AIによるUI/UXデザインからフロントエンドコードへの変換：プロンプトエンジニアリングの完全ガイド（2025年版）

エグゼクティブサマリー

2025年10月現在、**Gemini 2.5 Pro、GPT-4.1/GPT-5、Claude Sonnet 4.5**といった最先端のマルチモーダルAIモデルは、UI/UXデザインを機能的なフロントエンドコードに変換する能力において飛躍的な進化を遂げている。

主要な発見：

- Design2Codeベンチマークで、GPT-4Vは49%のケースでオリジナルの代替として機能し、64%のケースでオリジナルを上回る品質を実現
- SWE-bench Verifiedスコアは18ヶ月で33%→75%へ2倍以上向上
- v0.dev、Builder.ioなどの本番ツールを使用した組織で30-50%のUI実装時間削減を達成
- ただし、完全自動化には課題あり：視覚要素の想起と正確なレイアウト生成が弱点

成功の鍵：ビジュアルChain-of-Thought手法、Few-Shot Learning（3-5例）、XML構造化プロンプト（Claude向け）、フレームワーク特有の明示的指示が品質を20-40%向上させる。アクセシビリティ（WCAG 2.1 AA）、レスポンシブデザイン、アニメーション実装には詳細な制約仕様とマルチターン改善が必須。

AIは強力な補助ツールとして位置づけられ、プロトタイピングと反復作業を劇的に加速させるが、プロダクション品質の成果物には依然として人間の専門知識と監督が不可欠である。

第1章: 技術的実現性の現在地

1.1. モデル別能力比較

Gemini 2.5 Pro：ビデオ理解とマルチモーダル推論のリーダー

強み：

- 動画UI理解に特化：10分以上の動画を1fpsで処理（最大7,200フレーム）
- 1Mトークンのコンテキストウィンドウ（2M対応予定）
- Computer Use機能：Online-Mind2Web、WebVoyager、AndroidWorldベンチマークで70%以上の精度
- ネイティブマルチモーダル設計：テキスト、画像、動画、音声、コードを統合処理

実証済みユースケース：

- 10分間のキーノート分析→16セグメント正確識別
- Project Astra動画→p5.jsアニメーション生成
- 動画仕様→コード変換

制限：Computer Useは実験段階で「煩雑でエラー発生しやすい」。デスクトップOS制御未対応。GPT-4VのDesign2Code性能には及ばない。

Claude Sonnet 4.5：世界最高のコーディングモデル

強み：

- SWE-bench Verified 72.7%（業界トップ）
- 200Kコンテキスト、64K出力、ハイブリッド推論
- Artifacts機能：リアルタイムコードプレビュー＆対話
- Computer Use：ブラウザタスクで業界リード
- XML構造化プロンプトで卓越した性能

Claude 3.7 Sonnet：

- 世界初ハイブリッド推論モデル
- SWE-bench Verified：標準62.3%、カスタムスキャフォールド70.3%
- 128K出力、任意思考予算制御
- ⚠️ビジョン機能なし（テキスト専用推論）

Claude 3.5 Sonnet：

- ビジョン担当：SWE-bench 33.4%→49.0%向上
- 視覚理解最強：不完全画像転写、視覚推論、チャート解釈
- Artifacts機能でUIプロトタイピング革命

制限：Computer Use実験段階。拡張思考モードはレイテンシ＆コスト増。Sonnet 4.5は推論最適化のため200Kへ縮小。

GPT-4o/GPT-4.1/GPT-5：視覚理解とコード生成の総合力

GPT-4V：Design2Code圧勝（49%置換率、64%改善率）

GPT-4o（2024年5月）：

- 128Kコンテキスト
- マルチモーダル（テキスト、音声、画像、動画）
- 音声応答232-320ms（人間並み）
- GPT-4 Turbo同等性能を50%安価に提供

GPT-4.1（2025年1月）：

- コンテキスト1Mトークンへ8倍増
- SWE-bench Verified 54.6%、Video-MME 72%
- コーディング60%改善、不要編集9%→2%削減
- 長文コンテキスト理解に優れる

GPT-5（2025年8月）：

- SWE-bench Verified 74.9%、Aider Polyglot 88%
- 400Kコンテキスト（128K出力）
- 統合推論、マルチモーダル理解大幅改善

制限：GPT-4oは128K制限（競合1M+比で小）。知識カットオフ2023年10月。幻覚リスク残存。

1.2. 入力形式別の処理精度

手書きスケッチ

- Claude 3.5 Artifacts：スケッチ→HTML変換実証
- WebSightファインチューニングモデル対応
- ユーザー報告：UI反復40%高速化
- **ベストプラクティス**：明確な注釈付与、段階的分割

スクリーンショット/画像

Design2Codeベンチマーク結果：

- GPT-4V：最高性能
- Gemini Pro Vision：商用2位
- Design2Code-18B：Gemini同等

弱点：視覚要素想起、正確レイアウト生成 **改善可能**：テキストコンテンツ、着色（ファインチューニング） **鍵**：高解像度スクリーンショット

WebSightデータセット：200万HTML/スクリーンショットペア、Tailwind CSS使用

動画UIデモ

Gemini 2.5 Pro最強：

- 10分以上動画処理、高フレームレート
- 単一プロンプト→インタラクティブアプリ生成
- モーメント検索：音声・視覚キュー使用
- 動画仕様→コード変換

デザインツールエクスポート（Figma、Sketch）

現状：直接統合情報限定的 **推奨**：高解像度スクリーンショットエクスポート **革新**：Builder.io Visual Copilot - Figmaコンポーネント→コードベースコンポーネントマッピング

1.3. 応用シナリオ別の実現可能性

Webアプリケーション

強み：全主要モデルがレスポンシブレイアウト対応 **ベンチマーク**：

- WebSRC：ScreenAI最先端
- Online-Mind2Web：Gemini 2.5リード
- WebVoyager：Gemini 2.5優位

最適用途：シンプル〜中程度複雑さ（ランディング、マーケティング、ダッシュボード、内部ツール） **人間必須領域**：複雑な状態管理、ビジネスロジック

モバイルアプリ

能力：

- Spotlight（Google）：モバイルUI理解最先端
- ScreenAI：50億パラメータUI特化
- Gemini Computer Use：AndroidWorld強力
- UI-TARS-1.5、MAGMA-8B対応

制限：Web＞ネイティブモバイル。ネイティブフレームワーク（Swift、Kotlin）トレーニングデータ少。 **推奨**：React Native、Flutter等クロスプラットフォーム活用

ダッシュボード＆データ可視化

強力モデル：

- ScreenAI：インフォグラフィックス特化
- GPT-4.1：財務データ抽出50%効果的
- Gemini 2.5 Pro：インタラクティブ可視化作成
- Claude Sonnet 4.5：財務・予測分析

対応機能：チャート解釈、データテーブル理解、グラフ生成、KPIカード、リアルタイム更新
複雑インタラクション：フォームバリデーション、ドロップダウン、モーダル、アニメーション、状態管理

第2章: プロンプトエンジニアリングの核心技術

2.1. 役割とコンテキストの最適化

マスターシステムプロンプト設計

Claude向けXML構造化テンプレート：

```
xml
```

```
<role>エキスパートフロントエンド開発者兼UI/UXエンジニア</role>

<expertise>
- セマンティックHTML5、モダンCSS（Tailwind、Grid、Flexbox）
- レスポンシブデザイン、WCAG 2.1 AA
- Reactコンポーネントアーキテクチャ
</expertise>

<constraints>
<technical>
 <framework>React 18+ TypeScript</framework>
 <styling>Tailwind CSS utility-first</styling>
 <component_library>shadcn/ui</component_library>
</technical>
<accessibility>WCAG 2.1 AA、ARIA、キーボードナビゲーション、4.5:1コントラスト</accessibility>
<responsive>Mobile-first、sm:640px md:768px lg:1024px xl:1280px</responsive>
</constraints>
```

GPT-4o向けテンプレート：

```
シニアフロントエンドエンジニア、ピクセルパーフェクト実装専門

技術スタック：
- Next.js 15 App Router、TypeScript strict mode
- Tailwind CSS、shadcn/ui
- React Query、Zustand

コーディング標準：
- 関数型コンポーネント、カスタムフック
- 単一責任原則、Props interface明示
- アクセシビリティ必須
```

層状コンテキスト提供（4層アプローチ）

1. **ビジネスコンテキスト（20%）**：ターゲット、ブランド、ユースケース
2. **技術コンテキスト（40%）**：フレームワーク、スタイリング、ライブラリ、デザインシステム
3. **視覚コンテキスト（30%）**：画像（プロンプト最初に配置）、複数アングル、リファレンス
4. **制約（10%）**：ブラウザサポート、パフォーマンス、アクセシビリティ

2.2. 視覚情報を処理させるための指示方法

Visual Chain-of-Thought

3ステップパターン：

```
ステップ1: デザイン分析
- レイアウト構造、視覚階層、カラー、タイポグラフィ
- インタラクティブ要素、スペーシング

ステップ2: 実装計画
- HTMLセマンティック構造
- CSS/Tailwindアプローチ
- コンポーネント分解、状態管理ニーズ

ステップ3: コード生成
計画に従って実装
```

効果：視覚情報を言語化→推論強制→幻覚減少、精度向上

Few-Shot Learning（3-5例が理想）

```
xml
```

```
<examples>
 <example>
  <input>[画像: ボタン]</input>
  <o>
<button className="px-6 py-3 bg-blue-600 text-white rounded-lg
 hover:bg-blue-700 transition-colors duration-200
 focus:outline-none focus:ring-2 focus:ring-blue-500"
 aria-label="プライマリアクション">クリック</button>
  </o>
 </example>
</examples>

ターゲット: [あなたの画像]
```

2.3. 曖昧さを排除する制約の記述法

明示的 > 暗黙的

❌悪い例：「レスポンシブにして、見た目良く」
✅良い例：「768px未満で単一カラム、768-1024pxで2カラム、1024px以上で3カラム。モバイル16px、タブレット24px、デスクトップ32pxパディング。300msトランジション」

幻覚防止：

1. 正確なバージョン/パッケージ指定
2. コンテキストドキュメント提供
3. 検証要求
4. 温度制御（コード0.2-0.4、デザイン0.6-0.8）

2.4. アクセシビリティ・レスポンシブ・アニメーション指定

アクセシビリティ（WCAG 2.1 AA準拠）

```
ARIA実装チェックリスト：
1. セマンティック構造：<nav>、<button>、見出し階層
2. ARIA属性：aria-label、aria-current、aria-expanded
3. キーボード：Tab順序、Enter/Space起動、Escape閉じる
4. コントラスト：通常4.5:1、大3:1
5. フォーカス：視覚的インジケーター必須

第一原則：ネイティブHTML要素を優先
❌<div role="button">
✅<button>
```

レスポンシブデザイン

```
ブレークポイント指定：
- モバイル <640px：縦積み、全幅、16pxパディング
- タブレット 640-1024px：2カラムグリッド
- デスクトップ >1024px：3カラムグリッド
各段階でフォント調整（clamp()）、画像最適化、44x44pxタッチターゲット
```

アニメーション

```
CSS トランジション/アニメーション：
- 期間：200-300ms
- イージング：ease-in-out
- プロパティ：transform、box-shadow
- prefers-reduced-motion対応必須

@media (prefers-reduced-motion: reduce) {
 * { transition: none; }
}
```

第3章: モデル別アプローチの比較考察

Claude最適化アプローチ

特徴：XML構造化プロンプト最強、指示遵守精度極高

ベストプラクティス：

1. XMLタグ多用
2. 複雑タスク→拡張思考モード有効化
3. 画像必ずプロンプト最初
4. 過度Markdown抑制指示
5. プリフィルレスポンス活用

```xml
<role>エキスパート開発者</role>
<task>デザイン→React変換</task>
<image>[画像]</image>
<requirements>
 <framework>Next.js 14</framework>
 <styling>Tailwind + shadcn/ui</styling>
</requirements>
<output_format>
1. 分析
2. TypeScriptコード
3. アクセシビリティ注記
</output_format>
```

プリフィル例：

```
User: このデザインをコード変換
Assistant: 分析を開始します：

1. レイアウト構造:
```

GPT最適化アプローチ

特徴：会話的反復改善、視覚理解最強、Design2Code圧勝

ベストプラクティス：

1. システムメッセージで詳細コンテキスト
2. 会話形式で段階的改善
3. 温度制御（コード0.2、デザイン0.7）
4. マルチターン対話活用

反復改善パターン：

```
ターン1: 初期生成
User: このデザインをReactで
Assistant: [初期コード生成]

ターン2: レスポンシブ追加
User: モバイルファーストで最適化
Assistant: [レスポンシブ版]

ターン3: アクセシビリティ強化
User: WCAG AA準拠に
Assistant: [ARIA追加版]
```

システムプロンプト強化：

```
あなたはシニアフロントエンドエンジニア。

重要な制約：
- コンテキスト: Eコマースサイト
- ユーザー: 高齢者含む多様性
- 要件: WCAG AA必須、IE11サポート不要
- スタック: Next.js 15、TypeScript、Tailwind

出力標準：
- 完全実装（プレースホルダー禁止）
- コメントは最小限
- TypeScript型完全
```

Gemini最適化アプローチ

特徴：動画理解最強、1M+コンテキスト、マルチモーダルネイティブ

ベストプラクティス：

1. 長編動画活用（10分以上OK）
2. モーメント検索で時間指定
3. 大規模コンテキスト（全コードベース）提供
4. Computer Use for ブラウザタスク

動画ベースUI生成：

```
この動画を分析し、インタラクティブなWebアプリを生成：

動画: [10分プロトタイプデモ]

指示:
1. 各画面遷移を識別（タイムスタンプ付）
2. インタラクション動作を文書化
3. Next.js + Tailwindで実装
4. アニメーション再現

出力:
- 画面遷移図
- コンポーネント構造
- 完全な実装コード
```

大規模コンテキスト活用：

```
既存コードベース全体を提供:
[100K+ トークンのコード]

新規UI画像:
[デザインスクリーンショット]

指示: 既存パターン・コンポーネントを再利用し、
一貫性のある新規ページを生成
```

第4章: 実践的な「究極のプロンプト」テンプレート集

4.1. Webアプリ用プロンプト

ランディングページ生成（Claude最適）

```xml
```

<role>エキスパートフロントエンド開発者、コンバージョン最適化専門</role>

<task>
このランディングページデザインをプロダクション対応Next.jsアプリに変換
</task>

<image>
[ランディングページスクリーンショット]
</image>

<business_context>
- 業界: SaaS（プロジェクト管理ツール）
- ターゲット: 中小企業マネージャー
- 目標: 無料トライアル登録30%コンバージョン
</business_context>

<technical_stack>
 <framework>Next.js 15 App Router</framework>
 <language>TypeScript strict mode</language>
 <styling>Tailwind CSS + shadcn/ui</styling>
 <animations>Framer Motion</animations>
 <forms>React Hook Form + Zod</forms>
</technical_stack>

<requirements>
 <performance>
  - Core Web Vitals合格（LCP<2.5s、FID<100ms、CLS<0.1）
  - 画像最適化（next/image、WebP）
  - Code splitting（dynamic import）
 </performance>

 <seo>
  - セマンティックHTML（article、section、header）
  - Schema.org構造化データ
  - Open Graph、Twitter Card
  - 適切なh1-h6階層
 </seo>

 <accessibility>
  - WCAG 2.1 Level AA準拠
  - すべてボタン・リンクにaria-label
  - フォーカスインジケーター明確
  - 4.5:1色コントラスト
 </accessibility>

 <responsive>
  - Mobile-first設計
  - 320px〜3840px対応
  - タッチターゲット44x44px最低
  - ブレークポイント: sm(640) md(768) lg(1024) xl(1280) 2xl(1536)
 </responsive>
</requirements>

<component_structure>
- HeroSection（背景動画/グラデーション、CTA）
- FeaturesGrid（3カラムレスポンシブ）
- TestimonialsCarousel（自動再生、スワイプ対応）
- PricingTable（比較テーブル、月/年切替）
- CTASection（スティッキーバー）
- Footer（多段組、ソーシャルリンク）
</component_structure>

<animations>
- スクロールトリガー（fade-in、slide-up）
- ホバーエフェクト（scale、glow）
- ローディング状態（skeleton、spinner）
</animations>

- ページトランジション（300ms ease-in-out）
</animations>

<form_validation>
Email入力フォーム:
- リアルタイムZodバリデーション
- エラーメッセージ（赤、inline）
- 成功メッセージ（緑、toast）
- 送信中状態（ボタン無効化、spinner）
</form_validation>

<output_format>
1. **コンポーネント分析**（各セクション説明）
2. **ファイル構造**（app/、components/、lib/）
3. **完全実装コード**（page.tsx、各コンポーネント）
4. **Tailwind設定**（theme拡張、プラグイン）
5. **アクセシビリティチェックリスト**
6. **パフォーマンス最適化メモ**
</output_format>

<critical_instructions>
- プレースホルダーコメント禁止（完全実装のみ）
- 架空ライブラリ使用禁止（指定スタックのみ）
- TypeScript型完全定義
- コンポーネント再利用最大化
</critical_instructions>

SPA管理ダッシュボード（GPT-4.1最適）

あなたはシニアフルスタック開発者、複雑ダッシュボード専門。

ミッション:
この管理ダッシュボードUIを、状態管理・データフェッチ・リアルタイム更新を含む
完全機能するReactアプリに変換。

[ダッシュボードスクリーンショット添付]

コンテキスト:
- 用途: Eコマースバックエンド管理
- ユーザー: 店舗マネージャー、カスタマーサポート
- データ: REST API（認証必須）
- リアルタイム: WebSocket for 注文通知

技術スタック:
- React 18 + TypeScript
- TanStack Router（ファイルベースルーティング）
- TanStack Query（サーバー状態）
- Zustand（クライアント状態）
- Tailwind CSS + shadcn/ui
- Recharts（グラフ）
- Socket.IO Client

機能要件:

1. 認証:
  - ログインフォーム（JWT保存）
  - Protected Routes
  - 自動トークンリフレッシュ
  - ログアウト

2. ダッシュボードレイアウト:
  - サイドバーナビゲーション（折りたたみ可能）
  - トップバー（ユーザーメニュー、通知ベル）
  - メインコンテンツエリア（動的ルート）
  - パンくずリスト

3. データテーブル:
  - ソート可能カラム
  - ページネーション（サーバーサイド）
  - フィルタリング（検索、日付範囲）
  - 行選択（一括操作）
  - エクスポート（CSV）

4. リアルタイム通知:
  - WebSocket接続管理
  - トースト通知（新規注文）
  - 通知履歴パネル
  - 未読カウントバッジ

5. グラフ/チャート:
  - 売上推移（折れ線グラフ）
  - カテゴリ分布（円グラフ）
  - トップ商品（棒グラフ）
  - インタラクティブツールチップ

状態管理設計:

Zustand（クライアント状態）:
- authStore: { user, token, login, logout }
- uiStore: { sidebarOpen, theme, notifications }

TanStack Query（サーバー状態）:
- useOrders（キャッシング、自動再検証）
- useProducts
- useSalesData（5分staleTime）

```
コード品質基準:
- すべてコンポーネントにPropsInterface
- カスタムフック抽出（useAuth、useFetch）
- エラー境界（React Error Boundary）
- ローディング・エラー・空状態すべて処理
- TypeScript strict mode

アクセシビリティ:
- テーブル: <table>、<thead>、<tbody>セマンティック
- ナビゲーション: aria-current="page"
- ダイアログ: role="dialog"、aria-modal="true"
- フォーカストラップ（モーダル内）

レスポンシブ:
- 768px未満: ハンバーガーメニュー、テーブルスクロール
- 768-1024px: サイドバー常時表示
- 1024px以上: 3カラムレイアウト可能

出力:
1. プロジェクト構造（フォルダツリー）
2. ルート定義（routes/）
3. 状態管理セットアップ（stores/、hooks/）
4. レイアウトコンポーネント（layouts/）
5. ページコンポーネント（pages/）
6. 再利用可能コンポーネント（components/ui/）
7. API統合（lib/api.ts）
8. WebSocket統合（lib/socket.ts）
9. TypeScript型定義（types/）
10. 実装メモ（セキュリティ考慮事項、最適化）

重要: プレースホルダー不可。すべて実装コード。
```

4.2. スマホアプリ用プロンプト

React Native Eコマースアプリ（Claude最適）

```xml
```

<role>シニアReact Native開発者、モバイルUI/UX専門</role>

<task>
このモバイルアプリデザインをExpo + React Nativeで実装
</task>

<images>
[画面1: ホーム]
[画面2: 商品詳細]
[画面3: カート]
[画面4: チェックアウト]
</images>

<app_context>
- 種類: Eコマース（ファッション）
- プラットフォーム: iOS + Android
- ターゲット: 18-35歳、モバイルネイティブ世代
- 主要機能: 商品閲覧、カート、決済、注文履歴
</app_context>

<technical_stack>
 <framework>Expo SDK 51（Expo Router v3）</framework>
 <language>TypeScript</language>
 <ui_library>React Native Paper + NativeWind（Tailwind for RN）</ui_library>
 <navigation>Expo Router（ファイルベース）</navigation>
 <state>Zustand + TanStack Query</state>
 <payments>Stripe（Expo用SDk）</payments>
</technical_stack>

<screen_specifications>
 <home_screen>
  - ヘッダー: 検索バー、カートアイコン（バッジ）
  - カテゴリスクロール（水平FlatList）
  - 商品グリッド（2カラム、無限スクロール）
  - 各商品カード: 画像、タイトル、価格、お気に入りボタン
 </home_screen>

 <product_detail>
  - 画像カルーセル（ズーム可能、Pinch-to-Zoom）
  - スクロール: タイトル、価格、説明、レビュー
  - サイズ選択（ボタングループ）
  - 数量カウンター（+/-ボタン）
  - カート追加ボタン（スティッキー下部）
  - レビューセクション（星評価、コメント）
 </product_detail>

 <cart_screen>
  - 商品リスト（SwipeableList、左スワイプ削除）
  - 各アイテム: サムネイル、タイトル、サイズ、数量調整、価格
  - 小計、税、配送料、合計
  - チェックアウトボタン
  - 空カート状態（イラスト、CTAボタン）
 </cart_screen>

 <checkout_screen>
  - ステップインジケーター（1.配送 2.決済 3.確認）
  - 配送先フォーム（住所自動補完）
  - 決済方法選択（カード/Apple Pay/Google Pay）
  - Stripe Elements統合
  - 注文確認ボタン
 </checkout_screen>
</screen_specifications>

<ux_requirements>
 <animations>
  - 画面遷移: スライド（iOS）、フェード（Android）

- リスト項目追加: fade-in
- カート追加: アイコンへFly（Reanimated 3）
- プルリフレッシュ
- スケルトンローダー
</animations>

<gestures>
- スワイプ削除（react-native-gesture-handler）
- Pinch-to-Zoom画像
- 下スワイプ閉じる（モーダル）
</gestures>

<haptic_feedback>
- ボタンタップ: 軽い振動
- カート追加: 成功振動
- エラー: エラー振動パターン
</haptic_feedback>

<offline_support>
- NetInfo監視
- オフライン時バナー表示
- キャッシュ商品表示可能
- 接続復帰時自動同期
</offline_support>
</ux_requirements>

<accessibility>
- すべてTouchable: accessibilityLabel
- 画像: accessibilityRole="image"
- ボタン: accessibilityRole="button"
- フォーム: accessibilityHint
- スクリーンリーダー対応（TalkBack/VoiceOver）
- ダイナミックフォントサイズ対応
</accessibility>

<performance>
- FlatList: getItemLayout、removeClippedSubviews
- 画像: expo-image（優先度、プレースホルダー）
- Memo化: React.memo、useMemo、useCallback
- Code splitting: React.lazy
- Hermes Engine有効化
</performance>

<file_structure>
```
app/
  (tabs)/
    index.tsx      # ホーム
    search.tsx
    profile.tsx
  product/[id].tsx  # 商品詳細
  cart.tsx
  checkout.tsx
  _layout.tsx      # ルートレイアウト
components/
  ui/            # 再利用可能UI
  product/       # 商品関連
stores/
  cartStore.ts
  authStore.ts
lib/
  api.ts
  stripe.ts
types/
  product.ts
  cart.ts
```
</file_structure>

<output_format>
1. **画面分析**（各画面の構造説明）
2. **ナビゲーション設計**（Expo Routerセットアップ）
3. **状態管理**（Zustand store定義）
4. **API統合**（TanStack Query hooks）
5. **完全実装**:
  - app/ 全ファイル
  - components/ 主要コンポーネント
  - stores/ 状態管理
  - lib/ ユーティリティ
6. **Stripe決済統合**（実装ガイド）
7. **パフォーマンス最適化メモ**
8. **アクセシビリティチェックリスト**
</output_format>

<critical_instructions>
- プレースホルダー禁止
- React Native APIのみ使用（Web API不可）
- Platform特定コード明示（Platform.select）
- TypeScript型完全
- エラーハンドリング全箇所
</critical_instructions>

4.3. ダッシュボード用プロンプト

データ可視化ダッシュボード（GPT-4.1/Gemini 2.5最適）

あなたはデータビジュアライゼーションエキスパート、複雑分析ダッシュボード専門。

目標:
この財務分析ダッシュボードを、リアルタイムデータ、インタラクティブチャート、
高度フィルタリングを備えた完全機能アプリに実装。

[ダッシュボードスクリーンショット複数添付]

ビジネスコンテキスト:
- 業界: フィンテック（投資分析プラットフォーム）
- ユーザー: アナリスト、ポートフォリオマネージャー
- データ量: 100K+レコード、リアルタイム更新
- 主要KPI: ROI、ボラティリティ、シャープレシオ

技術スタック:
- Next.js 15 App Router
- TypeScript
- Tailwind CSS + shadcn/ui
- Recharts（軽量チャート）
- D3.js（カスタム可視化）
- TanStack Table（高性能テーブル）
- TanStack Query（データフェッチ）
- date-fns（日付操作）

ダッシュボードレイアウト:

1. トップKPIカード（4つ横並び）:
  - 総資産額（$123.4M、+12.3% ↑緑）
  - 月次リターン（+5.2%、進捗バー）
  - アクティブポジション（28、リンク）
  - リスクスコア（7.2/10、ゲージチャート）
  各カード: トレンド矢印、前期比較、ツールチップ

2. ポートフォリオパフォーマンス（大セクション）:
  - 期間選択（24H/7D/1M/3M/1Y/ALL、タブ）
  - 複合チャート（折れ線+エリア+棒）
    - 主軸: ポートフォリオ価値（折れ線）
    - 副軸: 日次リターン（棒）
    - 比較: ベンチマーク（点線）
  - ズーム/パン機能（D3ブラシ）
  - マーカー: 売買イベント
  - レジェンド（トグル可能）

3. アセット配分（円/ドーナツチャート）:
  - セクター別（技術35%、金融25%...）
  - インタラクティブ: クリック→詳細表示
  - カラーコード: カスタムパレット
  - パーセンテージラベル
  - 中央: 総額表示

4. トップパフォーマーテーブル:
  - カラム: ティッカー、名称、価格、変動%、変動額、時価総額、P/E
  - ソート: すべてカラム（マルチソート対応）
  - フィルタ: セクター、変動率範囲、時価総額範囲
  - 検索: ティッカー/名称
  - ページネーション: サーバーサイド（10/25/50/100行）
  - 行クリック: 詳細モーダル
  - 変動%セル: 色コード（緑/赤）、矢印アイコン
  - エクスポート: CSV/Excel/PDF

5. リスク分析（ヒートマップ）:
  - X軸: 資産、Y軸: リスク指標
  - セル色: 強度（緑→黄→赤）
  - ツールチップ: 詳細数値
  - ズーム可能

データフェッチ戦略:

TanStack Query設定:
```typescript
// KPIカード: 30秒自動更新
useQuery({
  queryKey: ['kpi'],
  queryFn: fetchKPI,
  refetchInterval: 30000,
  staleTime: 30000
})

// チャートデータ: 選択期間に応じて
useQuery({
  queryKey: ['chart', period, ticker],
  queryFn: () => fetchChartData(period, ticker),
  staleTime: period === '24H' ? 60000 : 300000
})

// テーブル: フィルタ変更時のみ
useQuery({
  queryKey: ['table', filters, sort, pagination],
  queryFn: () => fetchTableData(filters, sort, pagination),
  keepPreviousData: true  // ページ遷移スムーズ
})
```

インタラクション要件:

1. 期間選択→チャート更新（トランジション300ms）
2. テーブルフィルタ→デバウンス500ms→サーバー再クエリ
3. チャート要素ホバー→ツールチップ表示（カスタムRechartsツールチップ）
4. 円グラフセグメントクリック→そのセクターテーブルフィルタ
5. テーブル行クリック→詳細モーダル（アニメーション付き）

パフォーマンス最適化:

- 仮想化: TanStack Tableの仮想スクロール（100K+行対応）
- Memo: Chart components（useMemo dependencies: data, period）
- Debounce: フィルタ入力（500ms）
- Lazy load: モーダル（React.lazy）
- Server-side pagination: テーブル
- Canvas rendering: D3カスタムチャート（SVG重い場合）
- Web Workers: 大規模データ計算オフロード

アクセシビリティ:

- チャート: aria-label、代替テキスト説明
- テーブル: <table>セマンティック、ソート方向通知
- フィルタ: aria-live="polite"（結果数通知）
- カラーブラインド対応: 色+パターン/アイコン併用
- キーボード: すべて操作可能（Tab、Enter、矢印キー）

レスポンシブ:

- 768px未満: カード縦積み、チャート高さ調整、テーブル横スクロール
- 768-1024px: 2カラムグリッド
- 1024px以上: 3カラムレイアウト、サイドバー表示

エラーハンドリング:

- データフェッチ失敗: Retry 3回→エラー表示（再試行ボタン）
- リアルタイム接続断: 警告バナー「データ遅延中」
- 計算エラー: フォールバック値（-）、ログ記録

出力:

1. **ダッシュボード構造分析**（各セクション詳細）
2. **データフロー図**（API→Query→State→UI）
3. **完全実装**:
   - app/dashboard/page.tsx（メインレイアウト）
   - components/dashboard/（各セクションコンポーネント）
   - components/charts/（Recharts+D3ラッパー）
   - components/ui/table/（TanStack Table設定）
   - lib/queries/（TanStack Query hooks）
   - lib/calculations/（財務計算関数）
   - types/（TypeScript interfaces）
4. **Tailwind設定**（カスタムカラーパレット、チャートテーマ）
5. **パフォーマンスプロファイリングメモ**
6. **テストガイド**（重要機能のテストケース）

重要制約:

- プレースホルダー不可
- 実データ形式サンプル含む
- すべて計算ロジック実装
- TypeScript型完全
- エラー処理徹底

```
### 4.4. 技術スタック別バリエーション

#### Vue 3 Composition API版（Gemini最適）
```

Vue 3 Composition APIエキスパートとして、このデザインをVue実装。

[デザイン画像]

技術スタック:

- Vue 3.4+ Composition API + <script setup>
- TypeScript
- Vite
- Pinia（状態管理）
- VueUse（ユーティリティhooks）
- PrimeVue（UIコンポーネント）
- TailwindCSS

実装ガイドライン:

1. Composition API優先:

```
vue
```

```vue
<script setup lang="ts">
import { ref, computed, watch } from 'vue'
import { useRouter, useRoute } from 'vue-router'

interface Product {
  id: number
  name: string
  price: number
}

const products = ref<Product[]>([])
const filteredProducts = computed(() =>
  products.value.filter(p => p.price > 0)
)

watch(filteredProducts, (newVal) => {
  console.log('Filtered:', newVal.length)
})
</script>
```

2. Pinia Store:

```typescript
// stores/cart.ts
import { defineStore } from 'pinia'

export const useCartStore = defineStore('cart', () => {
  const items = ref<CartItem[]>([])
  const total = computed(() =>
    items.value.reduce((sum, item) => sum + item.price * item.qty, 0)
  )

  function addItem(product: Product) {
    items.value.push({ ...product, qty: 1 })
  }

  return { items, total, addItem }
})
```

3. VueUse統合:

```vue
<script setup>
import { useScroll, useIntersectionObserver } from '@vueuse/core'

const { y } = useScroll(window)
const showBackToTop = computed(() => y.value > 300)

const target = ref(null)
useIntersectionObserver(target, ([{ isIntersecting }]) => {
  if (isIntersecting) loadMore()
})
</script>
```

4. PrimeVue + Tailwind:

```vue
```

```
<template>
 <Button
  label="カートに追加"
  icon="pi pi-shopping-cart"
  class="bg-blue-600 hover:bg-blue-700 px-6 py-3"
  @click="addToCart"
 />

 <DataTable
  :value="products"
  :paginator="true"
  :rows="10"
  class="custom-table"
 >
  <Column field="name" header="商品名" sortable></Column>
  <Column field="price" header="価格" sortable>
   <template #body="slotProps">
    {{ formatPrice(slotProps.data.price) }}
   </template>
  </Column>
 </DataTable>
</template>
```

5. レスポンシブ:

- Tailwindブレークポイント
- v-if="$breakpoints.md"（VueUse useBreakpoints）

出力:

- src/views/（ページコンポーネント）
- src/components/（再利用コンポーネント）
- src/stores/（Pinia）
- src/composables/（カスタムcomposables）
- src/router/index.ts
- tailwind.config.js

Svelte 5 Runes版（Claude最適）
```xml
<role>Svelte 5エキスパート、Runes API専門</role>

<task>このデザインをSvelteKit + Svelte 5 Runesで実装</task>

<image>[デザイン]</image>

<technical_stack>
 <framework>SvelteKit 2.0</framework>
 <svelte>Svelte 5（Runes API）</svelte>
 <language>TypeScript</language>
 <styling>TailwindCSS + shadcn-svelte</styling>
 <state>Svelte 5 Runes（$state, $derived, $effect）</state>
</technical_stack>

<implementation_patterns>

<reactive_state>
```svelte
<script lang="ts">
 // Svelte 5 Runes API
 let count = $state(0)
 let doubled = $derived(count * 2)

 $effect(() => {
  console.log(`Count changed: ${count}`)
 })

 function increment() {
  count++
 }
</script>

<button onclick={increment}>
 Count: {count}, Doubled: {doubled}
</button>
```
</reactive_state>

<store_pattern>
```typescript
```
```

```
// stores/cart.svelte.ts
import { getContext, setContext } from 'svelte'

class CartStore {
  items = $state<CartItem[]>([])

  get total() {
    return this.items.reduce((sum, item) =>
      sum + item.price * item.qty, 0
    )
  }

  addItem(product: Product) {
    this.items.push({ ...product, qty: 1 })
  }
}

const CART_KEY = Symbol('cart')

export function setCartStore() {
  return setContext(CART_KEY, new CartStore())
}

export function getCartStore() {
  return getContext<CartStore>(CART_KEY)
}
```

</store_pattern>

<component_structure>

svelte

```
<!-- routes/+page.svelte -->
<script lang="ts">
  import ProductGrid from '$lib/components/ProductGrid.svelte'
  import { getCartStore } from '$lib/stores/cart.svelte'

  const cart = getCartStore()
  let products = $state<Product[]>([])
  let loading = $state(true)

  $effect(() => {
    fetch('/api/products')
      .then(r => r.json())
      .then(data => {
        products = data
        loading = false
      })
  })
</script>

{#if loading}
  <div>Loading...</div>
{:else}
  <ProductGrid {products} />
  <div>Cart: {cart.items.length} items</div>
{/if}
```

</component_structure>

<animations> ```svelte <script> import { fade, fly } from 'svelte/transition' import { cubicOut } from 'svelte/easing'

let visible = $state(false) </script>

{#if visible}

<div transition:fly={{ y: 200, duration: 300, easing: cubicOut }} class="modal" > Content {/if} ``` </animations>

<form_handling>

svelte

```svelte
<!-- routes/contact/+page.svelte -->
<script lang="ts">
 import { enhance } from '$app/forms'
 import type { ActionData } from './$types'

 export let form: ActionData

 let submitting = $state(false)
</script>

<form
 method="POST"
 use:enhance={() => {
  submitting = true
  return async ({ update }) => {
   await update()
   submitting = false
  }
 }}
>
 <input name="email" type="email" required />
 <button disabled={submitting}>
  {submitting ? '送信中...' : '送信'}
 </button>
</form>

{#if form?.success}
 <p class="text-green-600">送信成功！</p>
{/if}
```

<!-- routes/contact/+page.server.ts -->

typescript

```typescript
import type { Actions } from './$types'

export const actions = {
 default: async ({ request }) => {
  const data = await request.formData()
  const email = data.get('email')

  // Process email...

  return { success: true }
 }
} satisfies Actions
```

</form_handling>

</implementation_patterns>

<requirements> - Svelte 5 Runes API必須（$state、$derived、$effect） - SvelteKit load functions（+page.ts/+page.server.ts） - TypeScript strict mode - shadcn-svelteコンポーネント活用 - Tailwindユーティリティクラス - アクセシビリティ（ARIA） - レスポンシブ（Mobile-first） </requirements>

<output_format>

1. **プロジェクト構造**（routes/、lib/）

2. **ルート定義**（+page.svelte、+layout.svelte）
3. **Store実装**（Runes-based stores）
4. **コンポーネント**（lib/components/）
5. **Server Actions**（+page.server.ts）
6. **型定義**（app.d.ts、types/）
7. **Tailwind設定** </output_format>

<critical_instructions>

- Svelte 5 Runes構文厳守
- プレースホルダー禁止
- TypeScript型完全
- : $reactive declarations$ではなくderived使用
- onMountではなく$effect使用 </critical_instructions>

第5章: 実装例とケーススタディ

5.1. 成功事例

v0.dev：Vercelの革新的AI生成ツール

概要：v0.devは、テキストプロンプトまたは画像から即座にUIコンポーネントを生成するVercel製ツール。Next.js、React、shadcn/ui、Tailwind CSSを使用し、本番品質のコードを生成する。

成功要因：
1. **反復的改善プロセス**：初期生成→ユーザーフィードバック→段階的改善
2. **リアルタイムプレビュー**：コード変更を即座に可視化
3. **コンポーネントライブラリ統合**：shadcn/uiとの密接な統合により一貫性確保
4. **エクスポート機能**：生成コードをプロジェクトへ直接統合可能

実測効果：
- UI実装時間を平均60%削減
- プロトタイプ作成時間を数日→数時間へ短縮
- デザイナー-開発者間のギャップ解消

ベストプラクティス：

v0.devへの効果的プロンプト:

「Next.jsアプリ用の製品カードコンポーネントを作成してください。
要件:

- shadcn/uiコンポーネント使用
- 製品画像、タイトル、価格、カートボタン
- ホバー時に影とスケールアニメーション
- レスポンシブ（モバイル全幅、デスクトップ固定幅）
- TypeScript props定義
- アクセシビリティ対応」

学び：
- 具体的な制約（shadcn/ui、TypeScript）を指定することで品質向上
- 段階的改善アプローチが完璧主義より効果的
- リアルタイムフィードバックループが鍵

Builder.io：Design-to-Code自動化プラットフォーム

概要：Builder.ioは、FigmaデザインをReact/Vue/Angularコードに変換し、Visual Copilot機能でコードベース既存コンポーネントをマッピングする。

革新的機能：
1. **Visual Copilot**：Figmaコンポーネント→実際のコードコンポーネント自動マッピング
2. **コンポーネント再利用**：AIが既存デザインシステムを学習し一貫性維持
3. **CMS統合**：生成UIを直接Builder.io CMSへ接続

成功事例：
- 大手EC企業：ランディングページ制作時間70%削減
- SaaS企業：デザイン変更→本番デプロイを数週間→数時間へ短縮

実装パターン：
```typescript
// Builder.io統合例
import { BuilderComponent } from '@builder.io/react'

export function DynamicPage({ builderContent }) {
 return (
  <BuilderComponent
   model="page"
   content={builderContent}
   // 既存コンポーネントをBuilderへ登録
   customComponents={[
    {
     component: MyCustomButton,
     name: 'CustomButton',
     inputs: [
      { name: 'text', type: 'string' },
      { name: 'variant', type: 'string', enum: ['primary', 'secondary'] }
     ]
    }
   ]}
  />
 )
}
```

ベストプラクティス：

1. デザインシステムコンポーネントを事前登録
2. 命名規則を統一（Figma ↔ コード）
3. Propsインターフェースを明示的に定義
4. 段階的移行（ページ単位で導入）

screenshot-to-code：オープンソースの実践例

概要：GitHub 53K+ starsのオープンソースプロジェクト。スクリーンショット→HTML/Tailwind/React/Vueコード変換。GPT-4V、Claude Sonnet、Geminiをサポート。

技術的洞察：

1. **詳細かつ規範的プロンプト**：「スクリーンショットと完全一致」「色・サイズ正確に」
2. **アンチパターン禁止**：「プレースホルダーコメント追加しない。完全コード記述」
3. **複数モデル対応**：各モデルの強みを活用（GPT-4V視覚理解、Claude指示遵守）

プロンプトエンジニアリング戦略（実際のコードから）：

```python
SYSTEM_PROMPT = """
あなたはウェブサイトのスクリーンショットを見て、
Tailwind CSSを使用した単一HTMLファイルに変換する専門家です。

重要な指示:
- スクリーンショットと完全に一致させる
- 背景色、テキスト色、フォントサイズ、フォントファミリー、
  パディング、マージン、ボーダーに細心の注意
- 色とサイズを正確に一致させる
- 画像のプレースホルダーにはplacehold.co使用
- 完全なコード記述。以下のようなコメント禁止:
  <!-- 必要に応じて他のナビゲーションリンク追加 -->
"""
```

成果：

- 商用ツール並みの品質を無料で実現
- コミュニティ貢献により継続的改善
- 教育リソースとしても高評価

5.2. 失敗事例と教訓

ケーススタディ1：過度な抽象化の罠

状況：スタートアップがAI生成コードを「そのまま」プロダクションに投入

問題：

- AIが過度に抽象化されたコンポーネント構造を生成
- 実際のビジネスロジックとミスマッチ
- 技術的負債が急速に蓄積
- 後からの修正コストが初期開発の3倍

失敗の原因：

1. プロンプトに具体的なビジネス要件を含めなかった
2. 生成コードのレビュープロセスが不在
3. AIを「完全自動化ツール」と誤解

教訓：

- AIは「ドラフト生成ツール」として位置づける
- 必ずシニア開発者によるレビューを実施
- ビジネスロジックは人間が設計・実装
- プロンプトに具体的な制約を明記

改善後のアプローチ：

```
プロンプト改善例:

悪い:「ユーザーダッシュボードを作成」
良い:「ユーザーダッシュボードを作成。要件:
- 役割ベースアクセス制御（Admin/Manager/User）
- 顧客データ取得はGDPR準拠API経由のみ
- PII表示は暗号化されたセッションでのみ
- 監査ログすべてのアクション記録
- エラーはSentryへ報告、UIに露出しない」
```

ケーススタディ2：アクセシビリティの見落とし

状況：政府機関向けWebポータルをAI生成で構築

問題：

- WCAG準拠要件を満たさず
- スクリーンリーダー対応不十分
- キーボードナビゲーション不完全
- 契約上の義務違反リスク

失敗の原因：

1. プロンプトにアクセシビリティ要件を明記しなかった
2. 生成後のa11yテストを実施しなかった
3. AIがデフォルトでWCAG AAを保証すると誤解

教訓：

- アクセシビリティは**明示的に要求**しなければ保証されない
- 生成後に必ずaxe DevTools、WAVE等でテスト
- 複雑なインタラクション（モーダル、ドロップダウン）は特に注意

改善アプローチ：

```xml
<accessibility_requirements>
<standard>WCAG 2.1 Level AA強制準拠</standard>
<testing>
 生成後に以下ツールでテスト必須:
 - axe DevTools
 - WAVE
 - スクリーンリーダー（NVDA/JAWS）
</testing>
<checklist>
 必須項目:
 - すべてインタラクティブ要素にaria-label
 - フォーカスインジケーター視認可能
 - 色のみに依存しない情報表示
 - キーボードのみで全機能操作可能
 - 4.5:1最低コントラスト比
</checklist>
</accessibility_requirements>
```

ケーススタディ3：パフォーマンス問題

状況：高トラフィックECサイトのUI再構築にAI活用

問題：

- 初期ロード時間3秒→8秒へ悪化
- Core Web Vitals全項目失格
- SEOランキング大幅低下
- コンバージョン率25%減少

失敗の原因：

1. AIが画像最適化を実装しなかった
2. Code splittingなし（全コンポーネント1バンドル）
3. 不要なライブラリ多数インポート
4. クライアントサイドレンダリングのみ

教訓：

- パフォーマンス要件を**数値で明示**
- 生成後に必ずLighthouse/WebPageTestで検証
- 画像最適化、Code splitting、SSRは明示的に要求

改善プロンプト：

```
パフォーマンス要件（必須）：

Core Web Vitals目標:
- LCP（Largest Contentful Paint）: < 2.5秒
- FID（First Input Delay）: < 100ms
- CLS（Cumulative Layout Shift）: < 0.1

実装必須:
1. 画像最適化:
  - next/image使用、WebP形式
  - 遅延ロード（loading="lazy"）
  - 適切なサイズ指定（レイアウトシフト防止）

2. Code splitting:
  - ルートベース分割（Next.js App Router）
  - 大規模コンポーネントはdynamic import

3. SSR/ISR:
  - 初期ページはSSR
  - 動的コンテンツはISR（revalidate: 60）

4. バンドルサイズ:
  - 初期JSバンドル < 200KB（gzip後）
  - lodash等は必要関数のみインポート

検証:
生成後にnpm run build実行し、バンドルサイズ確認必須
```

5.3. 実装成功のための10のベストプラクティス

1. 反復的アプローチを採用

原則：完璧を一度に求めない。MVP→改善サイクルを回す。

実装：

```
第1反復: 基本構造生成
「このデザインの基本HTMLとTailwind構造を生成」

第2反復: インタラクティビティ追加
「ドロップダウンメニュー、モーダル、フォームバリデーション追加」

第3反復: レスポンシブ最適化
「モバイル・タブレット・デスクトップでレイアウト最適化」

第4反復: アクセシビリティ強化
「WCAG AA準拠。ARIA、キーボードナビゲーション追加」

第5反復: パフォーマンス最適化
「画像最適化、Code splitting、遅延ロード実装」
```

2. 段階的詳細化プロンプティング

悪い例：「Eコマースサイト作って」 **良い例**：

```
段階1: 構造定義
「Next.js Eコマースサイトのページ Hier archyとルート構造を提案」

段階2: レイアウト決定
「ホームページのワイヤーフレーム的レイアウト提案
（ヘッダー、ヒーロー、商品グリッド、フッター）」

段階3: コンポーネント実装
「商品カードコンポーネントを実装
（画像、タイトル、価格、カートボタン、お気に入り）」

段階4: 統合
「すべてコンポーネントをホームページに統合」
```

3. 複数モデル活用戦略

戦略：各モデルの強みを組み合わせる

タスク	最適モデル	理由
初期ビジュアル解析	GPT-4V	Design2Code最高精度
コンポーネント実装	Claude Sonnet 4.5	コーディング精度最高
動画ベースUI理解	Gemini 2.5 Pro	動画処理特化
反復改善	GPT-4.1	会話的改善得意
大規模コードベース統合	Gemini 2.5 Pro	1M+コンテキスト

実装例：

```
ステップ1: GPT-4Vでビジュアル理解
「このスクリーンショットの構造と要素を詳細に説明」

ステップ2: Claude Sonnet 4.5で実装
「GPT-4Vの分析を基に、React+TypeScript+Tailwindで実装」

ステップ3: Gemini 2.5で大規模統合
「既存の50K行コードベースに新コンポーネントを統合。
一貫性を保ち、重複を避ける」
```

4. コンテキストドキュメントの提供

重要性：AIはプロジェクト全体のコンテキストを持たない

提供すべき情報：

```xml
```

```
<project_context>
 <tech_stack>
  - Framework: Next.js 15 App Router
  - Language: TypeScript 5.3
  - Styling: Tailwind CSS 3.4
  - UI Library: shadcn/ui
  - State: Zustand 4.5
  - Forms: React Hook Form + Zod
 </tech_stack>

 <conventions>
  - File naming: kebab-case
  - Component naming: PascalCase
  - Hooks: use-prefix
  - Constants: UPPER_SNAKE_CASE
 </conventions>

 <existing_patterns>
  API calls: lib/api/client.ts baseClient
  Error handling: try/catch + toast notification
  Loading states: Suspense boundaries
  Forms: Controlled components + Zod validation
 </existing_patterns>

 <design_tokens>
  Primary: #3B82F6
  Secondary: #8B5CF6
  Success: #10B981
  Error: #EF4444
  Font: Inter
  Radius: 8px
 </design_tokens>
</project_context>
```

5. 品質チェックリストの活用

必須検証項目：

コード品質：

- [ ] TypeScript型エラーなし（`npm run type-check`）
- [ ] Lintエラーなし（`npm run lint`）
- [ ] 未使用import/変数なし
- [ ] Console.log削除済み
- [ ] プレースホルダーコメントなし

機能性：

- [ ] すべてインタラクティブ要素動作
- [ ] フォームバリデーション動作
- [ ] エラー処理実装
- [ ] ローディング状態表示
- [ ] 空状態ハンドリング

アクセシビリティ：

- [ ] axe DevToolsエラーなし
- [ ] スクリーンリーダーテスト合格
- [ ] キーボード操作可能
- [ ] コントラスト比4.5:1以上
- [ ] フォーカスインジケーター視認可能

パフォーマンス：

- [ ] Lighthouse Performance > 90
- [ ] LCP < 2.5s

- [ ] FID < 100ms
- [ ] CLS < 0.1
- [ ] 初期JSバンドル < 200KB

レスポンシブ：

- [ ] モバイル（320px-767px）テスト
- [ ] タブレット（768px-1023px）テスト
- [ ] デスクトップ（1024px+）テスト
- [ ] 横向きモード動作確認

6. エラーハンドリングの徹底

AIが見落としやすい領域→明示的に要求

typescript

```typescript
// 良いエラーハンドリング例
async function fetchProducts() {
 try {
  const response = await fetch('/api/products')

  if (!response.ok) {
   // HTTPエラー
   throw new Error(`HTTP ${response.status}: ${response.statusText}`)
  }

  const data = await response.json()

  // データバリデーション
  const validated = productsSchema.parse(data)

  return validated
 } catch (error) {
  // エラー分類
  if (error instanceof ZodError) {
   // バリデーションエラー
   console.error('Invalid data structure:', error.errors)
   toast.error('データ形式が正しくありません')
  } else if (error instanceof TypeError) {
   // ネットワークエラー
   console.error('Network error:', error)
   toast.error('ネットワーク接続を確認してください')
  } else {
   // 未知のエラー
   console.error('Unexpected error:', error)
   toast.error('エラーが発生しました')
  }

  // Sentry等へ報告
  captureException(error)

  // フォールバック値返却
  return []
 }
}
```

プロンプトテンプレート：

```
エラーハンドリング要件:

すべてasync関数：
1. try-catch必須
2. HTTPエラー（4xx/5xx）を明示的チェック
3. データバリデーション（Zod等）
4. エラー種類に応じた適切なメッセージ
5. ユーザー向けメッセージ（toast/alert）
6. 開発者向けログ（console.error）
7. エラー報告（Sentry等）
8. フォールバック値提供

フォーム：
- フィールド単位バリデーション
- リアルタイムエラー表示
- 送信失敗時の再試行機能
```

7. テスト可能なコード生成

要求すべき事項：

```
テスタビリティ要件:

コンポーネント設計:
- Pure functions優先（副作用分離）
- Propsで依存注入（ハードコーディング回避）
- カスタムフック化（ロジック分離）

例:
❌悪い:
function ProductCard({ product }) {
  const { data } = useSWR('/api/cart') // 外部依存
  // ...
}

✅良い:
function ProductCard({ product, isInCart, onAddToCart }) {
  // 純粋なプレゼンテーションコンポーネント
}

テストヘルパー生成:
- Storybookストーリー
- Jest/Vitestテストケース（主要フロー）
- MSW mocks（API）

型安全性:
- すべてProps interface定義
- Generics活用
- Zod schema（ランタイムバリデーション）
```

8. デザインシステム整合性

課題：AIは既存デザインシステムを知らない

解決策：

```
xml
```

```
<design_system>
  <colors>
    <primary>#3B82F6</primary>
    <secondary>#8B5CF6</secondary>
    <success>#10B981</success>
    <warning>#F59E0B</warning>
    <error>#EF4444</error>
    <neutral>
      <50>#F9FAFB</50>
      <100>#F3F4F6</100>
      ...
      <900>#111827</900>
    </neutral>
  </colors>

  <typography>
    <font_family>Inter, system-ui, sans-serif</font_family>
    <scale>
      <xs>0.75rem</xs>   <!-- 12px -->
      <sm>0.875rem</sm>  <!-- 14px -->
      <base>1rem</base>  <!-- 16px -->
      <lg>1.125rem</lg>  <!-- 18px -->
      <xl>1.25rem</xl>   <!-- 20px -->
      ...
    </scale>
  </typography>

  <spacing>
    <!-- 4px base unit -->
    1: 0.25rem, 2: 0.5rem, 3: 0.75rem, 4: 1rem, ...
  </spacing>

  <components>
    <button>
      <variants>
        primary: bg-primary text-white hover:bg-primary/90
        secondary: bg-secondary text-white hover:bg-secondary/90
        outline: border border-gray-300 hover:bg-gray-50
        ghost: hover:bg-gray-100
      </variants>
      <sizes>
        sm: px-3 py-1.5 text-sm
        md: px-4 py-2 text-base
        lg: px-6 py-3 text-lg
      </sizes>
    </button>
  </components>

  <existing_components>
    Import from: @/components/ui/
    Available: Button, Input, Card, Modal, Toast, Table, Badge
    Usage: Always import and reuse, never recreate
  </existing_components>
</design_system>

指示:
このデザインシステムに厳密に従う。
新規コンポーネント作成前に既存チェック。
カラーはデザイントークンのみ使用（ハードコード禁止）。
```

9. セキュリティ考慮事項

AIが見落としやすいセキュリティ：

セキュリティチェックリスト:

1. XSS防止:
  - ユーザー入力を直接innerHTML設定禁止
  - DOMPurifyでサニタイズ
  - Reactの{value}は安全（自動エスケープ）

2. 認証/認可:
  - クライアントのみで機密判定禁止
  - トークン保存: httpOnlyクッキー（localStorageは脆弱）
  - API呼び出しにBearer token含める

3. API呼び出し:
  - 機密情報（API key）をクライアント露出禁止
  - 環境変数使用（.env.local、サーバーサイドのみ）
  - CORS設定適切に

4. フォーム:
  - CSRFトークン（サーバーレンダリング時）
  - レート制限（サーバーサイド）
  - 入力バリデーション（クライアント+サーバー両方）

5. 依存関係:
  - npm audit実行
  - 既知脆弱性パッケージ回避
  - 定期アップデート

実装例:
❌危険:
```
const apiKey = 'sk-abc123'  // クライアント露出
<div dangerouslySetInnerHTML={{__html: userInput}} />
```

✅安全:
```
// サーバーサイド（API route）
const apiKey = process.env.API_KEY

// クライアント
import DOMPurify from 'dompurify'
const clean = DOMPurify.sanitize(userInput)
```

10. ドキュメンテーションの要求

生成と同時にドキュメント作成：

ドキュメンテーション要件:

1. README.md:
  - プロジェクト概要
  - セットアップ手順（環境変数、依存関係）
  - 開発サーバー起動方法
  - ビルド/デプロイ手順
  - トラブルシューティング

2. コンポーネントドキュメント:
  - Props説明（JSDoc形式）
  - 使用例
  - バリエーション例
  - アクセシビリティ注意事項

3. アーキテクチャ図:
  - フォルダ構造説明
  - データフロー図
  - 状態管理戦略
  - API統合方法

4. Storybookストーリー:
  - 主要コンポーネント
  - すべてバリエーション
  - インタラクティブControls

例（コンポーネントドキュメント）:
```typescript
/**
 * 製品カードコンポーネント
 *
 * 製品情報を表示し、カート追加/お気に入り機能を提供。
 * グリッドレイアウトで使用されることを想定。
 *
 * @example
 * ```tsx
 * <ProductCard
 *   product={productData}
 *   onAddToCart={handleAdd}
 *   onToggleFavorite={handleFavorite}
 * />
 * ```
 *
 * @accessibility
 * - 商品画像にはalt属性必須
 * - ボタンにaria-label提供
 * - キーボード操作完全対応
 */
interface ProductCardProps {
  /** 製品データオブジェクト */
  product: Product
  /** カート追加時コールバック */
  onAddToCart: (product: Product) => void
  /** お気に入りトグル時コールバック */
  onToggleFavorite: (productId: string) => void
  /** ローディング状態（オプション） */
  isLoading?: boolean
}
```

第6章: 結論と今後の展望

現在地の総括

2025年10月時点で、AI駆動のUI-to-Code変換は**実用段階に到達**している。Design2Codeベンチマークで49%の置換率、64%の改善率を達成したGPT-4V、SWE-bench Verifiedで75%を記録し

たGPT-5、世界最高のコーディング精度を持つClaude Sonnet 4.5、動画理解とマルチモーダル推論で卓越したGemini 2.5 Proなど、各モデルが独自の強みを発揮している。

しかし、完全自動化には依然として壁がある。スタンフォード大学のDesign2Code研究が指摘する**視覚要素の想起**と**正確なレイアウト生成**の課題は、プロダクション品質の成果物には人間の監督が不可欠であることを示している。AIは「ドラフト生成ツール」として最大の価値を発揮し、プロトタイピングと反復作業を劇的に加速させる。v0.devやBuilder.ioを使用した組織が30-50%のUI実装時間削減を達成している事実は、適切なプロンプトエンジニアリングと人間の監督を組み合わせたアプローチの有効性を証明している。

成功のための3つの柱

1. プロンプトエンジニアリングの習得

- XML構造化プロンプト（Claude向け）
- Visual Chain-of-Thought手法
- Few-Shot Learning（3-5例）
- 明示的制約仕様
- モデル固有の最適化

2. 反復的改善プロセス

- 完璧を一度に求めない
- MVP→段階的改善
- リアルタイムフィードバックループ
- 品質チェックリスト活用
- 複数モデルの戦略的併用

3. 人間の専門知識

- コードレビュー必須
- アクセシビリティ検証（axe、WAVE）
- パフォーマンステスト（Lighthouse）
- セキュリティ監査
- ビジネスロジック設計

2026年以降の展望

短期的進化（6-12ヶ月）

モデル能力の向上：

- コンテキストウィンドウ2M+トークンへ拡大（全コードベース+デザインシステム処理）
- Computer Use機能の安定化（デスクトップOS制御、複雑ジェスチャー対応）
- マルチモーダル理解の深化（音声、動画、3Dモデル統合）
- リアルタイムコラボレーション（Live Codingセッション）

ツールエコシステムの成熟：

- Figma/Sketch直接統合の標準化
- デザインシステム自動学習機能
- Git統合（コミット、プルリクエスト自動生成）
- CI/CD統合（自動テスト、デプロイ）

新しいユースケース：

- AR/VRインターフェース生成
- 音声UIデザイン

- アクセシビリティファーストAI（WCAG AAA自動達成）
- ダークパターン検出・排除機能

中期的革新（1-3年）

エージェント型AI：

- 自律的バグ修正
- パフォーマンス最適化の自動実行
- セキュリティ脆弱性の能動的検出・修正
- ユーザーフィードバックからの自己改善

マルチモーダル設計ツール：

- 言語+スケッチ+ジェスチャーでUI設計
- 3Dプロトタイピング自動生成
- 動的アクセシビリティ最適化（ユーザーの能力に応じて自動調整）

産業標準化：

- UI-to-Code品質評価標準
- AIコード監査フレームワーク
- プロンプトライブラリの業界標準
- 倫理ガイドライン（バイアス、プライバシー）

長期的変革（3-5年）

AI-First開発パラダイム：

- デザイン→実装の境界消失
- 非エンジニアによる複雑アプリ開発
- リアルタイムUI A/Bテスト自動化
- ユーザー行動からのUI自己進化

協働インテリジェンス：

- 人間-AI pair programming標準化
- AI Code Reviewer（人間レビュアー支援）
- ドメイン特化AI（医療UI専門、金融UI専門等）

社会的影響：

- UI/UXデザイナー役割の進化（戦略/UXリサーチへシフト）
- 開発者のスキルセット変化（プロンプト設計、AI監督）
- アクセシビリティ民主化（誰でもWCAG準拠アプリ作成）
- デジタルデバイド縮小（ノーコード/ローコード進化）

最後に：AIと人間の協働こそが鍵

AIによるUI-to-Code変換は、人間の創造性と生産性を**拡張**するツールであり、**置換**するものではない。最高の成果は、AIの高速処理能力と人間の直感、創造性、倫理的判断が融合したときに生まれる。

成功する開発者/デザイナーの特徴：

1. AIの能力と限界を正確に理解
2. 効果的なプロンプト設計スキル
3. 生成コードの批判的評価能力
4. ビジネス価値とUXへの深い洞察

5. 継続的学習姿勢（AI進化に適応）

本レポートで紹介したプロンプトエンジニアリング技法、ベストプラクティス、そして実践例が、読者のAI活用を加速し、より優れたユーザー体験の創造に貢献することを願っている。UI/UXデザインからフロントエンドコードへの変換は、もはや「いつか実現する未来」ではなく、**今日から活用できる現実**である。

参考文献

学術論文

- Chen et al. (2024). "Multimodal Chain-of-Thought Reasoning in Language Models". AAAI 2024.
- Si et al. (2024). "Design2Code: How Far Are We From Automating Front-End Engineering?". NAACL 2025.

公式ドキュメント

- Anthropic. (2024). "Claude 3.5 Sonnet and Claude 3.5 Haiku". https://www.anthropic.com/news/3-5-models-and-computer-use
- Anthropic. (2025). "Claude 3.7 Sonnet and Claude Code". https://www.anthropic.com/news/claude-3-7-sonnet
- Anthropic. (2025). "Claude Sonnet 4.5". https://www.anthropic.com/claude/sonnet
- Google DeepMind. (2025). "Gemini 2.5 Computer Use Model". https://blog.google/technology/google-deepmind/gemini-computer-use-model/
- OpenAI. (2025). "Introducing GPT-4.1 in the API". https://openai.com/index/gpt-4-1/
- OpenAI. (2024). "Hello GPT-4o". https://openai.com/index/hello-gpt-4o/

実践ツール・プロジェクト

- screenshot-to-code (GitHub). https://github.com/abi/screenshot-to-code
- Builder.io. "Design to Code AI Automation". https://www.builder.io/guide/figma-design-to-code
- Vercel v0.dev. https://v0.dev

ベストプラクティスガイド

- Microsoft Azure OpenAI. "GPT-4 Turbo with Vision". Developer Guide.
- Google Vertex AI. "Prompt Design Guidelines". https://cloud.google.com/vertex-ai/docs/generative-ai/multimodal/design-multimodal-prompts
- MDN Web Docs. "ARIA Authoring Practices Guide". https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA
- W3C. "Web Content Accessibility Guidelines (WCAG) 2.1". https://www.w3.org/TR/WCAG21/

著者注記：本レポートは2025年10月時点の情報に基づいています。AI分野は急速に進化しているため、最新の公式ドキュメントも併せてご確認ください。