何も作らないでいるうちに、次のような問題が明らかになります。
a) 技術的に可能なことは何でもやろうとしているが、うまくいっていない。
b) 技術で解決できる問題だと判明しても、結果を出せていない。
c) その根拠が明確で確実である。
となると、そろそろ何か作ってみようかなという気になってきたかもしれません。

では、作らなければならない場合、誰が作るのでしょうか？

↳ *プロジェクトとチームの構成*

上記のステップを経て、本当に価値のあるプロジェクトを見つけたという前提で話を進めましょう。次の問いは、プロジェクトの設計、コーディング、デプロイ、監視、開発サイクルを繰り返し行う人材を、どこで見つけ、どのようにチームを構成するかです。

技術力のない多くの組織が最初に思いつくのは、ほとんどの作業を外注しようとすることでしょう。外注は、戦術としては優れていますが、戦略としては最悪です。つまり、経験豊富で集中力のある上級エンジニアやデザイナーに個々の作業を外注すれば、莫大な利益を得ることができます。彼らは非常に柔軟性があり、予算を消費することなく、余った時間にプロジェクトに出入りすることができます。近年のハイレベルなフリーランサー・プラットフォームでは、特定のニーズに合わせて人材を見つけることができます。一方、外注は、どの業者に依頼するかという一点でプロジェクト全体の成否がほぼ決まってしまい、行き詰まります。一人のフリーランサーがうまくいかなければ、新しい人を探し、仕事のほとんどを中断することなく入れ替えることで、チームに迷惑をかけずに済みます。しかし、外注全体がうまくいかなければ、チーム全体を入れ替えなければならず、契約違反で法的な紛争に直面する可能性もあります。

外注は、戦術としては優れていますが、戦略としては最悪です。

Don’t build it

というのも、有能な最高技術責任者（CTO）がいないと、契約先の選定に失敗する可能性があるからです（詳しくは後述します）。提案書の技術的なメリットを判断したり、プロジェクトを抽象的な技術要素に分解したりする能力がないと、要件仕様を過小にも過剰にも設定してしまいます。過小に捉えてしまうと、契約上の救済措置がないため、最低の納品になります。過剰な場合は、契約上の救済措置という偽りの約束をしても、誰もあなたが作ったものに適応することができないので、さらに悪い結果を招くことになります。

しかし、提案依頼書（RFP）やその他の入札書類の主要な作成者が技術的な知識を持っていないことは、経験者が見れば明らかです。そのようなRFPを探すコンサルタントの仕事がありますが、彼らは、あなたが構築を避けられないと信じるプロジェクトを作る人ではありません（プロジェクトが選択肢の一つであるならば、あなたは作るべきではないからです）。

RFPを出した後、技術的なネットワークがない中で、どうやって業者を探すのでしょうか？また、業者が入札したら、選定時と初期のマイルストーンの両方で、あなたはどのように評価しますか？技術仕様書に変更を加えてくれた親切なエンジニアが、本当にやってくれるのでしょうか？もし、そうでなければ、ポートフォリオや専門知識に基づいて選択するしかないでしょう。技術力のない組織が、技術的に問題のあるエンジニアを雇うことはよくありますが、技術的な質問ができないまま、ポートフォリオを見て採用するのは、ランダムに選択するより悪手です。

なぜなら、セールスマンシップを選択することになるからです。つまり、どちらかといえば、技術的能力と逆相関しています。

たまに、どこかの組織が運良く、限られたプロジェクトで完璧な業者を見つけて成功することがあります。しかし、非常に幸運な場合を除いて、失敗する戦略は多かれ少なかれ、悲惨な戦略となります。

最高技術責任者（CTO）

プロジェクト間で、CTOの費用をどうやって負担するか悩んでいる組織もあるでしょう。そんな心配がある場合は、**プロジェクトを始めないでください**。あなたの組織は、優れたプロダクトを維持する準備ができていないのです。このような心配をするということは、技術が一度限りのプロジェクトとして構築され、公開された時点で完成し、反復し改善するための継続的な作業を必要としないと想定していることになります。もしそうであれば、問題が理解されていないか（振り出しに戻る）、技術自体が理解されていないかのいずれかであり、プロジェクトは最初から絶望的であると言えます。

もちろん、時間とともにリソースは変化するでしょう。しかし、有用なプロジェクトを開始してから少なくとも数年間、そのうちの半分以上の時間（多くの場合はそれ以上）は、優秀なCTOが必要です。子犬を飼ったのに、誰も散歩や餌やりをしないのと同じです。プロダクトを開発する際には、そのプロダクトが成長したときに面倒を見ることができる有能なCTOがいなければ、そのプロジェクトに着手するべきではありません。

プロジェクトが成熟してCTOに余力がある場合は、例えば、社内の技術アップグレード（パンデミックの際に多くの組織が数年前にやっておくべきだったと気づいたようなもの）に取り組んだり、他のパートナーとコストを分担して共同開発を行ったりすることができます。多くの組織では、熟練した質の高い技術マネージャーが必要です。

フルタイムエンジニア

チームの残りについては、プロダクトに合わせて構成します。フルタイムの人材は、彼らをチームに留めておくことができるので、時間をかけてスキルの成長に投資できますが、コストを考慮して慎重に追加します。特殊な技術で価値の高い短期間のプロジェクトに、優秀な人材を投入できるよう、十分な予算を残しておきます。例えば、優秀な若手のフロントエンドエンジニアと、経験豊富な上級フロントエンドエンジニアを1ヶ月ほど組み合わせて、コードの構造と最初のコアコンポーネントを一緒に構築し、残りの部分は、若手エンジニアを指導しながら構築を進めさせることができます。

たとえ若手であっても、フルタイムの人材を長期にわたって確保することは、投資に見合う価値があります。組織は知識を内面化し、長期にわたって保持することができます。上級エンジニアはその分野に関する深い知識を、社内のエンジニアはシステムに関する深い知識を持っています。医師を置かない、あるいは医師だけに頼ろうとするクリニックは失敗するように、どちらか一方では上手くいかないでしょう。

いろいろな組み合わせ方がありますが、基本は継続性を重視し、短期間でも重要なところでは高度な技術を適用し、余裕を持って行動することです。

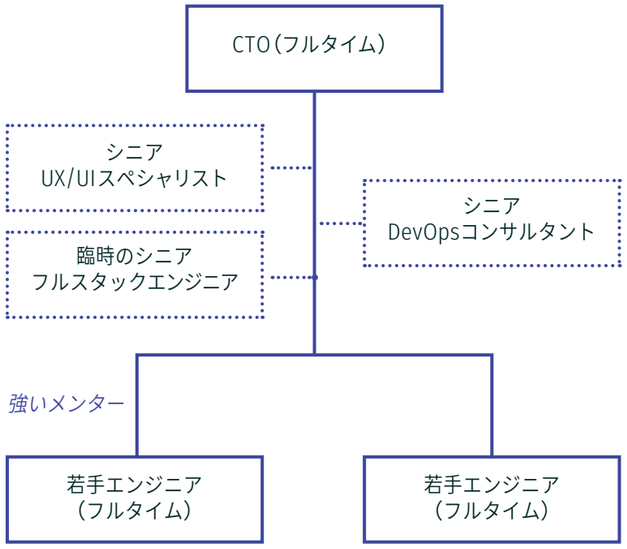
フルタイムの人材は、彼らをチームに留めておくことができるので、時間をかけてスキルの成長に投資できますが、コストを考慮して慎重に追加します。

DEFINITIONS

スタック：
製品やプロジェクトを構成する技術の組み合わせのこと。(比喩的に言えば、便利なものや楽しいものを作るために互いに積み重ねられたもの)

例えば、優れたチーム構成とは、まだ未熟な若手エンジニア、スタックするような最も重要な部分に精通した優秀なテクニカルリード、スタックの特定の部分に特化したシニアのフリーランサーが数週間または数日単位でプロジェクトに参加し、必要に応じて緊急支援のための予算を残しておく、といったものです。同様に、ひどいチーム構成の場合は、スタックの間違った部分を経験した3人の中堅ゼネラリストで構成され、余力が残っていないことです（図1）。共通しているのは、どのようにチームを構成しても、CTOがいなければ、プロジェクトは失敗に終わるということです。

優れたチーム構成



ひどいチーム構成

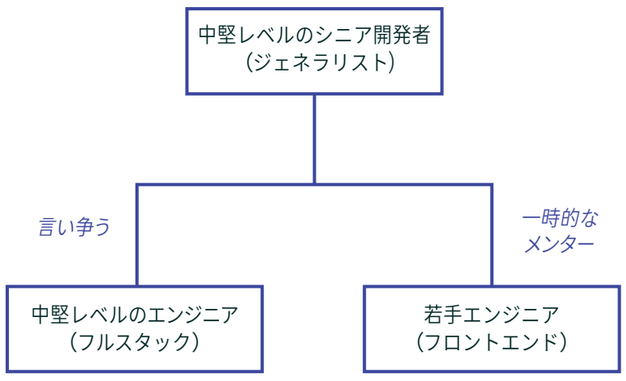


図 1 →

エンジニア、ユーザーエクスペリエンス、現場スタッフをつなぐ

チーム構成についての最後の注意点は、テクノロジーを構築できる人材が必要なのはもちろんですが、そのテクノロジーが使われる背景、つまりユーザーがいる国、コミュニティ、組織、人口統計などを理解している人材も必要だということです。

その一部は、熟練したUXデザイナーによる「ユーザーエクスペリエンス（UX）」の技術によって構築されるべきです（下記の「採用」のセクションを参照）。将来的には、フルタイムのUXスペシャリストを雇うことができるかもしれません。しかし、特に最初の段階では、UXスペシャリストをフルタイムで雇うことを正当化できるほどの需要があるとは思えませんし、チームメンバーとして毎回UXスペシャリストを雇えるほどの予算もないでしょう。現場のチームは、機能がどのように作られているかを理解すればするほど（理想的には、彼ら自身がUX技術を習得していれば）、開発チームにとって最も役に立つ自分たちの観察結果を擦り合わせることができます。また、エンジニアリングチームが現場で起こっていることを理解すればするほど、新しいアイデアや自分たちが作ったものの別の方向性を提案できる可能性が高くなります（あるいは健全な考え直しができるようになります）。

とはいえ、特に上級職の採用においては、コンテキストを堅苦しく解釈してはいけません。一般的に、適応性と学習の速さは、どの役割においても中核となる基準です。適応力があり、学習速度が速い人は、たとえ以前の現場経験がなくても、新しいコンテキストに合わせてデザインや構築を行うことができます。逆に、学習速度の遅い人は、コンテキストに対する理解が古く、その理解を更新するのに時間がかかり、自分とは異なる視点に抵抗を感じるかもしれません。

非常に若手の場合、あなたが築いているコミュニティの若いエンジニアを優先することで、コンテキストを重視することができますし、そうすべきです。儒教の古い言葉に「街には賢者が溢れている」というものがあります。探す努力をすれば、街にはエンジニアが溢れています。適切な能力と考え方を持ち、チームでの議論を豊かにするような経験を持つ人を見つけるために、履歴書を吟味する価値はあります。

しかし、プロジェクトのコンテキストに定期的に触れたり、潜在的なユーザーや実際のユーザーから直接フィードバックを受けたりしなければ、誰も適応したり学んだりすることはできません。大規模で成熟したチームでは、コードを書くこととその使用方法を考えることとは全く別のことなので、伝統的な大規模チームや企業での経験を持つエンジニアは、最初はコンテキストについて考えることに抵抗を感じるかもしれません。しかし、小規模なチームでは（あるいは大規模なチームでも）、全員がユーザーのニーズについて考えるべきなのです。

適応力と学習速度は、どの役割においてもコアとなります。

シビックテックでは、そのことはコミュニティ・アウトリーチ・チームと開発チームの間に密接な関係を築くことを意味します。また、ターゲットユーザーを深く理解し、誠実に対応するコミュニティチームや現場チームが必要となります。

現場チームとエンジニアチームの間で、情報が自然に流れるような関係を築くことに力を注ぎましょう。エンジニアが現場チームに同行することもあれば、エンジニア同士の機能に関する議論に現場チームを招き、貢献してもらうこともあるでしょう。エンジニアが来て「Xについてよくわかりません」と聞いてきたとき、エンジニアを現場チームのメンバーに引き合わせましょう。逆に、現場から「アプリにXをさせることができるだろうか」と言われたら、明らかに意味のあることであれば、現場チームのメンバーを、Xを実装できるエンジニアのところへ直接行ってもらうようにしましょう。
それは絶妙なバランスです ー CTOやいわゆる上級管理職は、頻繁にループに入る必要がありますが、その活動が作為的であれば無意味になりかねません。適切なバランスを見つけることは可能です（図2）。

私が働いたGrassrootでは、最初は比較的シニアに偏った構成で、多方面の才能を持つ契約者が多すぎました。時が経つにつれ、一度に1〜2人の若手エンジニアを配置し、そのうちの何人かは数年間チームに留まり、特定のタスクを達成するために数ヶ月間だけシニアのスペシャリストを補充するという体制に移行しました。CTOである私は、スペシャリストのコードをレビューしたり、若手エンジニアを指導したりしました。コミュニティチームは、3分の2の時間を現場で過ごし、現場にいないときは自分たちが見てきたことをエンジニアに伝えることを義務づけられていました。この体制に加えて、ユーザーエクスペリエンスのスペシャリストを随時投入し、ユーザーの行動を観察するトレーニングをコミュニティチームに提供しました。また、このような体制をとることで、他の人材を柔軟に取り入れることができました。例えば、いくつかの大学が開発した「コンピュータ・サイエンス・フォー・グッド」という優れたプログラム（私たちの場合はスタンフォード大学と協力しています）などです。

Don’t build it

無機質なチーム関係

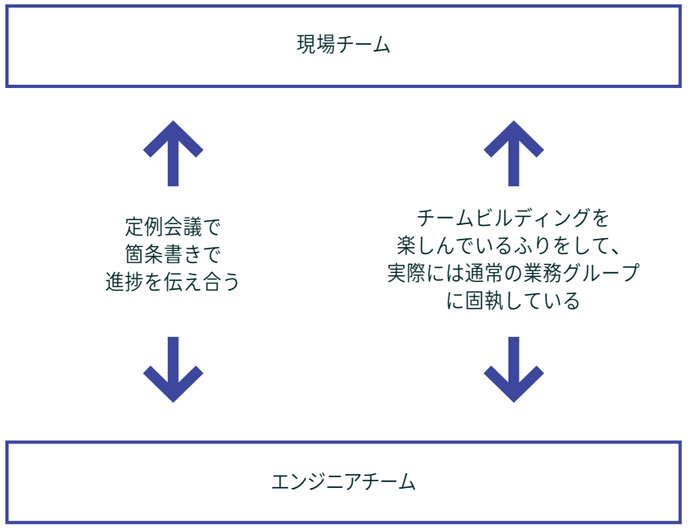
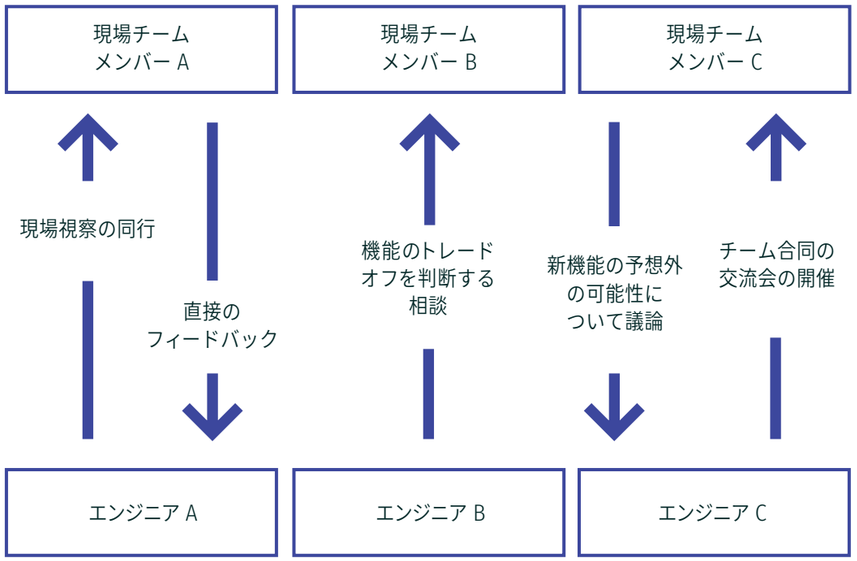
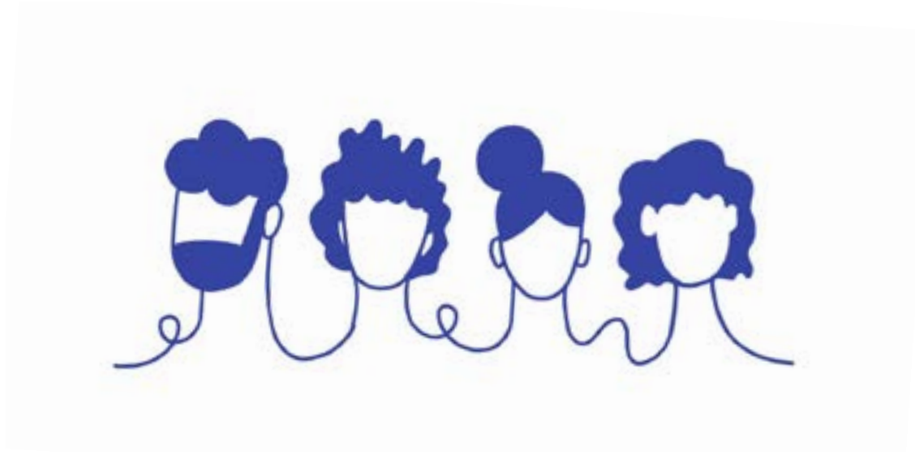


図2

有機的なチーム関係





最高技術責任者（CTO）

CTOを採用する際には、用語の混乱が邪魔になることがあります。大企業では、CTOの役割は、主にプロセスを監督し、中間管理職を雇用して監督することです。優れたCTOは、時には手を汚すこともありますが、主にいくつかのプロセスを構築するために監督します。しかし、スタートアップや非営利団体、あるいは政府のデジタルサービス部門では、求められる役割は異なるでしょうし、CTOは物を作り、物を作ることが好きでなければなりません。

適任者のバックグラウンドの例としては、強力な技術系企業でシニアエンジニアや「エンジニアリングマネージャー」を務めていたが、後者の役割が手続き的に重いために辞めたいと考えている人や、野心的で複雑なプロジェクトに取り組み、それを実現してきた実績を持つフリーランスの人などが挙げられます。主な基準は、新しい環境やチームに適応する能力、優れたプロセスの規律と柔軟性を兼ね備えていること、そして何よりも、ものを作り、人々がそれを使うのを見るのが好きであることです。

Don’t build it

CTOを見つけるにはリクルーターが役に立つでしょう。しかし、型にはまらない道や「ギザギザの履歴書」でも構わないことを説明することが大切でしょう。候補者が見つかったら、次のような質問をしてみましょう。

- *プロジェクトをうまい具合に中断させたことはありますか？*
- *これまでに経験した中で最高のチームメンバーはどんなメンバーでしたか？*
- *あなたが経験した最悪のチームメンバーと、それを避ける方法を教えてください。*
- *あなたは誰かを “最悪 “の状態から “最高 “の状態に導いたことがありますか？その時、あなたは何をしましたか？*
- *あるプロジェクトにおいて、ある技術をスタックに入れるかどうかをどのように判断しますか？*
- *プロジェクトが（非常に）遅れている場合、どのように対応しますか？また、誰も使っていない場合はどうしますか？*

これらの質問に対する完璧な答えはありません。あなたが求めているのは、自分のチームを大失敗や多くの無駄な努力から救うことができる思慮深い人、良い意味でも悪い意味でもチームを構成する方法を知っている人、若いチームメンバーを指導する方法を知っている人、新しい技術に常に注意を払いながらそれを現場に導入するための現実的な方法を知っている人、そして人生に何かが起こったときにうまく対応できる人です。あなたが活動する文脈に多少なりとも精通している人物を見つけることができれば、高く評価できるでしょうが、決定的なものではありません。適応力が高く、厳格さと創造性のバランスがとれている優秀なメンターでも、自分の置かれた状況にあまり精通していない人の方が、状況は深く知っているが適応も指導もできない人よりも、はるかにうまくいく可能性が高いです。

最後に、いくつかの注意点があります。勉強しなくなった人は避けましょう。自己アピールをする人は避けましょう。「コードを書くことよりも、チームを構築したり管理することのほうが価値があるからです」と言う人には注意が必要です。もちろん、チームが大きく成長すれば、コードを書くことがCTOの時間の使い方ではなくなります。しかし、少人数のチームで活躍するCTOは、自分でも多くのコードを書く必要があるでしょう。もし、そのような活動が好きではなく、構築よりも管理をしたいと考えているのであれば、それは良い結果にはならないでしょう。

あなたは私が書いたコードではなく、私が書かなかったコードに対して、より多くの報酬を支払っています。

上級エンジニア

私が一緒に仕事をした最高の上級エンジニアから、「あなたは私が書いたコードではなく、私が書かなかったコードに対して、より多くの報酬を支払っている」と言われたことがあります。それは大げさな表現でしたが、シニアエンジニアがもたらす価値の核心部分を要約していました。有望と思われる道でも結局は行き詰まる可能性がある。あるいは、他の道と比べれば短時間で希望通りではないが需要を満たし、結果、時間と労力を抑える道があることを知る経験になりました。

技術プロジェクトの管理は、トレードオフの連続です。非常に優秀な上級エンジニアは、可能なオプションの範囲を広げ、結果を達成するためのより複雑でない方法を見出す（あるいは、より複雑でない道がないという確信を深める）ことで、トレードオフの条件をより良くする手助けをします。契約者がこの技術にどれだけ長けているかを事前に完全に知ることは難しいですが、面接でそれをテストすることは可能であり、例えば次のように尋ねることができます。

- **チームが間違った選択をするのを防ぐことに成功した時のことを説明してください。**
- **あなたが経験した最悪のプロジェクトは何ですか？その中での最悪のトレードオフは何でしたか？**
- **自分が何かを複雑にしすぎているかどうかを、個人的にどうやって知ることができますか？あるいは、シンプルにしすぎているときは？**

人気のあるフレームワークやテクノロジーに対して、契約者が独断的な見解を示した場合は要注意です。若手のチームメンバーがいて、上級エンジニアが彼らを指導したり、知識を伝達したりする必要がある場合は、過去に同じような状況にどのように対処したかを聞いてみましょう。

上級エンジニアを見つけるには、ハイエンドの人材を見つけてフィルタリングすることに特化した非常に優れたプラットフォームがあります（例：Toptal）。高額なマージンを請求されることもありますが、短期間であればコストは管理可能です。彼らが行う質の高いフィルタリングと、ニッチなスキルを理解して見つけ出す能力には、価値があります。もちろん、彼らを利用するには、高品質の人材を柔軟に統合し、迅速に利用できる技術チームが必要です。プロセスが硬直しすぎていたり、チームの他のメンバーが独断的であったり、もちろんCTOがプロセス全体を監督していなければ、それは不可能です。

UX/UIデザイナー

優れたUX/UIデザイナーは、ユーザーの行動に合わせて、どのような状況にも適応できなければなりません。デザイナーを採用する際の課題は、完成したプロジェクトのどこが彼らのものなのかを見極めるのが非常に難しいことです。プロジェクトが終了する頃には、あまりにも多くの検討が行われているはずなので、元のデザインはとっくに埋もれてしまっています。最初のデザインと最後のデザインがまったく違うものになっているのは、出発点が悪かったのかもしれませんし、柔軟でクリエイティブなデザイナーが開発チームと緊密に連携して新しい方向性を打ち出しているのかもしれません。

このジレンマに対する本当の解決策は、おそらくないでしょう。面接では、デザイナーがどのように学習し、適応していくのかを理解するために、過去の実績をみます。私の友人は、ひとつのワイヤーフレームやスクリーンを徹底的に調べ上げるのが好きです。そうすることで、そのデザインについてどれだけ深く考えているか、どれだけ責任を持っているか、そしてそのデザインについてどれだけ会話ができるかがわかります。長期間の採用であれば、短い時間での持ち帰り課題を設定するのが妥当です。また、自分のやり方に非常に固執するデザイナーは避けたほうがいいでしょう。なぜペルソナがダメなのかを説明できない人は、大企業の組織には向いていますが、小規模な組織や曖昧な状況で活動するチームには向いていないでしょう。

さらに言えば、良いデザイナーを見つけたら、その人が自分と一緒に仕事をすることを楽しんでもらえるように全力を尽くすことです。この分野での採用がいかに難しいか、そしてそれがプロジェクトにどれほど大きな違いをもたらすかを考えると、あなたと一緒に仕事をするのが好きな優れたデザイナーほど、技術を作り上げることにおいて価値のあるリソースはありません。

若手エンジニア

世界中の大学やブートキャンプ、オンラインコースでは、コーディングは需要が高く、給料も高く、スキルを要するやりがいのある仕事だと聞いた人たちが毎年続々と集まってきます。ところが、修了後まもなく、彼らは何度も仕事を断られます。もしかしたら「技術面接」を受けて、自分がする必要のないことについてまったく無意味な質問をされ、不合格になるかも知れません。この不条理なファネルを通過した人たちは、巨大で無意味なプロジェクトの中で、取るに足らない仕事を任されることになります。そして誰もが、なぜこんなにひどいソフトウェアが毎年大量に作られているのかと不思議に思うのです。

この状況は悲惨でなければ滑稽です。しかし、予算内で開発チームを編成する人にとっては、非常に便利なものです。つまり、非常に才能のある若いエンジニアを、ほとんどの場合そのプロジェクトが運営されている地域から採用し、履歴書に2年間の経験を記入することで、入社時の不条理を乗り越えるルートを提供し、適度な給与を提示することができるのです。彼らが完全にスピードアップするには6ヶ月必要で、さらには1年半から2年で退職するでしょうが、下手な構成のチームでは考えられないようなペースと予算で構築することができるのです。

そのためには、大量の履歴書（数百通）を見なければなりません。このプロセスを自動化したり、外注したりすることはできません。そうしないと、せっかく採用できそうな優秀な若いエンジニアを、採用を気にかけない担当者がただ履歴書の山を処理したいだけのために捨ててしまうことになります。だから、あなたとCTOがやらなければならないのです。それは、あなたができる最も価値の高い仕事かもしれません。もしその時間と労力がなければ、そのプロジェクトを始めなければいいのです。

私が最も効果的だと感じた履歴書と面接の両方における基準は以下の通りです。

- **学ぶことへの渇望。** *従来の質問の中には、サイドプロジェクトや、勉強や仕事以外で学んだテーマ、失敗談など、純粋に役立つものがあります。また、過去のプロジェクトでチームが下した決定について、その理由を尋ねるのも有効です。仕事ですぐに学ぶのが好きな人は、その決定とその根拠を理解しようとするでしょうが、そうでない人は、ただそれを記録して先に進むだけでしょう。*

- **品質へのこだわり。** *コーディングには、近道や現実的なトレードオフがつきものです。大きな誘惑は、選択肢を適切に検討することなく、簡単な方法を取り、それをトレードオフとして正当化することです。外から見ただけでは、どちらが正しいかわからないことが多いので、若いエンジニアが直面する大きな誘惑は、実用性を言い訳にしてしまうことです。コードテストで少しは試すことができますが、ホワイトボードの上ではできません。ここでは、持ち帰りのコーディングテストが役立ちますが、それはその後に議論する場合に限ります。そうでなければ、これまでの人生でトレードオフを管理してきたことについて質問したり、想像上の（しかし具体的でもっともらしい）シナリオへの対応について仮説を立てたりすることが役に立ちます。*

- **基本的な技術力。** *これは当たり前のことですが、（経験があると思われる人でも）コードを書けない面接官が多いことに驚くことがあります。そこで、簡単なコーディングの質問と、いくつかの技術的な質問を用意しておくと便利です。*

私がこれまでに採用した中で最も良かったのは、最初に養鶏をしようとした人でした。南アフリカ政府は彼に養鶏のための補助金を与えましたが、病気の管理については何も教えてくれませんでした。鶏が死んでいく中、彼は独学でコーディングを学びました。私が彼にインタビューしたときには、彼は友人のためにいくつかのミニアプリケーションを作っていて、それらを作ったときの選択や、将来どのように違うものを作るかについて、まとまった話ができるようになっていました。彼を見つけたのは、同じように若くて才能がある異色のリクルーターで、私が変わった候補者を探すように伝えていました。

DEFINITIONS

Stack Overflow:
エンジニアが質問をしたり答えたりするサイト。ウィキペディアに次いで、インターネット上で最も価値のある公共財であることは間違いないでしょう。

工程表と予算

予算を設定することで、迅速なスタートを切ることができ、かつ時間をかけて開発サイクルを繰り返していくことができます。

あなたが構築しようとしているテクノロジーのシンプルなバージョンは、あなたが考えているよりも早くドアから出て、実際の人々の手に渡る（「ローンチ」と呼ばれる）ということがほとんどです。また、ほとんどの場合、テクノロジーが頻繁に使用され、バグが少ない状態（「成熟度」）にするには、あなたやあなたのスケジュールが見積もっているよりもはるかに長い時間がかかります。そのため、予算を設定することで、迅速なスタートを切ることができ、かつ時間をかけて開発サイクルを繰り返していくことができます。

ソフトウェアではたいていのことがうまくいきません。誰もあなたが作ったものを使いたくなかったり、世に出てきた途端、奇妙な理由で失敗します。なぜなら、ソフトウェアも人生と同じように複雑だからです。あるいは、プロダクトの作りが悪かったために失敗することもあります。しかし（1）優秀なCTOを雇い（2）「DevOps」や（まだチームの習慣になっていない場合は議論の余地がありますが）「テスト駆動開発」と呼ばれる手法を用いることで、バグの数を一桁減らすことができます。

しかし、最高の基準で作られたテクノロジーであっても、現実の世界に出てくると予期せぬバグが発生します。これにより、テクノロジーを構築するための実際の工程表が、ローンチ日を大幅に過ぎてしまうことがあります。ソフトウェアは、多くの小さな部品をつなぎ合わせて構成されており、複雑さに満ちた世界にリリースされます。ユーザーは、最初は理解し難く予想もできないような行動をとります。予期せぬ動作とコンポーネント間の複雑な相互作用の間で、何かが壊れている原因を特定することは非常に困難です。

Don’t build it

要約すると、ローンチの工程表は常に短くなり、成熟の工程表は常に伸びるということです。具体的にはどういうことでしょうか？こういったガイドのルールは、自分の首を絞めるような具体的な内容を言わないことです。ソフトウェア開発のルールは、「必要なだけ時間をかける」です。しかし、私はこの2つのルールを破ります。3〜4人の優秀なチームが適切なプラクティスに従えば、中程度の複雑さのプロダクトを約3ヶ月でローンチし、12ヶ月である程度の安定性と成熟度を得ることができるはずだと言いたいのです。もちろん、その後もユーザーがいればリスタートができます。ユーザーがいない場合は、6ヶ月で止めるか、より良いのは4ヶ月でやめることです。そして最善な方法は、**つくらないこと**です。

こういったガイドのルールは、自分の首を絞めるような具体的な内容を言わないことです。ソフトウェア開発のルールは、「必要なだけ時間をかける」です。しかし、私はこの2つのルールを破ります。

これらのスケジュールが予算に与える影響は、チームの構成によって異なります。Grassrootでは、4年間で合計約20万米ドル（約2800万円）を開発に費やしました。基幹プラットフォームの構築、大規模な国内キャンペーンを実行するための拡張、WhatsAppボットや機械学習システム、いくつかのAndroidアプリが含まれています。ただし、この組織では、エグゼクティブ・ディレクターとCTO（=私）を兼任して、相場よりもはるかに低いレートで雇っているために少し歪んでいます。これはあまりお勧めできません。にもかかわらず、私はこの金額の何倍もの金額で見積もっている組織があることを知っています。

先進国市場では、優秀なエンジニアリング・マネージャー（CTOになるためには、以下を参照）は月に約1万ドル（約140万円）（米国の大都市ではもっと高い）、経験の浅い優秀な若手は約4,000ドル（約56万円）（やはりニューヨークやサンフランシスコではもっと高い）が相場です。発展途上国の市場では、「インパクト投資ファンド」やベンチャーキャピタルの資金によって完全に歪められていなければ、この数字は5,000ドル（約70万円）や1〜2,000ドル（約14〜28万円）になるでしょう。例えば、東欧の優秀なシニア・コントラクターは、質の良いマッ

DEFINITIONS

テスト：
ソフトウェアに特定の機能を実行させ、それが正しく実行されたかどうかをチェックするコードのこと。

DEFINITIONS

テスト駆動開発（TDD）：
ソフトウェア自体を書く前に、ソフトウェアが合格するためのテストを書くこと。テスト駆動開発は、長期的には多くの時間を節約し、重要な機能が変更された場合にはすぐに明らかにすることができますが、前もって追加の時間が必要になります。

DevOps：
新しいコードを既存のコードに継続的に統合し、迅速かつ確実にデプロイするための一連の手法。コードを統合してデプロイするための具体的なツールを「継続的インテグレーション／継続的デプロイメント」（CI/CD）と呼びます。

チング・プラットフォームを通じて調達した場合は時給80～100ドル（約11,000～14,000円）直接調達した場合は時給40～60ドル（約5,500～8,500円）です（以前仕事をしたことがある請負業者か、強力な紹介がない限り、プラットフォームを使う価値があります）。全体的に見て、社内に技術的な専門知識がある場合を除き、中程度の複雑なプロジェクトを構築して成熟させるには、5～10万ドル（約700～1400万円）（多くのプロジェクトに分散されるため、CTOのコストは除く）かかると考えられます。それ以下の金額であれば、冗談のような話にしか聞こえませんし、それ以上であれば、おそらく誰かが必要以上に利益を得ていることでしょう。

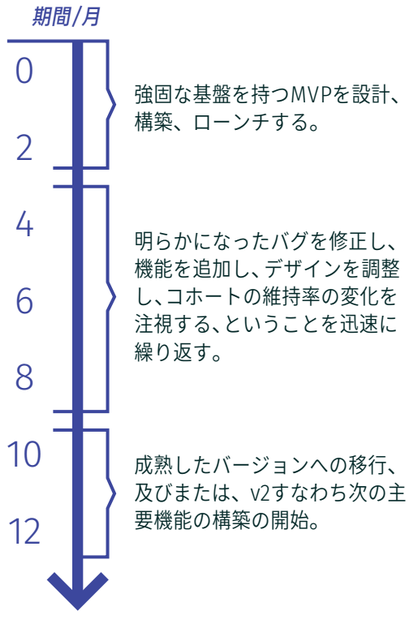
ここで最後に2つのポイントについて述べます。1つはプロジェクトの開始時、もう1つは成熟期に近い時です。DevOpsのパイプライン（定義を参照）を整備するだけでなく、基本的なドキュメント、コーディングスタイルの規約、新規エンジニアを立ち上げるためのスクリプト、コード品質のレポートや脆弱性スキャンなどの自動化なども含めて、「ツール」と呼ばれるものに投資することで、将来的に利益を得ることができます。このようなツールへの先行投資は、日常的な問題だけでなく、火事場の際にもチームの作業スピードを向上させます。時間をかければ、何倍にもなって返ってくることがあります。とはいえ、限界があります。もし誰かが「Kubernetes」というものを提案し始めたら、やめるように言ってください。

成熟に近い、おそらくソフトウェア開発で最も古い法則は「ブルックスの法則」、つまり、「遅れたソフトウェアプロジェクトに人手を加えると、そのプロジェクトはさらに遅くなる」というものです。もっと一般的に言えば、プロジェクトが大惨事になりかけている場合、予算や人員を追加することは最悪の事態を招く可能性があると言うことです。新たな人材の投入、新たな資金の使い道の検討、新たな報告書の作成などは、CTOやシニアチームの時間という最も希少なリソースを奪い、最も重要なタスクである核心的な問題の診断と解決からそらすことになります。プロジェクトが角を曲がって、慢性的な遅れや故障から、スケジュールは遅れているものの、迅速かつ健全な方法で進んでいる状態に変わったとき、リソースを追加することで加速することができます。角を曲がってからガソリンを入れることが大切です。角を曲がる前にガソリンを入れると、より強く壁にぶつかることになります。

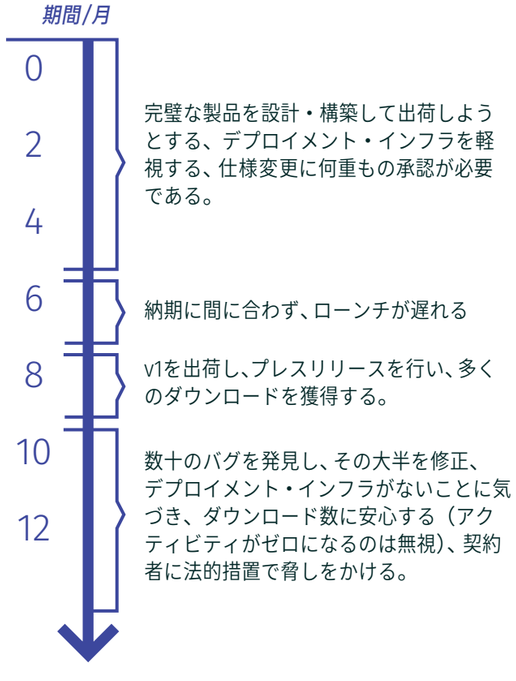
角を曲がってからガソリンを入れることが大切です。角を曲がる前にガソリンを入れると、より強く壁にぶつかることになります。

Don’t build it

良いタイムライン



できの悪いタイムライン（計画・実績）



DEFINITIONS

スクリプト：
例えば、エンジニアのマシーンに構築中のアプリケーションのプロトタイプを作成するなど、一連のコマンドや操作を指定したファイルのこと。環境構築、ログファイルのアーカイブ、コードの新バージョンのチェックなど、特定のタスクを自動化するためによく使用されます。

ツール：
エンジニアが他のプログラムやアプリケーションを作成、文書化、デバッグ、メンテナンス、デプロイなどのサポートを行うために使用するプログラム。例えば、ソースコードエディタや、コード品質の問題を検出してフラグを立てるためのツールなどがあります。

Kubernetes：
大規模なアプリケーション（数百万のアクティブユーザー）の展開を管理するための複雑なシステム。

ユーザー vs バニティメトリクス

証拠とは、十分な数のユーザーを対象に、十分な時間をかけて収集されたアプリ内のデータです。

シリコンバレーの言い伝えに役立つものがいくつかあります。最も実用的なのは、「プロダクトマーケットフィット」という考え方と、その対極にある「バニティメトリック」です。プロダクトマーケットフィットとは、人々が価値を感じて定期的に使用し、他の人にも教えたくなるようなものを作ったということです。言い換えれば、あなたが作っているものを人々が実際に使いたいと思っているかどうかということです。ここで重要なのは、人々があなたのプロダクトを使いたいと思うかどうかというあなたの意見、パートナーの意見、フォーカス・グループ（情報を収集するために集められた顧客のグループ）の意見でさえも、全て証拠として認められないということです。証拠とは、十分な数のユーザーを対象に、十分な時間をかけて収集されたアプリ内のデータです。

同様に重要なのは、「プロダクトを使用している人々」とは、「総ダウンロード数」や「ランディングページ数」、あるいは「作成されたアカウント数」を意味するものではないということです。「使っている」とは、中心となる機能を定期的に繰り返し実行すること、あるいは、さらに良いことに、その技術にお金を（たとえマイクロペイメントであっても）支払うことを意味します。これらと対極にあるメトリクス（評価尺度）が「バニティメトリクス」であり、気分を良くしたり、技術を良く見せたりするメトリクスを意味しますが、現実を反映していません。最も一般的な例は、総ユーザー数を測定することで、このメトリクスは、完全な大失敗の場合を除いて、常に上昇します。

Don’t build it

バニティメトリクスは、非営利団体や政府のプロジェクトでは特に危険です。なぜならば、自由な報道によって簡単に持ち上げられ、損益が発生しないため、そのようなメトリクスは学習を短絡的に進ませることができるからです。さらに、資金提供者や政府高官は、目標を達成しやすいメトリクスを使用するという強い動機づけを与えます（そして自然と報道されます）。しかし、プロジェクト自体にとっては、初期の迅速な学習ほど重要なものはなく、バニティメトリクスはそのような学習を省いてしまいます。可能であれば、製品市場への適合が近いと確信できるまで、プレスリリースを配信しないでください。また、他のステークホルダーをなだめるためにバニティメトリクスが必要であっても、チーム内の議論では禁止してください。バニティメトリクスは、特に困難な時期に最も誘惑的です。チームメンバーの一人が「そんなに落ち込むことはないだろう、総ユーザー数はまだすごく多いんだから！」と言うだけで、進歩を左右する、仕組みを遵守する誠実さが失われてしまうのです。

初期の迅速な学習ほど重要なものはなく、バニティメトリクスはそのような学習を省いてしまいます。

引き換えにダウンロードからサインアップ、定期的な使用や購読までのファネルを構築し、新規ユーザーがどのように進んでいくかを観察します。そのためには、ユーザーがどのようにアプリを使用しているか、匿名化・集計された詳細なデータが必要となります。これは「インストルメンテーション」と呼ばれ、ユーザーのグループ（コホート）の平均的な行動がどのように変化するかを観察することができます。Amplitudeのようなツールを使えば、自分で作らなくても簡単に設定でき、エンジニアではないチームメンバーでも利用状況の変化をモニターできます。たとえば、どの段階で何%の人が利用をやめてしまうのかを示すグラフを設定し、毎朝そのグラフを開きます。ソーシャルメディアの最新情報がバニティメトリクスに影響を与えようとも、状況が改善されないのであれば、修正する必要があります。コホート・リテンションが上がらないのであれば、あなたのプロジェクトは本来の用途に合っていないのです。

そんなとき、あなたは何をしますか？ユーザーエクスペリエンスとユーザーインターフェースデザイン(UX/UI)の一連のテクニックは、プロダクトのどこが不足しているかを理解するのに役立ちます（ただ人に聞くのは良いアイデアではありませんー親切すぎて、覚えていないかもしれません)。UX/UIの専門家がいなかったり、予算が足りない場合は、テクニックを勉強してみてください（私のUX/UIの友人には許してもらいたいですが)。ただし、本当に困っている場合は、予算を確保するか、UX/UIのボランティアを募ってくださ

DEFINITIONS

ファネル：
潜在的なユーザーが定期的に利用するようになるまでの一連のステップのことで、ウェブサイトやアプリのリストに掲載されてから（ファネルのトップ）、完全に設定が完了して定期的に利用されるようになるまで（ファネルのボトム）のこと。

DEFINITIONS

コホート：
ほぼ同じ時期に製品の使用を開始したユーザーのグループ。製品と市場の適合性が向上しているか、または悪化しているかを分析するために使用される。

コホート・リテンション：
あるコホートのうち、どのくらいの割合の人が後日も製品を使用しているか。

コンバージョン率：
例えば、アプリをダウンロードした人やホームページにアクセスした人のうち、何%がアカウントを作成したかなど。

インストルメンテーション：
ソフトウェア製品がどのように使用されているか、いつ、どこで障害が発生しているかをリアルタイムで計測するためにソフトウェア製品に組み込まれたツール。

UX/UI：
ユーザー体験をデザインし、ユーザーインターフェース（具体的なデザイン）にする専門職。

い。前もって良いツールと機器に投資しておけば、微調整を繰り返し、その結果をリアルタイムで観察することが簡単にできるはずです。忍耐力と柔軟性は必要ですが、それほど多くは必要ではないでしょう。

根気強さと愚直さのバランスを取るのは、難儀なことです。発売後の数ヶ月間は、コンバージョン率が上がったり下がったりするのは当たり前です。しかし、3ヶ月後にプロダクトマーケットフィットに近づいていなければ、振り出しに戻りましょう。最初のローンチから9カ月経っても、製品市場への適合性があまり高くない場合は、（プロジェクトの）電源プラグを抜いて他のことにリソースを割くべきでしょう。また、ローンチから数ヶ月後に、毎週のコホートの定着率が長期間にわたって低下し、問題解決のために3回以上繰り返しても効果が得られない場合も要注意です。

コホートが一貫して改善され、健全な割合のユーザーが固定客や有料ユーザーに変わってきたら、次に注目すべき指標は「ネットプロモータースコア」（NPS）です。これは、ユーザーが製品を他の人に推薦する可能性を測定するものです。NPSを測定する最も良い方法は、製品に紹介コードシステムを組み込み、その使用状況を監視することですが、時折行われる自動アンケート調査でも代用することができます。

あなたの業績を測る最も悪い尺度は、プレスカバレッジです。多くの報道がなされているプロジェクトは非常に疑わしいものです。それはチームがプロダクトを機能させるのではなく、バニティメトリクスを上げようとしていることを示唆しています。

技術の選択

プロジェクトに参加したり、自分でコードを書けるようになりたいと思っている、エンジニアではない友人から、”X言語は良い選択なのか？”とよく聞かれます。このような問いかけには、採用したCTOが答えるべきです。しかし、医師ではないのに医療機関を運営している場合、医療技術に関する高度な議論についていけるだけの健康知識を身につけておくと良いのと同様に、あなたが技術の選択についても少し知っておくと役に立ちます。

知っておいてほしいのは、大まかな条件の範囲内であれば、具体的な内容はあまり重要ではないということです。プロジェクトにどのプログラミング言語を使うか、どのデータベースを使うか、モバイルアプリやWebページに何を使うかといったことに時間を費やしてはいけません。その代わりに、いくつかの基本的な原則に従い、自分たちが選んだテクノロジーが完璧なソリューションであるとあなたを説得しようとしすぎる人は避けましょう（あなたに何かを売りつけようとしているか、経験が足りないかのどちらかです）。基本的な原則は以下の通りです。

(1) オープンソースにこだわる。 *非営利団体の場合、この選択は予算によって大きく左右されます。政府のプロジェクトでは、巨大なエンタープライズ・ベンダーが、先進国でオープンソースに取って代わりつつあることを補うために、発展途上国で積極的に販売しようとしているという動きがあります。彼らに耳を傾けてはいけません。PostgreSQL（またはMySQLやMongoDB、あなたのチームがそれらの経験が豊富であれば）を使えば良いのです。*

(2) 人気のある言語を選びましょう。 *大まかな目安として、機械学習が重要な場合はPython、何百万人ものユーザーを想定している場合はJavaやC++、それ以外の場合はJavascript系（NodeJS/Typescript）を使うといいでしょう。しかし、PHPを避けてさえいれば、何を使うかはあまり重要ではありません（独断は禁物と言いましたが、PHPは実際にひどいものです）。人気のある言語を選ぶということは、より多くの人たちの中から雇ったり契約したり、より多くのフレームワークから選択することが可能となり、パッケージに隠された不具合がより早く修正される可能性があるということです。*

上記の様々なセクションで説明したように、テスト駆動型の開発が必要だと思うでしょう。また、すべてが自動化されていることも確認しましょう。たとえば、リアルタイムの使用統計を監視し障害を検出できること、契約者を使用している場合にはドキュメントが適切に作成されていることを確認できることなどです。

重要なのは、技術の選択がセキュリティに与える影響はほとんどないということです。優れたセキュリティを実現するには、プロセスの規律とこだわりが必要です。もし誰かが、技術の選択によってセキュリティが向上すると信じているなら、そのプロダクトが使われる限り、ハッキングされてしまうでしょう。セキュリティは設計から始まります。「セキュリティ・バイ・デザイン」と呼ばれる手法では、実用的な範囲内でシステム全体を設計し、侵入を防ぎ、封じ込めることができます。例えば、個人を特定できるデータを含むテーブルは、通常のユーザー操作を処理するためのサーバーとは別のサーバーに保管し、そのような操作を処理する関数には、別のサーバーへのアクセス許可を与えないようにします。この方法は、「関心の分離」として知られる優れたソフトウェアエンジニアリングの一般原則と重なります。同様に、DevOpsが実施されていることを確認し（上記の「工程表と予算」を参照）、セキュリティスキャンがDevOpsプロセスに組み込まれていることを確認し、自動的にコードの脆弱性がスキャンされるようにします。一般的に、きれいなコードは良いコードであり、良いコードは（比較的）安全なコードです。良いチームは良いコードを作成し、安全なプラクティスを取り入れます。

とはいえ、いつかはハッキングされることを想定しておくべきです。予算のあるプロジェクトでは、ペネトレーションテストを依頼することもできます。そんな余裕があってもなくても、開発チームと定期的にセッションを持ち、「もし今夜、誰かがハッキングしたら、その侵入方法によって流出するデータはどれくらいだろうか？それをどうやって知ることができるだろうか？新たな脆弱性を生み出すような変更はあったのか？」と議論してください。このような議論は、CTOが主導すべきです。エンジニア以外のリーダーも参加して、細かい技術的な議論まではいかなくても、少なくともシステム設計のレベルで侵害を防ぐための可能性と手段を理解する必要があります。

Don’t build it

最後に、開発全体を外注することにした場合、それが最悪の戦略であったとしても、上記のすべてを作業範囲の要件に含むことができることを心に留めておいてください。CTOは、エンジニアがそれを守っているかどうかをチェックしなければなりません。もし開発全体を外注していて、CTOを雇っていない場合は、それらの要件を入れて、運良く契約者が注意を払ってくれることを期待しましょう。

DEFINITIONS

API：
Application Programming Interfaceの略。あるアプリケーション（例：モバイルアプリ）が別のアプリケーション（例：処理を行うクラウド機能）と通信するための手段。

フレームワーク：
エンジニアが新しいアプリケーションをすばやく作成できるように、あらかじめ組み込まれたコードパッケージのベース。例えば、ReactはWeb開発のためのフレームワークです。

ペネトレーションテスト：
雇われたハッカーや友好的なハッカー（「ホワイトハット」と呼ばれることもある）が、システムに侵入して脆弱性を明らかにし、それを修正できるようにすること。

結論、そして現実

ベストプラクティスや最善の意図を持っていても、私たちは現実の世界に生きています。マスコミの注目度、資金提供者の好み、チームの好み、組織の惰性などの影響を受けて、悪い理由で何らかのテクノロジーを構築することになるのは、多かれ少なかれ避けられません。「デジタルなことをやれ」「DX（デジタルトランスフォーメーション）をやれ」などと迫られることもあるでしょう。バニティメトリクスが蔓延するでしょう。誰かが報道発表することを提案するでしょう。いつの間にか、代表者のいない組織と責任のない機関を、使い物にならないチャットボットで結びつけ、試しに使ってみたものの二度と使わなかった数千人の人々がいたことで正当化することになるでしょう。あるいは、本当にやる価値のあるプロジェクトを見つけても、優秀なフルタイムの技術リーダーを雇うことができず、結局はパートタイムの、ほとんど興味のない契約社員と、指導を受けていない数人の若手エンジニアが担当することになり、プロダクトのローンチが遅れ、その後も修正されないことになるかもしれません。

現在、多くの寄付金や期待はプロダクトを成功させるための仕組みになっていません。Grassrootでは「中核的」な活動に資金を提供する場合でも、フルタイムとパートタイムの給与を区別する寄付金を受けて、柔軟で順応性のあるチーム構成を機能させるにはどうすればよいか、多くの時間をかけて検討しなければなりませんでした。若手エンジニアが退職し、契約社員が一時的に入り、まったく新しいユーザーの問題を発見して、別の契約社員が入るといった考え方は、多くの資金提供者にとって馴染みのあるモデルではありません。同様に、ユーザーの活動状況やエンゲージメントといった詳細なメトリクスでさえもインパクトにつながらないと知ったとき、一部の資金提供者は新しい地域に拡大することで虚栄心を高めようとしました。しかし、結果は良くてもバニティメトリクスの低いプロジェクトは買い手がつきません。また友人や仲間からアドバイスを求められて「作らない方がいい」と言っても、「それ」が何であれとりあえず作ってプレスリリースをして、そのまま放置されてしまうことが何度もありました。

Don’t build it

脱出できるルートは限られています。願わくば、このガイドの議論が悪いプロジェクトを回避し、チームを正しく構成してより早くローンチして学び、より長く成長させるための一助になればと思います。そうであれば、このガイドは何らかの目的を果たしたことになります。そうでなければ最後に、現実に合わせたダメージを低減するためのヒントがいくつかあります。1つ目は、可能な限り信頼できる優秀なエンジニアやデザイナー、チームリーダーのグループを育て、できる限り彼らに頼ることです。どうしてもプロジェクトをやらなければならず、うまくチームを構成できないが、優秀で信頼できる人材がいてもすぐには動けないという場合は、彼らが着手できるようになるまで遅らせることです。現場のチームと開発チームを継続的につなぐことです。技術Xを使わなければならないが、技術Xを使ってこのプロジェクトを遂行できる唯一の人物がまだいない場合、報告書の延長を要求したり予算にその時間を組み込んだりします。プロジェクトが立ち上がったときには、信頼できる協力者と一緒に、現実に課せられた制約の中で、最大限の戦略的柔軟性とプロジェクトをうまく実行する能力を得るための戦略を練るために、多くの時間を費やすのです。意識して慎重に行動し経験を積めば、驚くほどの余裕が生まれることもあるのです。

最後に、最も重要なことは、迅速にユーザーとの距離を縮めることです。ユーザーに近づくということは、専門のコミュニティ・エンゲージメント・チームを持ち、可能な限り現場に出向き、直接観察して開発プロセスに反映させることを意味します。それが早ければ早いほど、ユーザーが使いたいと思うものへの反復が早くなり、悪いアイデアを少なくとも部分的には使えるように調整することができるようになります。

迅速にユーザーとの距離を縮めることです。ユーザーに近づくということは、専門のコミュニティ・エンゲージメント・チームを持ち、可能な限り現場に出向き、直接観察して開発プロセスに反映させることを意味します。

では、みなさん

作ることをやめることができますか？

Don’t build it

もし作らなくてはならないなら、

- CTOを雇い
- 早くローンチし
- 長く成長させてください

↓

そして何があっても、

- 信頼できる仲間を集め
- 無駄をなくして迅速に構築し
- できるだけ早くユーザーに近づき、
  ユーザーとともに作り上げてください

↓

そうしなかった場合、

Bhagavad Gitaが言うとおり

**「あなたには仕事をする権利はあるが、
果実を手にする権利はない」**

Don’t build it

つくってはいけない

シビックテックの実践者／エンジニアのためのガイド

GRASSROOT

MIT GOV/LAB