

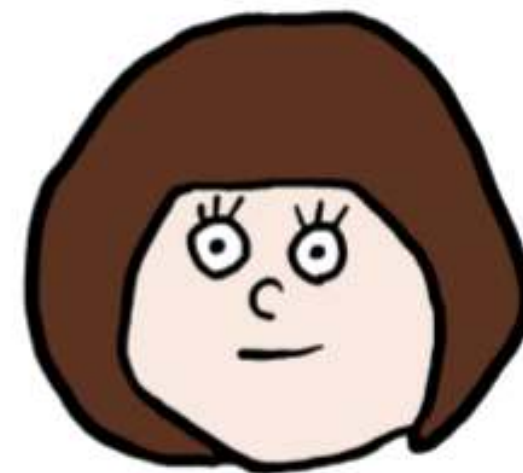
全コードレビューコメントを AI に分析させて 学びを得てみる

2025/06/13 noren.ts

@yoshiko_pg

よしこ [@yoshiko_pg](https://zenn.dev/yoshiko)

株式会社ナレッジワークのフロントエンドエンジニア
AIが好きでZennでAI関連の記事書いてます



<https://zenn.dev/yoshiko>

- 過去のレビューをどうやって分析させたか？
- マクロな分析
- ミクロな分析
- レビューコメントから得られた学び

過去のレビューをどうやって分析させたか？

GitHub API で取得ができます

rate limit に引っかからないようにじっくり取ってくる必要があります
reviewer でフィルタができないので、取得後に絞ります

2020年7月から現在までの私のレビューコメント:
4798件

json に対象ファイルパスが含まれているので `*.ts` と `*.tsx` だけに絞る

(今回のテーマが **React, TypeScript のコードレビュー** なので！)

結果:

4798件 → 3506 件 (73%)

30 件ずつ Gemini に送り、そのコメントの種類を分類してもらいます

各コメントを「指摘(A)」「質問(B)」「感想・その他(C)」のいずれかに分類してください。

[ヒント]

- 「？」で終わっていても指摘の場合あり。「～～。どうですか？」や「～かも？」など「？」を外したときに指摘と分類できるものは指摘に分類
- 「～と思いました」「～気がします」は感想ではなく指摘に分類
- 「おなじく」など他のコメントの参照は「感想・その他」に分類

[出力形式]

index ラベル 冒頭文字（改行区切り）

[出力例]

0 A e

1 B F

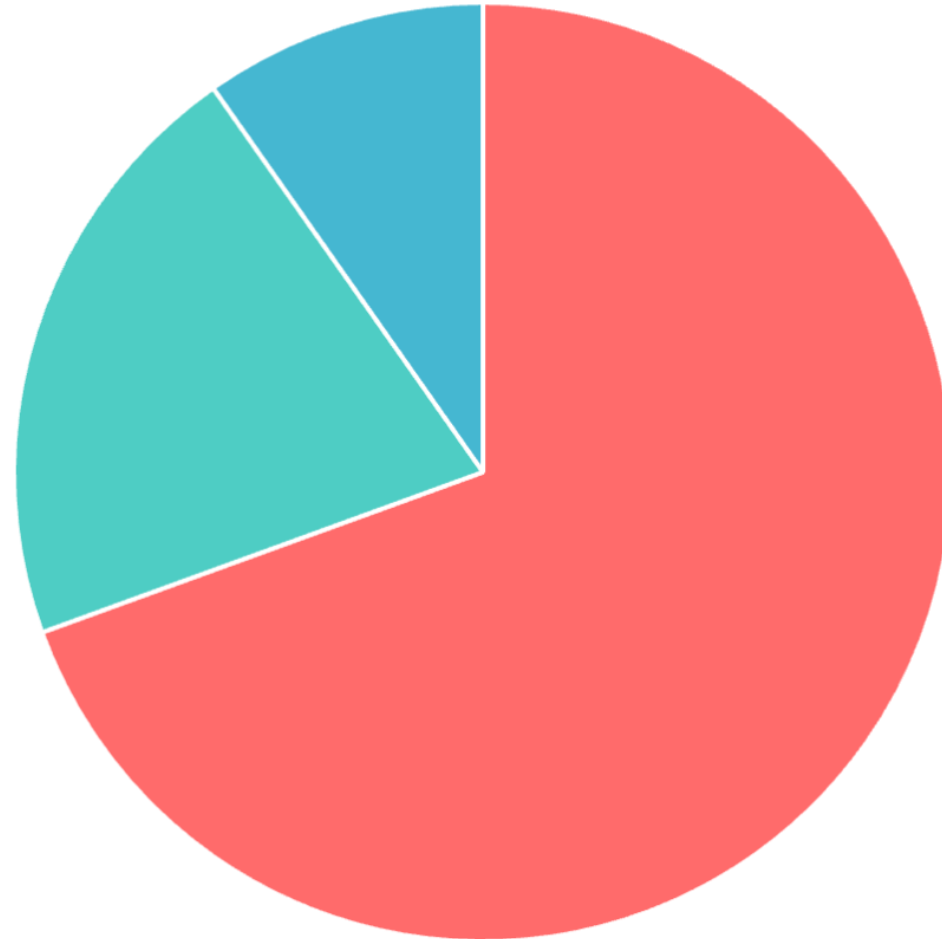
2 C (

[コメント本文]

{joined_comments}

全 3506 件中

- 指摘: 2435件 (69%)
- 感想その他: 729件 (21%)
- 質問: 342件 (10%)



マクロな分析

どんな種類のレビューがあったのか、ざっくり分布を見てみよう

指摘コメントだけを対象に、embedding してクラスタリングしてみます
コードコメントと、対象コードの diff 数行を含めます

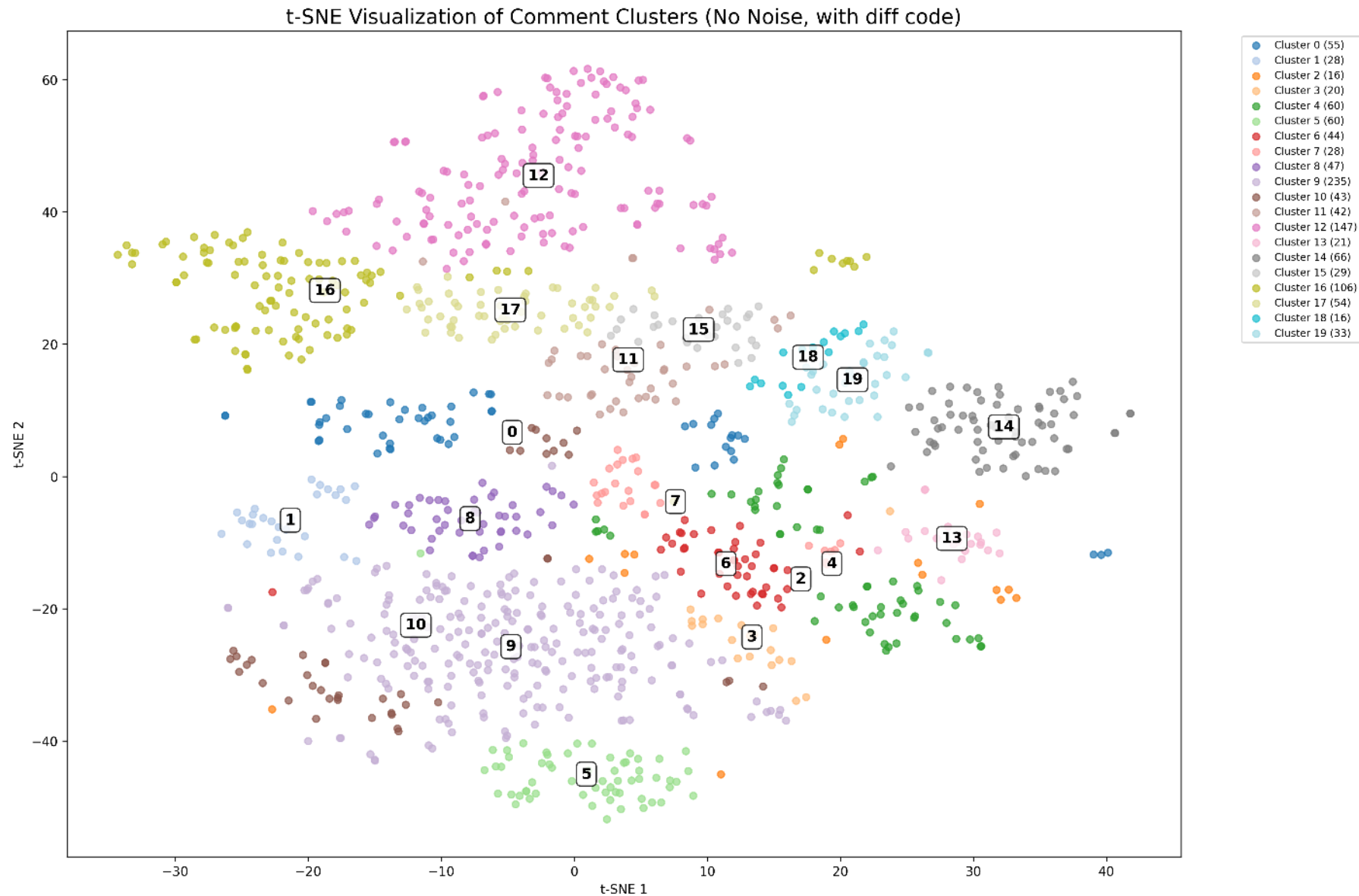
HDBSCAN という、クラスタ数を指定せずに自動決定されるやり方でやってみます

(全部 o3 & Claude Code がやってくれました)

結果:

ノイズ率 53%

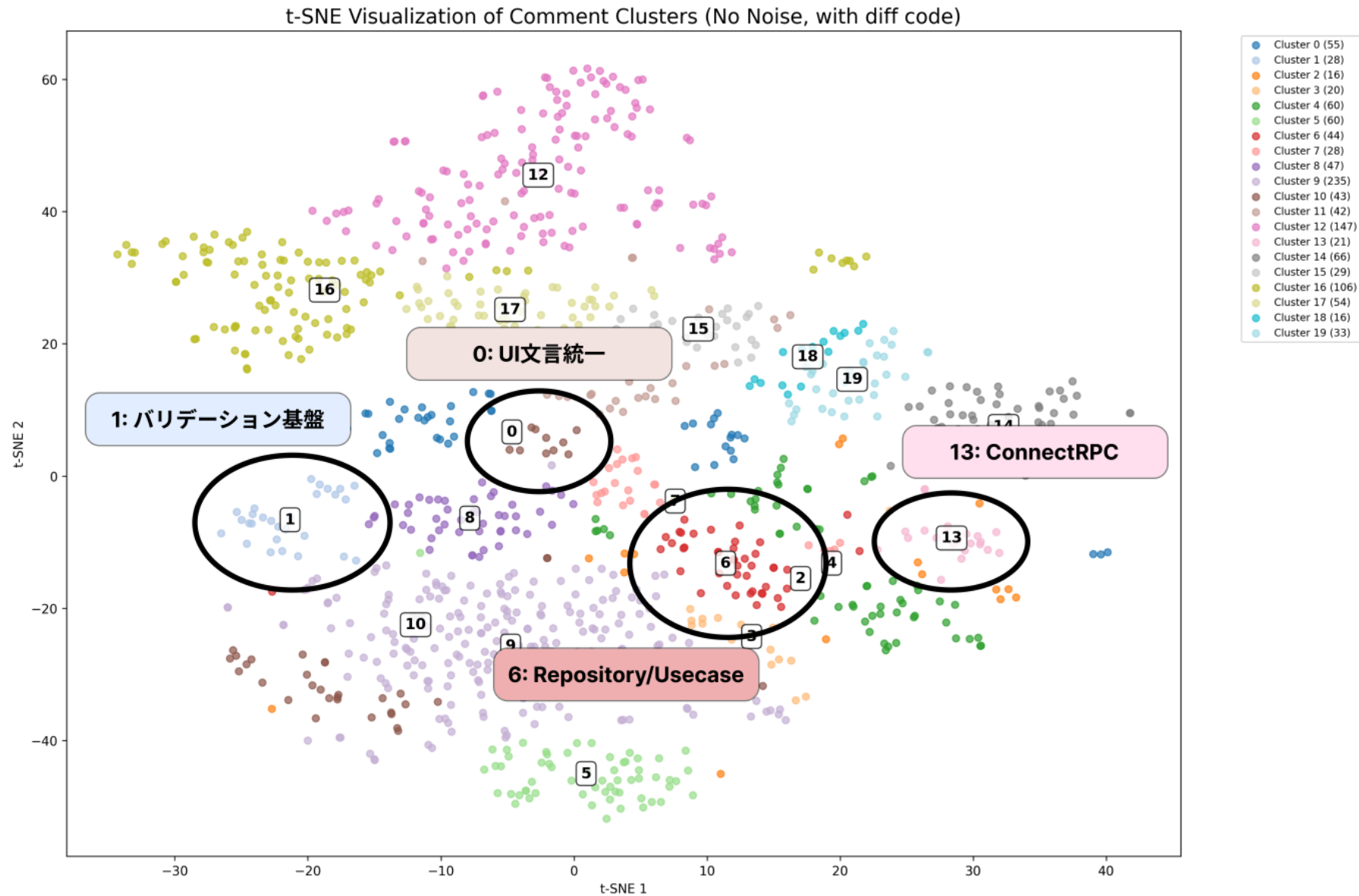
ノイズ除く1150件から 21クラスタが抽出できた！

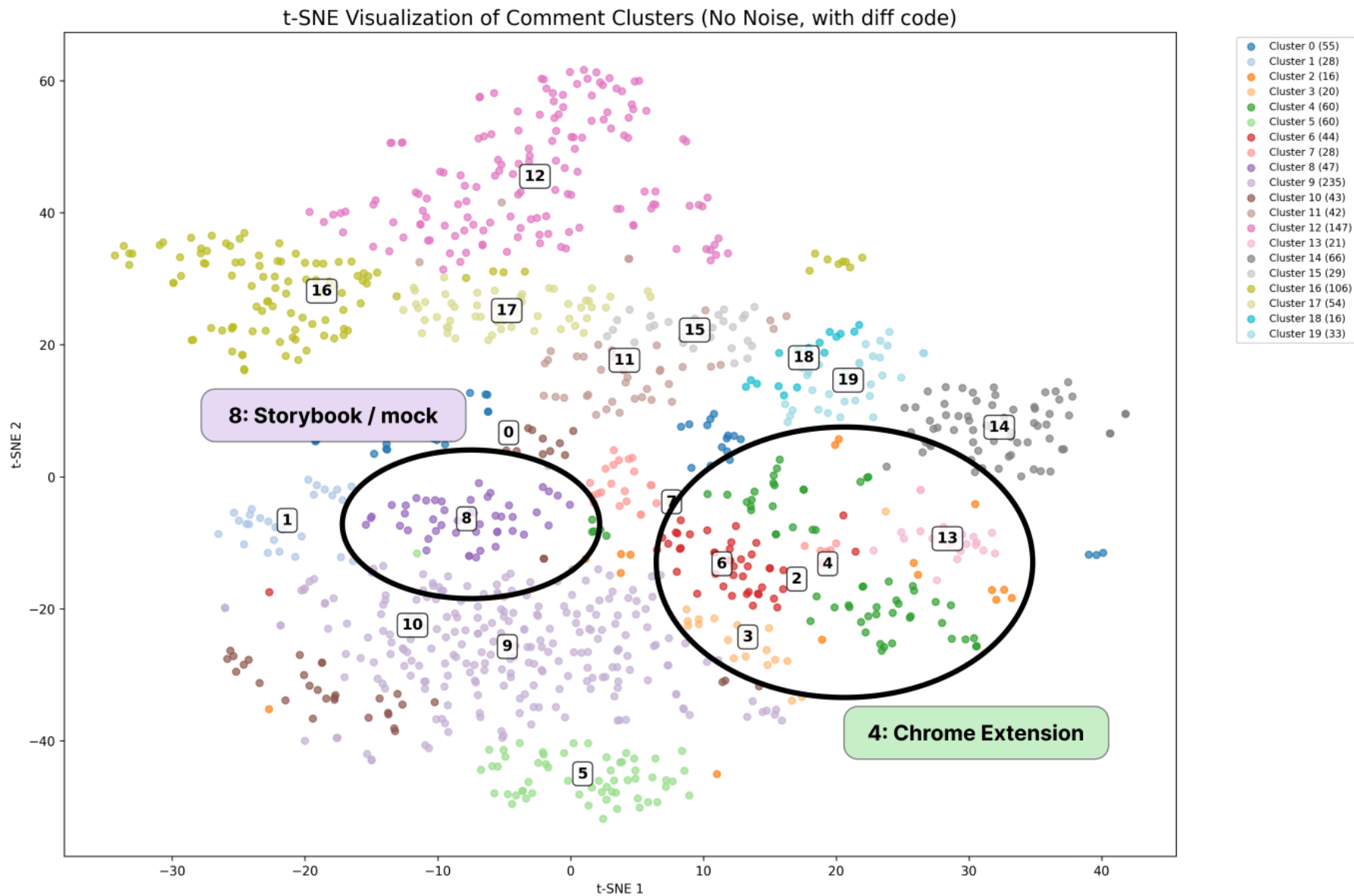


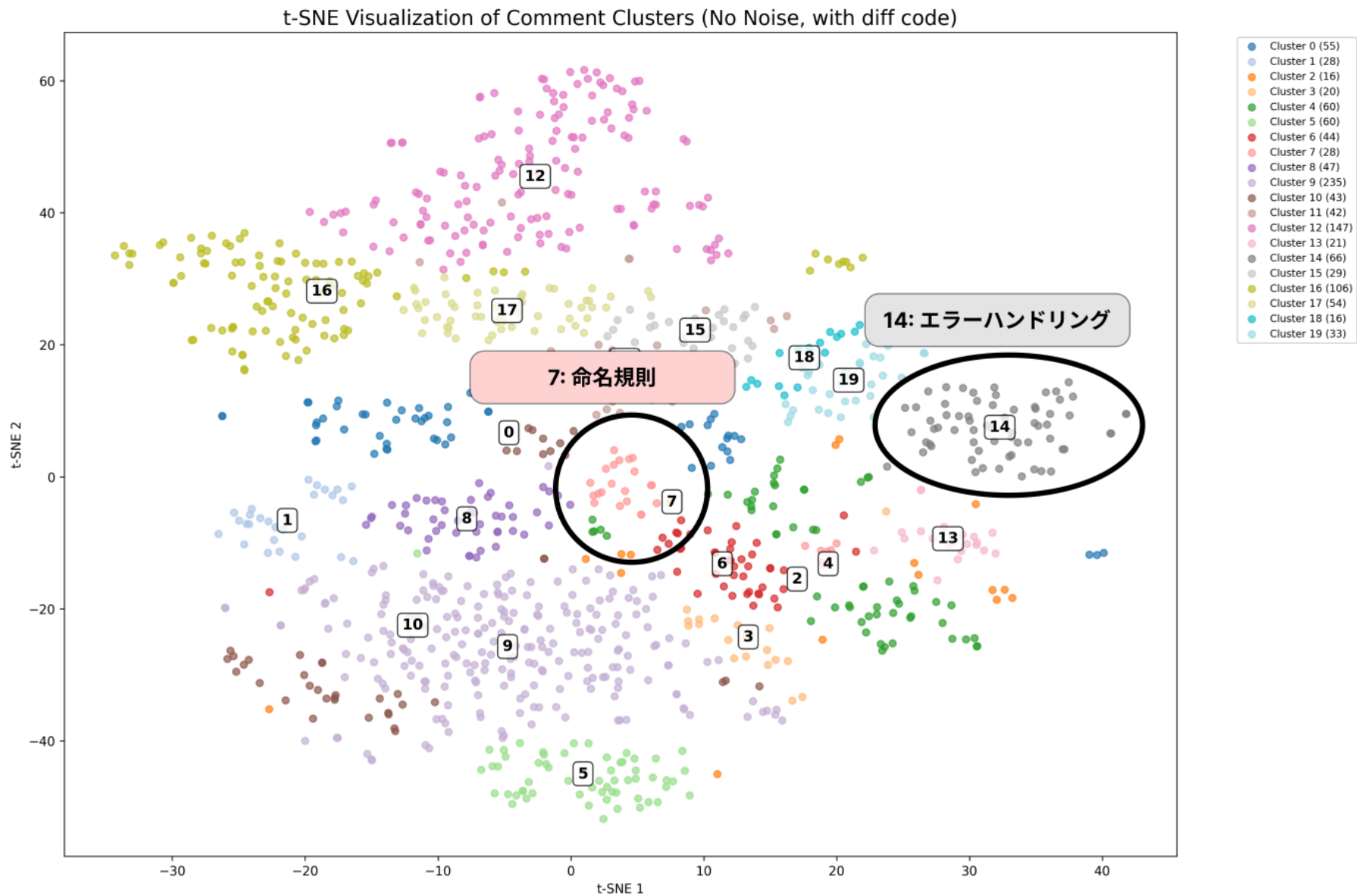
クラスタごとに o3 に生データを渡し
テーマとテーマの凝集度を判定させてみました
凝集度が一定以上高かったものを次に紹介します



うーん、評価 ★★★★★☆ 以上はあるかなー







1. 基盤整備: #1, #6, #13, #19
— バリデーション・リポジトリ境界・RPC クライアント・モデル必須化で土台作り
2. ユーザー体験: #2, #4, #10, #14, #15, #17
— 認証 UX、拡張機能、デザインシステム、エラー表示、一覧 UI など
3. 保守性向上: #5, #9, #12, #16
— 細部リファクタや命名統一など地味な指摘で保守性を保つ

ミクロな分析

具体的にどんなレビューがあったのか、学びになりそうなものを見よう

ひとつずつ見るの？



ソース



+ 追加

🔍 検索

すべてのソースを選択



cluster_0.md



cluster_1.md



cluster_10.md



cluster_11.md



cluster_12.md



cluster_13.md



cluster_14.md



cluster_15.md



cluster_16.md



cluster_17.md



チャット

_KNOWLEDGE WORK



Codebase Refinement and Best Practices

20 ソース

提供されたテキストは、ソフトウェア開発におけるコードレビューのコメントを集めたものです。これらのコメントは、機能の実装、UI/UXの改善、コードの品質（命名規則、型定義、リファクタリング）、テストとStorybookの拡充、パフォーマンスの最適化、エラーハンドリング、そして既存のFigmaデザインへの準拠といった多岐にわたる側面をカバーしています。特に、コンポーネントの再利用性や責務の明確化、データモデルの堅牢性に関する議論が多く見られます。コメントには、具体的な修正提案や技術的なアドバイス、今後の課題や改善点も含まれています。

メモに保存



メモを追加



音声概要



マインドマップ

入力を開始します...

20 個のソース

19

異なるコードベースのセクション間で、どのようにコンポーネント、ロジック、およびスタイルが再利用され、共通化され



型安全性を上げるために有益と思われるレビューを抽出してください

型安全性を向上させるために有益と思われるレビューは以下の通りです。

型定義・型設計の改善 1 ...

- Enumの代わりにstring union型を使用することを推奨しています。これにより、値が部分的なEnumとして扱われることを避け、より厳密な型チェックが可能になります 1。
- union型とexhaustiveチェック（assertNeverなど）を組み合わせることで、全てのケースが網羅的に処理されていることを保証し、将来的な型追加時のバグを防ぎます 2 >< 51 67。
- 定数から型を生成することで、型が漏れることなく、全ての値が網羅的に列挙されていることを確認できます 3 52。
- modelではnullableなプロパティをrequiredとして扱い、converterでnull値の処理を吸収することを基本としています。これにより、モデルの型定義がシンプルかつ厳密になります 5 ...。
- converter内でパーミッション判定ロジックをカプセル化し、frontend modelには結果のbooleanフラグ（例: canDownloadKnowledge: boolean）を持たせることで、関心事を分離し、型の意図を明確にします 4。
- あり得ない組み合わせを持つ複数のプロパティ（例: infiniteAmount: boolean と maxAmount: number）を避け、単一の数値プロパティ（例: maxAmount: number で Infinity を無制限として扱う）で状態を表現することで、不正な状態を型レベルで防止します 11。

型安全性を上げるために有益と思われるレビューを抽出してください

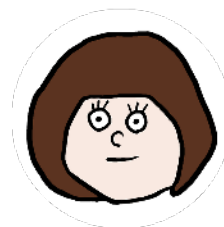
型安全性を向上させるために有益と思われるレビューは以下の通りです。

型定義・型設計の改善 1 ...

- Enumの代わりにstring union型を使用することを推奨しています。これにより、値が部分的なEnumとして扱われることを避け、より厳密な型チェックが可能になります 1。
- union型とexhaustiveチェック（assertNeverなど）を組み合わせることで、全てのケースが網羅的に処理されていることを保証し、将来的な型追加時のバグを防ぎます 2 >< 51 67。
- 定数から型を生成することで、型が漏れることなく、全ての値が網羅的に列挙されていることを確認できます 3 52。
- modelではnullableなプロパティをrequiredとして扱い、converterでnull値の処理を吸収することを基本としています。これにより、モデルの型定義がシンプルかつ厳密になります 5 ...。
- converter内でパーミッション判定ロジックをカプセル化し、frontend modelには結果のbooleanフラグ（例: canDownloadKnowledge: boolean）を持たせることで、関心事を分離し、型の意図を明確にします 4。
- あり得ない組み合わせを持つ複数のプロパティ（例: infiniteAmount: boolean と maxAmount: number）を避け、単一の数値プロパティ（例: maxAmount: number で Infinity を無制限として扱う）で状態を表現することで、不正な状態を型レベルで防止します 11。

レビューコメントから得られた学び

TypeScript ベスト3



わたしが選びました

相互排他的なパラメータにはunion型を使おう！

hogeId と fugaParams 系が同時にくることはないので、
とりうる値のunionにしたほうが使う時迷わないかなって思いました！

```
type SomeOption = {  
  hogeId?: Hoge['id']  
  fugaId?: Fuga['id']  
  fugaName?: string  
}
```



```
type SomeOption = {  
  hogeId: Hoge['id']  
} | {  
  fugaId: Fuga['id']  
  fugaName: string  
}
```

一部の組み合わせが存在しない複数パラメータにはunion型を使おう！

`hasAmount: false` と `infiniteAmount: true` の組み合わせはないので、
3種類のunionにしたらどうですか？

```
type SomeOption = {  
  hasAmount: boolean  
  // hasAmountがfalseの場合は必ずfalse  
  infiniteAmount: boolean  
}
```



```
type SomeOption = {  
  amount: 'NONE' | 'SPECIFIC' | 'INFINITE'  
}
```


プロパティからconst変数に入れると条件分岐での型絞り込み効くと思います！

ここに `as const` つけばいけます！

`satisfies` で担保できます！

型ガード関数あるとよさそうですね！

undefinedならthrowする `guardUndef` 関数があるので使ってくださいー！

`Object.entries` に型つけてある `objectEntries` というhelperがあります！

`AnimalType` に `& DogType` をつけておくことで内包が保証できてas不要になりそうです！

取り得るパターンが決まっているところは過不足を型で検出しよう！

以下のように、今後statusが増えたときに型エラーで気付けるようにしたいです！

```
switch (status) {  
  case 'QUEUING':  
    return '処理中です'  
  case 'ERROR':  
    return 'エラーが発生しました'  
  default:  
    return '' // 'SUCCESS'  
}
```



```
switch (status) {  
  case 'QUEUING':  
    return '処理中です'  
  case 'ERROR':  
    return 'エラーが発生しました'  
  case 'SUCCESS':  
    return ''  
  default: {  
    const check: never = status  
    throw new Error(`Unhandled status: ${status}`)  
  }  
}
```

取り得るパターンが決まっているところは過不足を型で検出しよう！

以下のように、今後statusが増えたときに型エラーで気付けるようにしたいです！

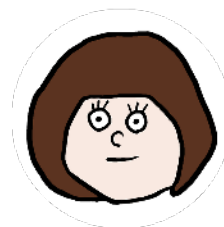
```
switch (status) {  
  case 'QUEUING':  
    return '処理中です'  
  case 'ERROR':  
    return 'エラーが発生しました'  
  default:  
    return '' // 'SUCCESS'  
}
```



```
return match(status)  
  .with('QUEUING', () => '処理中です')  
  .with('ERROR', () => 'エラーが発生しました')  
  .with('SUCCESS', () => '')  
  .exhaustive()  
  
// エラーにたくないなら .otherwise(fn) もある
```

レビューコメントから得られた学び

React ベスト3



わたしが選びました

微細なところはスルーですが、
custom hooksの返回值とContextのvalueはちょっと気にしてます

Contextのvalueに直接object渡しちゃうと別参照になっちゃいます！

```
const close = useCallback(() => ..., [...])
return (
  <ModalContext.Provider value={{ close }}>
    ...
  </ModalContext.Provider>
)
```



```
// object形式で { close } を返す
const value = useMemo(() => ..., [...])
return (
  <ModalContext.Provider value={value}>
    ...
  </ModalContext.Provider>
)
```

高頻度で発生するイベント内の処理に注意！

連続的に起きるイベントだったら、その中で毎回getBoundingClientRectとかonChangeCurrentTimeしてて重くならないかな？

```
const onPointerMove = (e: React.PointerEvent<HTMLDivElement>) => {  
  const { left } = e.currentTarget.getBoundingClientRect()  
  onChangeCurrentTime(calcSeekTime(e.clientX - left, e.currentTarget.clientWidth))  
}
```

- DOM Layout再計算がかからない値を使う
- stateの更新を間引く、もしくはCSSで表示を変えるなど

Loading範囲大きくなっちゃってるからここ `Suspense` で囲ったほうがよさそう！

Componentわけて、子でSuspenseしてErrorBoundaryでキャッチすればこのstate
いらなくなると思う！

ここ `useTransition` 使うと全体loadingになっちゃうの避けられそう！

適切に使えるとかなりシンプルに状態を減らしつつユーザー体験も向上できます

番外編

- とにかく消せるものがないか探す
- jsxが消えてたら
 - 独自classNameがついてないか
 - 専用Componentが含まれてないか
- importが消えてたら
 - そこ以外で使われているのかどうか

同一ファイルの未使用変数/importならlintで縛れますが、ファイルまたぐときが残りがち
(knipとかで自動でできるかも)

おわりに

今回の登壇のために色々なAIツールで過去のレビューを分析してみました。

NotebookLMは12万行のmarkdownも難なく扱っていてかなり強力だと思いました！

私はまだ学びの活用まで進めていないですが、次の名人さんが活用についてお話しされる
とのことですよ！

できる喜びが巡る日々を届ける

Deliver the joy of enablement