# 証券CRMシステム開発計画書

## Claude Code活用によるフェーズ2達成プロジェクト

**作成日**: 2025年9月29日  
**プロジェクト期間**: 10週間（約2.5ヶ月）  
**目標**: 実用レベルのCRMシステム（フェーズ2）の完成

## 目次

1. [プロジェクト概要](#_1._プロジェクト概要)
2. [技術スタック](#_2._技術スタック)
3. [詳細スケジュール](#_3._詳細スケジュール)
4. [Claude Codeプロンプト集](#_4._Claude_Codeプロンプト集)
5. [手動作業手順書](#_5._手動作業手順書)
6. [品質管理](#_6._品質管理)
7. [リスク管理](#_7._リスク管理)
8. [成果物チェックリスト](#_8._成果物チェックリスト)
9. [ClaudeCode活用のベストプラクティス](#_9._Claude_Code活用のベストプラクティス)
10. [プロジェクト完了時の評価](#_10._プロジェクト完了時の評価)

## 1. プロジェクト概要

### 1.1 目標システム

証券会社向けCRMシステムのMVP（実用レベル）を構築する。

**実装する主要機能**: - 顧客マスタ管理（CRUD、詳細検索） - 商談履歴・面談記録管理 - タスク・リマインダー機能 - 承認ワークフロー（商品提案承認） - ロールベースアクセス制御（4役割） - 操作ログ記録・監査機能 - ダッシュボード・レポート - データエクスポート機能 - 外部API連携基盤（モック実装）

### 1.2 開発体制

**推奨構成**: - プロジェクトリーダー：1名（全体統括、レビュー） - 開発エンジニア：1-2名（Claude Code活用） - 業務知識者：1名（要件確認、受け入れテスト）

**Claude Codeの役割**: - コード生成（60-70%） - テスト作成・実行（80-90%） - リファクタリング・バグ修正（70-80%） - ドキュメント生成（80-90%）

## 2. 技術スタック

### 2.1 選定理由

Claude Codeとの相性、学習コスト、エコシステムの充実度を考慮。

### 2.2 技術構成

**フロントエンド**: - React 18.x + TypeScript - Next.js 14.x（App Router） - Tailwind CSS - shadcn/ui（UIコンポーネント） - React Hook Form + Zod（フォーム・バリデーション） - TanStack Query（データフェッチング）

**バックエンド**: - Node.js 20.x + TypeScript - Express.js - Prisma ORM - PostgreSQL 16.x - JWT（認証） - bcrypt（パスワードハッシュ）

**テスト**: - Jest（ユニット・統合テスト） - Playwright（E2Eテスト） - Supertest（APIテスト）

**開発ツール**: - ESLint + Prettier - Husky（Git hooks） - Docker Compose - GitHub Actions（CI/CD）

## 3. 詳細スケジュール

### Week 1-2: プロジェクト基盤構築

#### Day 1-3: 環境セットアップ

**目標**: 開発環境の構築とプロジェクト初期化

**作業内容**: 1. リポジトリ作成 2. 技術スタックのセットアップ 3. データベース設計 4. 基本的なプロジェクト構造構築

**成果物**: - 動作するHello World - データベーススキーマ設計書 - プロジェクト構成ドキュメント

**Claude Codeへのプロンプト**: [→ セクション4.1参照](#Xa469236276a982230a5299677bbbe537c4fbe86)

**手動作業**: [→ セクション5.1参照](#Xa79c6e828704ad71f54e8df8f6d7edecf1e03a3)

#### Day 4-7: 認証システム実装

**目標**: JWT認証とユーザー管理機能の完成

**作業内容**: 1. ユーザーモデルとAPIの実装 2. JWT認証フローの構築 3. ログイン・ログアウト画面 4. 認証ミドルウェア

**成果物**: - 動作する認証システム - ログイン画面 - ユーザー管理API

#### Day 8-10: 顧客マスタ基本機能

**目標**: 顧客情報のCRUD機能完成

**作業内容**: 1. 顧客モデルとAPI実装 2. 顧客一覧・詳細画面 3. 顧客登録・編集フォーム 4. 基本的な検索機能

**成果物**: - 顧客管理機能（CRUD完成） - テストコード（カバレッジ60%以上）

### Week 3-4: コア業務機能実装

#### Day 11-14: 商談履歴管理

**目標**: 商談記録とタスク管理機能

**作業内容**: 1. 商談履歴モデルとAPI 2. 商談記録画面（リッチテキスト対応） 3. 顧客詳細画面との統合 4. 商談履歴一覧・検索

**成果物**: - 商談履歴管理機能 - 顧客との関連表示

#### Day 15-20: タスク・リマインダー機能

**目標**: タスク管理とアラート機能

**作業内容**: 1. タスクモデルとAPI 2. タスク一覧・カレンダー表示 3. 期限アラート機能 4. タスクの完了・延期機能 5. ダッシュボードへのタスク表示

**成果物**: - タスク管理機能 - ダッシュボード初期版

### Week 5-6: セキュリティ・権限管理

#### Day 21-25: ロールベースアクセス制御

**目標**: 4つの役割による権限管理

**役割定義**: - ADMIN: 全機能アクセス - MANAGER: チーム管理、承認権限 - SALES: 自分の顧客のみ - COMPLIANCE: 全体閲覧（読取専用）

**作業内容**: 1. 役割・権限テーブル設計 2. 権限チェックミドルウェア 3. APIレベルの権限制御 4. UI上の権限による表示制御 5. 役割管理画面（ADMIN用）

**成果物**: - 完全な権限管理システム - 権限テストスイート

#### Day 26-30: 操作ログ・監査機能

**目標**: 全操作の記録と監査画面

**作業内容**: 1. 監査ログテーブル設計 2. 自動ログ記録ミドルウェア 3. ログ閲覧・検索画面（COMPLIANCE用） 4. 変更履歴の表示（顧客詳細画面） 5. ログエクスポート機能

**成果物**: - 監査ログシステム - ログ閲覧UI

### Week 7-8: ワークフロー・レポート機能

#### Day 31-37: 承認ワークフロー

**目標**: 商品提案の承認フロー実装

**ワークフロー定義**: 1. SALES → 商品提案申請 2. MANAGER → 承認/差し戻し/却下 3. 承認後 → 顧客提案可能

**作業内容**: 1. 承認申請モデルとAPI 2. 申請作成画面 3. 承認待ち一覧画面 4. 承認アクション画面 5. 通知機能（メール・画面内通知） 6. 承認履歴表示

**成果物**: - 承認ワークフローシステム - 通知機能

#### Day 38-42: レポート・ダッシュボード

**目標**: データ可視化とレポート機能

**実装するレポート**: 1. 営業実績サマリ 2. 顧客セグメント分析 3. タスク消化率 4. 承認待ち案件サマリ 5. 月次活動レポート

**作業内容**: 1. レポートAPI実装 2. グラフ・チャート表示（Recharts） 3. ダッシュボード画面の完成 4. レポートのフィルタ・期間指定 5. PDFエクスポート（将来対応の基盤）

**成果物**: - 完成したダッシュボード - レポート機能

### Week 9-10: 外部連携・品質向上

#### Day 43-49: 外部API連携基盤

**目標**: 外部システム連携の実証

**作業内容**: 1. モックAPIサーバー構築 2. 勘定系システム想定のAPI呼び出し 3. 市場データAPI連携（モック） 4. エラーハンドリングとリトライ 5. API呼び出しログ記録 6. 接続設定管理画面

**成果物**: - 外部API連携基盤 - モックAPIサーバー

#### Day 50-56: テスト・品質保証

**目標**: 包括的なテストとドキュメント整備

**作業内容**: 1. ユニットテストの追加（カバレッジ70%以上） 2. E2Eテストスイート作成 3. セキュリティスキャン実施 4. パフォーマンステスト 5. バグ修正 6. ドキュメント整備 7. デモ環境構築

**成果物**: - 完全なテストスイート - API仕様書 - ユーザーマニュアル - セットアップガイド - デモ環境

#### Day 57-60: 最終調整・レビュー

**目標**: プロダクション準備完了

**作業内容**: 1. コードレビュー 2. UI/UXの最終調整 3. パフォーマンス最適化 4. セキュリティ最終チェック 5. デプロイ手順確認 6. 最終デモ準備

## 4. Claude Codeプロンプト集

### 4.1 Week 1-2: 基盤構築

#### プロンプト1: プロジェクト初期化

Next.js 14とTypeScriptを使用した証券CRMシステムのプロジェクトを初期化してください。  
  
要件:  
- Next.js 14 (App Router)  
- TypeScript  
- Tailwind CSS  
- ESLint, Prettierの設定  
- Husky (pre-commit hooks)  
- 基本的なフォルダ構造 (app/, components/, lib/, types/)  
  
また、以下を含めてください:  
- .env.exampleファイル  
- README.mdに開発環境セットアップ手順  
- package.jsonに必要なscripts

#### プロンプト2: バックエンド初期化

Express + TypeScript + Prismaを使用したバックエンドAPIサーバーを構築してください。  
  
要件:  
- Express.js  
- TypeScript  
- Prisma ORM (PostgreSQL)  
- 基本的なミドルウェア (cors, helmet, morgan)  
- エラーハンドリングミドルウェア  
- 環境変数管理 (dotenv)  
  
フォルダ構造:  
- src/routes/ (APIルート)  
- src/controllers/ (ビジネスロジック)  
- src/middleware/ (認証等)  
- src/services/ (データアクセス)  
- src/types/ (型定義)  
- prisma/ (スキーマ)

#### プロンプト3: データベーススキーマ設計

証券CRMシステムのPrismaスキーマを作成してください。  
  
必要なモデル:  
1. User (ユーザー)  
 - id, email, password, name, role (ADMIN/MANAGER/SALES/COMPLIANCE)  
 - createdAt, updatedAt  
  
2. Customer (顧客)  
 - id, name, email, phone, address  
 - investmentProfile (conservative/moderate/aggressive)  
 - riskTolerance, investmentExperience  
 - assignedSalesId (担当営業)  
 - createdAt, updatedAt  
  
3. Meeting (商談履歴)  
 - id, customerId, salesId, date, summary, nextAction  
 - createdAt, updatedAt  
  
4. Task (タスク)  
 - id, userId, customerId, title, description, dueDate, status  
 - priority, createdAt, updatedAt  
  
5. ApprovalRequest (承認申請)  
 - id, requesterId, approverId, customerId  
 - productName, amount, status (PENDING/APPROVED/REJECTED)  
 - requestedAt, processedAt  
  
6. AuditLog (監査ログ)  
 - id, userId, action, resourceType, resourceId  
 - changes (JSON), ipAddress, timestamp  
  
リレーションも適切に設定してください。

#### プロンプト4: 認証システム実装

JWT認証システムを実装してください。  
  
実装内容:  
1. POST /api/auth/register - ユーザー登録  
 - パスワードをbcryptでハッシュ化  
 - バリデーション（email形式、パスワード強度）  
  
2. POST /api/auth/login - ログイン  
 - メール・パスワード認証  
 - JWTトークン発行（有効期限24時間）  
 - リフレッシュトークン対応  
  
3. POST /api/auth/logout - ログアウト  
  
4. GET /api/auth/me - 現在のユーザー情報取得  
  
5. 認証ミドルウェア  
 - Bearerトークン検証  
 - req.userにユーザー情報を設定  
  
6. ユニットテスト  
 - 各APIエンドポイントのテスト  
 - 認証ミドルウェアのテスト

### 4.2 Week 3-4: コア業務機能

#### プロンプト5: 顧客管理API

顧客管理のRESTful APIを実装してください。  
  
エンドポイント:  
1. GET /api/customers - 顧客一覧取得  
 - ページネーション (page, limit)  
 - 検索 (name, email)  
 - フィルタ (investmentProfile, assignedSalesId)  
 - ソート (createdAt, name)  
  
2. GET /api/customers/:id - 顧客詳細取得  
 - 関連する商談履歴も含める  
  
3. POST /api/customers - 顧客作成  
 - バリデーション（Zod使用）  
 - assignedSalesIdは現在のユーザー  
  
4. PUT /api/customers/:id - 顧客更新  
 - 権限チェック（自分の顧客のみ、またはMANAGER以上）  
  
5. DELETE /api/customers/:id - 顧客削除（論理削除）  
  
6. GET /api/customers/:id/meetings - 顧客の商談履歴  
7. GET /api/customers/:id/tasks - 顧客のタスク一覧  
  
各エンドポイントのテストコードも作成してください。

#### プロンプト6: 顧客管理UI（フロントエンド）

顧客管理のUIコンポーネントを実装してください。  
  
必要な画面:  
1. 顧客一覧ページ (/customers)  
 - データテーブル（shadcn/ui使用）  
 - 検索ボックス  
 - フィルタ（投資プロファイル、担当営業）  
 - ページネーション  
 - 新規登録ボタン  
  
2. 顧客詳細ページ (/customers/[id])  
 - タブUI（基本情報、商談履歴、タスク、変更履歴）  
 - 編集ボタン  
 - 商談記録追加ボタン  
  
3. 顧客登録・編集フォーム  
 - React Hook Form + Zod  
 - バリデーションエラー表示  
 - 投資プロファイル選択（ラジオボタン）  
 - リスク許容度（スライダー）  
  
4. 顧客検索コンポーネント  
 - あいまい検索  
 - 検索結果のオートコンプリート  
  
TanStack Queryを使用してデータフェッチを実装してください。

#### プロンプト7: 商談履歴機能

商談履歴管理機能を実装してください。  
  
バックエンド:  
1. POST /api/meetings - 商談記録作成  
2. GET /api/meetings/:id - 商談詳細  
3. PUT /api/meetings/:id - 商談記録更新  
4. DELETE /api/meetings/:id - 商談記録削除  
5. GET /api/customers/:customerId/meetings - 顧客の商談一覧  
  
フロントエンド:  
1. 商談記録フォーム  
 - 日付選択  
 - リッチテキストエディタ（Tiptapまたはシンプルなtextarea）  
 - 次回アクション入力  
 - 次回アクション日時  
  
2. 商談履歴一覧  
 - タイムライン表示  
 - 最新順にソート  
 - 商談内容のサマリ表示  
  
3. 商談記録後の自動タスク作成  
 - 次回アクションが入力されていればタスク自動生成  
  
テストコードも含めてください。

#### プロンプト8: タスク管理機能

タスク管理機能を完全実装してください。  
  
バックエンド:  
1. POST /api/tasks - タスク作成  
2. GET /api/tasks - タスク一覧（自分のタスク、またはチームのタスク）  
3. GET /api/tasks/:id - タスク詳細  
4. PUT /api/tasks/:id - タスク更新  
5. DELETE /api/tasks/:id - タスク削除  
6. PUT /api/tasks/:id/complete - タスク完了  
7. GET /api/tasks/overdue - 期限切れタスク  
  
フロントエンド:  
1. タスク一覧ページ (/tasks)  
 - ステータス別タブ（未完了、完了、期限切れ）  
 - 優先度による色分け  
 - 期限によるソート  
  
2. タスクカレンダービュー  
 - 月間カレンダー表示  
 - 日付クリックでタスク追加  
  
3. タスク作成・編集ダイアログ  
 - 顧客紐付け（検索機能）  
 - 期限日時選択  
 - 優先度選択  
 - 説明入力  
  
4. ダッシュボードウィジェット  
 - 今日のタスク  
 - 期限切れタスク（アラート表示）  
 - 今週のタスク数  
  
通知機能（期限30分前にアラート）も実装してください。

### 4.3 Week 5-6: セキュリティ・権限管理

#### プロンプト9: RBAC実装

ロールベースアクセス制御（RBAC）を実装してください。  
  
役割定義:  
- ADMIN: すべての機能にアクセス可能  
- MANAGER: チーム管理、承認権限、配下メンバーの顧客閲覧  
- SALES: 自分の顧客のみアクセス可能  
- COMPLIANCE: 全顧客閲覧可能（読取専用）、監査ログアクセス  
  
実装内容:  
1. 権限チェックミドルウェア  
 ```typescript  
 requireRole(['ADMIN', 'MANAGER'])  
 requireOwnershipOr(['ADMIN', 'MANAGER'])

1. APIレベルの権限制御
   * 顧客APIに権限チェック追加
   * 商談・タスクAPIに権限チェック追加
2. データフィルタリング
   * SALESは自分の顧客のみ返す
   * MANAGERはチームメンバーの顧客も返す
3. フロントエンド権限制御
   * usePermissionフック作成
   * 役割に応じたUI表示制御
   * ボタン・メニューの表示/非表示
4. 役割管理画面（ADMIN用）
   * ユーザー一覧
   * 役割変更機能

包括的な権限テストも作成してください。

#### プロンプト10: 監査ログシステム

包括的な監査ログシステムを実装してください。  
  
要件:  
1. 自動ログ記録ミドルウェア  
 - すべてのCRUD操作を記録  
 - 変更前後のデータ（JSON形式）  
 - ユーザー情報、IPアドレス、タイムスタンプ  
 - リクエストメソッド、パス  
  
2. ログ記録対象:  
 - 顧客の作成・更新・削除  
 - 商談記録の追加・更新  
 - タスクの作成・完了  
 - 承認申請の処理  
 - ユーザー認証（ログイン・ログアウト）  
 - 役割変更  
  
3. 監査ログAPI  
 - GET /api/audit-logs - ログ一覧（COMPLIANCE, ADMIN）  
 - GET /api/audit-logs/user/:userId - ユーザー別ログ  
 - GET /api/audit-logs/resource/:type/:id - リソース別ログ  
 - フィルタ（日付範囲、アクション種別、ユーザー）  
 - エクスポート機能（CSV）  
  
4. 監査ログUI  
 - ログ閲覧画面（COMPLIANCE専用）  
 - 高度な検索・フィルタ  
 - タイムライン表示  
 - 変更差分表示（before/after）  
  
5. 顧客詳細画面への統合  
 - 変更履歴タブ  
 - 誰がいつ何を変更したか表示  
  
ログの改ざん防止も考慮してください。

### 4.4 Week 7-8: ワークフロー・レポート

#### プロンプト11: 承認ワークフロー

商品提案の承認ワークフローを実装してください。  
  
ワークフロー:  
SALES作成 → MANAGER承認 → 顧客提案可能  
  
実装内容:  
1. バックエンドAPI  
 - POST /api/approval-requests - 承認申請作成  
 - GET /api/approval-requests - 申請一覧（役割により異なる）  
 - GET /api/approval-requests/:id - 申請詳細  
 - PUT /api/approval-requests/:id/approve - 承認  
 - PUT /api/approval-requests/:id/reject - 却下  
 - PUT /api/approval-requests/:id/recall - 取り下げ（申請者のみ）  
  
2. ステータス管理  
 - PENDING: 承認待ち  
 - APPROVED: 承認済み  
 - REJECTED: 却下  
 - RECALLED: 取り下げ  
  
3. 通知機能  
 - 申請時にMANAGERへ通知  
 - 処理時に申請者へ通知  
 - メール通知（実装は後回し、インターフェースのみ）  
  
4. フロントエンド  
 - 申請作成フォーム  
 - 承認待ち一覧（MANAGER向け）  
 - 自分の申請一覧（SALES向け）  
 - 承認処理画面（理由入力）  
 - 承認履歴表示  
  
5. ダッシュボード統合  
 - 承認待ち件数バッジ  
 - 自分が承認すべき案件リスト  
  
包括的なテストを含めてください。

#### プロンプト12: ダッシュボード・レポート

データ可視化とレポート機能を実装してください。  
  
ダッシュボード要件:  
1. 営業実績サマリ  
 - 今月の商談件数  
 - 今月の新規顧客数  
 - 承認済み提案数  
 - グラフ: 月別商談推移（過去6ヶ月）  
  
2. タスク状況  
 - 今日のタスク数  
 - 期限切れタスク数（赤色アラート）  
 - タスク完了率（今週）  
  
3. 顧客セグメント分析  
 - 投資プロファイル別円グラフ  
 - リスク許容度分布  
 - 顧客数推移  
  
4. チーム実績（MANAGER向け）  
 - メンバー別商談件数ランキング  
 - メンバー別タスク完了率  
  
5. コンプライアンス指標（COMPLIANCE向け）  
 - 承認申請処理時間（平均）  
 - 高リスク取引数  
 - 監査ログサマリ  
  
実装:  
1. レポートAPI  
 - GET /api/reports/sales-summary  
 - GET /api/reports/customer-segments  
 - GET /api/reports/team-performance  
 - GET /api/reports/compliance-metrics  
 - 日付範囲フィルタ対応  
  
2. ダッシュボード画面  
 - Rechartsでグラフ描画  
 - レスポンシブグリッドレイアウト  
 - 役割に応じたウィジェット表示  
 - リアルタイム更新（polling or WebSocket）  
  
3. レポートエクスポート  
 - CSV形式  
 - 将来のPDF対応のための基盤  
  
パフォーマンスを考慮したクエリ最適化も実装してください。

### 4.5 Week 9-10: 外部連携・品質向上

#### プロンプト13: 外部API連携基盤

外部システム連携の基盤を構築してください。  
  
モックAPIサーバー構築:  
1. 勘定系システムAPI（モック）  
 - GET /mock-api/accounts/:customerId - 口座情報  
 - GET /mock-api/accounts/:customerId/balance - 残高  
 - GET /mock-api/accounts/:customerId/transactions - 取引履歴  
  
2. 市場データAPI（モック）  
 - GET /mock-api/market/stock/:symbol - 株価情報  
 - GET /mock-api/market/indices - 市場指標  
  
CRM側の実装:  
1. APIクライアント  
 - src/lib/externalApi.ts  
 - リトライロジック（exponential backoff）  
 - タイムアウト設定  
 - エラーハンドリング  
  
2. API呼び出しログ  
 - すべての外部API呼び出しを記録  
 - レスポンスタイム  
 - ステータスコード  
 - エラー詳細  
  
3. 接続設定管理  
 - 環境変数での設定  
 - ADMIN向けAPI設定画面  
 - 接続テスト機能  
  
4. 顧客詳細画面への統合  
 - 口座情報タブ  
 - リアルタイム残高表示  
 - 最近の取引表示  
  
5. エラー時のフォールバック  
 - キャッシュ機能  
 - エラー時のUI表示  
  
包括的なテストコードを含めてください。

#### プロンプト14: E2Eテストスイート

Playwrightを使用した包括的なE2Eテストスイートを作成してください。  
  
テストシナリオ:  
1. 認証フロー  
 - ログイン成功  
 - ログイン失敗（無効な資格情報）  
 - ログアウト  
 - 未認証時のリダイレクト  
  
2. 顧客管理フロー  
 - 顧客新規登録  
 - 顧客一覧表示・検索  
 - 顧客詳細閲覧  
 - 顧客情報編集  
 - 顧客削除  
  
3. 商談管理フロー  
 - 商談記録作成  
 - 商談履歴閲覧  
 - 商談記録編集  
 - 商談からタスク自動生成  
  
4. タスク管理フロー  
 - タスク作成  
 - タスク一覧表示  
 - タスク完了  
 - 期限切れタスクの表示  
  
5. 承認ワークフロー  
 - SALES: 承認申請作成  
 - MANAGER: 承認処理  
 - ステータス更新確認  
  
6. 権限制御  
 - SALESが他人の顧客にアクセス不可  
 - MANAGERがチームの顧客にアクセス可能  
 - COMPLIANCEが監査ログにアクセス可能  
  
7. ダッシュボード  
 - 各役割のダッシュボード表示  
 - データの正確性確認  
  
設定:  
- テストユーザーのシードデータ作成  
- テスト用データベースの自動セットアップ・クリーンアップ  
- スクリーンショット撮影（失敗時）  
- 並列実行対応  
  
実行コマンド:  
- npm run test:e2e  
- npm run test:e2e:ui (UI mode)

#### プロンプト15: セキュリティスキャンと最適化

セキュリティスキャンとパフォーマンス最適化を実施してください。  
  
セキュリティチェック:  
1. npm audit実行と修正  
 - 脆弱性のある依存関係の更新  
 - audit reportの生成  
  
2. ESLintセキュリティルール  
 - eslint-plugin-security導入  
 - 危険なパターンの検出  
  
3. OWASP Top 10対応確認  
 - SQL Injection対策（ORMで担保）  
 - XSS対策（React自動エスケープ）  
 - CSRF対策（SameSite cookie）  
 - 認証・認可の再確認  
  
4. セキュリティヘッダー  
 - helmetミドルウェアの設定  
 - CSP（Content Security Policy）  
  
パフォーマンス最適化:  
1. データベースクエリ最適化  
 - スロークエリの特定  
 - インデックス追加  
 - N+1問題の解消  
  
2. フロントエンド最適化  
 - コード分割（dynamic import）  
 - 画像最適化  
 - バンドルサイズ分析  
  
3. APIレスポンス最適化  
 - ページネーション実装確認  
 - 不要なデータの除外  
 - gzip圧縮  
  
4. パフォーマンステスト  
 - 1万件の顧客データで検索速度測定  
 - 同時アクセステスト（50ユーザー）  
 - レスポンスタイム目標: 3秒以内  
  
レポート生成:  
- セキュリティスキャン結果  
- パフォーマンステスト結果  
- 改善推奨事項

#### プロンプト16: ドキュメント生成

包括的なドキュメントを生成してください。  
  
1. API仕様書（OpenAPI/Swagger形式）  
 - すべてのエンドポイント  
 - リクエスト・レスポンス例  
 - 認証方法  
 - エラーレスポンス  
  
2. データモデル図  
 - ER図（Mermaid形式）  
 - リレーション説明  
  
3. アーキテクチャドキュメント  
 - システム構成図  
 - フロー図（認証、承認ワークフロー）  
 - フォルダ構造説明  
  
4. セットアップガイド  
 - 開発環境構築手順  
 - 環境変数一覧  
 - データベースマイグレーション手順  
 - シードデータ投入方法  
  
5. ユーザーマニュアル  
 - 各機能の使い方（スクリーンショット付き）  
 - 役割別の機能説明  
 - よくある質問（FAQ）  
  
6. デプロイガイド  
 - 本番環境デプロイ手順  
 - Docker Composeでのデプロイ  
 - 環境変数設定  
 - バックアップ手順  
  
7. コントリビューションガイド  
 - コーディング規約  
 - ブランチ戦略  
 - プルリクエストプロセス  
 - テスト要件  
  
Markdown形式で、読みやすく構造化してください。

## 5. 手動作業手順書

### 5.1 Week 1-2: 環境構築

#### 手順1: GitHubリポジトリ作成

**所要時間**: 10分

1. GitHubで新規リポジトリ作成
   * リポジトリ名: securities-crm-system
   * Visibility: Private
   * Initialize with README: チェック
   * .gitignore: Node
   * License: MIT
2. ローカルにクローン

* git clone https://github.com/your-username/securities-crm-system.git  
  cd securities-crm-system

1. ブランチ戦略設定

* git checkout -b develop  
  git push -u origin develop

#### 手順2: 開発環境準備

**所要時間**: 30分

**必要なツール**: - Node.js 20.x - PostgreSQL 16.x - Docker Desktop（推奨） - VS Code + 拡張機能 - ESLint - Prettier - Prisma - TypeScript

**インストール確認**:

node --version # v20.x.x  
npm --version # 10.x.x  
psql --version # 16.x  
docker --version

**PostgreSQLセットアップ（Docker使用）**:

# docker-compose.ymlを作成  
mkdir docker  
cd docker

# docker/docker-compose.yml  
version: '3.8'  
services:  
 postgres:  
 image: postgres:16-alpine  
 environment:  
 POSTGRES\_USER: crm\_user  
 POSTGRES\_PASSWORD: crm\_password  
 POSTGRES\_DB: crm\_development  
 ports:  
 - "5432:5432"  
 volumes:  
 - postgres\_data:/var/lib/postgresql/data  
  
volumes:  
 postgres\_data:

# 起動  
docker-compose up -d  
  
# 接続確認  
psql -h localhost -U crm\_user -d crm\_development

#### 手順3: Claude Codeでプロジェクト初期化

**所要時間**: 15分

1. Claude Codeを起動

* # ターミナルでプロジェクトルートに移動  
  cd securities-crm-system

1. Claude Codeに**プロンプト1**を入力
   * Next.jsプロジェクト初期化
   * 必要な依存関係インストール
2. 生成されたファイルを確認
   * package.json
   * tsconfig.json
   * next.config.js
   * .eslintrc.json
   * tailwind.config.ts
3. 開発サーバー起動確認

* npm run dev  
  # http://localhost:3000 でアクセス確認

#### 手順4: バックエンド初期化

**所要時間**: 20分

1. バックエンド用フォルダ作成

* mkdir backend  
  cd backend

1. Claude Codeに**プロンプト2**を入力
   * Expressサーバー構築
2. 環境変数設定

* # backend/.env  
  DATABASE\_URL="postgresql://crm\_user:crm\_password@localhost:5432/crm\_development"  
  JWT\_SECRET="your-secret-key-change-in-production"  
  PORT=4000  
  NODE\_ENV=development

1. バックエンドサーバー起動確認

* cd backend  
  npm run dev  
  # http://localhost:4000 でアクセス確認

#### 手順5: データベーススキーマ適用

**所要時間**: 15分

1. Claude Codeに**プロンプト3**を入力
   * Prismaスキーマ生成
2. スキーマレビュー
   * backend/prisma/schema.prismaを確認
   * リレーションの妥当性チェック
   * インデックス追加検討
3. マイグレーション実行

* cd backend  
  npx prisma migrate dev --name init

1. Prisma Studioで確認

* npx prisma studio  
  # http://localhost:5555 でDB確認

#### 手順6: 初期コミット

**所要時間**: 10分

git add .  
git commit -m "chore: initial project setup with Next.js and Express"  
git push origin develop

### 5.2 Week 3-4: 機能開発時の手順

#### 日次開発フロー

**毎朝のルーティン**（15分）: 1. 最新コードをプル bash git checkout develop git pull origin develop

1. 今日のタスク確認
   * スケジュール表を参照
   * 実装する機能を明確化
2. フィーチャーブランチ作成

* git checkout -b feature/customer-management

**開発中**: 1. Claude Codeに該当プロンプトを入力 2. 生成されたコードをレビュー 3. 必要に応じて調整・リファクタリング 4. テスト実行 bash npm test 5. 動作確認（ブラウザ/Postman）

**夕方のルーティン**（20分）: 1. コードレビュー（セルフまたはペア） 2. コミット bash git add . git commit -m "feat: implement customer CRUD operations" 3. プッシュ bash git push origin feature/customer-management 4. プルリクエスト作成（必要に応じて）

### 5.3 Week 5-6: セキュリティレビュー手順

#### セキュリティチェックリスト

**毎週金曜日実施**（60分）:

1. **認証・認可の確認**
   * すべてのAPIエンドポイントに認証が必要
   * 権限チェックが適切に機能
   * JWTトークンの有効期限設定
   * パスワードが適切にハッシュ化
2. **入力バリデーション**
   * すべてのAPI入力がバリデーション済み
   * SQLインジェクション対策（ORM使用）
   * XSS対策（React自動エスケープ）
   * ファイルアップロード検証（該当する場合）
3. **データ保護**
   * 機密情報が環境変数で管理
   * .envファイルが.gitignoreに含まれる
   * APIレスポンスにパスワードが含まれない
   * エラーメッセージで内部情報が漏洩しない
4. **監査ログ**
   * すべての重要操作がログ記録
   * ログ改ざん防止措置
   * ログ保存期間の設定
5. **依存関係**

* npm audit  
  # 脆弱性があれば修正  
  npm audit fix

**レビュー記録**:

# セキュリティレビュー - Week X  
  
実施日: YYYY-MM-DD  
実施者: [名前]  
  
## チェック結果  
- [ ] 認証・認可: OK  
- [ ] 入力バリデーション: OK  
- [ ] データ保護: OK  
- [ ] 監査ログ: OK  
- [ ] 依存関係: 2件の脆弱性修正  
  
## 発見事項  
1. [問題点]  
 - 対応: [修正内容]  
  
## 次回アクション  
- [残課題]

### 5.4 Week 7-8: レビュー会議の進め方

#### 2週間ごとのデモ・レビュー会議

**準備**（会議の1日前、60分）: 1. デモ環境の準備 bash # デモ用データベースにシードデータ投入 npm run seed:demo

1. デモシナリオ作成
   * 実装した機能のユーザーストーリー
   * デモの順序
   * 想定される質問と回答
2. プレゼンテーション資料
   * 今回実装した機能
   * 次回の予定
   * 課題・リスク

**会議当日**（60分）: 1. 進捗報告（10分） - 完了した機能 - スケジュール達成度

1. デモンストレーション（30分）
   * 実際の画面操作
   * 主要機能の実演
2. フィードバック収集（15分）
   * 業務知識者からの意見
   * UI/UXの改善点
3. 次回計画（5分）
   * 次の2週間の目標確認

**会議後**（30分）: 1. フィードバックをIssueに登録 2. 優先順位付け 3. スケジュール調整（必要に応じて）

### 5.5 Week 9-10: 最終調整手順

#### リリース前チェックリスト

**Day 50-53: 機能完成度確認**

1. **機能テストマトリクス作成**

* | 機能 | ADMIN | MANAGER | SALES | COMPLIANCE | 備考 |  
  |------|-------|---------|-------|------------|------|  
  | 顧客一覧 | ✓ | ✓ | ✓ | ✓ | |  
  | 顧客作成 | ✓ | ✓ | ✓ | - | |  
  | 顧客編集 | ✓ | ✓ | 自分のみ | - | |  
  ...

1. **全機能の手動テスト**
   * テストユーザーでログイン
   * 各役割で全機能を操作
   * バグを発見したらIssue登録
2. **ブラウザ互換性確認**
   * Chrome
   * Firefox
   * Safari
   * Edge

**Day 54-56: パフォーマンステスト**

1. **テストデータ準備**

* # 10,000件の顧客データを生成  
  npm run seed:performance

1. **パフォーマンス測定**
   * 顧客検索: X秒
   * ダッシュボード読み込み: X秒
   * レポート生成: X秒
   * 目標: 3秒以内
2. **ボトルネック特定と改善**
   * スロークエリログ確認
   * インデックス追加
   * 再測定

**Day 57-58: ドキュメント最終確認**

1. **README.md更新**
   * 最新のセットアップ手順
   * スクリーンショット更新
2. **API仕様書確認**
   * すべてのエンドポイントが記載されているか
   * 例が正確か
3. **ユーザーマニュアル**
   * 各機能の操作手順
   * スクリーンショット

**Day 59-60: デモ準備**

1. **デモ環境構築**

* # 本番に近い環境でデプロイ  
  docker-compose -f docker-compose.prod.yml up -d

1. **デモシナリオ準備**
   * 30分のデモフロー
   * 各役割での操作デモ
   * 質疑応答準備
2. **プレゼンテーション資料**
   * プロジェクトサマリ
   * 実装機能一覧
   * 技術スタック
   * 今後の拡張可能性
   * Claude Codeの活用効果

## 6. 品質管理

### 6.1 コード品質基準

#### 静的解析

**毎コミット前に実行**:

# Lint check  
npm run lint  
  
# Type check  
npm run type-check  
  
# Format check  
npm run format:check

**自動修正**:

npm run lint:fix  
npm run format

#### コードレビュー基準

**必須チェック項目**: - [ ] TypeScriptの型定義が適切 - [ ] エラーハンドリングが実装されている - [ ] セキュリティリスクがない - [ ] パフォーマンスへの配慮 - [ ] テストコードが含まれている - [ ] コメントが必要な箇所に記載 - [ ] 命名規則に従っている

### 6.2 テスト基準

#### カバレッジ目標

* **ユニットテスト**: 70%以上
* **統合テスト**: 主要APIエンドポイント100%
* **E2Eテスト**: クリティカルパス100%

#### テスト実行タイミング

# 開発中（Watch mode）  
npm run test:watch  
  
# コミット前  
npm run test  
  
# プッシュ前  
npm run test:all # ユニット + 統合 + E2E

### 6.3 パフォーマンス基準

#### レスポンスタイム目標

| 操作 | 目標 | 許容 |
| --- | --- | --- |
| ページ読み込み | 1秒 | 2秒 |
| API応答（単純） | 100ms | 500ms |
| API応答（複雑） | 1秒 | 3秒 |
| 検索（10,000件） | 2秒 | 3秒 |
| レポート生成 | 2秒 | 5秒 |

#### データベース

* インデックス設定
* N+1問題の排除
* ページネーション実装

## 7. リスク管理

### 7.1 主要リスクと対策

#### リスク1: スケジュール遅延

**発生確率**: 中  
**影響度**: 高

**原因**: - Claude Codeの生成コード品質のばらつき - 予期しない技術的課題 - スコープの拡大

**対策**: - バッファを各週に20%確保 - 週次で進捗確認、遅延時は即座にスコープ調整 - 優先順位の明確化（Must/Should/Could）

#### リスク2: セキュリティ脆弱性

**発生確率**: 中  
**影響度**: 致命的

**原因**: - AIが生成したコードのセキュリティホール - 金融規制への理解不足

**対策**: - 週次セキュリティレビュー - 自動セキュリティスキャン（GitHub Actions） - 専門家による最終レビュー

#### リスク3: Claude Codeの制約

**発生確率**: 低  
**影響度**: 中

**原因**: - 複雑なビジネスロジックの実装困難 - 大規模リファクタリングの難しさ

**対策**: - 複雑な部分は人間が設計 - 小さな単位で段階的に実装 - コードレビューの徹底

### 7.2 課題管理

#### Issue管理

**ラベル分類**: - bug: バグ - feature: 新機能 - enhancement: 改善 - security: セキュリティ - documentation: ドキュメント - priority:high: 高優先度 - priority:medium: 中優先度 - priority:low: 低優先度

**週次トリアージ**: 毎週月曜日にIssueを確認し、優先順位を更新

## 8. 成果物チェックリスト

### 8.1 コード成果物

#### フロントエンド

* Next.js アプリケーション
* 認証画面（ログイン・ログアウト）
* 顧客管理画面（一覧・詳細・登録・編集）
* 商談履歴管理画面
* タスク管理画面
* 承認ワークフロー画面
* ダッシュボード
* 監査ログ画面（COMPLIANCE）
* レスポンシブ対応

#### バックエンド

* Express API サーバー
* 認証API（JWT）
* 顧客管理API
* 商談管理API
* タスク管理API
* 承認ワークフローAPI
* レポートAPI
* 監査ログAPI
* 外部API連携基盤
* ロールベースアクセス制御

#### データベース

* Prismaスキーマ
* マイグレーションファイル
* シードデータ
* インデックス最適化

### 8.2 テスト成果物

* ユニットテスト（カバレッジ70%以上）
* 統合テスト（主要API）
* E2Eテスト（Playwright）
* パフォーマンステスト結果
* セキュリティスキャン結果
* テストレポート

### 8.3 ドキュメント成果物

* README.md（プロジェクト概要）
* SETUP.md（環境構築手順）
* API仕様書（OpenAPI）
* データモデル図（ER図）
* アーキテクチャ図
* ユーザーマニュアル
* デプロイガイド
* セキュリティ対策ドキュメント

### 8.4 デモ・プレゼン資料

* プロジェクトサマリ資料
* デモ環境
* デモシナリオ
* デモ動画（録画）
* Claude Code活用効果レポート

## 9. Claude Code活用のベストプラクティス

### 9.1 効果的なプロンプトの書き方

#### 良いプロンプトの例

顧客管理APIを実装してください。  
  
要件:  
1. エンドポイント:  
 - GET /api/customers (一覧取得、ページネーション、検索、フィルタ)  
 - GET /api/customers/:id (詳細取得)  
 - POST /api/customers (新規作成)  
 - PUT /api/customers/:id (更新)  
 - DELETE /api/customers/:id (削除)  
  
2. バリデーション:  
 - Zodを使用  
 - emailは形式チェック  
 - nameは必須、2-50文字  
  
3. 権限制御:  
 - SALESは自分の顧客のみ  
 - MANAGERはチーム全体  
 - ADMINは全顧客  
  
4. テスト:  
 - Jestでユニットテスト  
 - Supertestで統合テスト  
 - 各エンドポイントのテストケース  
  
5. エラーハンドリング:  
 - 適切なHTTPステータスコード  
 - エラーメッセージの統一フォーマット

#### 避けるべきプロンプト

顧客管理機能を作って

→ 曖昧すぎて品質が不安定

### 9.2 コードレビューのポイント

#### Claude Codeが生成したコードで確認すべき点

1. **セキュリティ**
   * SQLインジェクション対策
   * XSS対策
   * 認証・認可の実装
2. **エラーハンドリング**
   * try-catchの適切な使用
   * エラーメッセージの適切性
   * ログ出力
3. **パフォーマンス**
   * N+1問題
   * 不要なデータ取得
   * キャッシング
4. **コード品質**
   * 命名規則
   * 重複コード
   * 複雑度

### 9.3 トラブルシューティング

#### よくある問題と解決策

**問題1**: Claude Codeが古いライブラリの使い方を提案 **解決策**: 最新のドキュメントURLを提供して再生成依頼

**問題2**: テストが失敗する **解決策**: エラーメッセージをClaude Codeに渡して修正依頼

**問題3**: 生成されたコードが要件を満たしていない **解決策**: より具体的な要件を追加してプロンプトを再実行

## 10. プロジェクト完了時の評価

### 10.1 評価指標

#### 機能完成度

* 必須機能100%実装
* Should機能80%以上実装
* バグ重大度High: 0件
* バグ重大度Medium: 3件以下

#### コード品質

* テストカバレッジ70%以上達成
* ESLint違反: 0件
* TypeScript strict mode: エラー0件
* セキュリティスキャン: クリティカル脆弱性0件

#### パフォーマンス

* すべての操作が目標レスポンスタイム内
* 10,000件データで問題なく動作
* 同時50ユーザーアクセスで安定動作

#### ドキュメント

* すべての成果物ドキュメント完成
* 第三者が環境構築可能なレベル

### 10.2 Claude Code活用効果測定

#### 定量評価

* 総開発時間: X時間
* Claude Codeが生成したコード行数: X行（全体のY%）
* バグ修正にかかった時間: X時間
* 従来手法との比較（推定）: Z%の時間短縮

#### 定性評価

* Claude Codeが特に有効だった領域
* 人間の介入が必要だった領域
* 改善提案

### 10.3 次フェーズへの推奨事項

**フェーズ3で実装すべき機能**: 1. 本格的な外部システム連携 2. 高度な分析・レポート機能 3. メール通知システム 4. モバイルアプリ 5. AIによる顧客インサイト

**技術的改善**: 1. マイクロサービス化の検討 2. GraphQL導入 3. リアルタイム通信（WebSocket） 4. より高度なセキュリティ対策

## 付録

### A. コマンドリファレンス

# 開発サーバー起動  
npm run dev # フロントエンド（Next.js）  
cd backend && npm run dev # バックエンド（Express）  
  
# テスト実行  
npm test # 全テスト  
npm run test:unit # ユニットテストのみ  
npm run test:integration # 統合テストのみ  
npm run test:e2e # E2Eテストのみ  
npm run test:watch # Watch mode  
npm run test:coverage # カバレッジレポート  
  
# データベース  
npx prisma migrate dev # マイグレーション実行  
npx prisma studio # Prisma Studio起動  
npx prisma generate # Prismaクライアント生成  
npm run seed # シードデータ投入  
  
# コード品質  
npm run lint # Lint実行  
npm run lint:fix # Lint自動修正  
npm run format # Prettier実行  
npm run type-check # TypeScriptチェック  
  
# ビルド・デプロイ  
npm run build # 本番ビルド  
npm start # 本番サーバー起動  
docker-compose up -d # Docker環境起動

### B. 環境変数一覧

#### フロントエンド (.env.local)

NEXT\_PUBLIC\_API\_URL=http://localhost:4000  
NEXT\_PUBLIC\_APP\_NAME=Securities CRM

#### バックエンド (.env)

# Database  
DATABASE\_URL=postgresql://crm\_user:crm\_password@localhost:5432/crm\_development  
  
# Authentication  
JWT\_SECRET=your-secret-key-minimum-32-characters  
JWT\_EXPIRES\_IN=24h  
  
# Server  
PORT=4000  
NODE\_ENV=development  
  
# External APIs (Mock)  
MOCK\_ACCOUNT\_API\_URL=http://localhost:5000  
MOCK\_MARKET\_API\_URL=http://localhost:5001  
  
# Logging  
LOG\_LEVEL=debug

### C. プロジェクト構造

securities-crm-system/  
├── frontend/  
│ ├── app/  
│ │ ├── (auth)/  
│ │ │ ├── login/  
│ │ │ └── layout.tsx  
│ │ ├── (dashboard)/  
│ │ │ ├── customers/  
│ │ │ │ ├── page.tsx  
│ │ │ │ ├── [id]/  
│ │ │ │ └── new/  
│ │ │ ├── meetings/  
│ │ │ ├── tasks/  
│ │ │ ├── approvals/  
│ │ │ ├── reports/  
│ │ │ ├── audit-logs/  
│ │ │ └── layout.tsx  
│ │ ├── layout.tsx  
│ │ └── page.tsx  
│ ├── components/  
│ │ ├── ui/ # shadcn/ui components  
│ │ ├── customers/  
│ │ ├── meetings/  
│ │ ├── tasks/  
│ │ ├── approvals/  
│ │ └── layout/  
│ ├── lib/  
│ │ ├── api.ts # API client  
│ │ ├── auth.ts # Auth utilities  
│ │ └── utils.ts  
│ ├── hooks/  
│ │ ├── useAuth.ts  
│ │ ├── useCustomers.ts  
│ │ └── useTasks.ts  
│ ├── types/  
│ │ └── index.ts  
│ └── package.json  
│  
├── backend/  
│ ├── src/  
│ │ ├── routes/  
│ │ │ ├── auth.ts  
│ │ │ ├── customers.ts  
│ │ │ ├── meetings.ts  
│ │ │ ├── tasks.ts  
│ │ │ ├── approvals.ts  
│ │ │ ├── reports.ts  
│ │ │ └── audit-logs.ts  
│ │ ├── controllers/  
│ │ ├── services/  
│ │ ├── middleware/  
│ │ │ ├── auth.ts  
│ │ │ ├── rbac.ts  
│ │ │ ├── audit.ts  
│ │ │ └── errorHandler.ts  
│ │ ├── types/  
│ │ ├── utils/  
│ │ └── server.ts  
│ ├── prisma/  
│ │ ├── schema.prisma  
│ │ ├── migrations/  
│ │ └── seed.ts  
│ ├── tests/  
│ │ ├── unit/  
│ │ ├── integration/  
│ │ └── fixtures/  
│ └── package.json  
│  
├── e2e/  
│ ├── tests/  
│ │ ├── auth.spec.ts  
│ │ ├── customers.spec.ts  
│ │ ├── meetings.spec.ts  
│ │ ├── tasks.spec.ts  
│ │ ├── approvals.spec.ts  
│ │ └── rbac.spec.ts  
│ ├── fixtures/  
│ └── playwright.config.ts  
│  
├── docs/  
│ ├── api/  
│ │ └── openapi.yaml  
│ ├── architecture/  
│ │ ├── system-design.md  
│ │ └── data-model.md  
│ ├── setup/  
│ │ └── getting-started.md  
│ └── user-manual/  
│ └── guide.md  
│  
├── docker/  
│ ├── docker-compose.yml  
│ ├── docker-compose.prod.yml  
│ └── Dockerfile  
│  
├── .github/  
│ └── workflows/  
│ ├── test.yml  
│ └── deploy.yml  
│  
└── README.md

### D. 役割別アクセス権限マトリクス

| 機能 | ADMIN | MANAGER | SALES | COMPLIANCE |
| --- | --- | --- | --- | --- |
| **顧客管理** |  |  |  |  |
| 顧客一覧閲覧 | 全顧客 | チーム | 自分のみ | 全顧客（RO） |
| 顧客詳細閲覧 | ✓ | チームのみ | 自分のみ | ✓（RO） |
| 顧客作成 | ✓ | ✓ | ✓ | ✗ |
| 顧客編集 | ✓ | チームのみ | 自分のみ | ✗ |
| 顧客削除 | ✓ | ✗ | ✗ | ✗ |
| **商談管理** |  |  |  |  |
| 商談記録作成 | ✓ | ✓ | ✓ | ✗ |
| 商談履歴閲覧 | 全て | チーム | 自分のみ | 全て（RO） |
| 商談記録編集 | ✓ | 自分のみ | 自分のみ | ✗ |
| **タスク管理** |  |  |  |  |
| タスク作成 | ✓ | ✓ | ✓ | ✗ |
| タスク閲覧 | 全て | チーム | 自分のみ | ✗ |
| タスク編集 | ✓ | 自分/部下 | 自分のみ | ✗ |
| **承認ワークフロー** |  |  |  |  |
| 申請作成 | ✓ | ✓ | ✓ | ✗ |
| 申請承認 | ✓ | ✓ | ✗ | ✗ |
| 承認履歴閲覧 | ✓ | ✓ | 自分のみ | ✓（RO） |
| **レポート** |  |  |  |  |
| 個人実績 | ✓ | ✓ | ✓ | ✗ |
| チーム実績 | ✓ | ✓ | ✗ | ✗ |
| 全体実績 | ✓ | ✗ | ✗ | ✗ |
| **監査ログ** |  |  |  |  |
| ログ閲覧 | ✓ | ✗ | ✗ | ✓ |
| ログエクスポート | ✓ | ✗ | ✗ | ✓ |
| **システム管理** |  |  |  |  |
| ユーザー管理 | ✓ | ✗ | ✗ | ✗ |
| 役割変更 | ✓ | ✗ | ✗ | ✗ |
| システム設定 | ✓ | ✗ | ✗ | ✗ |

RO = Read Only（読取専用）

### E. テストデータ例

#### 初期ユーザー（シードデータ）

// backend/prisma/seed.ts  
const users = [  
 {  
 email: 'admin@example.com',  
 password: 'Admin123!',  
 name: '管理者太郎',  
 role: 'ADMIN'  
 },  
 {  
 email: 'manager@example.com',  
 password: 'Manager123!',  
 name: 'マネージャー花子',  
 role: 'MANAGER'  
 },  
 {  
 email: 'sales1@example.com',  
 password: 'Sales123!',  
 name: '営業一郎',  
 role: 'SALES'  
 },  
 {  
 email: 'sales2@example.com',  
 password: 'Sales123!',  
 name: '営業二郎',  
 role: 'SALES'  
 },  
 {  
 email: 'compliance@example.com',  
 password: 'Compliance123!',  
 name: 'コンプライアンス三郎',  
 role: 'COMPLIANCE'  
 }  
];

#### サンプル顧客データ

const customers = [  
 {  
 name: '山田太郎',  
 email: 'yamada@example.com',  
 phone: '090-1234-5678',  
 investmentProfile: 'conservative',  
 riskTolerance: 3,  
 investmentExperience: '5年',  
 assignedSalesId: 'sales1-user-id'  
 },  
 {  
 name: '佐藤花子',  
 email: 'sato@example.com',  
 phone: '090-2345-6789',  
 investmentProfile: 'moderate',  
 riskTolerance: 5,  
 investmentExperience: '10年',  
 assignedSalesId: 'sales1-user-id'  
 },  
 // ... more customers  
];

### F. よくある質問（FAQ）

#### Q1: Claude Codeが生成したコードが動かない場合は？

**A**: 以下の手順で対処してください： 1. エラーメッセージをコピー 2. Claude Codeに「以下のエラーが発生しました。修正してください：[エラーメッセージ]」と依頼 3. 生成されたコードで置き換え 4. それでも解決しない場合は、該当部分を手動デバッグ

#### Q2: データベースマイグレーションエラーが出た場合は？

**A**:

# データベースをリセット（開発環境のみ）  
npx prisma migrate reset  
  
# 新しいマイグレーション作成  
npx prisma migrate dev --name fix-migration  
  
# それでもダメなら手動でDBを削除して再作成  
dropdb crm\_development  
createdb crm\_development  
npx prisma migrate dev

#### Q3: テストが失敗し続ける場合は？

**A**: 1. テストデータベースが正しくセットアップされているか確認 2. 環境変数が正しく設定されているか確認 3. 他のテストとの干渉がないか確認（テストの独立性） 4. Claude Codeに失敗したテストコードとエラーを渡して修正依頼

#### Q4: パフォーマンスが目標に達しない場合は？

**A**: 1. データベースのスロークエリログを確認 2. N+1問題がないかチェック 3. 必要なインデックスが追加されているか確認 4. 不要なデータ取得がないか確認 5. Claude Codeに「このクエリを最適化してください」と依頼

#### Q5: セキュリティスキャンで脆弱性が見つかった場合は？

**A**:

# 詳細確認  
npm audit  
  
# 自動修正試行  
npm audit fix  
  
# 手動で依存関係更新  
npm update [package-name]  
  
# それでもダメなら代替パッケージを検討

#### Q6: スケジュールが遅れている場合は？

**A**: 1. 遅延の原因を特定（技術的課題 vs スコープ拡大） 2. 優先順位を再確認（Must機能に集中） 3. Should/Could機能を次フェーズに延期 4. チームメンバーを追加（可能であれば） 5. ステークホルダーに早めに報告

### G. 成功のためのヒント

#### ヒント1: 小さく始めて段階的に拡張

最初から完璧を目指さず、動く最小限の機能から始めましょう。 - Day 1: Hello World - Day 2: ログイン機能 - Day 3: 顧客一覧表示 - …

#### ヒント2: 毎日動くものを作る

毎日の終わりには必ず動くデモを準備できる状態にしておきます。

#### ヒント3: テストを後回しにしない

コードを書いたらすぐにテストを書く習慣をつけましょう。後でまとめて書くと時間がかかります。

#### ヒント4: Claude Codeに具体的な指示を出す

曖昧な指示より、具体的で詳細な要件を伝える方が良い結果が得られます。

#### ヒント5: 定期的にコードレビュー

週に1回は必ずコード全体をレビューし、品質を確認しましょう。

#### ヒント6: ドキュメントを継続的に更新

最後にまとめて書くのではなく、機能を作るたびにドキュメントも更新します。

#### ヒント7: 早めのフィードバック

業務知識者からのフィードバックは早い段階で受けましょう。後戻りを防げます。

### H. チェックリスト（印刷用）

#### 毎日のチェックリスト

□ 朝: 今日のタスクを明確化  
□ コーディング前にブランチ作成  
□ 定期的にコミット（小さく頻繁に）  
□ テストを書く・実行する  
□ コードレビュー（セルフまたはペア）  
□ 夕方: プッシュとドキュメント更新  
□ 明日のタスクを準備

#### 毎週のチェックリスト（金曜日）

□ 週次進捗レビュー  
□ セキュリティスキャン実行  
□ パフォーマンステスト実行  
□ npm audit実行・修正  
□ ドキュメント更新  
□ 次週の計画確認  
□ Issues整理・優先順位付け

#### マイルストーン完了時チェックリスト

□ 全機能が要件通り動作  
□ テストカバレッジ目標達成  
□ コード品質基準クリア  
□ セキュリティスキャン合格  
□ ドキュメント更新完了  
□ デモ準備完了  
□ ステークホルダーレビュー実施  
□ フィードバック反映

## おわりに

この開発計画書は、Claude Codeを最大限活用して証券CRMシステムを構築するための包括的なガイドです。

**重要なポイント**:

1. **段階的な開発**: 一度にすべてを作ろうとせず、週ごとに着実に機能を積み上げる
2. **品質重視**: 速度だけでなく、セキュリティとコード品質を常に意識
3. **継続的なレビュー**: 人間によるレビューでAIの限界を補完
4. **柔軟な対応**: 計画は目安であり、状況に応じて調整可能

**成功の鍵**: - Claude Codeと人間の強みを組み合わせる - 小さく始めて段階的に拡張 - 定期的なフィードバックと改善 - ドキュメントの継続的な更新

このプロジェクトを通じて、AIを活用した開発の可能性と限界を実証できることを期待しています。

**プロジェクトの成功を祈っています！**

**改訂履歴**: - v1.0 (2025-09-29): 初版作成

**連絡先**: 質問や問題が発生した場合は、プロジェクトのIssueトラッカーに登録してください。