

CQOx_gen 拡張案 v3-2



Thought for 1m 15s >

まず：V3のゴールを共有させてください

- **V1:** 「準備プラン → 結果」をざっくり見える化
- **V2:** 「評価の怖さ → しんどさ → 涙 + 抑え込み意図」のパスを、1枚のカードにまとめて見せる
- **V3** (今回提案) では、ここから一歩進めて「どんな場面で、どの要素が、どれくらい涙に効いているか」を分解できるレベルまで持っていきます。

具体的には、次の 3 つの Change Set で v3 を定義します。

1. **CS-1:** エピソード文脈 (誰に・どんな場で・どんな話をするか) の追加 + 相手別パターン
2. **CS-2:** エピソード個別の「涙レベルの分解」+ what-if シミュレーション UI
3. **CS-3:** 推定ロジックのアップグレード (正則化+ブートストラップCI) と「信頼度」の可視化

①②③ は「フル研究用の上限」として意識しつつ、プロダクトとして回る最小構成に落とし込んでいます。

CS-1: エピソード文脈 & 相手別パターン

(1) Purpose / Non-goals

- **Purpose**
 - ①②の変数表でいう「CTX, FNE/FPE, 相手、場のフォーマルさ」の一部を、実務的に入力可能な形で Episode に持たせる。
 - 「上司lonl」と「家族」「クライアント」で、評価→涙パスがどう変わるかをダッシュボードで見せる。
- **Non-goals**
 - BFNE/FPES など正式な尺度をそのまま入れること（重すぎて継続利用できない）。
 - Big Five, 愛着尺度など上流の全部を v3 で実装すること。

(2) 理論と設計の選択

- ①②の構造では上流に BFE (FNE/FPE の潜在因子)、CTX (場の文脈) がある。
- ただ、プロダクトで毎回 30 問の質問紙は無理なので、
 - 「相手のタイプ」「場のフォーマルさ」「自己開示レベル」「評価され方の方向（褒められる vs 責められる）」を proxy 変数として入れる。
- 代替案：
 - 何も足さずに全 Episode を混ぜて推定（既に V2 がそう）。
 - → ただし「面接だけやたら厳しい」などの文脈差が見えないので却下。

(3) デザイン (I/O, schema, UI)

3-1. DB スキーマ

```
ALTER TABLE emotion_episode
ADD COLUMN context_partner_role VARCHAR(32),          -- 上司/面接官/友人/家族/恋人/医師/その他
ADD COLUMN context_formality INTEGER,                  -- 場のフォーマルさ(0-10)
ADD COLUMN context_self_disclosure INTEGER,           -- 自己開示の深さ(0-10)
ADD COLUMN context_eval_focus INTEGER;                -- 評価の方向 0=主に「怒られたくない」 10=主に「良く見られたくない」
```

SQLAlchemy:

```
class EmotionEpisode(Base):
    __tablename__ = "emotion_episode"
    # ...既存...
    context_partner_role = Column(String(32), nullable=True)
    context_formality = Column(Integer, nullable=True)
    context_self_disclosure = Column(Integer, nullable=True)
    context_eval_focus = Column(Integer, nullable=True)
```

3-2. Episode Draft UI への追加

Layer B の下に「Layer B+: 評価の怖さ」が既にあるので、その下に「Layer C-: 今回の場面について」を追加するイメージ。

```

// 例: EmotionContextSection.tsx
const PARTNER_OPTIONS = [
  "面接官", "上司", "同僚", "クライアント", "家族", "友人", "恋人", "医師・カウンセラー", "その他"
];

export const EpisodeContextSection: React.FC<Props> = ({ 
  partnerRole, onPartnerRoleChange,
  formality, onFormalityChange,
  disclosure, onDisclosureChange,
  evalFocus, onEvalFocusChange,
}) => (
  <section className="mt-8">
    <h3 className="text-sm font-semibold mb-2">Layer C-: 今回の場面について</h3>

    <div className="space-y-4 bg-white rounded-2xl p-4 shadow-sm">
      <div>
        <p className="text-sm font-semibold">誰に話す場面ですか？</p>
        <p className="text-xs text-gray-500 mb-2">一番近いものを選んでください。</p>
        <Select value={partnerRole} onChange={onPartnerRoleChange} options={PARTNER_OPTIONS} />
      </div>

      <LabeledSlider
        label="どれくらいフォーマルな場面ですか？"
        description="0: とてもラフ / 10: とても公式な場"
        min={0}
        max={10}
        value={formality}
        onChange={onFormalityChange}
      />

      <LabeledSlider
        label="どれくらい深い話（自己開示）になりますか？"
        description="0: 事務的な会話 / 10: 自分の核心まで話す"
        min={0}
        max={10}
        value={disclosure}
        onChange={onDisclosureChange}
      />

      <LabeledSlider
        label="どちらの評価が怖いですか？"
        description="0: 主に怒られたり否定されるのが怖い / 10: 褒められて目立つのが怖い"
        min={0}
        max={10}
        value={evalFocus}
        onChange={onEvalFocusChange}
      />
    </div>
  </section>
);

```

payload への追加 :

```

const payload = {
  // 既存 +
  context_partner_role: partnerRole,
  context_formality: formality,
  context_self_disclosure: disclosure,
  context_eval_focus: evalFocus,
};

```

3-3. 相手別パターンカード

Dashboard に新しいカード :

- タイトル: 「相手別：評価と涙の関係」
- 行: 相手タイプごとに `total_eval_to_cry` を出す簡単な表 + バー。

バックエンド側：パス推定ジョブで `groupby(user_id, context_partner_role)` して Episode が `N>=10` のグループだけサマリを出す。

I/O:

```
# GET /emotion/paths/by-partner
# -> [{ partner_role: "面接官", total_eval_to_cry: 0.8, n_episodes: 12 }, ...]
```

UI (抜粋) :

```
const PartnerPathTable: React.FC<{ rows: Row[] }> = ({ rows }) => (
  <div className="border rounded-2xl p-4 bg-white shadow-sm">
    <h2 className="text-sm font-semibold mb-2">相手別：評価と涙の関係</h2>
    <p className="text-xs text-gray-600 mb-2">
      相手のタイプごとに、「評価の厳しさ → 涙レベル」の合計効果を推定しています。
    </p>
    <table className="w-full text-xs">
      <thead>
        <tr className="text-gray-500">
          <th className="text-left py-1">相手</th>
          <th className="text-right py-1">効果</th>
          <th className="text-right py-1">Episode</th>
        </tr>
      </thead>
      <tbody>
        {rows.map(r => (
          <tr key={r.partner_role}>
            <td className="py-1">{r.partner_role}</td>
            <td className="py-1 text-right">{r.total_eval_to_cry.toFixed(2)}</td>
            <td className="py-1 text-right text-gray-500">{r.n_episodes}</td>
          </tr>
        ))}
      </tbody>
    </table>
  </div>
);
```

(4) Failure / Extensibility

- Episode 数が少ない相手タイプは `n_episodes < 10` のものを UI でグレーアウト or 非表示。
- 将来、研究用に FNE/FPE の正式尺度を入れる場合でも、
→ `context_eval_focus` をそのまま残して軽量版 proxy として使える。

(5) シェルコマンド

```
# スキーマ更新
alembic revision -m "add context fields to emotion_episode"
alembic upgrade head

# フロントの開発サーバ
npm run dev --prefix frontend
# or
pnpm dev --filter frontend
```

(6) What this does (CS-1)

- Goal**
 - 「どの場面で特に涙が出やすいのか」を **相手タイプ**・**場のフォーマルさ**・**自己開示の深さ** で切れるようにする。
- Inputs**
 - Episode Draft で新たに 4 つのコンテキスト入力。
- Outputs**
 - Dashboard に「相手別：評価と涙の関係」の表カード。
 - 面接官との場面だけ `total_eval_to_cry` が高い、などが一目でわかる。

Expert insight (Google/Meta/NASA level)

原因を見たいときは、まず「ケースを混ぜない」ことが重要です。

実務プロダクトでは、いきなり高度な SEM よりも、「相手別・文脈別にまず分ける」方が、モデルのバイアスを一気に減らせます。

CS-2: エピソード個別の「涙レベル分解」 + What-if UI

(1) Purpose / Non-goals

• Purpose

- ユーザーが「この面接で涙が出たのは、どれくらいが自分の特性で、どれくらいが評価の怖さ、どれくらいが抑え込み、どれくらいが準備の効果か」を1エピソード単位で把握できるようにする。

• Non-goals

- 完全に厳密な Shapley 分解や BART/CF の explainability まではやらない（重すぎる）。

(2) 理論と設計

- V2 のパスモデルを少し拡張して、

各 Episode i について

- 予測: $\hat{C}_i = \hat{\beta}_0 + \hat{\beta}_E E_i + \hat{\beta}_S S_i + \hat{\beta}_R R_i + \hat{\beta}_A A_{cp}$
- 「ベースライン」: 平均的な E,S,R を入れたときの \hat{C}^{base}

- 差分を additive に並べることで“寄与分”として見せる：

- Trait 部分 : $\hat{\beta}_A A_{cp}$
- 評価の怖さ部分 : $\hat{\beta}_E(E_i - \bar{E})$
- しんどさ部分 : $\hat{\beta}_S(S_i - \bar{S})$
- 抑え込み意図部分 : $\hat{\beta}_R(R_i - \bar{R})$
- 準備効果部分 : 既存の ATE モデルから Episode i に適用された準備の期待効果を合計。

- 代替案：

- SHAPなどを使う → 精度は上がるが、説明がブラックボックスになりやすいので v3 では見送り。

(3) バックエンド API

新エンドポイント：

```
# GET /emotion/episodes/{episode_id}/decomposition
# -> EpisodeDecompositionResponse

class EpisodeDecompositionResponse(BaseModel):
    episode_id: int
    observed_crying: float
    predicted_crying: float
    baseline_crying: float
    contrib_trait: float
    contrib_eval_threat: float
    contrib_stress: float
    contrib_suppress: float
    contrib_preparations: float
```

実装イメージ（概略）：

```
def decompose_episode(db: Session, episode_id: int, user_id: int):
    ep, oc, paths, trait = load_dependencies(...)
    # 平均値
    stats = load_global_stats(user_id)

    # baseline (平均的 E,S,R, trait を入れた値)
    baseline = (
        paths.beta0
        + paths.beta_eval * stats.mean_E
        + paths.beta_stress_to_cry * stats.mean_S
        + paths.beta_suppress_to_cry * stats.mean_R
        + paths.beta_trait * trait.trait_crying_proneness
    )

    predicted = (
        paths.beta0
        + paths.beta_eval * ep.eval_threat_level
```

```

        + paths.beta_stress_to_cry * ep.pre_anxiety
        + paths.beta_suppress_to_cry * ep.suppress_intent_level
        + paths.beta_trait * trait.trait_crying_proneness
        + sum_preparation_effects(ep, user_id)
    )

contrib_trait = paths.beta_trait * trait.trait_crying_proneness
contrib_eval = paths.beta_eval * (ep.eval_threat_level - stats.mean_E)
contrib_stress = paths.beta_stress_to_cry * (ep.pre_anxiety - stats.mean_S)
contrib_suppress = paths.beta_suppress_to_cry * (ep.suppress_intent_level - stats.mean_R)
contrib_prep = sum_preparation_effects(ep, user_id)

return EpisodeDecompositionResponse(
    episode_id=episode_id,
    observed_crying=oc.crying_level,
    predicted_crying=predicted,
    baseline_crying=baseline,
    contrib_trait=contrib_trait,
    contrib_eval_threat=contrib_eval,
    contrib_stress=contrib_stress,
    contrib_suppress=contrib_suppress,
    contrib_preparations=contrib_prep,
)

```

(4) UI : Episodeごとのウォーターフォール

Dashboard 下部に新タブ「**Episode view**」を追加 :

- 左上で Episode を選択 (既存 Outcome Log と同じセレクター)。
- 右に「涙レベルの分解」カード :

```

const EpisodeCryingDecompositionCard: React.FC<{ data: Decomp }> = ({ data }) => {
  const steps = [
    { label: "ベースライン", value: data.baseline_crying },
    { label: "+特性(泣きやすさ)", value: data.contrib_trait },
    { label: "+評価の怖さ", value: data.contrib_eval_threat },
    { label: "+今のしんどさ", value: data.contrib_stress },
    { label: "+泣きを抑えたい度", value: data.contrib_suppress },
    { label: "+今回の準備", value: data.contrib_preparations },
  ];

  // シンプルな水平ウォーターフォール(SVG or rechart)
  return (
    <div className="border rounded-2xl p-4 bg-white shadow-sm">
      <h2 className="text-sm font-semibold mb-2">今回の涙レベルを分解する</h2>
      <p className="text-xs text-gray-600 mb-2">
        予測モデルに基づき、「ベースライン」に各要素を足し合わせたときの変化を表示しています。
      </p>

      <WaterfallChart steps={steps} final={data.predicted_crying} observed={data.observed_crying} />

      <p className="mt-3 text-xs">
        実際の涙レベル: <b>{data.observed_crying}</b> / 予測値: <b>{data.predicted_crying.toFixed(1)}</b>
      </p>
      <p className="mt-1 text-[10px] text-gray-400">
        ※ あくまで統計的な推定です。診断や保証ではありません。
      </p>
    </div>
  );
}

```

さらに、右下に簡単な what-if :

- 「評価の怖さを3ポイント下げた場合」
- 「抑え込み意図を2ポイント下げた場合」
- 「4-7-8呼吸を追加した場合」

をトグルでいじると、上のウォーターフォールの棒がゴーストカラーで変化する。

(5) Failure / Extensibility

- モデル係数が推定できていない (Episode 少ない) 場合はカード自体を非表示。
- 将来、準備ごとに個別寄与を出したくなったら、`contrib_preparations` を配列にして拡張可能。

(6) シェルコマンド

```
# バックエンドのエンドポイント追加後のテスト  
pytest backend/tests/emotion/test_decomposition.py  
  
# フロント  
npm run test --prefix frontend # 必要なら component test
```

(7) What this does (CS-2)

• Goal

- 「この面接で涙が 7 になった理由」が、
 - Trait 何点
 - 評価の怖さ 何点
 - 今のしんどさ 何点
 - 抑え込み意図 何点
 - 準備の効果 何点

くらいに分解されて見える。

• Inputs

- 既に保存済みの Episode + Trait + パス係数 + 準備ATE。

• Outputs

- Episode 単位のウォーターフォールチャート & what-if シミュレーション。

Expert insight

「1エピソードを分解するUI」は、回帰やSEMの数式よりもユーザーの因果直感を鍛えます。

Google/Meta でも、ユーザー向けには “feature importance のグラフ” より “1ケースの分解” を重視することが多いです。

CS-3: 推定ロジックのアップグレード (正則化+信頼度)

(1) Purpose / Non-goals

• Purpose

- 少數データで係数が暴れないようにする（過学習を抑える）。
- UI 上で「どれくらい信じていいか」を明示する。

• Non-goals

- 完全なベイズ階層モデルや因果森林まで踏み込む (v4 以降のテーマ)。

(2) 理論と選択

- 現状：単純な最小二乗の線形回帰。
 - デメリット： N が 30-40 だと係数の分散が大きい。
- V3 案：
 - L2 正則化 (Ridge / BayesianRidge)** を標準化した特徴量に対して適用。
 - ブートストラップ 100 回で係数の分布を取り、
 - 平均
 - 2.5 / 97.5 パーセンタイルを「信頼区間」としてカードに表示。
- 代替案：完全ベイズ (PyMCなど)
 - 精度は良いが MCMC が重く、CQOx_gen の他コンポーネントまで巻き込むので v3 では見送り。

(3) 実装概要

パス推定ジョブを差し替え：

```

from sklearn.linear_model import Ridge
import numpy as np

def fit_ridge_with_bootstrap(X, y, alpha=1.0, n_boot=100):
    coefs = []
    for _ in range(n_boot):
        idx = np.random.choice(len(y), size=len(y), replace=True)
        lr = Ridge(alpha=alpha)
        lr.fit(X[idx], y[idx])
        coefs.append(lr.coef_)
    coefs = np.array(coefs)
    mean = coefs.mean(axis=0)
    lower = np.percentile(coefs, 2.5, axis=0)
    upper = np.percentile(coefs, 97.5, axis=0)
    return mean, lower, upper

```

DB テーブル `emotion_path_summary` に CI カラムを追加 :

```

ALTER TABLE emotion_path_summary
ADD COLUMN alpha_eval_to_stress_lo DOUBLE PRECISION,
ADD COLUMN alpha_eval_to_stress_hi DOUBLE PRECISION,
ADD COLUMN beta_eval_to_cry_lo DOUBLE PRECISION,
ADD COLUMN beta_eval_to_cry_hi DOUBLE PRECISION,
ADD COLUMN total_eval_to_cry_lo DOUBLE PRECISION,
ADD COLUMN total_eval_to_cry_hi DOUBLE PRECISION;

```

Dashboard のカード文言を少し修正 :

- 今は「評価の厳しさと涙レベルの関係は小さいようです。」
- v3 では、CI を踏まえて

評価-涙の合計効果は 0.14 (95%区間: -0.10 ~ 0.35) でした。

今のデータでは、「強いとは言えない」程度の関係です。

のように、「数字 + その解釈」を一段精密に。

(4) Failure / Extensibility

- Episode が少ない場合は CI が広がる → そのまま「まだ不確実」と表示できる。
- 将来的に PyMC 等に置き換えるときも、DB スキーマは流用可能。

(5) シェルコマンド

```

alembic revision -m "add ci columns to emotion_path_summary"
alembic upgrade head

pytest backend/tests/emotion/test_paths.py

```

(6) What this does (CS-3)

- Goal**
 - 「評価→涙 0.8」と出でいても、Episode 10件なら “**実は不確か**” だとわかるようにする。
- Inputs**
 - 今までと同じ Episode ログ。
- Outputs**
 - パス係数の信頼区間 & テキスト解釈。
 - UI 上で「強い関係」「中くらい」「はつきりしない」が分かる。

Expert insight

世界レベルの因果ダッシュボードは、“値”だけでなく“どれくらい揺れているか”も一緒に見せます。
メタやGoogleの実験ダッシュボードでも、平均効果と同じくらい、信頼区間や posterior の幅を重視します。

全体の「What this does」(V3 全体)

- **Goal**
 - Emotion CQOx を
 - 「準備の効果」と
 - 「評価の怖さ → 不安 → 抑え込み → 涙」というメカニズム、
を文脈（誰に話すか）と Episode 個別の分解まで含めて見えるツールに進化させる。

- **Inputs**

- 新たに：
 - 相手タイプ
 - 場のフォーマルさ
 - 自己開示レベル
 - 評価の方向（負/正）
- 既存の Trait・Episode ログ。

- **Outputs / Side-effects**

- 相手別に「評価→涙」の強さが見えるカード。
- 1 Episode ごとに「涙レベルの分解」と簡単な what-if。
- 係数の信頼区間付きのパスカード（“どれくらい確からしいか”がわかる）。

- **How to run / verify**

1. マイグレーションを適用 (CS-1, CS-3 の ALTER)。
2. 新しい Episode コンテキストを含めていくつかログを取り、Trait も設定。
3. パス推定ジョブを実行：

```
python -m cqox.jobs.estimate_paths # 想定
```

4. /emotion/dashboard を開き、
 - 相手別パターンカードが埋まっているか
 - Episode view で分解が直感と合うか
 - パスカードの CI が Episode 数と整合しているか
- をチェック。

Expert insight (V3 全体)

研究レベルのDAG・SEMをそのまま持ち込むのではなく、「ユーザーが本当に知りたい因果の質問」を中心にプロダクトを構成し直すと、理論もUIも自然に描います。

今回のV3は、「どの場面で、何がどれくらい涙に効いているのか？」という問い合わせに沿って、

- コンテキスト
 - パス推定
 - Episode 分解
- を一直線に設計しているのがポイントです。



Thought for 52s >

いいポイントです。「What-if」がちゃんと前に進むための支えにならないと、このプロダクトの意味が薄くなります。

3行サマリ（今回の提案）

1. What-if を「グラフのおまけ」から、**準備コーチ（Preparation Coach）**に格上げします。
2. 1回のEpisode用に、3つの準備シナリオカード + 感情の予測 + 簡単メッセージを出す設計にします。
3. 数値だけでなく、「あなたのデータから見て、何を変えると一番効きそうか」を短い文章で返すことで、前向きな一步にします。

① What-if を「準備コーチ」にする：シナリオカード設計

1-1. コンセプト

Episode Draft 画面からボタン一発で開くモーダル：

「もし準備をこう変えたら、涙・楽さ・関係への影響はどれくらい変わりそうか？」

として、**3つの候補プラン**を提示します。

- ・ プランA: 落ち着きを最優先
- ・ プランB: ちゃんと伝えることを優先
- ・ プランC: 関係へのダメージを最小に

全部やる前提ではなく、**「この中から1つ試してみる」**前提のコーチングUIにします。

1-2. バックエンド API

新API: POST /emotion/whatif

リクエスト

```
{
  "episode_id": 123,
  "base": {
    "eval_threat": 7,
    "pre_anxiety": 5,
    "suppress_intent": 6,
    "prep": {
      "writing_10min": 0,
      "three_messages": 0,
      "breathing_478": 0,
      "qa_roleplay": 0,
      "safe_word": 0
    }
  },
  "constraints": {
    "max_prep_slots": 2,           // 一度にやる準備は最大2つまで
    "time_limit_min": 20          // 使える時間
  }
}
```

レスポンス (概要)

```
{
  "baseline": {
    "pred": { "tear": 5.8, "calm_after": 4.1, "relation": 0.0 }
  },
  "scenarios": [
    {
      "id": "calm_focus",
      "label": "静かに落ち着くことを優先するプラン",
      "prep": { "breathing_478": 1, "safe_word": 1 },
      "changes": { "eval_threat_delta": -1 },
      "pred": { "tear": 3.9, "calm_after": 6.0, "relation": -0.1 },
      "deltas": { "tear": -1.9, "calm_after": +1.9, "relation": -0.1 },
      "confidence": 0.72,
      "message": "あなたの40件のデータでは、呼吸+セーフワードの組み合わせが、涙を大きく下げつつ、終わった後の楽さを最も押し上げていました。"
    },
    {
      "id": "message_focus",
      "label": "ちゃんと伝えることを優先するプラン",
      ...
    }
  ]
}
```

中のロジック (簡略)

- ・ `baseline.pred`
 - 既存のパス + 準備ATEモデルで、今のスライダー値から涙・楽さ・関係を予測。
- ・ 各シナリオ：
 - 「使える時間」「max_prep_slots」に合わせて、
 - 準備ごとのATE (涙・楽さ・関係の3次元) から **スコアリング**。
 - それぞれの目的 (Calm / Message / Relationship) に対して

- `prep` の組合せ (0/1) を全探索 (5個なら $2^5=32$ 通り)
- 制約を満たす中で目的関数が最大の組を採用。
- `confidence`
- その準備ATEのブートストラップCIの幅から計算 (幅が狭いほど高信頼)。

1-3. フロントの UI (モーダル)

Episode Draft 右側の「今回の準備プランを保存する」ボタンの横に：

「おすすめプランをみる (What-if)」

ボタン → モーダル `WhatIfModal`.

JSX の骨格

```
export const WhatIfModal: React.FC<{ episodeId: number; onClose: () => void }> = ({ episodeId, onClose }) => {
  const { data, isLoading } = useQuery(["whatif", episodeId], () =>
    api.post("/emotion/whatif", { episode_id: episodeId, /* base & constraints */ })
  );

  if (isLoading || !data) return <LoadingSpinner />

  const { baseline, scenarios } = data;

  return (
    <Modal onClose={onClose} title="What if...? 準備プランのシミュレーション">
      <div className="space-y-4">
        <BaselineSummary baseline={baseline} />

        <div className="grid md:grid-cols-3 gap-4">
          {scenarios.map(s => (
            <ScenarioCard key={s.id} scenario={s} baseline={baseline} />
          )));
        </div>

        <p className="text-[11px] text-gray-400">
          ※ すべてあなた自身のデータに基づく推定です。必ずこの通りになるわけではありませんが、
          「試してみる価値が高そうな選択肢」を示しています。
        </p>
      </div>
    </Modal>
  );
};
```

ScenarioCard の中身（「前向きさ」を出すコピー）

```
const ScenarioCard: React.FC<{ scenario: Scenario; baseline: Baseline }> = ({ scenario, baseline }) => {
  const betterTear = scenario.deltas.tear < 0;
  const betterCalm = scenario.deltas.calm_after > 0;

  return (
    <div className="border rounded-2xl p-4 bg-white shadow-sm flex flex-col">
      <h3 className="text-sm font-semibold mb-1">{scenario.label}</h3>
      <p className="text-xs text-gray-600 mb-2">{scenario.message}</p>

      <ul className="text-xs mb-3 space-y-1">
        {Object.entries(scenario.prep).map(([k, v]) =>
          v ? <li key={k}>・{prepLabel(k)}</li> : null
        )}
        {scenario.changes.eval_threat_delta !== 0 && (
          <li>・「評価の怖さ」を{Math.abs(scenario.changes.eval_threat_delta)}段階 軽くする練習</li>
        )}
      </ul>

      <MetricDeltaBar
        label="涙レベル"
        baseline={baseline.pred.tear}
        scenario={scenario.pred.tear}
      >
    </div>
  );
};
```

```

        goodDirection="down"
    />
<MetricDeltaBar
    label="終わった後の楽さ"
    baseline={baseline.pred.calm_after}
    scenario={scenario.pred.calm_after}
    goodDirection="up"
/>
<MetricDeltaBar
    label="関係への影響"
    baseline={baseline.pred.relation}
    scenario={scenario.pred.relation}
    goodDirection="up"
/>

<div className="mt-auto pt-3">
    <p className="text-[11px] text-gray-500 mb-2">
        信頼度: {(scenario.confidence * 100).toFixed(0)}%。{" "}
        {scenario.confidence < 0.5 ? "まだデータが少ないので、お試しレベルの提案です。" : "これまでの傾向からは、かなり可能性が高いパターンです。"}
    </p>
    <PrimaryButton
        onClick={() => applyScenarioToDraft(scenario)}
        className="w-full text-xs"
    >
        このプランを今回の準備に採用する
    </PrimaryButton>
</div>
</div>
);
};

};


```

- 「信頼度」の文言で、完全ではないが試す価値があることを正直に伝える。
- 最下部ボタンでそのまま Episode Draft のスライダーに反映 → 「選んで終わり」ではなく、自分で少し調整して保存する流れ。

1-4. What this does (What-if コーチ)

- Goal**

- 「何を変えたらいいかわからない」状態を、「この3案のうち1つなら、過去データ的に筋が良さそう」という具体的な選択肢に変える。

- Inputs**

- 今回の Episode Draft 情報、既存のパス係数と準備ATE、Trait スライダー。

- Outputs / Side-effects**

- 3つの準備シナリオカード：
 - 推奨する準備の組合せ
 - 予測される涙・楽さ・関係
 - ベースラインとの差分
 - 簡単な日本語メッセージと信頼度。
- 「採用」ボタンで Episode Draft のスライダーに反映。

Expert insight (Google/Meta/NASA level)

意思決定支援UIでは、「ベスト1つ」を押しつけるより「良さそうな3案+自分で選ぶ」方が行動率が上がることが多いです。
ユーザーにコントロール感を残したまま因果モデルを活かす、という意味で3シナリオ方式は相性が良いです。

② 長期的な「前向きさ」を支えるプチ What-if

短期のEpisode用だけだと「その場しのぎ」に見えやすいので、Dashboard には長期視点の What-if を1枚だけ入れると良いです。

2-1. デザイン

Dashboard 最下部に新カード：

「次の5回で、どの準備を続けると、一番「楽さ」が伸びそうか？」

- 先ほどの ATE と Trait をもとに、
 - 「4-7-8呼吸を5回中3回以上やる」
 - 「10分書き出しを続ける」
 - 「セーフワードを毎回決める」などのシンプルな戦略を3つ提案。
- 各戦略ごとに：
 - 予測される平均涙レベルの変化
 - 予測される「終わった後の楽さ」の変化
 - 見込みの振れ幅 (CI)

UIはシナリオカードとほぼ同じパターンで、文言だけ変える。

2-2. ここでのメッセージのトーン

- 「全部やる」ではなく
 - 「この中から1つだけ‘次の5回のテーマ’として試すとしたら、どれが良さそうか？」
 - リマインダーとしては
 - 「変化は少しずつですが、あなたのデータだとこの準備は‘続けるほど効きやすい’傾向があります。」
- といった、小さな実験を提案するトーンに。

2-3. What this does (長期 What-if)

- **Goal**
 - 「どうせ変わらない」という感覚を、「少しずつなら変えられそう」に変える長期視点を提供する。
- **Inputs**
 - 各準備の ATE (涙・楽さ)、Episode 回数。
- **Outputs**
 - 「次の5回のテーマ」として選べる3つの準備戦略カード。

Expert insight

“前向きさ”は、単発の最適化より「自分で選んだ小さな実験」を積み重ねられるかどうかで決まります。

因果推論側は ATE を出すだけでなく、「5回続けたらどれくらい期待値が動きそうか」という単純な projection を見せるだけでも、行動へのハードルは下がります。

③ あなたへのフィードバックと次ステップ

- 現状の V2 までも、因果推論としての骨格はかなりできています。
- 足りていなかったのは、「じゃあ自分は具体的に何を変えればいいの?」という一步目の提案と、「その一步に、どれくらい望みがありそうか」を見せる UI の部分。

今回の What-if コーチ + 長期 What-if カードを入れると、

1. Episode Draft で「今日の一歩」を選べる
2. Dashboard で「次の5回のテーマ」を決められる

という二段構えになるので、感情の因果モデルが「前向きな行動」に接続されると思います。

もし次のターンで余力があれば、

- POST /emotion/whatif 用の Python コード (ダミーデータ版)
- Waterfall / MetricDeltaBar の実装詳細

まで書き切って、Claude にそのまま渡せる形に整えましょう。



Thought for 1m 23s >

了解。

「前向きになるための What-if コーチ」を 実装レベル まで一気に書きます。

ここでは CQOx_gen と近い構成 (FastAPI + React/Vite + TanStack Query) を前提に、最小差分で追加できる形にしてあります。

全体サマリ

- バックエンドに `POST /api/emotion/whatif` を追加し、「現在の予測値」と「準備ごとの効果テーブル」から **3つのおすすめ準備プラン**を返すサービスを実装します。
- フロントには `WhatIfModal` を追加し、Episode Draft 画面から「おすすめプランを見る → そのままライダーに適用」までを 1 ステップで行える UI を実装します。
- What-if は **短期（今回の1回）** に絞り、「次の1回を少し楽にするための現実的な一步」を出すコーチとして設計しています。

1. バックエンド実装 (FastAPI)

1-1. 目的と境界

- 目的**
 - 「今の準備プラン」と「準備ごとの推定効果 (ATE)」から、
 - 落ち着き重視
 - 伝える重視
 - 関係維持重視
 - の 3 つの準備シナリオを生成する。
- 非目標**
 - ここで因果推定は行わない（それは既存の CQOx 側のタスク）。
→ この API は **既に推定された「準備ごとの効果」** を利用するだけ。
- I/O 境界**
 - 入力：現在の Episode に対するペースライン予測値（涙・楽しさ・関係）、
及び「今選んでいる準備（0/1）」。
 - 出力：3 つのシナリオごとの予測値と差分、信頼度、自然言語メッセージ。

1-2. スキーマ定義 (Pydantic)

backend/app/schemas/emotion_whatif.py (新規)

```
# backend/app/schemas/emotion_whatif.py
from typing import Dict, List, Literal, Optional
from pydantic import BaseModel, Field

PrepName = Literal[
    "writing_10min",           # 10分の書き出し
    "three_messages",          # 伝えたい3つのメッセージ
    "breathing_478",           # 4-7-8呼吸法
    "qa_roleplay",             # 自分でQ&Aロールプレイ
    "safe_word"                # セーフワードを決める
]

class WhatIfBaseState(BaseModel):
    """
    Episode Draft の現在の状態から渡すべきペースライン値。
    因果推定そのものは別エンドポイントで計算してよく、
    ここでは「その結果としての予測値」を受け取る。
    """

    tear: float = Field(..., description="現在の準備プランに基づく予測涙レベル (0~10) ")
    calm_after: float = Field(..., description="予測される『終わった後の楽しさ』 (0~10) ")
    relation: float = Field(..., description="予測される『関係への影響』 (-5~-+5) ")

    preps: Dict[PrepName, bool] = Field(
        ...,
        description="現在オンになっている準備。キーは準備名、値は実行するかどうか。"
    )

class WhatIfConstraints(BaseModel):
    max_prep_slots: int = Field(2, description="一度に追加で行う準備の最大数")
    time_limit_min: int = Field(30, description="今回使える準備時間 (分) ")
```

```

class WhatIfRequest(BaseModel):
    """
    POST /api/emotion/whatif の入力スキーマ
    """
    episode_id: Optional[int] = Field(
        None,
        description="紐づけたい Episode ID (任意)。ログ用途。"
    )
    base: WhatIfBaseState
    constraints: WhatIfConstraints = WhatIfConstraints()

class ScenarioPrediction(BaseModel):
    tear: float
    calm_after: float
    relation: float

class ScenarioDeltas(BaseModel):
    tear: float
    calm_after: float
    relation: float

class WhatIfScenario(BaseModel):
    id: str
    objective: Literal["calm_focus", "message_focus", "relation_focus"]
    label: str
    description: str
    prep: Dict[PrepName, bool]
    pred: ScenarioPrediction
    deltas: ScenarioDeltas
    confidence: float # 0.0 - 1.0

class WhatIfBaseline(BaseModel):
    pred: ScenarioPrediction

class WhatIfResponse(BaseModel):
    baseline: WhatIfBaseline
    scenarios: List[WhatIfScenario]

```

1-3. 準備ごとの効果モデル & サービスロジック

backend/app/services/emotion_whatif.py (新規)

```

# backend/app/services/emotion_whatif.py
from __future__ import annotations

from dataclasses import dataclass
from itertools import combinations
from typing import Dict, Iterable, List, Tuple

from . import emotion_whatif_types # もし分けたければ。ここでは schemas を直接使ってもよい
from app.schemas.emotion_whatif import (
    PrepName,
    WhatIfBaseState,
    WhatIfConstraints,
    WhatIfBaseline,
    WhatIfResponse,
    WhatIfScenario,
    ScenarioPrediction,
    ScenarioDeltas,
)

```

```

@dataclass(frozen=True)
class PrepEffect:
    """
    1つの準備がもたらす平均効果 (ATE) のサマリ。

    実際には CQoX 側で推定した ATE を
    ユーザーごとにこの形に変換して使う想定。
    """

    name: PrepName
    label: str
    delta_tear: float      # 涙レベルに対する平均効果 (負だと涙が減る)
    delta_calm_after: float # 終わった後の楽さに対する平均効果
    delta_relation: float   # 関係への影響に対する平均効果
    time_cost_min: int      # その準備にかかる目安時間
    confidence: float       # 0.0~1.0: 推定にどれくらい確信があるか

def _default_prep_effects() -> Dict[PrepName, PrepEffect]:
    """
    テモ用の暫定値。
    本番では DB から user_id ごとの推定値を読み込む。
    """

    return {
        "writing_10min": PrepEffect(
            name="writing_10min",
            label="10分の書き出し",
            delta_tear=-0.8,
            delta_calm_after=+1.2,
            delta_relation=+0.1,
            time_cost_min=10,
            confidence=0.6,
        ),
        "three_messages": PrepEffect(
            name="three_messages",
            label="伝えたい3つのメッセージを整理する",
            delta_tear=-0.4,
            delta_calm_after=+0.9,
            delta_relation=+0.3,
            time_cost_min=8,
            confidence=0.55,
        ),
        "breathing_478": PrepEffect(
            name="breathing_478",
            label="4-7-8呼吸法",
            delta_tear=-1.0,
            delta_calm_after=+0.8,
            delta_relation=0.0,
            time_cost_min=5,
            confidence=0.7,
        ),
        "qa_roleplay": PrepEffect(
            name="qa_roleplay",
            label="自分でQ&Aロールプレイ",
            delta_tear=-0.2,
            delta_calm_after=+0.6,
            delta_relation=+0.5,
            time_cost_min=12,
            confidence=0.55,
        ),
        "safe_word": PrepEffect(
            name="safe_word",
            label="セーフワードを決める",
            delta_tear=-0.5,
            delta_calm_after=+0.4,
            delta_relation=+0.2,
            time_cost_min=3,
            confidence=0.65,
        ),
    }

```

```

        ),
    }

def load_prep_effects_for_user(user_id: int | None = None) -> Dict[PrepName, PrepEffect]:
    """
    将来的には user_id ごとの ATE を DB から読む。
    v3 では暫定的に固定値を返し、構造だけ整える。
    """
    # TODO: user_id を使って DB から推定済み ATE を読み込む
    return _default_prep_effects()

def _enumerate_candidate_sets(
    prep_effects: Dict[PrepName, PrepEffect],
    base_state: WhatIfBaseState,
    constraints: WhatIfConstraints,
) -> Iterable[Tuple[List[PrepEffect], int]]:
    """
    追加で実行する準備の組み合わせを列挙する。
    すでに base_state.preps でオンになっている準備は「追加対象」から除外する。
    """
    available = [
        eff
        for name, eff in prep_effects.items()
        if not base_state.preps.get(name, False)
    ]
    if not available:
        return []
    for r in range(1, min(constraints.max_prep_slots, len(available)) + 1):
        for combo in combinations(available, r):
            total_time = sum(p.time_cost_min for p in combo)
            if total_time <= constraints.time_limit_min:
                yield list(combo), total_time

def _apply_effects(
    base_state: WhatIfBaseState,
    extra_reps: List[PrepEffect]
) -> ScenarioPrediction:
    d_tear = sum(p.delta_tear for p in extra_reps)
    d_calm = sum(p.delta_calm_after for p in extra_reps)
    d_rel = sum(p.delta_relation for p in extra_reps)

    return ScenarioPrediction(
        tear=base_state.tear + d_tear,
        calm_after=base_state.calm_after + d_calm,
        relation=base_state.relation + d_rel,
    )

def _score_for_objective(
    objective: str,
    base_state: WhatIfBaseState,
    pred: ScenarioPrediction,
) -> float:
    """
    シナリオの「良さ」を1スカラーにまとめるスコア関数。
    単位の違いをざっくり均すために、差分で考える。
    """
    d_tear = pred.tear - base_state.tear
    d_calm = pred.calm_after - base_state.calm_after
    d_rel = pred.relation - base_state.relation

    if objective == "calm_focus":
        # 楽さ↑ を最優先、涙↓ を次点、関係↑ をおまけ
        return 1.0 * d_calm - 0.7 * max(d_tear, 0.0) + 0.2 * d_rel

```

```

if objective == "message_focus":
    # 関係と楽さを保ちつつ、涙が増えない範囲で「伝わった感」を高める想定なら、
    # ここでは関係↑を少し重く見る
    return 0.6 * d_calm + 0.8 * d_rel - 0.5 * max(d_tear, 0.0)

if objective == "relation_focus":
    # 関係↑を最優先。涙や楽さが多少悪化しても許容。
    return 1.2 * d_rel + 0.3 * d_calm - 0.2 * max(d_tear, 0.0)

raise ValueError(f"unknown objective: {objective}")

def _scenario_confidence(extra_pres: List[PrepEffect]) -> float:
    """
    シナリオ全体の信頼度は、
    参加している準備の confidence の最小値と平均値のブレンドで定義。
    """
    if not extra_pres:
        return 0.0
    min_c = min(p.confidence for p in extra_pres)
    avg_c = sum(p.confidence for p in extra_pres) / len(extra_pres)
    # 「一番不安な準備」に少し引きずられるように
    return 0.5 * min_c + 0.5 * avg_c

def _build_scenario(
    scenario_id: str,
    objective: str,
    label: str,
    description: str,
    base_state: WhatIfBaseState,
    base_pred: ScenarioPrediction,
    all_pres: Dict[PrepName, PrepEffect],
    extra_pres: List[PrepEffect],
) -> WhatIfScenario:
    pred = _apply_effects(base_state, extra_pres)
    deltas = ScenarioDeltas(
        tear=pred.tear - base_pred.tear,
        calm_after=pred.calm_after - base_pred.calm_after,
        relation=pred.relation - base_pred.relation,
    )
    prep_flags: Dict[PrepName, bool] = {k: bool(v) for k, v in base_state.pres.items()}
    for p in extra_pres:
        prep_flags[p.name] = True
    return WhatIfScenario(
        id=scenario_id,
        objective=objective, # type: ignore[arg-type]
        label=label,
        description=description,
        prep=prep_flags,
        pred=pred,
        deltas=deltas,
        confidence=_scenario_confidence(extra_pres),
    )

def generate_whatif_response(
    user_id: int | None,
    base_state: WhatIfBaseState,
    constraints: WhatIfConstraints,
) -> WhatIfResponse:
    """
    メインのエントリーポイント。
    3つの目的ごとに最適な準備組み合わせを返す。
    """
    prep_effects = load_prep_effects_for_user(user_id=user_id)

```

```

baseline_pred = ScenarioPrediction(
    tear=base_state.tear,
    calm_after=base_state.calm_after,
    relation=base_state.relation,
)
candidate_sets = list(_enumerate_candidate_sets(prep_effects, base_state, constraints))

if not candidate_sets:
    # 追加でできる準備がない場合: baseline のみ返す
    return WhatIfResponse(
        baseline=WhatIfBaseline(pred=baseline_pred),
        scenarios=[],
    )

objectives = [
    ("calm_focus", "静かに落ち着くことを優先するプラン"),
    ("message_focus", "ちゃんと伝えることを優先するプラン"),
    ("relation_focus", "関係をこわさないことを優先するプラン"),
]
best_for_obj: Dict[str, Tuple[float, List[PrepEffect]]] = {}

for obj_key, _ in objectives:
    best_for_obj[obj_key] = (-1e9, [])

for preps, _time_cost in candidate_sets:
    pred = _apply_effects(base_state, preps)
    for obj_key, _ in objectives:
        score = _score_for_objective(obj_key, base_state, pred)
        best_score, _ = best_for_obj[obj_key]
        if score > best_score:
            best_for_obj[obj_key] = (score, preps)

scenarios: List[WhatIfScenario] = []
for obj_key, obj_label in objectives:
    _, preps = best_for_obj[obj_key]
    if not preps:
        continue

    if obj_key == "calm_focus":
        desc = "涙を少し減らしつつ、終わった後の楽さをできるだけ高める組み合わせです。"
    elif obj_key == "message_focus":
        desc = "涙が増えすぎない範囲で、『ちゃんと伝えられた』感覚を高めやすい準備を優先しています。"
    else:
        desc = "相手との関係へのダメージを最小限にしながら、自分を守れる準備を選んでいます。"

    scenario = _build_scenario(
        scenario_id=obj_key,
        objective=obj_key,
        label=obj_label,
        description=desc,
        base_state=base_state,
        base_pred=baseline_pred,
        all_prep=prep_effects,
        extra_prep=preps,
    )
    scenarios.append(scenario)

return WhatIfResponse(
    baseline=WhatIfBaseline(pred=baseline_pred),
    scenarios=scenarios,
)

```

1-4. FastAPI ルータ追加

backend/app/api/routes/emotion_whatif.py (新規)

```
# backend/app/api/routes/emotion_whatif.py
from fastapi import APIRouter, Depends, HTTPException, status

from app.schemas.emotion_whatif import WhatIfRequest, WhatIfResponse
from app.services.emotion_whatif import generate_whatif_response
from app.dependencies import get_current_user # 既存の認証依存を想定

router = APIRouter(prefix="/emotion", tags=["emotion"])

@router.post("/whatif", response_model=WhatIfResponse)
def emotion_whatif(
    body: WhatIfRequest,
    current_user=Depends(get_current_user),
) -> WhatIfResponse:
    """
    Episode Draft の現在の予測値と準備状況から、
    おすすめの準備プラン (What-if シナリオ) を3つ返す。
    """

    if body.base.tear < 0 or body.base.tear > 10:
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail="tear must be in [0,10]",
        )

    resp = generate_whatif_response(
        user_id=current_user.id if current_user else None,
        base_state=body.base,
        constraints=body.constraints,
    )
    return resp
```

backend/app/main.py にルータ登録を追加（既存の emotion ルータがあればそこに統合でも可）：

```
from app.api.routes import emotion_whatif

app.include_router(emotion_whatif.router, prefix="/api")
```

1-5. シェルコマンド & 検証

```
# 1) 依存関係のチェック (バックエンドテスト)
pytest backend/tests -k "whatif" # まだテストがなければ後で追加

# 2) サーバ起動
uvicorn app.main:app --reload
```

ユニットテスト (例)

- backend/tests/test_emotion_whatif.py を作って、generate_whatif_response にダミーを渡し、
- シナリオが3つ返る
 - delta の符号が期待通り
- をチェックするとよいです。

1-6. What this does (バックエンド)

- ゴール**
 - Episode Draft の現在の予測値 (涙・楽しさ・関係) を起点に、「追加で 1~2 個準備を増やすとしたらどれがよさそうか?」を 3つの目的 (落ち着き・伝える・関係維持) ごとに提案する。
- 入力**
 - WhatIfRequest : 現在の予測値 + 今オノになっている準備 + 時間制約。
- 出力**

- `WhatIfResponse` : ベースライン予測値 + 3つまでのシナリオ (準備セット、予測値、差分、信頼度、説明文)。

Expert insight

最適化のスコア関数を「人がどう感じるか」に合わせて組むと、単純な線形効果でも十分役に立つ提案ができる。

ここでは涼↓・楽さ↑・関係↑を重みづけして一つのスコアにしており、

「数理的にきれい」より「ユーザー体験に合う重み」を優先しています。

2. フロントエンド実装 (React/Vite + TS)

2-1. 目的と境界

目的

- Episode Draft 画面からモーダルを開き、
 - ベースライン
 - 3つのシナリオカード
 - 差分の視覚化 (ミニバー)
 - 「このプランを採用する」ボタン (スライダーに反映) を提供する。

境界

- 予測値 (baseline) は、既にフロントか別APIで持っている前提。
→ 今は Episode Draft 右側の「予測される結果」と同じ値を流用する想定。

2-2. MetricDeltaBar コンポーネント

`frontend/src/features/emotion/components/MetricDeltaBar.tsx` (新規)

```
// frontend/src/features/emotion/components/MetricDeltaBar.tsx
import React from "react";

type Props = {
  label: string;
  baseline: number;
  scenario: number;
  goodDirection: "up" | "down";
  min?: number;
  max?: number;
};

export const MetricDeltaBar: React.FC<Props> = ({label, baseline, scenario, goodDirection, min = 0, max = 10}) => {
  const delta = scenario - baseline;
  const isGood =
    (goodDirection === "up" && delta >= 0) ||
    (goodDirection === "down" && delta <= 0);

  const clamp = (v: number) => Math.max(min, Math.min(max, v));
  const baselinePct = ((clamp(baseline) - min) / (max - min)) * 100;
  const scenarioPct = ((clamp(scenario) - min) / (max - min)) * 100;

  return (
    <div className="mb-2">
      <div className="flex items-baseline justify-between">
        <span className="text-[11px] text-gray-600">{label}</span>
        <span className="text-[11px] text-gray-500">
          {baseline.toFixed(1)} →{" "}
          {isGood ? "text-emerald-600" : "text-gray-800"}>
        </span>
      </div>
    </div>
  );
}
```

```

        {scenario.toFixed(1)}
      </span>{" "}
    ({delta >= 0 ? "+" : ""})
    {delta.toFixed(1)})
  </span>
</div>
<div className="relative h-2 mt-1 bg-gray-100 rounded-full overflow-hidden">
  <div
    className="absolute inset-y-0 left-0 bg-gray-300"
    style={{ width: `${baselinePct}%` }}
  />
  <div
    className={`absolute inset-y-0 left-0 ${isGood ? "bg-emerald-400" : "bg-slate-500"}
    opacity-80`}
    style={{ width: `${scenarioPct}%` }}
  />
</div>
</div>
);
};

```

2-3. WhatIfModal 実装

frontend/src/features/emotion/components/WhatIfModal.tsx (新規)

```

// frontend/src/features/emotion/components/WhatIfModal.tsx
import React from "react";
import { useMutation } from "@tanstack/react-query";
import { MetricDeltaBar } from "./MetricDeltaBar";
import { apiClient } from "@/lib/apiClient"; // axios ラッパなどを想定

type PrepName =
  | "writing_10min"
  | "three_messages"
  | "breathing_478"
  | "qa_roleplay"
  | "safe_word";

type BaseState = {
  tear: number;
  calm_after: number;
  relation: number;
  preps: Record<PrepName, boolean>;
};

type Constraints = {
  max_prep_slots: number;
  time_limit_min: number;
};

type ScenarioPrediction = {
  tear: number;
  calm_after: number;
  relation: number;
};

type ScenarioDeltas = ScenarioPrediction;

type WhatIfScenario = {
  id: string;
  objective: "calm_focus" | "message_focus" | "relation_focus";
  label: string;
  description: string;
  prep: Record<PrepName, boolean>;
  pred: ScenarioPrediction;
  deltas: ScenarioDeltas;
};

```

```

confidence: number;
};

type WhatIfResponse = {
  baseline: { pred: ScenarioPrediction };
  scenarios: WhatIfScenario[];
};

type Props = {
  episodeId: number | null;
  baseState: BaseState;
  constraints?: Constraints;
  onApplyScenario: (scenario: WhatIfScenario) => void;
  onClose: () => void;
};

const prepLabel = (name: PrepName): string => {
  switch (name) {
    case "writing_10min":
      return "10分の書き出し";
    case "three_messages":
      return "伝えたい3つのメッセージを整理する";
    case "breathing_478":
      return "4-7-8呼吸法";
    case "qa_roleplay":
      return "自分でQ&Aロールプレイ";
    case "safe_word":
      return "セーフワードを決める";
  }
};

const fetchWhatIf = async (
  episodeId: number | null,
  baseState: BaseState,
  constraints?: Constraints
): Promise<WhatIfResponse> => {
  const body = {
    episode_id: episodeId,
    base: baseState,
    constraints: constraints ?? {
      max_prep_slots: 2,
      time_limit_min: 30,
    },
  };
  const res = await apiClient.post<WhatIfResponse>("/emotion/whatif", body);
  return res.data;
};

export const WhatIfModal: React.FC<Props> = ({
  episodeId,
  baseState,
  constraints,
  onApplyScenario,
  onClose,
}) => {
  const { mutate, data, isLoading, isError } = useMutation({
    mutationFn: () => fetchWhatIf(episodeId, baseState, constraints),
  });

  React.useEffect(() => {
    mutate();
  }, [mutate]);

  const baseline = data?.baseline.pred;

  return (
    <div className="fixed inset-0 z-40 flex items-center justify-center bg-black/40">
      <div className="w-full max-w-5xl mx-4 bg-white rounded-2xl shadow-xl p-6">
        <div className="flex justify-between items-center mb-3">

```

```
<div>
  <h2 className="text-sm font-semibold">
    What if...? 準備プランのシミュレーション
  </h2>
  <p className="text-xs text-gray-600">
    今のお手本をベースに、「少しだけ変えるとしたらどんな組み合わせが良さそうか」を
    あなたのデータから提案します。
  </p>
</div>
<button
  onClick={onClose}
  className="text-xs text-gray-500 hover:text-gray-800"
>
  × 閉じる
</button>
</div>

{isLoading && <p className="text-xs text-gray-500">計算中です...</p>}
{isError && (
  <p className="text-xs text-red-500">
    シミュレーションの取得に失敗しました。
    時間をおいてもう一度お試しください。
  </p>
) }

{baseline && data && (
  <>
  <div className="mb-4 border rounded-xl p-3 bg-slate-50">
    <p className="text-xs font-semibold mb-1">
      現在のプラン (ベースライン)
    </p>
    <div className="grid grid-cols-3 gap-2 text-[11px]">
      <div>
        減レベル: <b>{baseline.tear.toFixed(1)}</b>
      </div>
      <div>
        終わった後の楽さ: <b>{baseline.calm_after.toFixed(1)}</b>
      </div>
      <div>
        関係への影響: <b>{baseline.relation.toFixed(1)}</b>
      </div>
    </div>
    <p className="mt-1 text-[11px] text-gray-500">
      ここから「できそうな準備を 1~2 個だけ増やす」ことを想定しています。
    </p>
  </div>

  <div className="grid md:grid-cols-3 gap-4">
    {data.scenarios.map((scenario) => (
      <div
        key={scenario.id}
        className="border rounded-2xl p-4 bg-white shadow-sm flex flex-col">
        <h3 className="text-xs font-semibold mb-1">
          {scenario.label}
        </h3>
        <p className="text-[11px] text-gray-600 mb-2">
          {scenario.description}
        </p>

        <div className="mb-3">
          <p className="text-[11px] font-semibold mb-1">
            追加でやること
          </p>
          <ul className="text-[11px] text-gray-700 space-y-0.5">
            {Object.entries(scenario.prep)
              .filter(([_, v]) => v)
              .map(([name]) => (
                <li key={name}>· {prepLabel(name as PrepName)}</li>
              ))
          </ul>
        </div>
      </div>
    ))}
  </div>
)}
```

```

        ))}
      </ul>
    </div>

    <MetricDeltaBar
      label="涙レベル"
      baseline={baseline.tear}
      scenario={scenario.pred.tear}
      goodDirection="down"
      min={0}
      max={10}
    />
    <MetricDeltaBar
      label="終わった後の楽さ"
      baseline={baseline.calm_after}
      scenario={scenario.pred.calm_after}
      goodDirection="up"
      min={0}
      max={10}
    />
    <MetricDeltaBar
      label="関係への影響"
      baseline={baseline.relation}
      scenario={scenario.pred.relation}
      goodDirection="up"
      min={-5}
      max={5}
    />

<div className="mt-auto pt-3">
  <p className="text-[11px] text-gray-500 mb-2">
    信頼度:{" "}
    <b>{Math.round(scenario.confidence * 100)}%</b>{" "}
    {scenario.confidence < 0.5
      ? "（まだデータが少ないので、お試しレベルの提案です）"
      : "（これまでの傾向から見ると、かなり可能性が高いパターンです）"
    }
  </p>
  <button
    className="w-full text-xs py-2 rounded-full bg-indigo-600 text-white hover:bg-indigo-700 transition"
    onClick={() => onApplyScenario(scenario)}
  >
    このプランを今回の準備に採用する
  </button>
</div>
</div>
))}

</div>

{data.scenarios.length === 0 && (
  <p className="text-xs text-gray-500 mt-2">
    追加ができる準備がないため、今回はベースラインのみを表示しています。
  </p>
)}

<p className="mt-4 text-[11px] text-gray-400">
  ※ すべてあなた自身のデータから学習した平均的な傾向に基づく予測です。
  「絶対こうなる」という保証ではありませんが、
  次の一歩を考えるヒントとして使ってください。
</p>
</>
)
</div>
</div>
);
};

```

2-4. Episode Draft への組み込み

```
frontend/src/features/emotion/pages/EpisodeDraftPage.tsx (既存ファイルに追記の例)
```

```
// import に追加
import { WhatIfModal } from "../components/WhatIfModal";

// コンポーネント内
const [showWhatIf, setShowWhatIf] = React.useState(false);

// 既存の「予測される結果」計算ロジックから baseline を取得している前提
// 例: prediction = { tear, calmAfter, relation }

// Episode Draft で保持している準備スライダーの state を前提に BaseState を構成
const baseStateForWhatIf: BaseState = {
  tear: prediction.tear,
  calm_after: prediction.calmAfter,
  relation: prediction.relation,
  preps: {
    writing_10min: writing10min > 0,
    three_messages: threeMessages > 0,
    breathing_478: breathing478 > 0,
    qa_roleplay: qaRoleplay > 0,
    safe_word: safeWord > 0,
  },
};

// 「今回の準備プランを保存する」ボタン付近に What-if ボタンを追加
<div className="flex flex-col md:flex-row items-stretch md:items-center gap-2">
  <button
    className="flex-1 md:flex-none px-4 py-2 text-xs rounded-full bg-indigo-600 text-white hover:bg-indigo-700"
    onClick={handleSavePlan}
  >
    今回の準備プランを保存する
  </button>
  <button
    type="button"
    className="flex-1 md:flex-none px-4 py-2 text-xs rounded-full border border-indigo-200 text-indigo-700 hover:bg-indigo-50"
    onClick={() => setShowWhatIf(true)}
  >
    おすすめプランを見る (What-if)
  </button>
</div>

// 最後の JSX にモーダル描画
{showWhatIf && (
  <WhatIfModal
    episodeId={currentEpisodeId}
    baseState={baseStateForWhatIf}
    onApplyScenario={(scenario) => {
      // シナリオの prep を Episode Draft のスライダーに反映
      const p = scenario.prep;
      setWriting10min(p.writing_10min ? 5 : 0);
      setThreeMessages(p.three_messages ? 5 : 0);
      setBreathing478(p.breathing_478 ? 5 : 0);
      setQaRoleplay(p.qa_roleplay ? 5 : 0);
      setSafeWord(p.safe_word ? 5 : 0);
      setShowWhatIf(false);
    }}
    onClose={() => setShowWhatIf(false)}
  />
)}
```

* スライダーを 0/5/10 の 3 段階で持っているなら、5 を「やる」に対応させるなど、細かい値は実装に合わせて調整してください。

2-5. シェルコマンド & 検証

```
# フロントエンド起動  
npm run dev --prefix frontend  
# or  
pnpm dev --filter frontend  
  
# 型チェック  
npm run typecheck --prefix frontend
```

プラウザで Episode Draft を開き：

1. スライダーを少し動かして「予測される結果」が出ている状態で「おすすめプランを見る (What-if)」をクリック。
2. モーダルが開き、ベースラインと3つのシナリオが表示されるか。
3. 「このプランを採用する」を押すと、スライダー側が反映され、予測結果が変化するかを確認。

2-6. What this does (フロント)

- ゴール
 - 「結局どの準備をやればいいのか」が分からぬ状態を、**3つの現実的な準備プラン**にまで落として見せる。
- 入力
 - Episode Draft 内で計算されている予測値 (涙・楽さ・関係) と、各準備スライダーの状態。
- 出力 / 副作用
 - What-if モーダル：
 - ベースライン予測値
 - 3つのシナリオカード (差分と信頼度、自然言語の説明)
 - 「採用」ボタンでスライダーと予測値が更新される。

Expert insight

「おすすめを1つだけ出す」より、「3つの良さそうな選択肢+ユーザーが選ぶ」方が、実際に行動してもらえる確率が高い。
ここでは UI レベルでそのパターンを実装し、因果推定から得た ATE を「小さな行動の選択肢」に変換しています。

最後に

この実装で、Emotion CQOx に

- 短期 (今回の1回) の「準備コーチ」としての役割
- 「次に何を変えればいいか」が数値+言葉で見える支援

が入ります。

次のステップとしてもしやるなら：

- 準備ごとの ATE を、実際の Episode データから推定して **PrepEffect** に流し込むレイヤー
- 長期的な「次の5回のテーマ」用 What-if (Dashboard 側)

を追加していくと、さらに「前向きな一步」を後押しできると思います。