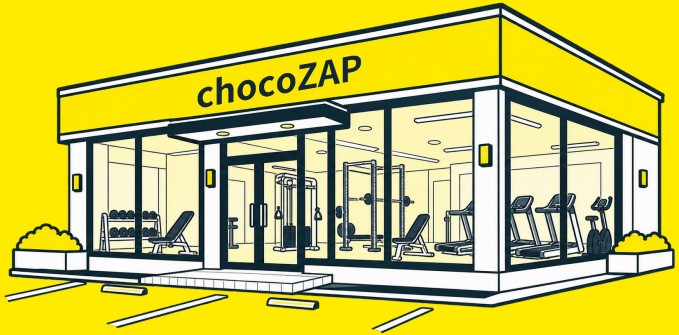
chocoZAPを支える AI活用開発術



Rubyセミナー 大阪 2026.10.29

RIZAPテクノロジーズ株式会社 梅田 智大

© RIZAP TECHNOLOGIES

RIZAPで、エンジニアをしています



梅田 智大

RIZAPテクノロジーズ バックエンドユニット長

新卒入社7年目

1〜2年目

プロテイン・サプリなどの商品開発。
ITとは無縁の世界にいました。

3年目〜現在

エンジニアに転身。
chocoZAP のシステムを開発しています。

© RIZAP TECHNOLOGIES

RIZAPが考えた、無人運営のコンビニジム。

chocoZAP

カンタン　便利　たのしい



スマホ一つで通える。

© RIZAP TECHNOLOGIES

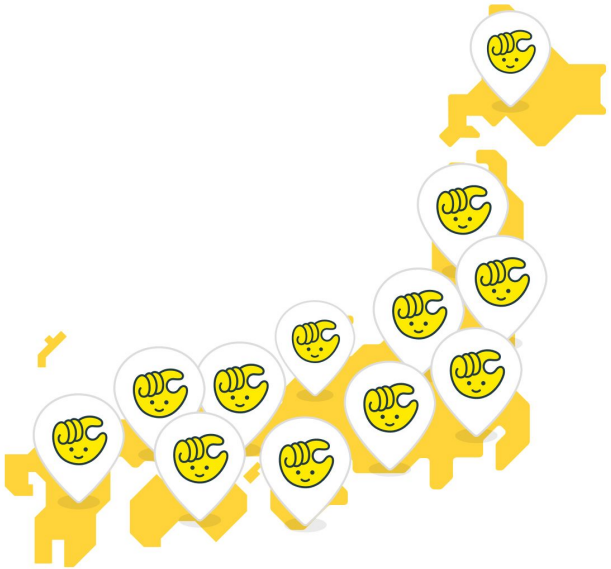
全国に2,000店舗超、会員は110万人超。

店舗数

2,000店舗超

会員数

110万人超



※ 2026年8月14日時点。店舗数は国内（FC店舗を含む、海外店舗を除く）。
出典: RIZAPグループ株式会社 2027年3月期 第1四半期 決算説明資料（2026年8月14日）

© RIZAP TECHNOLOGIES

でも実は、
1店舗の運営だけでも大変。

© RIZAP TECHNOLOGIES

店を1つ回すだけで、これだけの仕事がある。

店をきれいに保つ

- 清掃
- 消耗品の補充
- ゴミの回収
- 備品の在庫と配送
- 忘れ物の対応

マシンと設備を動かし続ける

- マシンの点検
- 故障の受付と修繕
- 空調・照明・水回りの保守
- マシンの入替・搬入
- 法定点検・届出

安全を守る

- 店内の見守り（カメラ）
- 入退館の管理
- トラブル・緊急時の対応
- 鍵・セキュリティ
- 体調不良・事故への対応

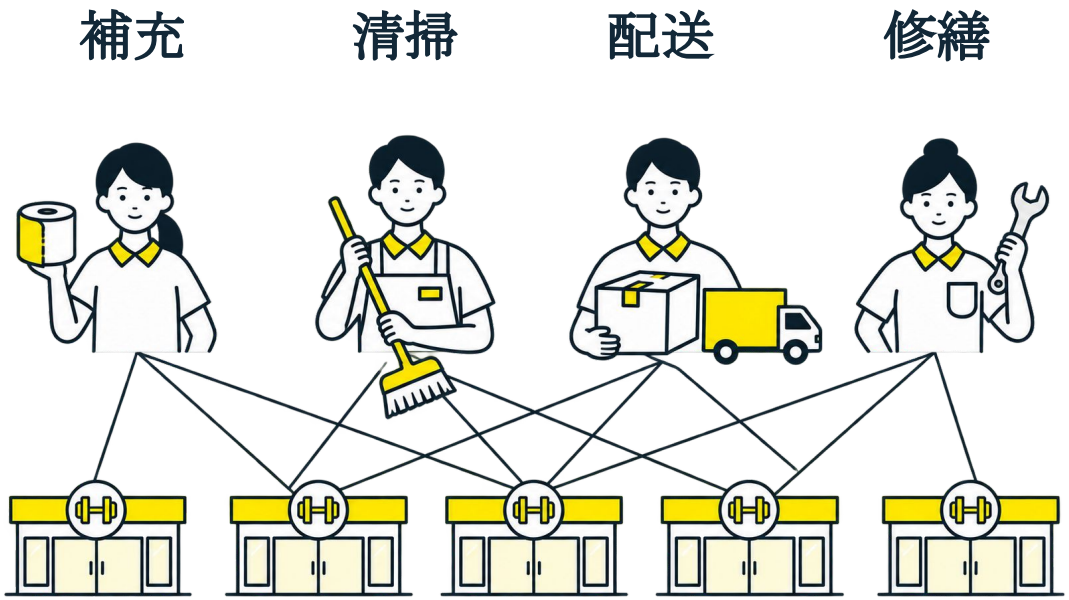
会員に伝える

- 店内のサイネージ・掲示
- マシンの使い方の案内
- キャンペーン・告知
- 営業時間・休館の案内
- 問い合わせ対応

これを、約2,000店舗で、毎日。

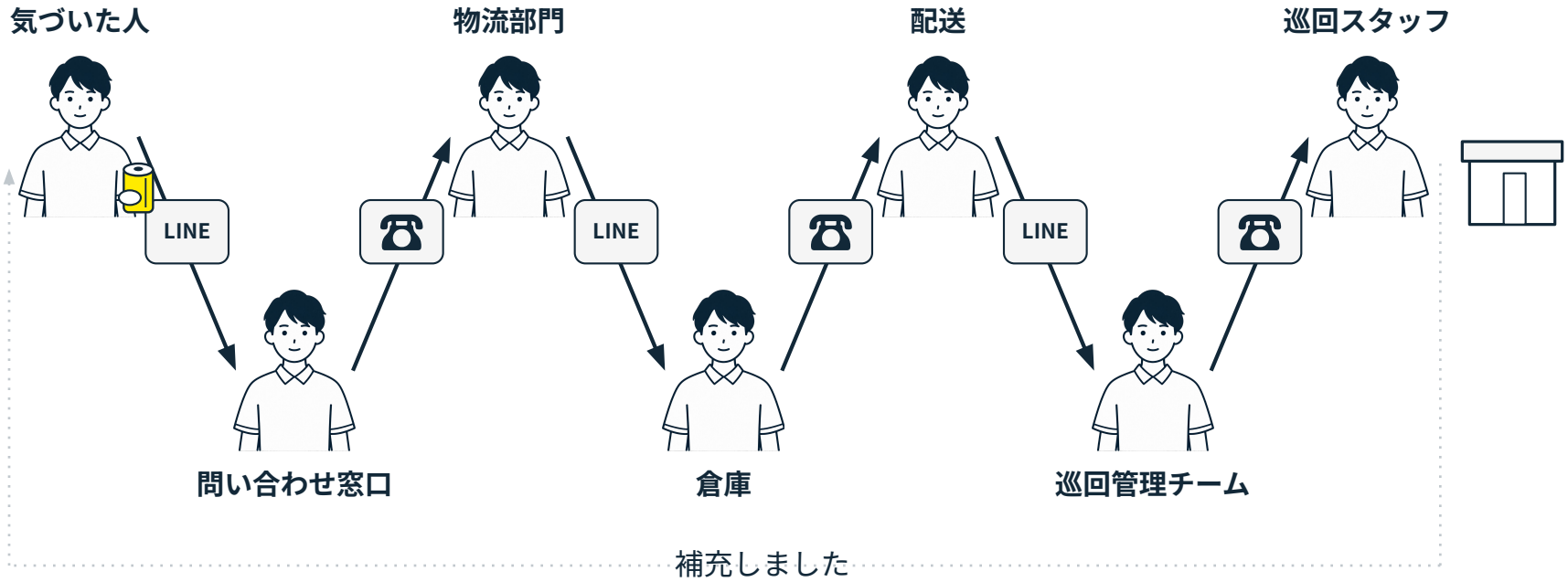
© RIZAP TECHNOLOGIES

仕事ごとに別の人が、いくつもの店舗を受け持つ。



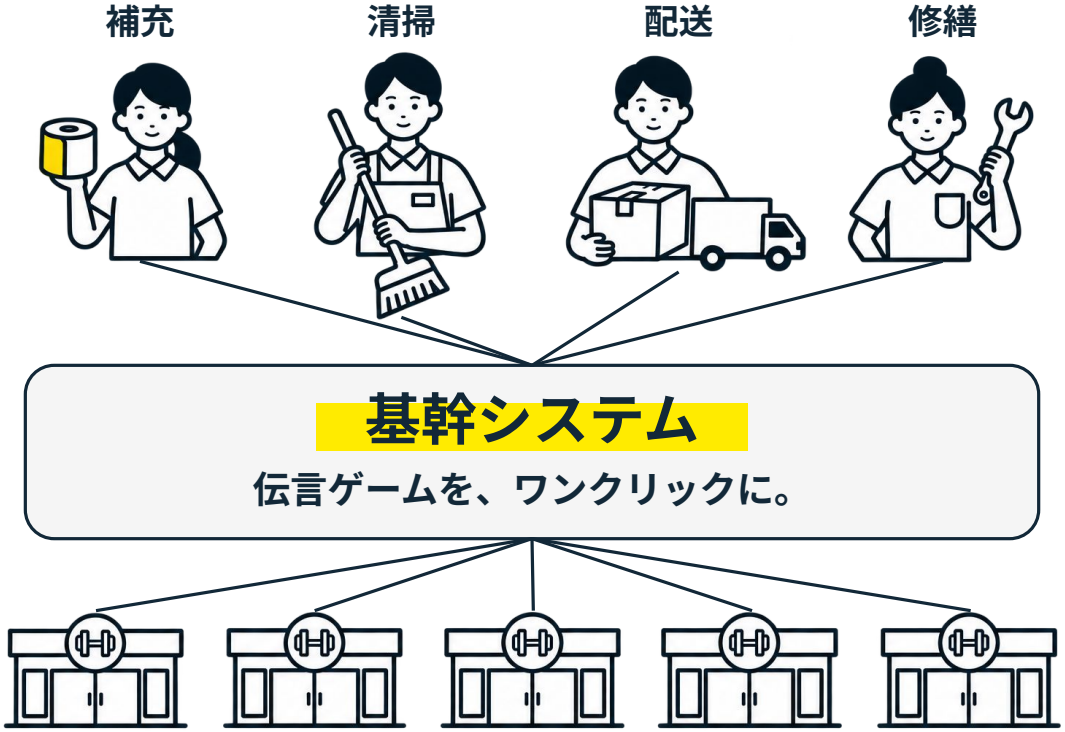
© RIZAP TECHNOLOGIES

トイレットペーパー1つ補充するにも、伝言ゲーム。



© RIZAP TECHNOLOGIES

店舗開発・運営を、テクノロジーで支える。



店舗開発・運営の基幹システム。いま、開発中。

© RIZAP TECHNOLOGIES

外注とSaaSから、社内で作る、へ。

	以前	いま
	外部	社内で作る
内製比率	23% （2022年10月）	約100% （2024年2月）

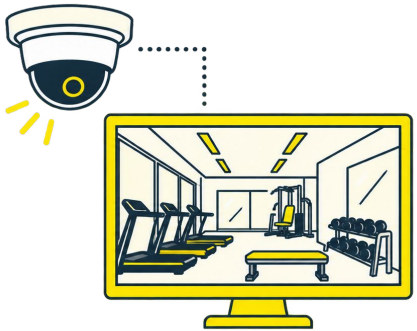
2021年まで、開発組織はゼロだった。

© RIZAP TECHNOLOGIES

社内で作った例を、2つ。

chocoCAM

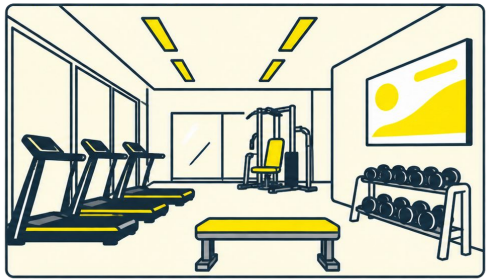
内製



店内のカメラ映像で、
安全を見守る

chocoVISION

内製



店内のサイネージに、
案内と告知を配信する

© RIZAP TECHNOLOGIES

これを、誰が作っているのか。

© RIZAP TECHNOLOGIES

Ruby バックエンドチーム

新卒3年目		3人
新卒2年目		3人
新卒1年目		3人
私		新卒7年目、エンジニア歴4年

© RIZAP TECHNOLOGIES

Ruby バックエンドチーム

新卒3年目		3人
新卒2年目		3人
新卒1年目		3人
私		新卒7年目、エンジニア歴4年

＋ AI

© RIZAP TECHNOLOGIES

内製で開発できているのは
AIの力によるところが大きいです

© RIZAP TECHNOLOGIES

とはいえ、コードを一切見ずに
AIだけではシステムは作れません

© RIZAP TECHNOLOGIES

AIがいる前提で、開発のやり方を組み直した。

エンジニア　×　AI　→　運用保守できるシステム

要件定義　設計　実装

全部の工程を、組み直した。

© RIZAP TECHNOLOGIES

AIがいる前提で、開発のやり方を組み直した。

エンジニア　×　AI　→　運用保守できるシステム

要件定義　**設計**　**実装**

全部の工程を、組み直した。

正解は、私にもまだわかりません。

懇親会で、皆さんのやり方を教えてください。

© RIZAP TECHNOLOGIES

まず、何を作るかを
決めるところから。

要件定義

設計

実装

© RIZAP TECHNOLOGIES

要件は、文章と静止画を見て、「はず」で固めていた。

要件定義書は、文章と静止画で作成



要件定義書　静止画

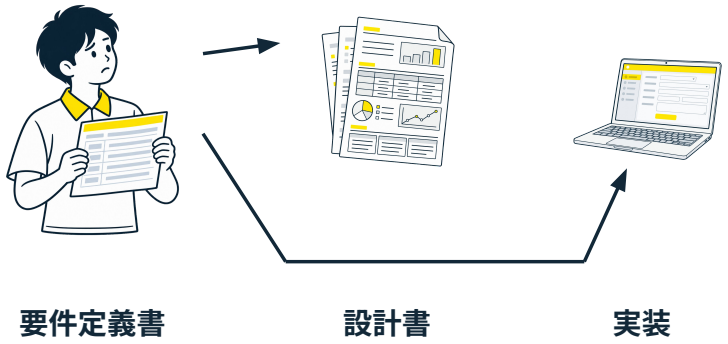
「このボタンを押したら、
こう動くはず」という想像で、
要件を固めていた

© RIZAP TECHNOLOGIES

設計と実装も、「はず」で進めていた。

エンジニアも、

要件定義書から
画面の動きを想像して、
設計・実装していた

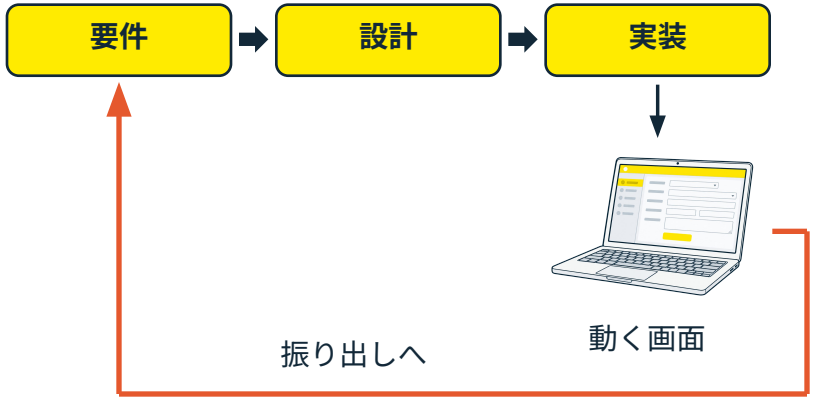


© RIZAP TECHNOLOGIES

その結果、実装ができてから、違いに気づく。

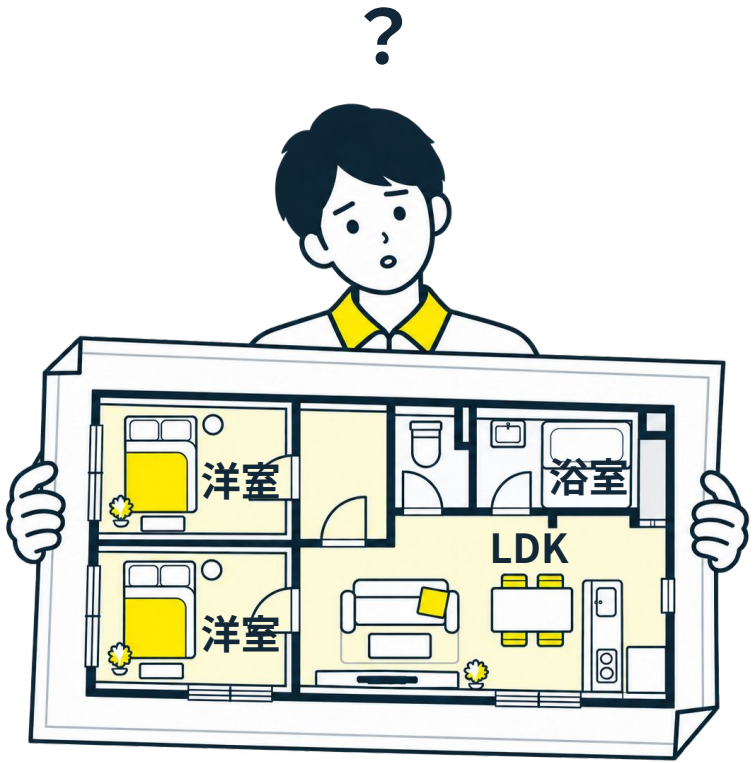
実装が終わってから、
「ここ、違くない？」と
ズレに気づく

リリース直前に、
大幅な手戻り



© RIZAP TECHNOLOGIES

間取り図だけで、家を買っているようなもの。



© RIZAP TECHNOLOGIES

AIにより、動く画面を作る
（家を建てる）コストが、
ほぼゼロになった。

© RIZAP TECHNOLOGIES

要件定義書は、動くコードから書く。

以前



要件定義書

書いてから、作る



実装

© RIZAP TECHNOLOGIES

要件定義書は、動くコードから書く。

以前



書いてから、作る





要件定義書

実装

いま



作ってから、書く



要件定義書

実装

文書より先に、動くコードがある。

© RIZAP TECHNOLOGIES

いきなり、エンジニアがプロトタイプを作る。



企画書



AI



動くプロトタイプ

画面も API も、エンジニア1人で

コードは、一行も読まない。バイブコーディング。

© RIZAP TECHNOLOGIES

動く画面を触ると、会話が弾む。



会議は、全部文字起こしする。

© RIZAP TECHNOLOGIES

「会議の宿題、やっといて」

AI への指示は、これだけ。

© RIZAP TECHNOLOGIES

文字起こしを、丸投げする。



文字起こし

そのまま渡す

AI

会議の30分後



直ったプロトタイプ

文字起こしには、**何を・なぜ・どう直すか**が詰まっている

↓

だから、そのまま渡すと、AI は議論のとおりに直してくれる

© RIZAP TECHNOLOGIES

要件は、触って固めて、最後に書き起こす。





AI



いきなり、プロトタイプを作る

エンジニア1人が、AI で

動く画面を触って、議論する

全員で

「会議の宿題、やっといて」

文字起こしを、AI に

プロトタイプをもとに、要件定義書を書き起こす

AI が書く

© RIZAP TECHNOLOGIES

AIも、「はず」では書けない。

自然言語だけで頼む

こういう画面にしたい
こういう機能にしたい

文章の指示

AI



要件定義書

「はず」だらけの要件定義書

解釈がブレる。指示が漏れる。

© RIZAP TECHNOLOGIES

AIも、「はず」では書けない。

自然言語だけで頼む

こういう画面にしたい
こういう機能にしたい

文章の指示

→ AI →



要件定義書

「はず」だらけの要件定義書

解釈がブレる。指示が漏れる。

プロトタイプを渡して頼む



動くコード

→ **AI** →



要件定義書

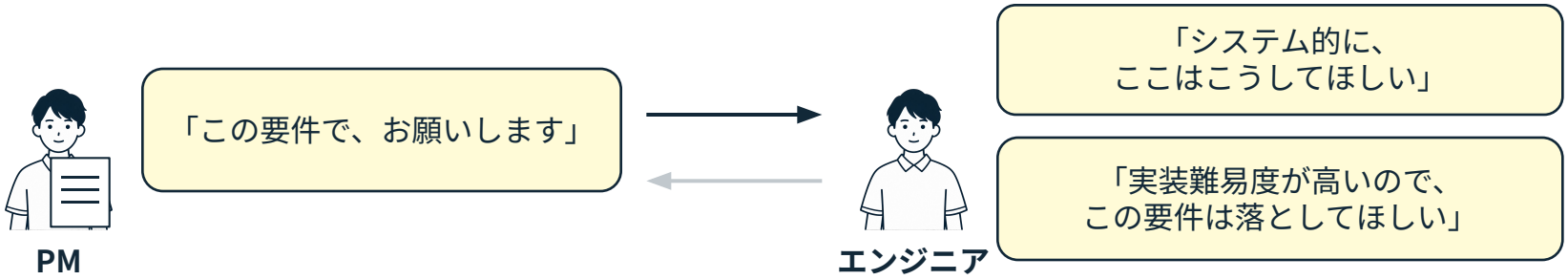
正確な要件定義書

どこを押すと、どう動くか。
どの条件で、どう動くか。
コードから、AIが100%読み取れる。

© RIZAP TECHNOLOGIES

システムの観点が、最初から要件に入る。

以前



設計・実装の段階で、要件に戻るラリー。

© RIZAP TECHNOLOGIES

システムの観点が、最初から要件に入る。

以前



PM

「この要件で、お願いします」



エンジニア

「システム的に、
ここはこうしてほしい」

「実装難易度が高いので、
この要件は落としてほしい」

設計・実装の段階で、要件に戻るラリー。

いま



エンジニア



プロトタイプ



要件定義書

作ってみたら、作れなかった

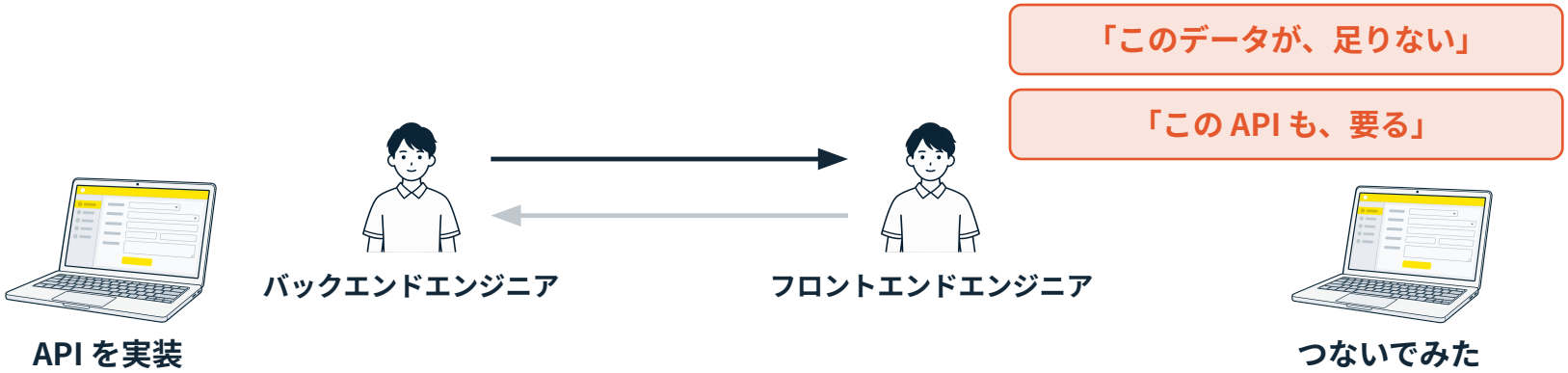
作ってみたら、性能が出なかった

プロトタイプが、検証を兼ねる。

© RIZAP TECHNOLOGIES

どの画面に、どのデータが要るかが、確定する。

以前



つないでから分かって、やりとりが発生する。

© RIZAP TECHNOLOGIES

どの画面に、どのデータが要るかが、確定する。

以前



API を実装



バックエンドエンジニア　フロントエンドエンジニア

「このデータが、足りない」

「この API も、要る」



つないでみた

つないでから分かって、やりとりが発生する。

いま



動くプロトタイプ

どの画面に、どのデータと API が要るかが、明確になる

© RIZAP TECHNOLOGIES

動く画面から、具体的な要件ができた。



動くプロトタイプ

触って議論・検証する

利用者が求める動き

システムとしての実現性

画面に必要なデータとAPI

確認した内容をAIが整理



要件定義書

動作・条件・必要なデータが、具体的に書かれる。

要件が具体的だから、設計・実装で迷いにくい。

© RIZAP TECHNOLOGIES

次は、固めた要件を
設計に落とす話。

要件定義

設計

実装

© RIZAP TECHNOLOGIES

設計書は大事。
でも、書けていない。

© RIZAP TECHNOLOGIES

設計書はメンテされず、実装とずれていく。



設計書 v1.0

=



実装 v1.0

最初は、合っている。

© RIZAP TECHNOLOGIES

設計書はメンテされず、実装とずれていく。



設計書 v1.0

≠

エンハンス

バグ修正

仕様変更



実装 v1.3

設計書と実装が次第にズレる。
設計書と実装のどちらが正しいのかわからなくなる。

© RIZAP TECHNOLOGIES

AI で書くと、設計書が無限に膨らむ。

AI



要件定義書

ユーザー一覧API 設計書

概要

管理画面のユーザー一覧で、名前・メール・状態・最終ログインを表示するために使用する API。ユーザー・所属・ログイン履歴のデータを返す。

リクエストパラメータ

レスポンス

© RIZAP TECHNOLOGIES

AI で書くと、設計書が無限に膨らむ。

AI



要件定義書

文章量が多すぎて、
大事な箇所が分からない

v1.1 → v1.8

ユーザー一覧API 設計書

概要

管理画面のユーザー一覧で、名前・メール・状態・最終ログインを表示するために使用する API。ユーザー・所属・ログイン履歴のデータを返す。

リクエストパラメータ

レスポンス

エンハンスのたびに AI が追記

© RIZAP TECHNOLOGIES

RIZAP TECHNOLOGIES

設計書の読み手は、
人から AI に変わった。

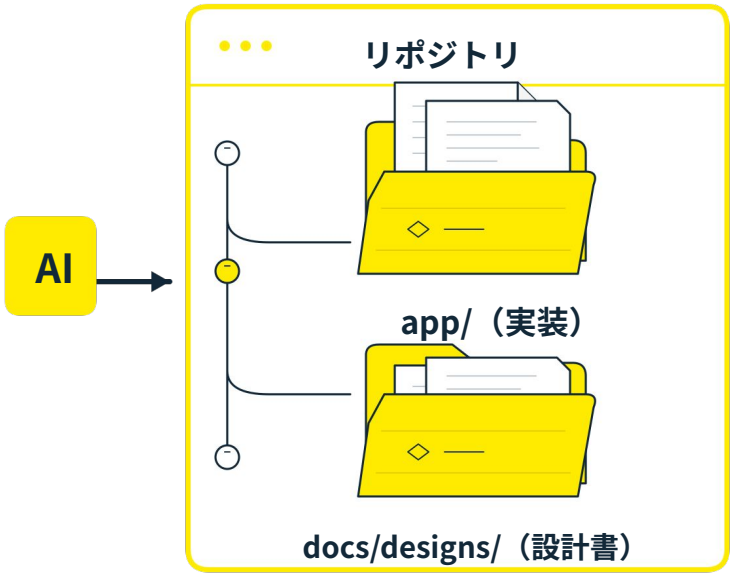
© RIZAP TECHNOLOGIES

設計書の読み手は、
人から AI に変わった。

設計書は、AI が読める場所に、
100% 読める形で書く。

© RIZAP TECHNOLOGIES

設計書は、実装と同じリポジトリに置く。



設計書も実装も、
同じリポジトリで Git 管理する

AI が迷わず読める。
設計書と実装のズレも、
PR で見える

© RIZAP TECHNOLOGIES

API 設計書は、OpenAPI で書く。

```yaml
summary: ユーザー一覧API
description: 条件に合うユーザーを返す。
parameters:
  - name: name
    in: query
    description: ユーザー名の部分一致で絞り込む
    schema: { type: string }
responses:
  "200":
    schema:
      additionalProperties: false
      required: [users, pagination]
      properties:
        users:
          items:
            properties:
              status:
                enum: [active, suspended]
```

設計書として、API としての契約（パラメータ・レスポンス・型・必須・取りうる値）が決まれば十分

決まった仕様で書くから、AIの解釈がブレない。

© RIZAP TECHNOLOGIES

OpenAPI は、AI が一瞬で書く。書くコストが、かからない。

AI

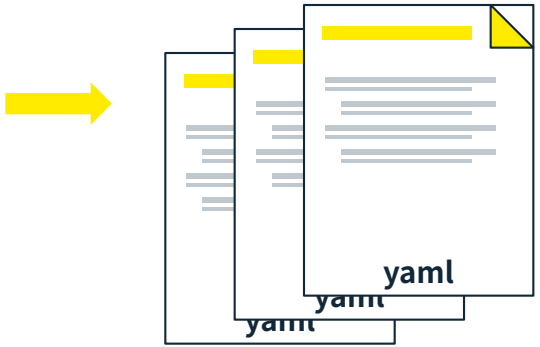




要件定義書

プロトタイプ（動くコード）

このプロトタイプと要件定義書から、
API の契約を起こして



以前: 人が手書き。書くのも直すのも、苦労した

いま: AI が一瞬で書く。設計レビューで直すときも、すぐ直る

© RIZAP TECHNOLOGIES

OpenAPI を起点に、バックエンドとフロントエンドが同時に開発する。



バックエンドエンジニア
OpenAPI どおりに実装する

OpenAPI

フロントエンドエンジニア
モックサーバーで画面を作る

OpenAPI が決まった日に、両方が作り始める

↓

フロントエンドは、バックエンドの実装を待たない

© RIZAP TECHNOLOGIES

文章に書くのは、OpenAPI に書けないことだけ。

```
app/controllers/admin/v1/
  users_controller.rb
docs/designs/admin/v1/
  users/
    index.yaml    ← 設計書（API の入出力）
    index.md      ← 判定と理由（必要なときだけ）
```

ディレクトリは、Controller と同じ構成。1アクションに1ファイル

OpenAPI に書けないことだけ、同じ場所に md で残す。**必要最低限**

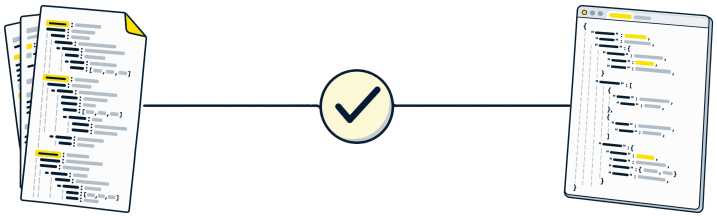
md がある = 特殊な処理をしている API。人の目が、大事な箇所にいく

© RIZAP TECHNOLOGIES

設計書と実装が一致することを、自動テストで担保する。

```ruby
test "ユーザー一覧APIが設計書通りのレスポンスを返すこと" do
  get admin_v1_users_path, as: :json

  assert_schema_conform(200)
end
```



設計書　　実装

committee で、
設計書（OpenAPI）と
API のレスポンスが
一致することを、
自動テストする

**設計書と実装が
一致していないと、
CI が落ちて、PR が
マージできなくなる**

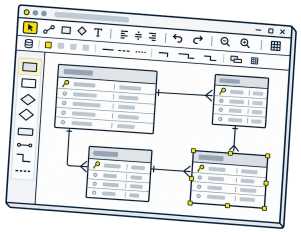
不一致は機械が検知する。だから AI が、自分で直す。

© RIZAP TECHNOLOGIES

テーブル設計書は、スキーマファイルで書く。

以前





テーブル定義書を、文章と表で書く

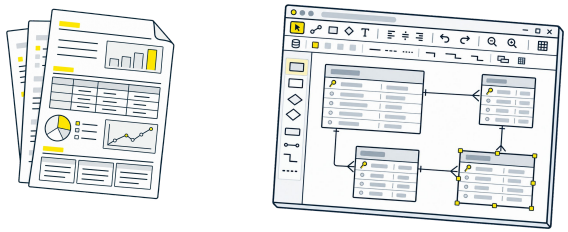
ER 図を、作図ツールで人が描く

テーブル定義書は、一致しているかの確認も大変
ER 図はメンテし続けるのも大変

© RIZAP TECHNOLOGIES

テーブル設計書は、スキーマファイルで書く。

以前



テーブル定義書を、文章と表で書く

ER 図を、作図ツールで人が描く

テーブル定義書は、一致しているかの確認も大変
ER 図はメンテし続けるのも大変

いま



設計書 = スキーマファイル（DDL）

形式が決まっているから、
AI がブレずに読める
実装との一致を機械的に判定できる

© RIZAP TECHNOLOGIES

ER 図は、スキーマファイルから AI が HTML で起こす。



スキーマファイル

機械的に起こす

AI

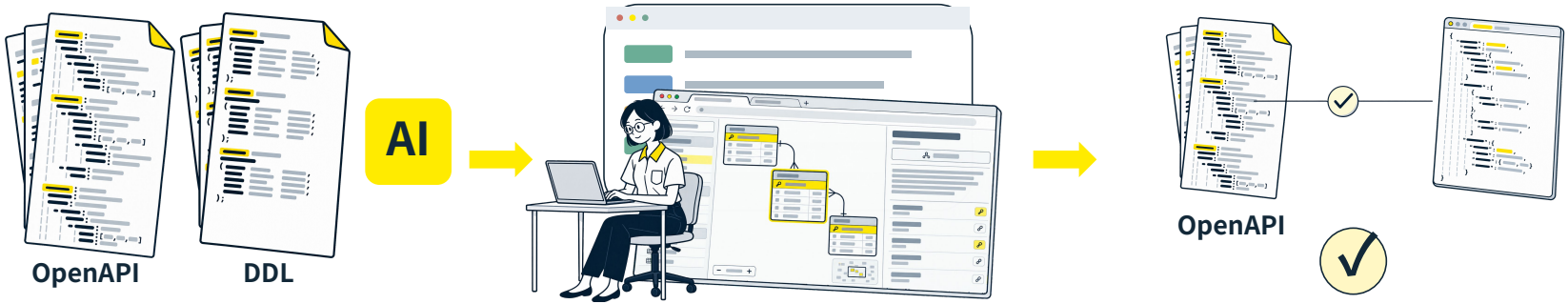


スキーマファイルは、AI が 100% 読める。
だから ER 図を起こすのは、解釈の要らない機械的な作業

書くのはスキーマ、見るのは HTML。更新したら、起こし直すだけ

© RIZAP TECHNOLOGIES

設計書は、AI が読み、人が見て、機械が守る。



AI が読む

API 設計書は OpenAPI、
テーブル設計書は
スキーマファイルで書く。
実装と同じリポジトリに置く

人が見る

Swagger や HTML の
見やすい画面にして、
レビューする

機械が守る

設計書と実装の一致を、
CI で担保する

だから、設計書が腐らない。膨らまない。

© RIZAP TECHNOLOGIES

設計書を、そのまま AI に渡して実装する。

要件定義

設計

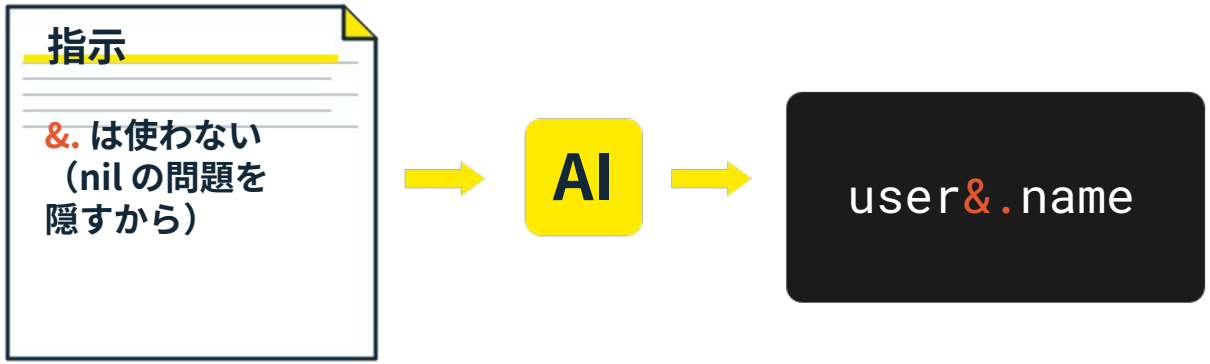
実装

© RIZAP TECHNOLOGIES

AI が、指示どおりに
コードを書いてくれない。

© RIZAP TECHNOLOGIES

指示しても、油断すると、すぐ使い始める。



AI は、世の中に多い
書き方を出しやすい。

© RIZAP TECHNOLOGIES

指示しても、油断すると、すぐ使い始める。

指示

&. は使わない
（nil の問題を
隠すから）

→ AI →

```
user&.name
```

AI は、世の中に多い
書き方を出しやすい。



毎回、同じ指摘

油断すると、すぐ使い始める

日本語の指示は、人間のレビュアーにしか効かない。

© RIZAP TECHNOLOGIES

ガイドラインの読み手も、
人から AI に変わった。

© RIZAP TECHNOLOGIES

ガイドラインの読み手も、
人から AI に変わった。

ガイドラインは、
AI が 100% 守る形で渡す。

© RIZAP TECHNOLOGIES

ガイドラインの1行を、カスタム Cop にする。

Ruby のクラス1つ

```ruby
# == 禁止理由
# ぼっち演算子（`&.`）は例外処理の握りつぶしであり、
# 意図しない不具合の混入やメンテナンス性を損ないます。
# `nil` が返る可能性がある場合は、事前に `nil` チェックを行うか、
# オブジェクトが `nil` にならないよう設計を見直してください。
#
# @example NG
#   user&.name
#   order&.items&.first&.price
#
# @example OK
#   user.name if user  # nil チェックを明示的に行う
#   user.name          # nil にならないことが保証されている
class NoSafeNavigation < Base
  def on_csend(node)
    add_offense(node, message: MSG)
  end
end
```

ガイドライン

ぼっち演算子（safe navigation）の利用禁止

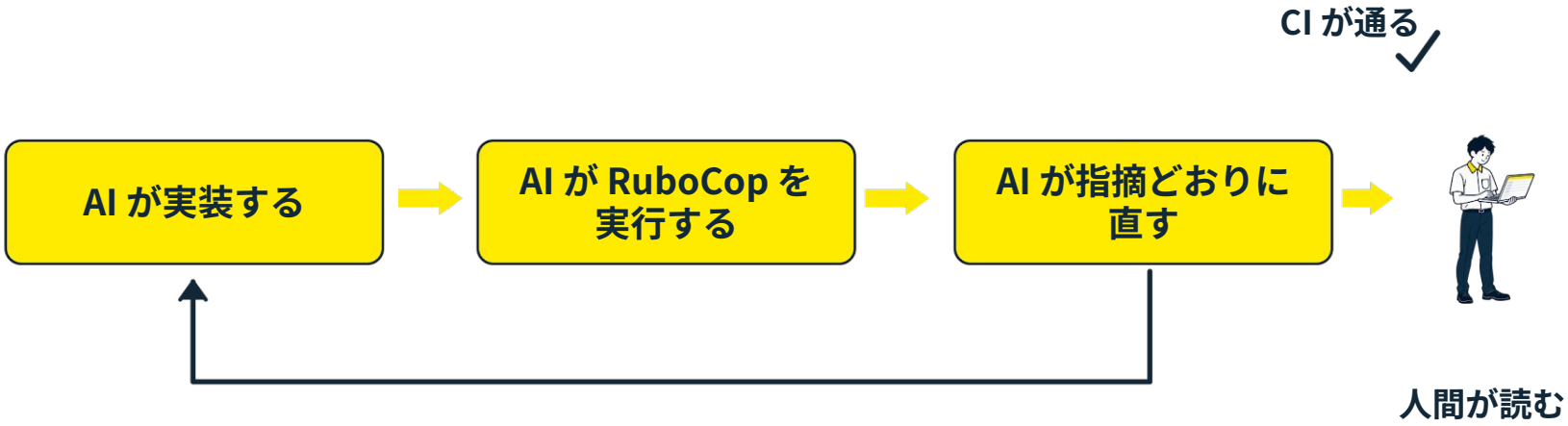
ガイドラインの1行を、1つの Cop にする

↓

なぜダメか、何が NG で何が OK かまで、Cop に書く

© RIZAP TECHNOLOGIES

ガイドライン違反は、AI が自分で見つけて、自分で直す。



ガイドライン違反は、RuboCop が検出する

↓

AI は、ガイドラインに違反したコードを出してこない

© RIZAP TECHNOLOGIES

Cop を作るコツは、指摘に理由と直し方を書くこと。

NG

理由

直し方

```
app/models/user.rb:12:7: C:
Rizap/NoSafeNavigation:
ぼっち演算子（`&.`）を使用しないでください。
例外処理の握りつぶしであり、
意図しない不具合の混入やメンテナンス性を損なうため、
`nil` チェックを明示的に行ってください
（Ruby 開発ガイドライン 禁止事項「ぼっち演算子（safe
navigation）の利用禁止」）。

    user&.name
    ^^^^^^^^^^
```

→ AI →

```
user.name if user
```

AI が直したコード

NG だけ伝えると、AI は直し方に迷う

↓

理由と直し方まで伝えると、AI の修正の精度が上がる

© RIZAP TECHNOLOGIES

Cop は、守らせる手段にすぎない。
大事なのは、ガイドラインをどう作るか。

© RIZAP TECHNOLOGIES

ガイドラインは、日々の開発の中で生まれる。

この書き方で実装したら、
バグが出た。
だから、この書き方はやめよう



ルール

この機能の作り方、散々議論した。
次に同じ機能を作るとき、
同じ議論を繰り返さないように、
このケースはこう実装する、と残そう



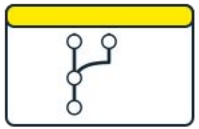
マニュアル

誰かが考え、誰かと議論して、決めたこと。

© RIZAP TECHNOLOGIES

議論の中身を、実装と同じリポジトリに残す。

レビューは対面。終わったら、文字起こしも同じリポジトリで管理する



PR を作る

実装者



対面レビュー

ビデオ会議で。
文字起こし中



**議事録 md に
書き起こす**

AI



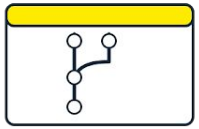
**同じリポジトリに
コミット**

PR からリンク

© RIZAP TECHNOLOGIES

議論の中身を、実装と同じリポジトリに残す。

レビューは対面。終わったら、文字起こしも同じリポジトリで管理する



PR を作る

実装者



対面レビュー

ビデオ会議で。
文字起こし中



議事録 md に書き起こす

AI



同じリポジトリにコミット

PR からリンク



コミット

どこを、どう実装し、
どう直したか

履歴



対面レビューの文字起こし

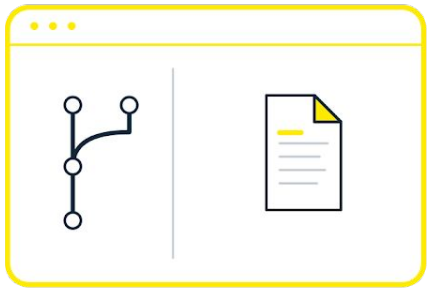
どんな議論の末、
その実装に至ったか

背景

実装の履歴と、議論の背景が、同じリポジトリに揃う。

© RIZAP TECHNOLOGIES

材料は、全部揃っている。あとは、定期的に AI に頼むだけ。

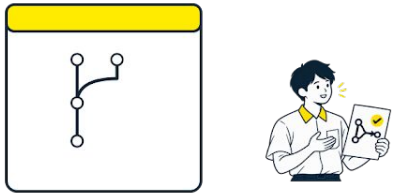


実装の履歴 ＋ 議論の背景

AI

定期的に

レビューの記録から、新しい
ガイドライン・Cop を作って



新しい Cop とガイドラインの PR

採用するかを決めて、
approve

AI が記録を読んで、ガイドラインと Cop の PR を作る

↓

人間は、PR を見て採用を決めるだけ

© RIZAP TECHNOLOGIES

個人の知見が、チーム全体の知見になる。



あるレビューで、誰かが指摘



Cop になる



エンジニア全員

AI

次からは、全員に
同じ指摘が届く

誰かが一度した指摘が、次からは Cop から全員に届く

↓

理由と直し方まで届くから、新人の教材にもなる

人間が一度言ったことは、二度言わなくていい。

© RIZAP TECHNOLOGIES

まとめ

© RIZAP TECHNOLOGIES

AI を前提に、開発の考え方を根本から変えた。

	人のやり方のまま	AI を前提に
要件定義	要件定義書を書いてから、作る	**作ってから、要件定義書を書く**
設計	人が読む文章で、設計書を書く	**AI が 100% 読める形で、設計書を書く**
実装	人が AI に、都度指示して直させる	**機械が指摘し、 AI が自分で判断して直す**

© RIZAP TECHNOLOGIES

AI を前提に、開発の考え方を根本から変えた。

	人のやり方のまま	AI を前提に
要件定義	要件定義書を書いてから、作る	**作ってから、要件定義書を書く**
設計	人が読む文章で、設計書を書く	**AI が 100% 読める形で、設計書を書く**
実装	人が AI に、都度指示して直させる	**機械が指摘し、 AI が自分で判断して直す**

人のやり方のまま AI にやらせても、人以上の成果は出ない

↓

AI が最大限の力を出せるように、やり方そのものを組み直す。それが、AI ネイティブ。

© RIZAP TECHNOLOGIES