大きなプロダクトの育て方

@yo_waka

2021.04.10

freee 株式会社

freee株式会社
執行役員 プロダクトコア開発本部本部長

若原 祥正

@yo_waka

2013年6月、freee株式会社へ入社。

会計freeeのエンジニア
-> 会計freeeのエンジニアマネージャ
-> 既存プロダクト全体のマネージャという変遷を辿っています。

開発に関しては攻めたがり、組織に関しては守りたがり。
デカいサービスでもアーキテクチャの変更を入れていったり、メイン機能をリファクタリングしていくのが好きです。

今日は会計freeeに関するその辺の話をしていこうと思っています！

アジェンダ

1. 会計freeeについて
2. 規模の変遷
3. 起きた問題と対応事例
4. 開発/運用のバランス
5. さらなる進化

Mission

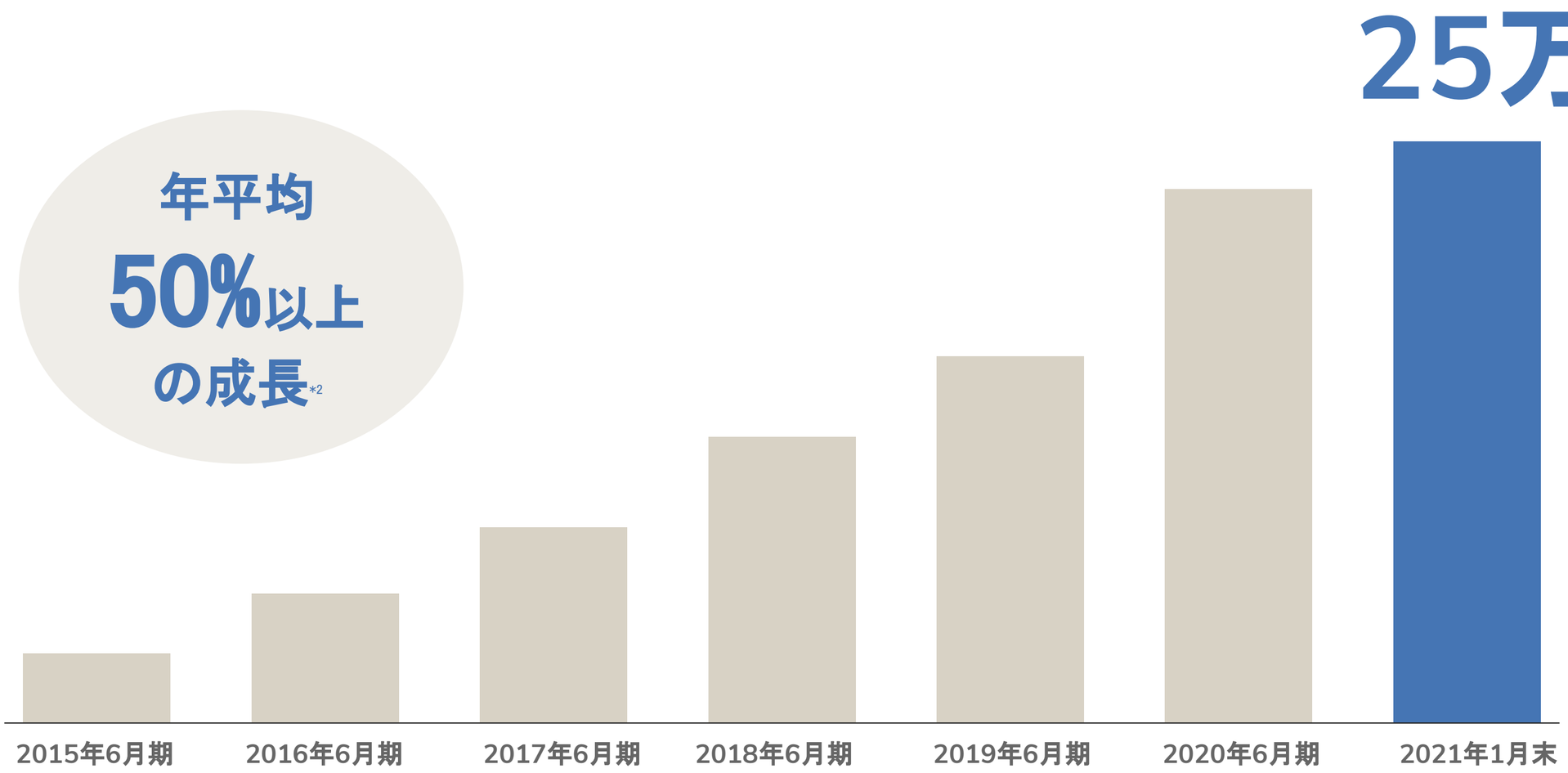
スモールビジネスを、
世界の主役に。

freeeは「スモールビジネスを、世界の主役に。」をミッションに掲げ、
「アイデアやパッションやスキルがあればだれでも、
ビジネスを強くスマートに育てられるプラットフォーム」
の実現を目指してサービスの開発及び提供をしております。

大胆に、スピード感をもってアイデアを具現化することができる
スモールビジネスは、様々なイノベーションを生むと同時に、
大企業を刺激して世の中全体に新たなムーブメントを起こすことができる存在だと考えております。

有料課金ユーザー企業数

有料課金ユーザー企業*1は年平均成長率50%以上で増加し、直近では25万社を超え



*1 2021年1月末時点。有料課金ユーザー企業数には個人事業主を含む。千単位から四捨五入。
*2 2015年6月期から2020年6月期までの年平均成長率

スモールビジネス向けに統合型クラウドERPを提供

統合型クラウド(1)会計ソフト

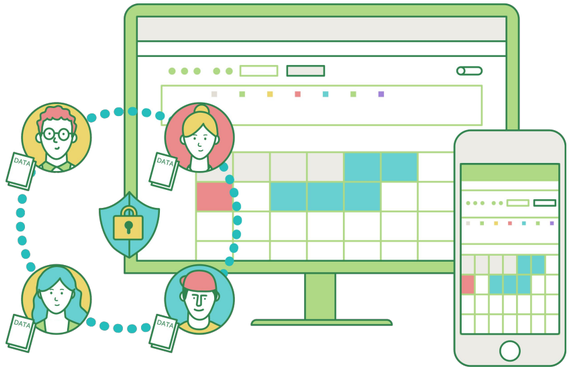
freee 会計フリー

2013年3月～

日本のクラウド市場シェアNo.1 (2)

請求書 | 経費精算 | 決算書 | 予実管理 | 内部統制

統合型クラウド人事労務ソフト



freee 人事労務フリー

2014年10月～

日本のクラウド市場シェアNo.1 (3)

勤怠管理 | 入退社管理 | 給与計算 | 年末調整
マイナンバー管理

その他サービス

freee プロジェクト管理フリー	freee 会社設立フリー	freee 開業フリー	freee 申告フリー	freee マイナンバー管理フリー	freee フリーカード
プロジェクト管理	会社設立	開業	税務申告	マイナンバー管理	クレジットカード

注:
1. クラウドサービス:ソフトウェアやハードウェアを所有することなく、ユーザーがインターネットを経由してITシステムにアクセスを行えるサービス
2. 株式会社BCN「クラウド会計ソフトを導入している従業員数300名未満の企業又は個人事業主へのWeb調査(2017年9月実施、2017年10月公表))」(N=418)
3. クラウド給与計算ソフトの市場シェア:株式会社MM総研「日本におけるクラウド給与計算ソフトの利用状況調査に関するWeb調査(2016年3月実施)」(N=4,168)

会計freeeの基本的な構成

- バックエンドはほぼ Rails
  - 巨大なモノリス＋いくつかのマイクロサービス
  - 人事労務 freee など、他の社内プロダクトとの連携あり
  - 銀行などの明細を取得する部分も含んでいる
    - これはマイクロサービスへの切り出しが進んでいる
- フロントエンドは React
  - 一部coffeescriptが残っている（以前のRailsの名残）
- データストアはRDBにMySQL, そのほかElasticSearchやRedis等もある

そこまで変わった構成ではないです

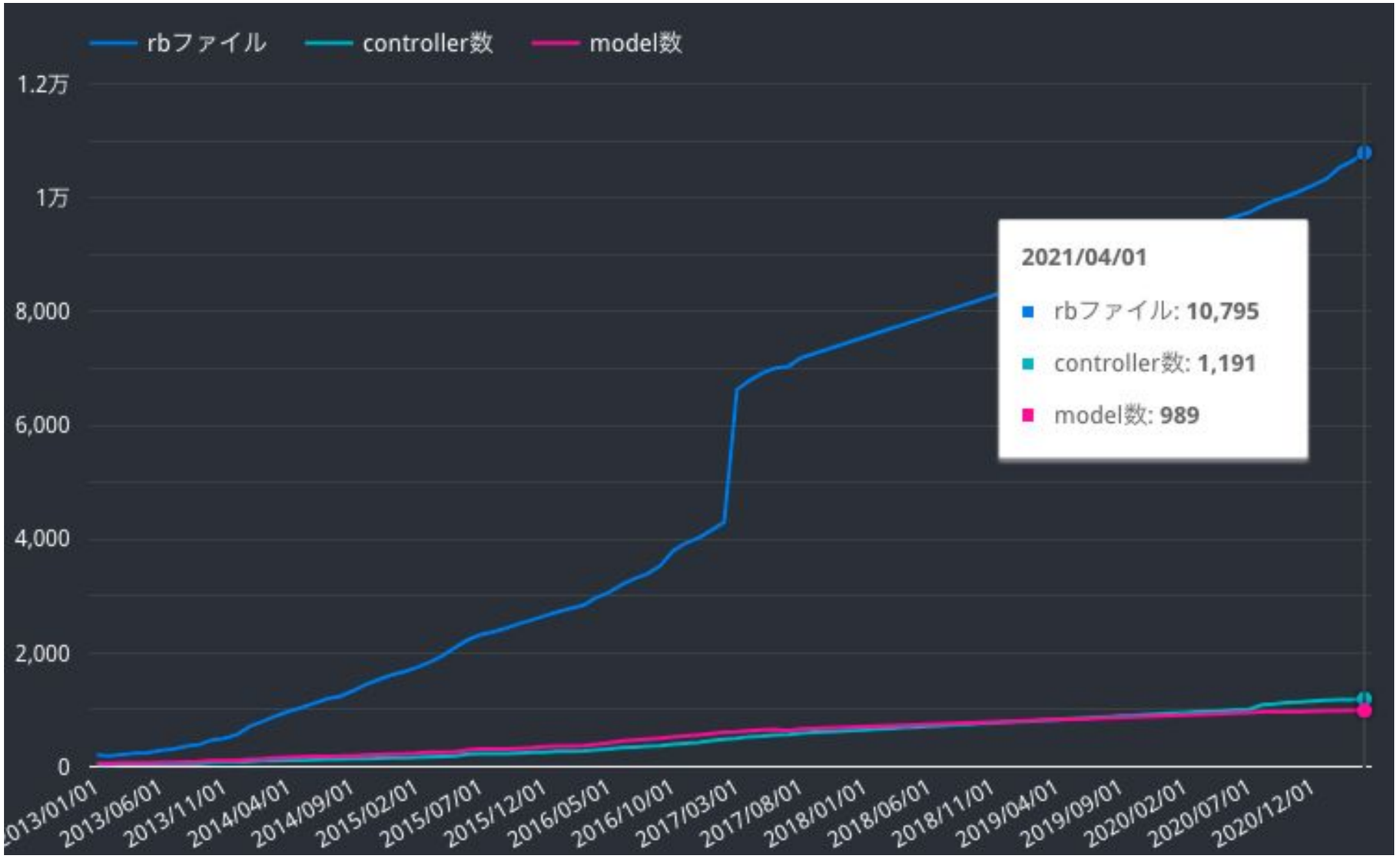
規模の変遷

会計 freee の開発に関する数字

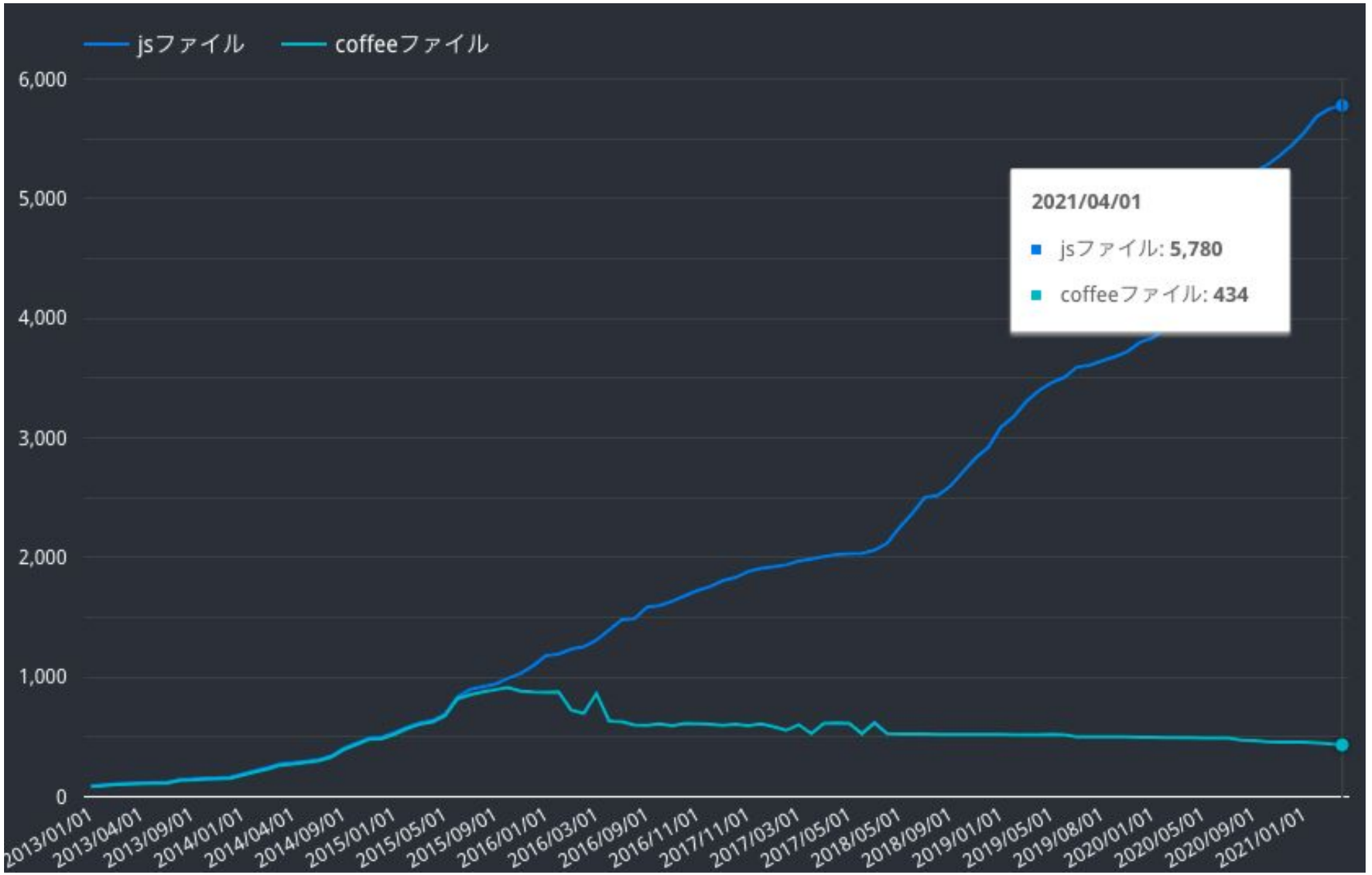
2021/03時点

テーブル数	1000くらい
Ruby ファイル数	11000くらい
JavaScript ファイル数	6000くらい
routes	4000くらい
1ヶ月のコミット数	2000くらい
1ヶ月のプルリクエスト数	1000くらい

会計freeeのRubyファイル数変遷



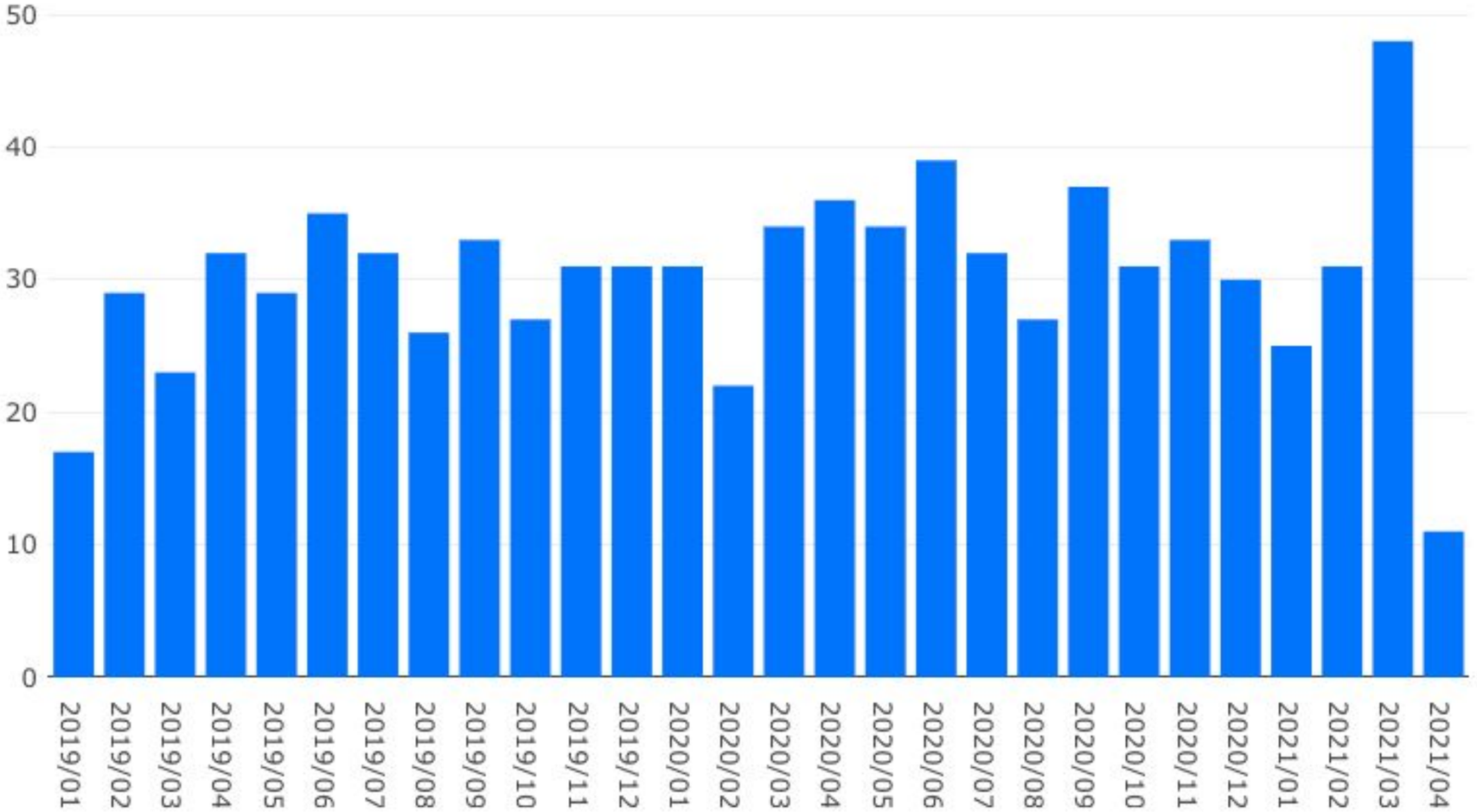
会計freeeのJSファイル数変遷



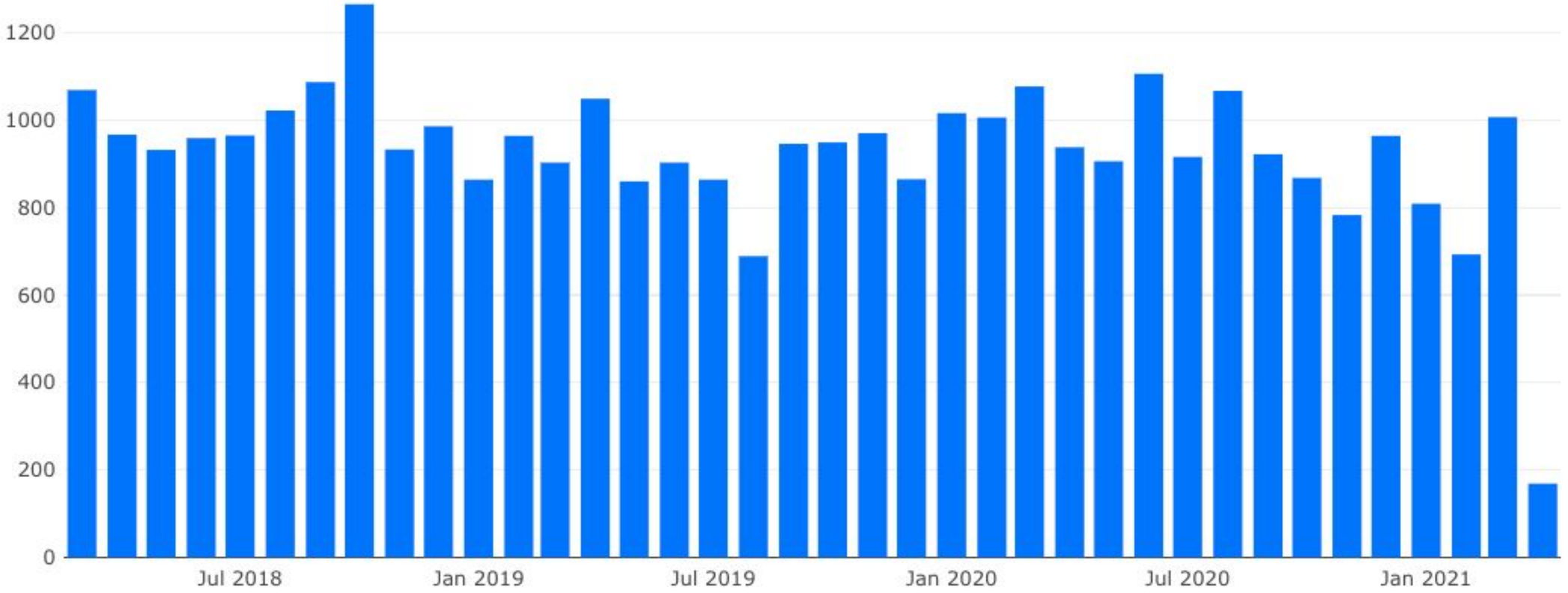
会計freeeのデータストア

- 最もレコード数の多いテーブルで15億超レコードくらい
  - auto increment idはIntegerの最大値を超えた
- 1億レコード超えてるテーブルはさっき数えたら21テーブルくらい
- ちょっと時系列グラフに出せていないのですが、毎年130%くらいのペースで増え続けています

会計freeeのデプロイ回数



会計freeeのpull request数



会計freeeの規模変遷

- コードベースはリリースから8年経った今も初期と変わらない成長ペースで増加
- データストアの規模は毎年130%ペースで増加
- 開発者は増えているがデプロイ回数は減らさずいけてる
- 開発者は増えているがPullRequest数は変わらない

起きた問題と対応事例

規模が大きくなるにつれ起きた課題

- コード規模が増えると
  - 生産性、品質の課題
    - 開発速度↓
    - 不具合&問い合わせ↑
- データ規模が増えると
  - パフォーマンスの課題
    - レスポンスタイム↓

これらの課題で具体的にやっていったことを紹介します

ちなみにRuby/Railsゆえに発生した大きな問題はありません

生産性の課題

コード量が増えると起きる問題は様々

- CI/CD
  - **テストの実行に30分かかる**
  - **デプロイするのに1時間かかる**
  - **1回のデプロイに参加する人が多すぎて動作確認取るのに数時間かかる**
- 開発
  - **社内の他サービスに似たような実装が出現しがちになる**
  - **フロントエンドがデカすぎてビルドに10分かかるので待ち時間でコーヒーを飲みに行く**
  - **新しい技術と古い技術が混在してスイッチングコストがかかる**

生産性の取り組み

組織的に継続改善しないといけないものが多い
=> 委員会制度でボトムアップに解決

- CI/CD
  - **生産性向上委員会**
- 開発
  - **マイクロサービス委員会**
  - **フロントエンド委員会**

その領域に関心が強い人たちが定期的に集まって課題出し->改善していく。
これらの委員会でToDoに決まったものはプロダクトロードマップとは別枠にある程度工数とってOK（1月以上かかるようなものはロードマップに乗せて工数取る）。

アウトプットはプラスの評価として扱うことで継続するインセンティブを作る。

品質の取り組み

- 品質改善をミッションとして取り組むチームを作った
  - **QA**
    - **デグレの削減**
      - **リグレッション**
      - **E2E**
    - **重篤度の定義**
  - **CRE（Customer Reliability Engineer）**
    - **問い合わせの削減**
      - **サポートと連携**
- 品質の見える化
  - **問い合わせを全てJIRAに集約**
  - **社内からの専用登録フォームを作って情報の過不足をなくした**
  - **チケットを分析可能にするためにRedshiftに連携**

機能開発チームとQA/CREが一緒になって品質改善に取り組むような体制に

品質の取り組み

機能開発チームでは

- 業務ドメインでチームを分ける
  - **ドメイン知識を溜めることで不具合が混入されにくくする**
    - **開発スピードも品質も上がったのでオススメ**
- どのドメインにも属さない機能の扱い
  - **チケット数が均等になるように各チームに担当を振り分ける**
    - **ドメインに閉じすぎてプロダクト全体の品質に目がいかなくなるのを防ぐ**
    - **対応難易度、発生数を見てちょくちょく振り分けは変える**
- とにかくみんなで分析したものを定点観測する
  - **見てるだけでも意識の上がる人は上がる**
  - **減っていくのがわかると品質へのモチベーションもUP**

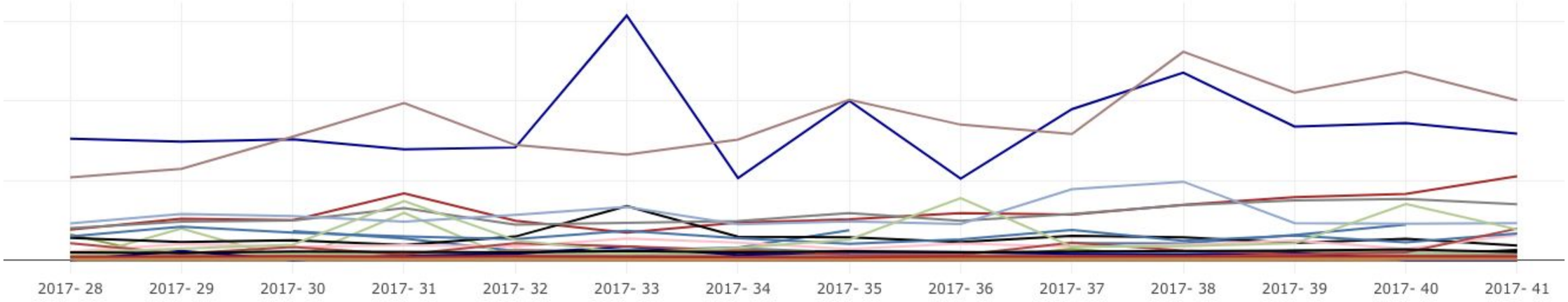
パフォーマンスの課題

データ規模によって問題が変わった

- まだ規模が小さい頃（レコード数 ~1000万くらい）
  - **N+1問題が起きがち（DBやネットワーク）**
    - **そこまで気にしなくても影響は少ない**
- 規模が大きくなってきた（レコード ~1億くらい）
  - **適当に貼ってたインデックスが猛威を振るいサービスを落としかける**
    - **実はインデックスが貼られてなかったなんてことも**
- **規模が大きくなってきた（レコード 1億~くらい）**
  - **SQLの改善だけではどうにも早くならない問題が起きる**

チューニングだけでは無理だったその1

飛び抜けて遅いエンドポイントが2つあることが分かる
その事業所の1年分のデータを集計してドリルダウンで分析可能にする機能が遅い



- これまで
  - **リアルタイムに会計データを愚直に集計**
  - **レンジが1年間なことが多いので、業種によってはものすごい数のレコードを集計する必要があった**
- 改善後
  - **トランザクションのコミットフックで事前に集計データを作るような仕組みに変更**
  - **業種によっては集計レコード数を1000倍以上圧縮**
    - **最大で15倍速**

	期首	2017-08	2017-09	2017-10	2017-11	2017-12	2018-01	2018-02
						資産の部		
▼流動資産								
▶現金	-12,000	-12,000	-12,000	-12,000	-12,000	-12,000	-12,000	-12,000
▶りそな（法人）	0	0	0	0	0	0	0	0
▶受取手形	0	0	0	0	0	-10,000	-10,000	-10,000
▶不渡手形	0	0	0	0	0	0	5,000	5,000
▶売掛金	10,000	10,000	10,000	10,000	10,000	16,080	16,080	37,680
▶仕掛品	0	0	0	0	0	0	10,000	10,000
▶立替金	56,410	56,410	56,410	56,410	56,410	56,410	56,410	56,410
仮払消費税	3,713	4,453	4,453	4,453	4,453	4,453	3,342	4,823
流動資産	**58,123**	**58,863**	**58,863**	**58,863**	**58,863**	**54,943**	**68,832**	**91,913**
▶有価証券	0	0	0	0	0	5,000	5,000	5,000
▼固定資産								

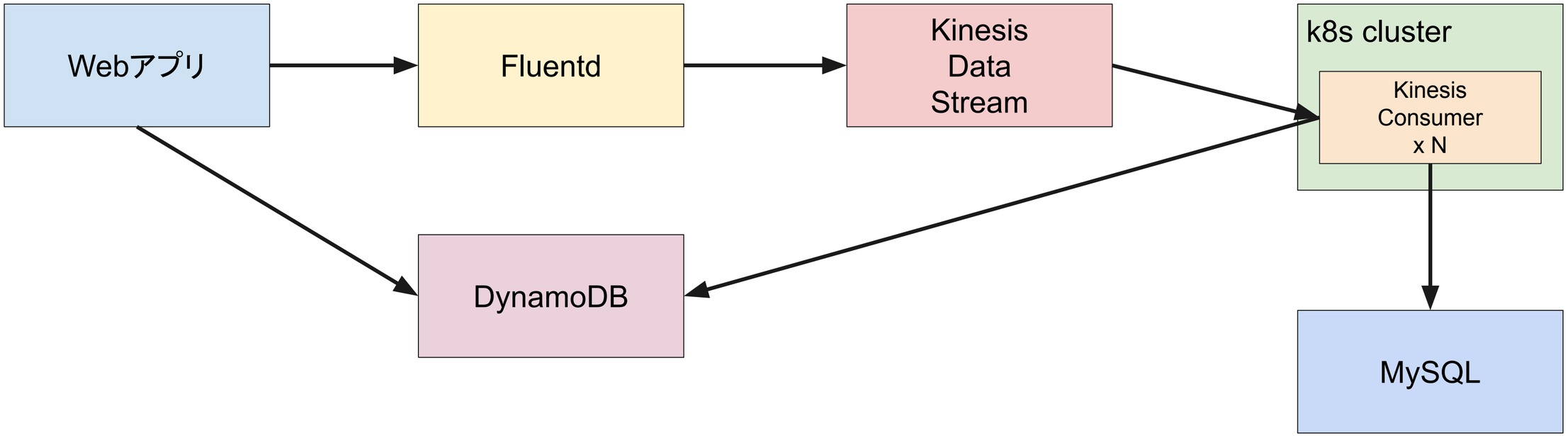
チューニングだけでは無理だったその2

トランザクションのコミットフックで集計データを作るように
=> 1トランザクションあたりの時間が伸びたことでロック待ちエラーが頻発

- 楽観ロックによるエラーが頻発
  - **整合性を取るために楽観ロックをかけていた**
  - **複数ユーザー/複数タブ/一括処理等で並列に処理が走ると、同じ集計データが更新対象となるケースは容易に起こりうる**
- ネクストキーロック+挿入インテンションギャップロックによるエラーが稀に発生
  - **業務サービスではインポート処理等の一括処理がとても多い**
  - **また、インポート処理をやり直すこともとても多い**
  - **複数のユーザーが同時多発的にdelete~insertを行うと、発生しうる**

イベントソーシングな仕組みへの乗せ替え

- 集計データをイベントソーシングな仕組みに乗せて分散させることで、ロックエラーを防ぐ
- 分散させるためのキーは事業所ごとにユニークになるようなもので振り分ける
  - **事業所内は直列処理させつつ負荷を分散させて全体を待たせないことが可能に**



ようやく平穏が訪れた
（ここまで2年かかってます）

開発と運用のバランス

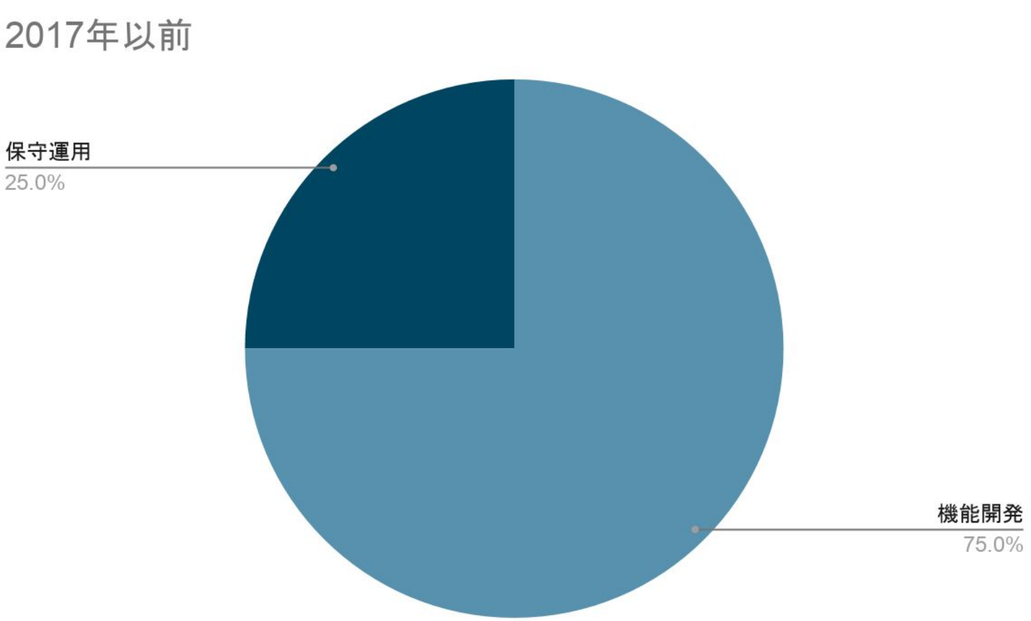
ユーザー課題と技術課題

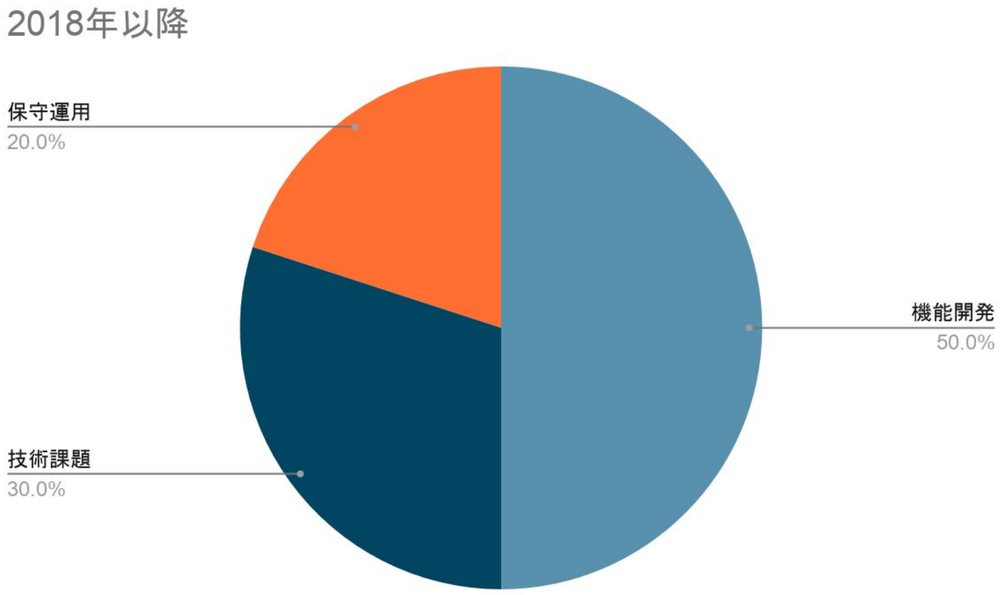
どちらかにフォーカスしすぎると、どちらかに限界がきて結果生産性が下がる。

特にスタートアップでは技術課題だけに取り組んでいるとユーザー課題の改善が
進まずにサービスが伸びなくなるリスク。
とはいえユーザー課題だけやっていると、負債も溜まるし、開発速度も落ちる。
出してしまった不具合を直す保守の工数も取らないと・・・

機能と技術と保守のバランスを取る必要がある。

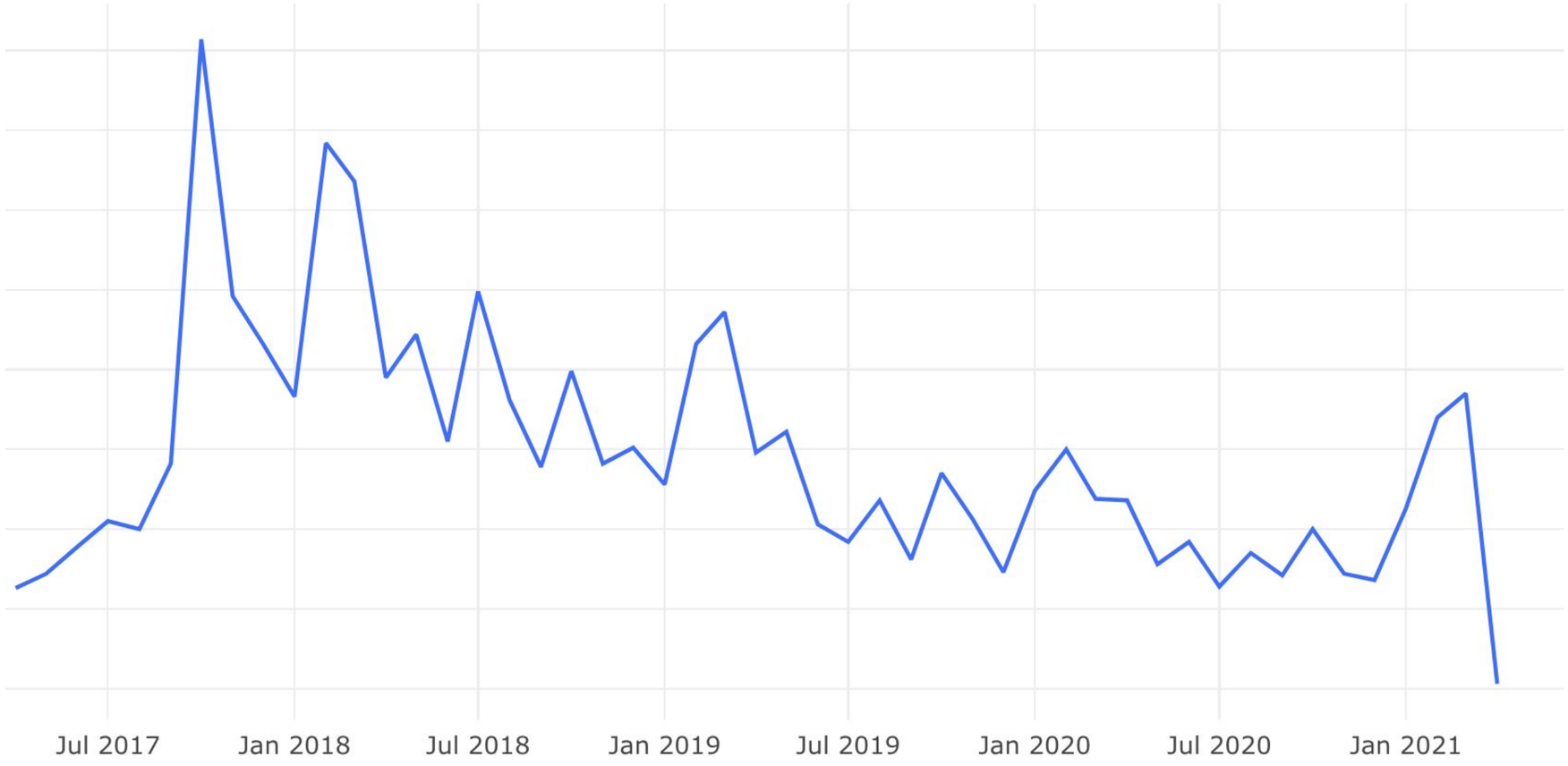
2017年くらいまでは会社のフェーズ的にもとにかく機能開発に振っていました。
2018年くらいからは毎年 技術課題の改善に3ヶ月/年くらい使っています。
（売れる状態になるまでギリギリ粘った）
保守運用は不具合を減らしていくことで25%から20%に改善





運用に一定工数を割くことで生まれる効果

4年かけて75%不具合チケットを削減。
この辺はやれば成果が出るので、減らした分 X%機能開発に割く時間が増やせるということをアピールして工数取り続けるの大事。



技術課題に取り組む工数を取るために

機能開発と同じで、技術課題に取り組むことでどういう効果があるのかを説明する必要がある。
（組織のフェーズ、風土によって説明難度は変わる）

freeeでは、全社目標に品質改善テーマを入れてもらうことで、機能開発と同等に重要なものであるという空気を出して、先に紹介したような開発起因の大きな課題にも工数を割けるようになりました。

ただ、パフォーマンスのように定量的に効果測れる課題はやりやすいのですが、大きなリファクタリングなんかは効果が測りづらいので、えいやで計画に入れて押し通すことも・・

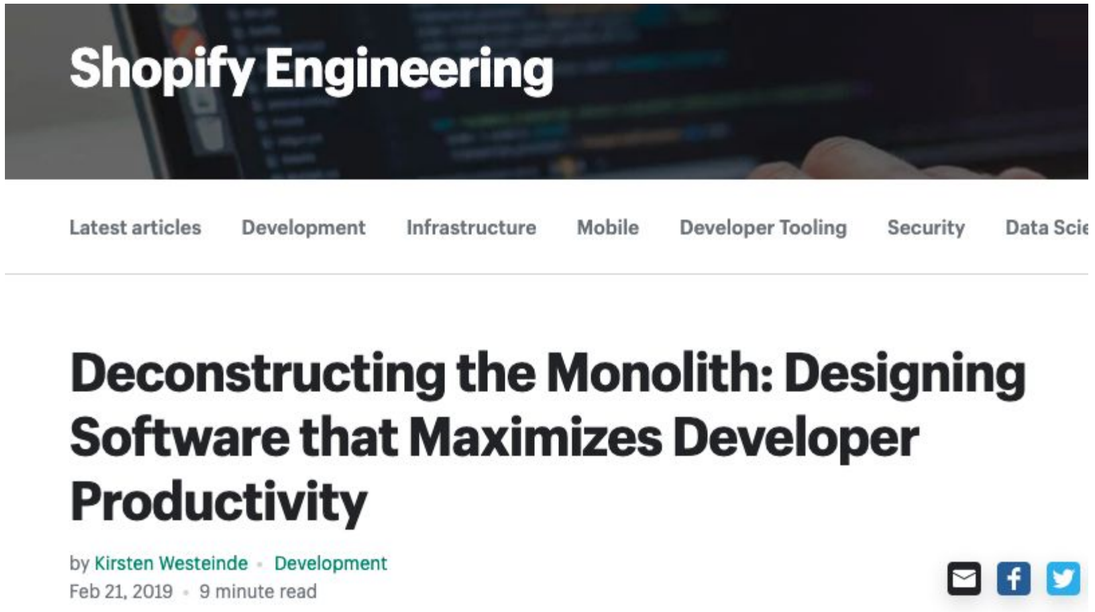
さらなる進化

あらためて継続成長に向き合う

- コード規模増加で起きつつある問題
  - **影響範囲が読みづらい**
    - **知らないコードが増える**
      - **調査が大変**
      - **過剰に慎重になってしまうバイアスがかかる**
  - **リファクタリングがしづらい**
  - **同様にAPIの変更が難しい**

モジュラーモノリスの採用

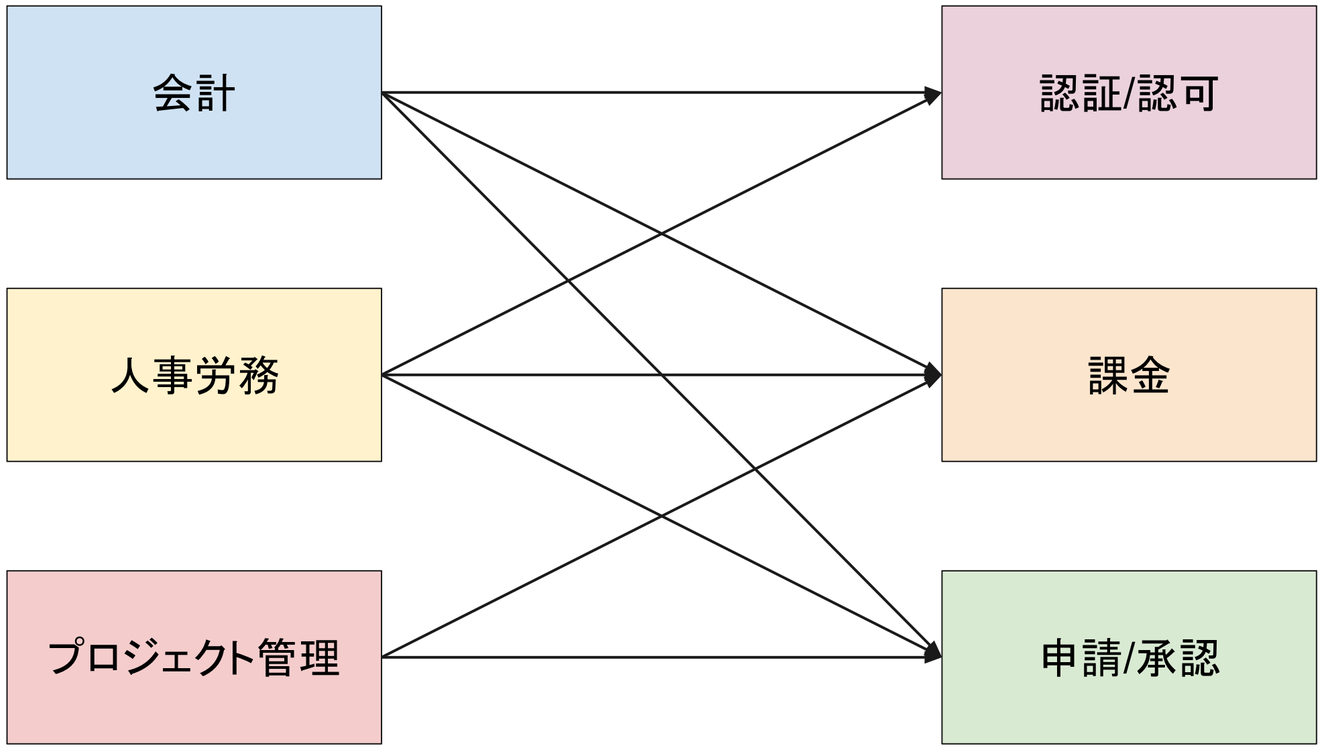
ShopifyのEngineering Blogで提唱された手法



モノリスなアプリケーション内で、ドメイン単位のモジュールに分解することで、
デプロイは単一でありつつも、モジュールごとの独立性を担保するアーキテクチャ

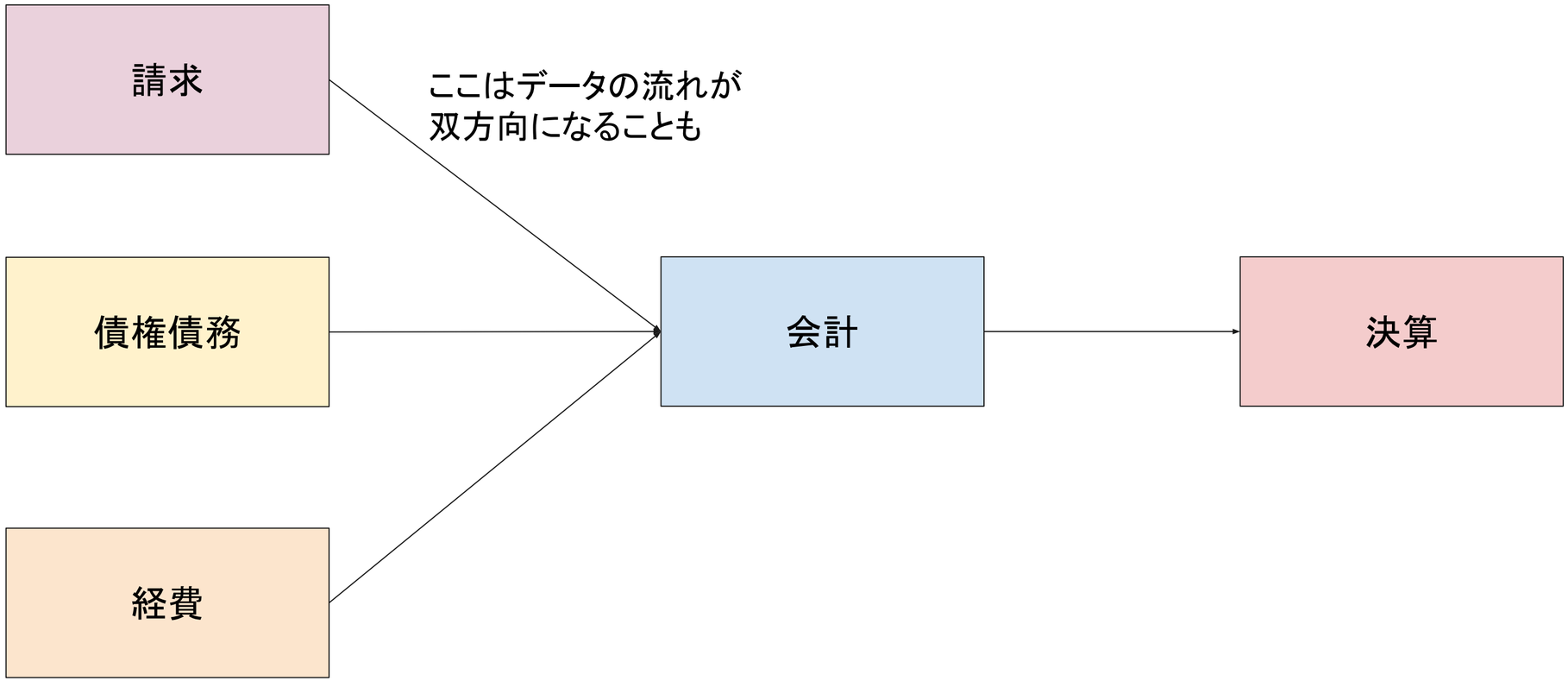
freeeでのマイクロサービス

コードを疎にしたいという理由より、複数のサービスから共通で使われるものを
切り出す方針
サービス間でメッシュになるのが理想、というイメージ



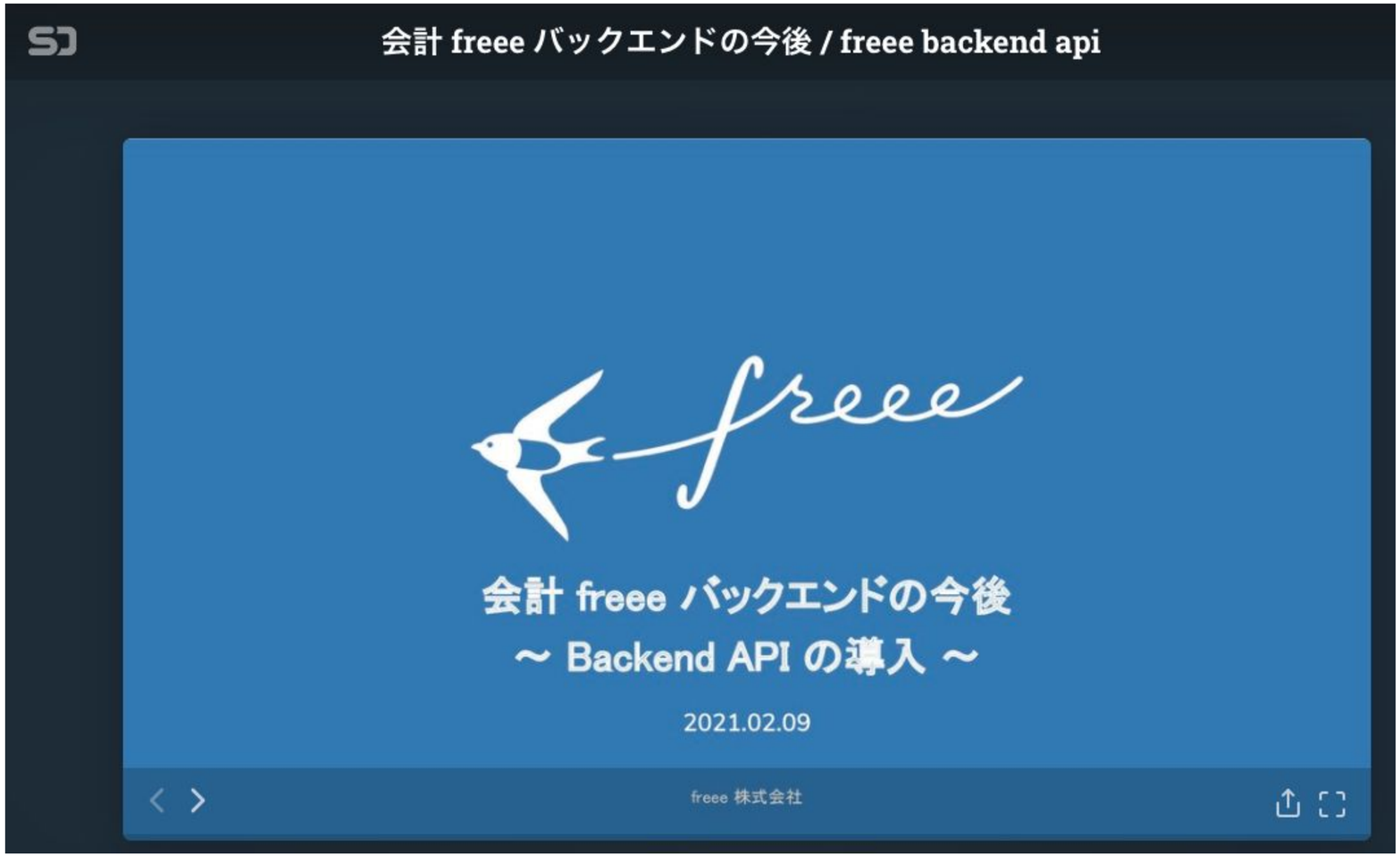
freeeでのモジュラーモノリス

サービス内の複数の業務ドメインをいい感じに管理するために採用
複数ドメインが連携して1トランザクションで両方更新することが多いため、マイクロサービス化を単純に進めると難易度が一気に上がる



freeeでのモジュラーモノリス

時間の都合上内部の詳細は弊社テックリードが発表しているスライドがSpeakerDeckから見れるので、そちらも興味があれば見てみてください



スモールビジネスを、
世界の主役に。