# Miyabi

## Claude Code & Codex
## マルチエージェント協調アーキテクチャ

CCG (Claude Code) + CG (Codex) サブエージェントシステム

Miyabi 開発チーム

2025 年 11 月

**Abstract**

本ドキュメントは、Miyabi 自律型開発プラットフォームにおける Claude Code（CCG）と OpenAI Codex（CG）エージェントの協調に関する包括的な技術仕様を提供します。両システムのサブエージェントアーキテクチャを詳述し、協調パターンを定義し、マルチエージェントオーケストレーションのベストプラクティスを確立します。目標は、Claude と OpenAI モデルの両方の強みを活用するハイブリッド AI 開発ワークフローを実現することです。

## Contents

# 1 はじめに

## 1.1 Miyabi プラットフォーム概要

Miyabi は完全自律型 AI 開発オペレーションプラットフォームです：

- **GitHub-as-OS**: Issue = タスク、PR = 成果物、Actions = CI/CD
- **21 個の AI エージェント**: コーディング 7 個 + ビジネス 14 個
- **59 個の Rust Crate**: モジュラーアーキテクチャ（コンパイル 50%以上高速化）
- **28 個以上の MCP サーバー**: 拡張可能なツールエコシステム
- **A2A プロトコル**: エージェント間通信標準

## 1.2 CCG と CG の定義

| 略称 | 正式名称 | 説明 |
|---|---|---|
| CCG | Claude Code Guide | Claude Code サブエージェントシステム |
| CG | Codex Guide | OpenAI Codex サブエージェントシステム |

Table 1: エージェントシステム略称

## 1.3 本ドキュメントの目的

1. CCG と CG のサブエージェント機能と制限を定義
2. ハイブリッドワークフローの協調パターンを確立
3. Miyabi 統合の実装ガイドラインを提供
4. マルチエージェントオーケストレーションのベストプラクティスを文書化

# 2 Claude Code（CCG）アーキテクチャ

## 2.1 コアメカニズム

Claude Code の Task ツールは一時的なワーカー生成システムとして動作します：

- 各タスクは**独立した 200k コンテキストウィンドウ**を受け取る
- サブエージェントは親から**独立した状態**を維持
- 最大 **10 個の並列タスク**
- **サブエージェントのネスト不可**：サブエージェントは他のサブエージェントを生成できない

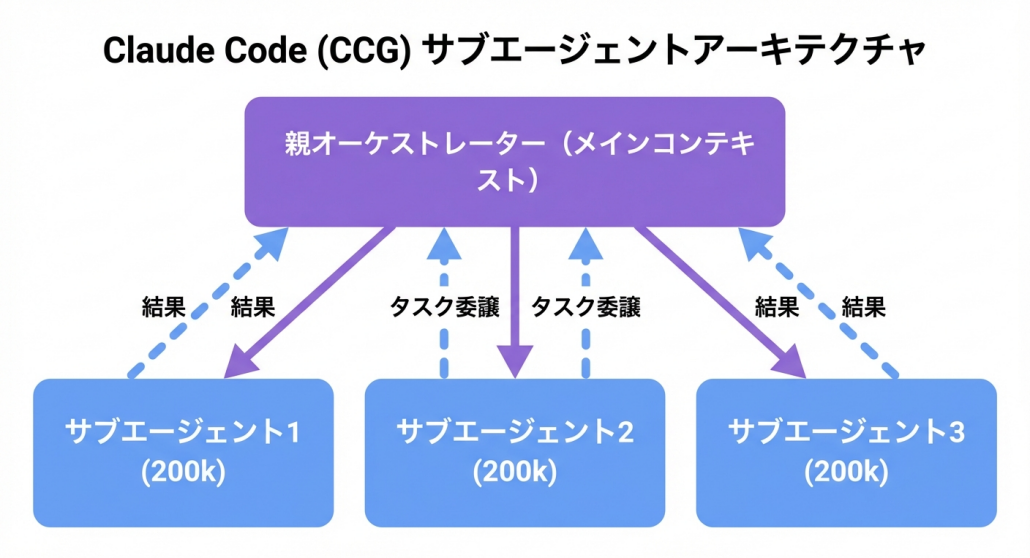


Figure 1: CCG サブエージェント実行モデル（Gemini 3 Pro 生成）

| タイプ | モデル | ツール | 用途 |
|---|---|---|---|
| General-Purpose | Sonnet | 全て | フルスタック実装 |
| Plan | Sonnet | 読み取り専用 | アーキテクチャ分析 |
| Explore | Haiku | 読み取り専用 | 高速コードベース検索 |
| claude-code-guide | — | ドキュメント | ドキュメント参照 |

Table 2: CCG 組み込みサブエージェントタイプ

### 2.2 組み込みサブエージェントタイプ

### 2.3 カスタムサブエージェント設定

カスタムサブエージェントは Markdown ファイルの YAML フロントマターで定義：

Listing 1: カスタム CCG サブエージェント定義

```
---
name: security-reviewer
description: "セキュリティ監査専門家"
model: sonnet
tools:
  - Read
  - Glob
  - Grep
  - Bash
permissionMode: default
---

あなたはコードの脆弱性を分析するセキュリティ専門家です...
```

**保存場所:**

- プロジェクトレベル: `.claude/agents/`（バージョン管理）
- ユーザーレベル: `~/.claude/agents/`（グローバル）

### 2.4 Claude Agent SDK アーキテクチャ

**SDK コンポーネント**

- `ClaudeSDKClient`: ステートフルなマルチターンセッション
- `query()`: ステートレスな単発クエリ
- `ClaudeAgentOptions`: 設定オブジェクト
- Hooks: PreToolUse, PostToolUse, SessionStart など

#### 2.4.1 Python SDK 使用例

Listing 2: Claude Agent SDK 例

```
from claude_agent_sdk import ClaudeSDKClient, ClaudeAgentOptions

options = ClaudeAgentOptions(
    allowed_tools=["Read", "Write", "Bash"],
    model="sonnet",
    permission_mode="acceptEdits",
    max_turns=20
)

async with ClaudeSDKClient() as client:
    await client.connect(initial_prompt)
    async for message in client.query(task):
        process_message(message)
```

### 2.5 通信パターン

**主要原則:**

1. サブエージェントは**タスク関連コンテキストのみ**を受け取る
2. 結果は親に**コンテキスト汚染なし**で返される
3. サブエージェント間の**ピアツーピア通信なし**
4. オーケストレーターは**関心の分離**を維持

## 3 OpenAI Codex（CG）アーキテクチャ

### 3.1 コアメカニズム

Codex CLI は Model Context Protocol（MCP）サーバーアーキテクチャで統合：

- `codex()` と `codex-reply()` ツールを公開
- エージェントターン間で永続的なセッション
- 設定可能なサンドボックスと承認ポリシー
- ネイティブ Agents SDK 統合

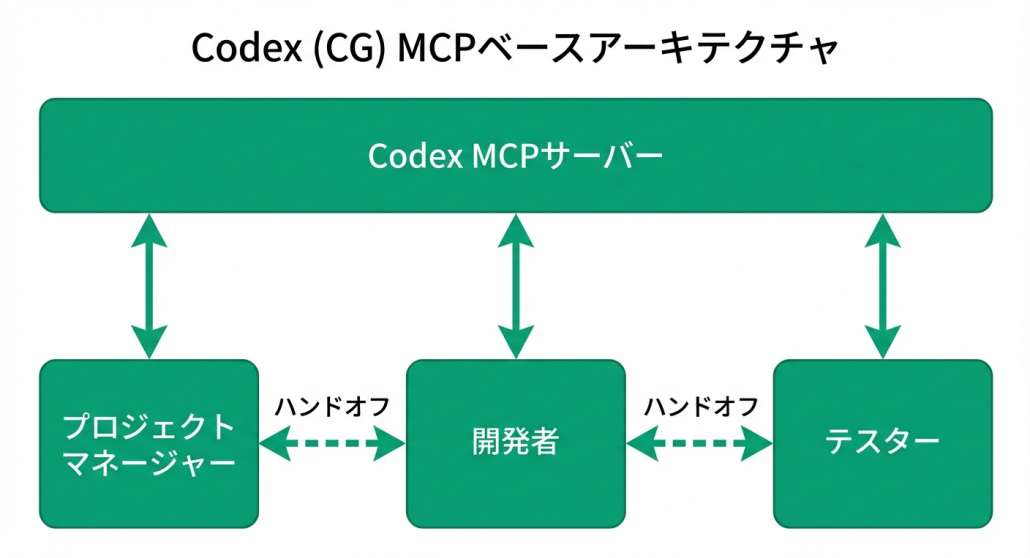


Figure 2: CG MCP ベースアーキテクチャ（Gemini 3 Pro 生成）

### 3.2 エージェント設定

Codex エージェントは`~/.codex/agents.toml`で TOML ファイルとして設定：

Listing 3: Codex エージェント設定（agents.toml）

```
[agents.code-reviewer]
name = "Code Reviewer"
system_prompt = """
あなたは以下に焦点を当てたコードレビュー専門家です：
- セキュリティ脆弱性
- パフォーマンス問題
- コード品質と保守性
"""
model = "gpt-5"

[agents.security-auditor]
name = "Security Auditor"
system_prompt = "あなたはセキュリティ専門家です..."
```

### 3.3 マルチエージェントオーケストレーション

**Codex オーケストレーション機能**

- **階層的調整**: プロジェクトマネージャーがハンドオフを管理
- **並列実行**: 独立したエージェントが同時実行
- **ファイルベースチェックポイント**: 成果物が進行を制御
- **双方向ハンドオフ**: 検証のため PM に戻る

#### 3.3.1 Agents SDK 統合

Listing 4: Codex + Agents SDK

```
from agents import Agent, Runner
from agents.mcp import MCPServerStdio

async with MCPServerStdio(
    name="Codex CLI",
    params={"command": "npx", "args": ["-y", "codex", "mcp"]},
) as codex_mcp:

    developer = Agent(
        name="Developer",
        instructions="仕様に基づいて機能を構築...",
        mcp_servers=[codex_mcp],
        model="gpt-5"
    )

    reviewer = Agent(
        name="Reviewer",
        instructions="コード品質をレビュー...",
        mcp_servers=[codex_mcp],
        handoffs=[developer]
    )

    result = await Runner.run(reviewer, task)
```

### 3.4 サンドボックスと権限

| 設定 | 説明 |
|---|---|
| approval-policy: never | 自律実行 |
| approval-policy: always | ユーザー承認必須 |
| sandbox: workspace-write | ワークスペース内のみ書き込み |
| sandbox: none | フルシステムアクセス |

Table 3: Codex サンドボックス設定

## 4 CCG + CG 協調アーキテクチャ

### 4.1 ハイブリッドオーケストレーションモデル

Miyabi プラットフォームは統一されたオーケストレーション層を通じて CCG と CG を調整：

### 4.2 役割定義

### 4.3 協調パターン

#### 4.3.1 パターン 1: シーケンシャルハンドオフ

Listing 5: シーケンシャル CCG から CG へのハンドオフ

```
# フェーズ1: CCG計画
plan = await ccg_planner.query("機能Xを分析・設計")

# フェーズ2: CG実装
```

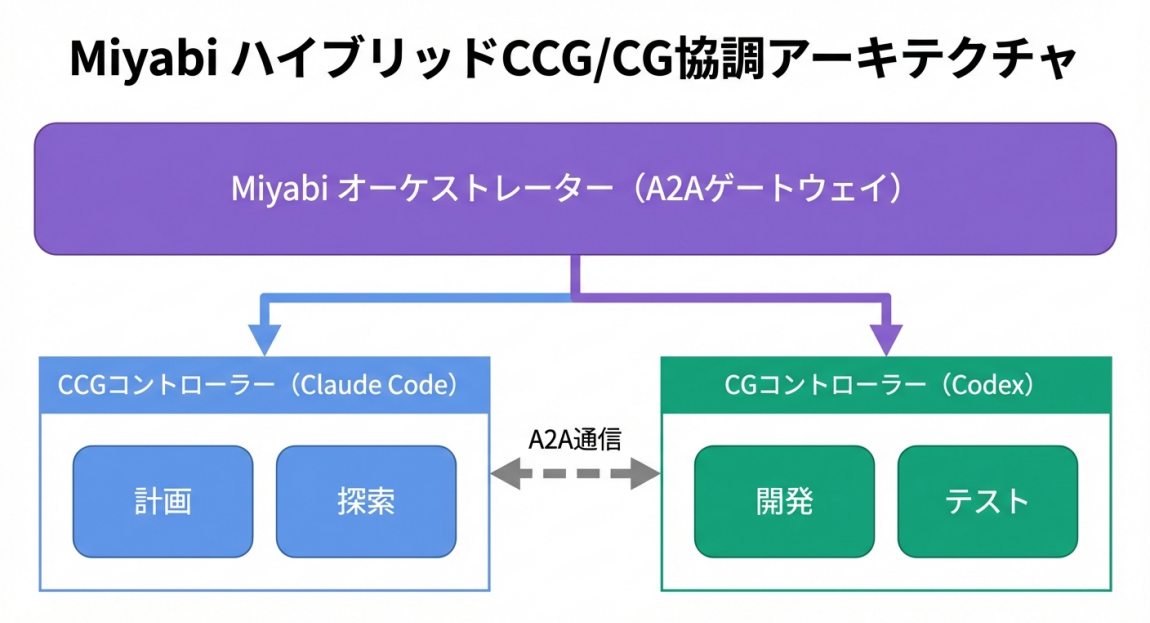


Figure 3: Miyabi ハイブリッド CCG/CG アーキテクチャ（Gemini 3 Pro 生成）

| 役割 | システム | 責務 | 強み |
|---|---|---|---|
| コーディネーター | CCG | タスク分解、DAG | 長いコンテキスト、推論 |
| プランナー | CCG | アーキテクチャ設計 | 計画モード |
| エクスプローラー | CCG | コードベース分析 | 高速 Haiku 検索 |
| デベロッパー | CG | コード生成 | GPT-5 創造性 |
| テスター | CG | テスト作成 | 構造化出力 |
| レビュアー | CCG | 品質保証 | Opus 深い分析 |

Table 4: ハイブリッドエージェント役割割り当て

```
impl = await cg_developer.query(f"実装: {plan}")

# フェーズ3: CCGレビュー
review = await ccg_reviewer.query(f"レビュー: {impl}")
```

#### 4.3.2 パターン 2: 並列実行

Listing 6: CCG と CG の並列実行

```
# CCGとCGエージェントを並列実行
results = await asyncio.gather(
    ccg_explore.query("認証パターンを検索"),
    cg_developer.query("認証ボイラープレートを生成"),
    ccg_security.query("セキュリティ要件をレビュー")
)

# 結果を統合
synthesis = await ccg_coordinator.query(
    f"調査結果を統合: {results}"
)
```

#### 4.3.3 パターン 3: 競合検証

Listing 7: クロスシステム検証

```
# 一方のシステムで生成
ccg_impl = await ccg_developer.query("機能を実装")

# 他方で検証
cg_review = await cg_reviewer.query(f"レビュー: {ccg_impl}")

# または逆方向
cg_impl = await cg_developer.query("機能を実装")
ccg_review = await ccg_reviewer.query(f"レビュー: {cg_impl}")
```

### 4.4 A2A プロトコル統合

Miyabi の A2A プロトコルは標準化された通信を実現：

Listing 8: A2A ツール命名規則

```
# CCGエージェント
a2a.claude_code_planning_agent.analyze_architecture
a2a.claude_code_review_agent.review_code

# CGエージェント
a2a.codex_development_agent.generate_code
a2a.codex_testing_agent.create_tests

# A2A Gateway経由のクロスシステムルーティング
a2a_gateway.route(
    source="ccg.planner",
    target="cg.developer",
    payload=plan_document
)
```

## 5 実装ガイドライン

### 5.1 Miyabi 統合手順

1. **CCG エージェントの設定**
   - `.claude/agents/`ディレクトリを作成
   - YAML フロントマターでエージェント仕様を定義
   - Claude Agent SDK フックを設定
2. **CG エージェントの設定**
   - `~/.codex/agents.toml` を作成
   - エージェントシステムプロンプトを定義
   - MCP サーバー統合を設定
3. **A2A Gateway の実装**
   - CCG と CG の両エージェントを登録

- ルーティングルールを定義
- メッセージ変換を実装

4. **オーケストレーションワークフローの作成**
   - タスク分解ルールを定義
   - 並列実行ハンドラーを実装
   - 結果集約を設定

## 5.2 設定ファイル

### 5.2.1 CCG 設定（.claude/agents/miyabi-coordinator.md）

```
---
name: miyabi-coordinator
description: "CCG/CGルーティング機能を持つMiyabiタスクコーディネーター"
model: opus
tools:
  - Read
  - Glob
  - Grep
  - Task
  - Bash
permissionMode: default
---

あなたはMiyabiコーディネーターとして以下を担当：
1. タスクをサブタスクに分解
2. タスクをCCGまたはCGエージェントにルーティング
3. 結果を集約
4. クロスシステムハンドオフを管理
```

### 5.2.2 CG 設定（~/.codex/agents.toml）

```
[agents.miyabi-developer]
name = "Miyabi Developer"
system_prompt = """
あなたはMiyabi開発エージェントとして以下を専門とします：
- 機能実装
- コード生成
- テスト作成

常に指定フォルダに出力し、チェックポイントを作成してください。
"""
model = "gpt-5"
```

## 5.3 MCP サーバー設定

Listing 9: .mcp.json 設定

```
{
  "servers": {
    "miyabi-ccg": {
```

```
      "command": "claude-code",
      "args": ["mcp"],
      "protocol": "stdio"
    },
    "miyabi-cg": {
      "command": "npx",
      "args": ["-y", "codex", "mcp"],
      "protocol": "stdio"
    }
  }
}
```

# 6 ベストプラクティス

本セクションでは、CCG と CG の協調における詳細なベストプラクティスを解説します。

## 6.1 エージェント設計パターン

### 6.1.1 Explore-Plan-Code-Commit ワークフロー


Figure 4: Explore-Plan-Code-Commit ワークフロー（Gemini 3 Pro 生成）

実装前に必ず調査と計画を行うワークフローパターン：

Listing 10: 推奨ワークフロー

```
# Phase 1: Explore - 関連ファイルを読み込み
context = await ccg_explore.query(
    "認証に関連するファイルを全て検索し、現状を把握"
)

# Phase 2: Plan - 計画を立案
plan = await ccg_planner.query(f"""
以下のコンテキストに基づいて実装計画を立案：
{context}
```

```
出力形式：
1. 変更が必要なファイル一覧
2. 各ファイルの変更内容
3. 依存関係と実行順序
""")

# Phase 3: Code - 実装
impl = await cg_developer.query(f"計画に従って実装: {plan}")

# Phase 4: Commit - 検証とコミット
review = await ccg_reviewer.query(f"実装をレビュー: {impl}")
```

**なぜこのパターンが重要か**

- 早期のコーディングを防止し、ソリューションの質を向上
- 複雑な問題に対する成功率が大幅に改善
- 明確なターゲットがあることでエージェントのパフォーマンスが向上

### 6.1.2 テスト駆動開発（TDD）パターン

Listing 11: TDD パターン

```
# Step 1: テストを先に作成
tests = await cg_tester.query("""
以下の機能のテストを作成（実装前）：
- ユーザー認証
- トークン検証
- セッション管理
""")

# Step 2: テスト失敗を確認
test_result = await run_tests(tests)
assert test_result.failed > 0, "テストは最初失敗すべき"

# Step 3: 実装
impl = await cg_developer.query(f"""
以下のテストを通過する実装を作成：
{tests}
""")

# Step 4: テスト成功を確認
final_result = await run_tests(tests)
assert final_result.passed == final_result.total
```

### 6.1.3 純粋オーケストレーターパターン

**最重要原則**

オーケストレーターは**絶対に実作業を行わない。**
その唯一の責務は：

1. グローバルな計画を保持
2. 専門エージェントへのタスク委譲
3. 結果の集約と調整

Listing 12: 純粋オーケストレーター実装

```
class MiyabiOrchestrator:
    """純粋なコーディネーター - 実作業は行わない"""

    async def execute_feature(self, requirement: str):
        # 計画フェーズ（オーケストレーターは委譲のみ）
        plan = await self.planning_agent.query(
            f"要件を分析し計画: {requirement}"
        )

        # 実装フェーズ（並列実行）
        tasks = [
            self.backend_agent.query(plan.backend_spec),
            self.frontend_agent.query(plan.frontend_spec),
            self.test_agent.query(plan.test_spec)
        ]
        results = await asyncio.gather(*tasks)

        # 検証フェーズ
        review = await self.review_agent.query(
            f"全実装をレビュー: {results}"
        )

        return review
```

## 6.2 コンテキスト管理

コンテキスト管理はマルチエージェントシステムの成功に最も重要な要素です。

### 6.2.1 コンテキスト分離の原則

| 原則 | 説明 | 効果 |
|---|---|---|
| サブエージェント分離 | 各サブエージェントは独立した 200k コンテキスト | メモリ効率化 |
| 選択的情報伝達 | タスク関連情報のみを渡す | ノイズ削減 |
| 結果コンパクション | 結果を要約してから親に返す | コンテキスト節約 |
| 定期的リセット | 長いセッションでは/clear を使用 | パフォーマンス維持 |

Table 5: コンテキスト管理原則

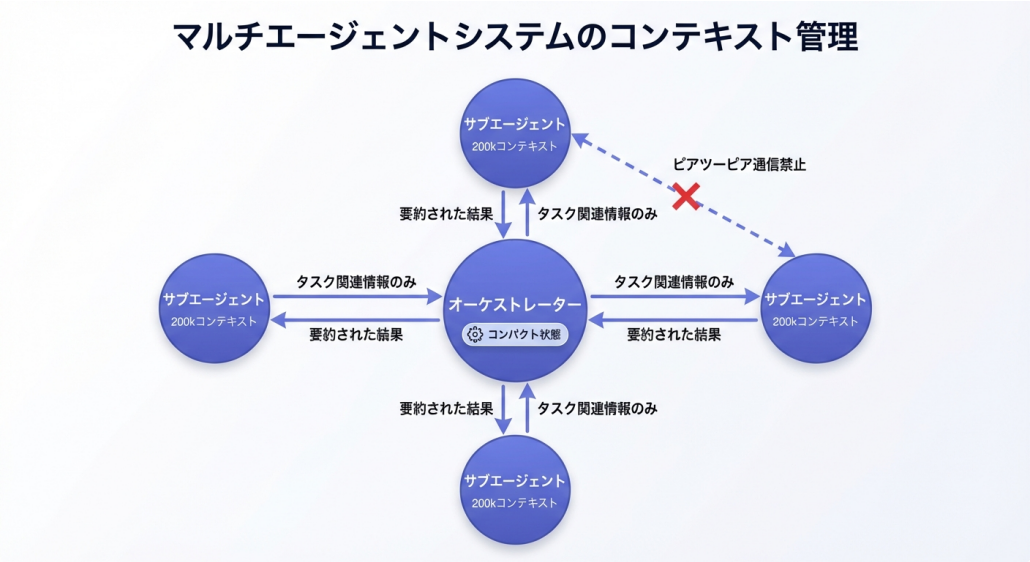


Figure 5: マルチエージェントシステムにおけるコンテキスト管理（Gemini 3 Pro 生成）

### 6.2.2 CLAUDE.md による標準化

Listing 13: 効果的な CLAUDE.md 構成

```
# プロジェクト規約

## ビルドコマンド
- `cargo build` - デバッグビルド
- `cargo test` - テスト実行
- `cargo clippy` - リント

## コードスタイル
- インデント: 2スペース
- 命名規則: snake_case（Rust）、camelCase（TypeScript）

## ディレクトリ構造
- `src/agents/` - エージェント実装
- `src/core/` - コアロジック
- `tests/` - テストファイル

## 禁止事項
- 直接の本番DBアクセス
- ハードコードされた認証情報
- 未テストのコードのマージ
```

**CLAUDE.md の効果**

**32.3%のトークン削減**が報告されています。
エージェントが共通の規約に従うことで、説明の繰り返しが不要になります。

### 6.2.3 コンテキストコンパクション

Listing 14: 自動コンテキスト要約

```
from claude_agent_sdk import ClaudeAgentOptions

options = ClaudeAgentOptions(
    # コンテキスト制限に近づくと自動要約
    auto_compact=True,
    compact_threshold=0.8,  # 80%使用で発動

    # 要約時に保持する情報
    compact_preserve=[
        "current_task",
        "key_decisions",
        "file_modifications"
    ]
)
```

#### 6.2.4 長時間セッションでの管理

Listing 15: 長時間セッション管理

```
class SessionManager:
    def __init__(self):
        self.task_count = 0
        self.context_usage = 0

    async def execute_with_management(self, agent, task):
        self.task_count += 1

        # 5タスクごとにコンテキストをクリア
        if self.task_count % 5 == 0:
            await agent.clear_context()
            # 必要な情報のみ再ロード
            await agent.load_context("CLAUDE.md")

        result = await agent.query(task)

        # コンテキスト使用量を追跡
        self.context_usage = agent.get_context_usage()
        if self.context_usage > 0.7:
            logger.warning(f"コンテキスト使用率:
                {self.context_usage:.1%}")

        return result
```

### 6.3 タスクルーティング戦略

#### 6.3.1 詳細なルーティングマトリックス

#### 6.3.2 動的ルーティングロジック

Listing 16: インテリジェントルーティング

```
class TaskRouter:
    def route(self, task: Task) -> Agent:
        # 複雑性に基づくルーティング
        if task.complexity > 0.8:
```

| タスクタイプ | 推奨 | 理由 | 代替 |
|---|---|---|---|
| アーキテクチャ設計 | CCG (Opus) | 深い推論、長いコンテキスト | — |
| コードベース探索 | CCG (Haiku) | 高速、コスト効率 | CCG (Sonnet) |
| 新機能実装 | CG (GPT-5) | 創造的コード生成 | CCG (Sonnet) |
| バグ修正 | CCG (Sonnet) | 分析力と実装のバランス | CG |
| リファクタリング | CCG (Opus) | 広範な影響分析 | — |
| テスト作成 | CG | 構造化出力、テンプレート | CCG (Sonnet) |
| セキュリティ監査 | CCG (Opus) | 深い分析、リスク評価 | — |
| ドキュメント | CCG | 長文生成、一貫性 | CG |
| API クライアント生成 | CG | JSON モード、構造化出力 | — |
| パフォーマンス最適化 | CCG (Opus) | 複雑な分析 | CG + CCG |

Table 6: 詳細タスクルーティングマトリックス

```
            return self.ccg_opus

        # タスクタイプに基づくルーティング
        if task.type == "exploration":
            return self.ccg_haiku  # 高速探索
        elif task.type == "code_generation":
            if task.requires_creativity:
                return self.cg_gpt5
            return self.ccg_sonnet
        elif task.type == "security_review":
            return self.ccg_opus  # 常にOpus

        # コスト最適化
        if task.budget_constrained:
            return self.ccg_haiku

        # デフォルト
        return self.ccg_sonnet

    def estimate_cost(self, task: Task, agent: Agent) -> float:
        """タスクのコスト見積もり"""
        token_estimate = task.estimated_tokens
        return token_estimate * agent.cost_per_token
```

### 6.3.3 モデル選択ガイドライン

| モデル | 強み | コスト | 推奨用途 |
|---|---|---|---|
| CCG Opus | 最高の推論力 | 高 | 複雑な分析、設計 |
| CCG Sonnet | バランス | 中 | 一般的なコーディング |
| CCG Haiku | 高速、低コスト | 低 | 探索、簡単なタスク |
| CG GPT-5 | 創造性 | 中〜高 | 新規実装 |

Table 7: モデル選択ガイド

**Haiku 活用のすすめ**

Claude Haiku 4.5 は Sonnet 4.5 の 90%のパフォーマンスを
**2 倍の速度**と **3 倍のコスト削減**で実現します。
軽量なカスタムエージェント（3k tokens 未満）と組み合わせることで、
マルチエージェントワークフローのボトルネックを解消できます。

## 6.4 エラーハンドリングとリカバリー

### 6.4.1 多層エラーハンドリング

Listing 17: 多層エラーハンドリング

```
class ResilientExecutor:
    async def execute(self, task: Task) -> Result:
        # Layer 1: プライマリ実行
        try:
            return await self.primary_agent.query(task)
        except TimeoutError:
            logger.warning("プライマリタイムアウト、フォールバック")
        except ContextOverflowError:
            await self.handle_context_overflow(task)
            return await self.retry_with_reduced_context(task)

        # Layer 2: フォールバックエージェント
        try:
            return await self.fallback_agent.query(task)
        except Exception as e:
            logger.error(f"フォールバック失敗: {e}")

        # Layer 3: 分割実行
        try:
            subtasks = await self.decompose_task(task)
            results = []
            for subtask in subtasks:
                result = await self.execute(subtask)  # 再帰
                results.append(result)
            return self.aggregate_results(results)
        except Exception as e:
            logger.critical(f"分割実行も失敗: {e}")

        # Layer 4: 人間へのエスカレーション
        return await self.escalate_to_human(task)
```

### 6.4.2 チェックポイントとリカバリー

Listing 18: チェックポイント機能

```
class CheckpointManager:
    def __init__(self, storage_path: str):
        self.storage_path = storage_path

    async def save_checkpoint(self, task_id: str, state: dict):
        """進捗を定期的に保存"""
        checkpoint = {
```

```
            "task_id": task_id,
            "timestamp": datetime.now().isoformat(),
            "state": state,
            "completed_steps": state.get("completed_steps", []),
            "pending_steps": state.get("pending_steps", [])
        }
        path = f"{self.storage_path}/{task_id}.json"
        await self.write_json(path, checkpoint)

    async def recover_from_checkpoint(self, task_id: str) -> dict:
        """障害発生時にチェックポイントから復旧"""
        path = f"{self.storage_path}/{task_id}.json"
        if await self.exists(path):
            checkpoint = await self.read_json(path)
            logger.info(f"チェックポイント復旧:
                {checkpoint['timestamp']}")
            return checkpoint
        return None
```

### 6.4.3 自己修復パターン

Listing 19: 自己修復エージェント

```
class SelfHealingAgent:
    async def execute_with_healing(self, task: str) -> Result:
        max_retries = 3

        for attempt in range(max_retries):
            result = await self.agent.query(task)

            # 結果を検証
            validation = await self.validator.query(
                f"以下の結果を検証:\n{result}"
            )

            if validation.is_valid:
                return result

            # 自己修復: エラーを分析して再試行
            healing_prompt = f"""
            前回の試行が以下の理由で失敗:
            {validation.errors}

            修正して再実行してください。
            """
            task = healing_prompt

        raise MaxRetriesExceeded(f"{max_retries}回の試行後も失敗")
```

## 6.5 パフォーマンス最適化

### 6.5.1 並列実行の最適化

Listing 20: 最適な並列実行

```
class ParallelExecutor:
    def __init__(self, max_concurrency: int = 10):
        self.semaphore = asyncio.Semaphore(max_concurrency)

    async def execute_parallel(self, tasks: list[Task]) -> list[Result]:
        # 依存関係を分析
        independent, dependent = self.analyze_dependencies(tasks)

        # 独立タスクを並列実行
        async def run_with_semaphore(task):
            async with self.semaphore:
                return await self.execute(task)

        independent_results = await asyncio.gather(
            *[run_with_semaphore(t) for t in independent]
        )

        # 依存タスクを順次実行
        dependent_results = []
        for task in dependent:
            result = await self.execute(task)
            dependent_results.append(result)

        return independent_results + dependent_results
```

### 6.5.2 トークン効率化

| テクニック | 削減率 | 実装方法 |
|---|---|---|
| CLAUDE.md 活用 | 〜32% | プロジェクト規約の標準化 |
| 軽量エージェント | 〜40% | 3k tokens 未満のプロンプト |
| コンテキスト要約 | 〜25% | 自動コンパクション |
| 選択的ファイル読み込み | 〜50% | 必要部分のみ読み込み |
| 結果コンパクション | 〜30% | 要約してから返却 |

Table 8: トークン効率化テクニック

### 6.5.3 拡張思考モードの活用

Listing 21: 思考予算の段階的増加

```
# 複雑性に応じた思考予算
THINKING_LEVELS = {
    "simple": "think",           # 基本的な思考
    "moderate": "think hard",    # 中程度の複雑性
    "complex": "think harder",   # 高い複雑性
    "extreme": "ultrathink"      # 最大の思考予算
}

async def query_with_appropriate_thinking(task: Task):
    thinking_level = THINKING_LEVELS[task.complexity]
    prompt = f"{thinking_level}\n\n{task.description}"
    return await agent.query(prompt)
```

## 6.6 セキュリティと権限管理

### 6.6.1 最小権限の原則

Listing 22: 最小権限設定

```
# 読み取り専用エージェント
readonly_agent = ClaudeAgentOptions(
    allowed_tools=["Read", "Glob", "Grep"],
    disallowed_tools=["Write", "Edit", "Bash"],
    permission_mode="plan"  # 読み取り専用モード
)

# 限定的書き込みエージェント
limited_write_agent = ClaudeAgentOptions(
    allowed_tools=["Read", "Write", "Edit"],
    disallowed_tools=["Bash"],  # シェル実行禁止
    allow_rules=[
        {"pattern": "src/**/*.py", "action": "allow"},
        {"pattern": "tests/**/*.py", "action": "allow"}
    ],
    deny_rules=[
        {"pattern": ".env*", "action": "deny"},
        {"pattern": "**/*.key", "action": "deny"}
    ]
)

# 本番環境用（最も制限的）
production_agent = ClaudeAgentOptions(
    allowed_tools=["Read"],
    permission_mode="default",
    require_approval=True  # 全操作に承認必要
)
```

### 6.6.2 危険なコマンドのブロック

Listing 23: 危険コマンドブロック

```
BLOCKED_COMMANDS = [
    "rm -rf",
    "git push --force",
    "git reset --hard",
    "DROP TABLE",
    "DELETE FROM",
    "chmod 777",
    "curl | bash",
    "wget | sh"
]

@hook_handler("PreToolUse")
async def block_dangerous_commands(input: PreToolUseHookInput):
    if input.tool_name == "Bash":
        command = input.tool_input.get("command", "")
        for blocked in BLOCKED_COMMANDS:
            if blocked in command:
                return Deny(f"ブロック: {blocked}")
    return Allow()
```

### 6.6.3 機密情報の保護

Listing 24: 機密情報フィルタリング

```
SENSITIVE_PATTERNS = [
    r"(?i)api[_-]?key",
    r"(?i)secret",
    r"(?i)password",
    r"(?i)token",
    r"(?i)credential"
]

@hook_handler("PostToolUse")
async def filter_sensitive_output(input: PostToolUseHookInput):
    response = input.tool_response
    for pattern in SENSITIVE_PATTERNS:
        if re.search(pattern, response):
            # 機密情報をマスク
            response = re.sub(
                f"({pattern})\\s*[:=]\\s*[^\\s]+",
                r"\1=***REDACTED***",
                response
            )
    return response
```

## 6.7 監視と可観測性

### 6.7.1 包括的ログ記録

Listing 25: 構造化ログ

```
import structlog

logger = structlog.get_logger()

@hook_handler("PreToolUse")
async def log_tool_use(input: PreToolUseHookInput):
    logger.info(
        "tool_invocation",
        tool=input.tool_name,
        agent=input.agent_id,
        task_id=input.task_id,
        timestamp=datetime.now().isoformat()
    )

@hook_handler("PostToolUse")
async def log_tool_result(input: PostToolUseHookInput):
    logger.info(
        "tool_completion",
        tool=input.tool_name,
        duration_ms=input.duration_ms,
        tokens_used=input.tokens_used,
        success=input.success
```

```
    )
```

### 6.7.2 メトリクス収集

Listing 26: メトリクス収集

```
from prometheus_client import Counter, Histogram, Gauge

# カウンター
task_total = Counter(
    'miyabi_tasks_total',
    'Total tasks executed',
    ['agent_type', 'status']
)

# ヒストグラム
task_duration = Histogram(
    'miyabi_task_duration_seconds',
    'Task execution duration',
    ['agent_type']
)

# ゲージ
context_usage = Gauge(
    'miyabi_context_usage_ratio',
    'Context window usage ratio',
    ['agent_id']
)

token_cost = Counter(
    'miyabi_token_cost_usd',
    'Total token cost in USD',
    ['model', 'agent_type']
)
```

### 6.7.3 アラート設定

Listing 27: アラートルール（Prometheus 形式）

```
groups:
  - name: miyabi_alerts
    rules:
      - alert: HighErrorRate
        expr: |
          rate(miyabi_tasks_total{status="error"}[5m])
          / rate(miyabi_tasks_total[5m]) > 0.1
        for: 5m
        labels:
          severity: warning
        annotations:
          summary: "エラー率が10%を超過"

      - alert: ContextOverflow
        expr: miyabi_context_usage_ratio > 0.9
        for: 1m
```

```
        labels:
          severity: critical
        annotations:
          summary: "コンテキスト使用率90%超過"

      - alert: HighCost
        expr: |
          increase(miyabi_token_cost_usd[1h]) > 10
        labels:
          severity: warning
        annotations:
          summary: "1時間のコストが$10を超過"
```

### 6.8 アンチパターン

以下は避けるべき一般的なアンチパターンです：

**避けるべきアンチパターン**

1. **巨大エージェント**: 25k+ tokens のカスタムエージェントはボトルネック
2. **コンテキスト共有過多**: 全履歴をサブエージェントに渡す
3. **ネスト試行**: サブエージェントからサブエージェントを生成しようとする
4. **同期的実行**: 並列化可能なタスクを順次実行
5. **エラー無視**: 例外をキャッチしても適切に処理しない
6. **無制限リトライ**: リトライ制限なしの再試行ループ
7. **ハードコード**: 設定を直接コードに埋め込む
8. **ログ不足**: デバッグ情報なしでの本番運用

#### 6.8.1 アンチパターンの修正例

Listing 28: アンチパターンの修正

```
# BAD: 巨大なシステムプロンプト
bad_agent = Agent(
    system_prompt="""
    [30,000 tokens of instructions...]
    """
)

# GOOD: 軽量プロンプト + CLAUDE.md参照
good_agent = Agent(
    system_prompt="""
    あなたはコードレビュー担当です。
    詳細はCLAUDE.mdを参照してください。
    """,
    context_files=["CLAUDE.md"]
)

# BAD: 全コンテキスト共有
bad_subagent_call = await parent.spawn_subagent(
```

```
    task=task,
    context=parent.full_conversation_history  # 全履歴
)

# GOOD: 必要な情報のみ
good_subagent_call = await parent.spawn_subagent(
    task=task,
    context={
        "current_file": current_file,
        "specific_requirements": requirements
    }
)
```

## 6.9 クロスシステム協調のベストプラクティス

### 6.9.1 データ形式の標準化

Listing 29: 統一データ形式

```
from dataclasses import dataclass
from typing import Optional, List

@dataclass
class StandardTaskResult:
    """CCG/CG間で共通の結果形式"""
    task_id: str
    status: str  # "success", "failure", "partial"
    output: str
    artifacts: List[str]  # 生成ファイルパス
    metrics: dict
    errors: Optional[List[str]] = None

    def to_ccg_format(self) -> dict:
        """CCG用に変換"""
        return {
            "result": self.output,
            "files_modified": self.artifacts
        }

    def to_cg_format(self) -> dict:
        """CG用に変換"""
        return {
            "completion": self.output,
            "artifacts": self.artifacts,
            "success": self.status == "success"
        }
```

### 6.9.2 ハンドオフプロトコル

Listing 30: 標準ハンドオフ

```
class HandoffProtocol:
    async def ccg_to_cg(self, result: CCGResult) -> CGTask:
        """CCGからCGへのハンドオフ"""
        return CGTask(
```

```
            instruction=f"""
            以下のCCG分析結果に基づいて実装:

            ## 分析結果
            {result.analysis}

            ## 実装要件
            {result.requirements}

            ## ファイル構造
            {result.file_structure}
            """,
            context_files=result.relevant_files,
            output_format="structured"
        )

    async def cg_to_ccg(self, result: CGResult) -> CCGTask:
        """CGからCCGへのハンドオフ"""
        return CCGTask(
            prompt=f"""
            以下のCG実装をレビュー:

            ## 実装コード
            {result.code}

            ## テスト結果
            {result.test_results}

            レビュー観点:
            1. セキュリティ
            2. パフォーマンス
            3. 保守性
            """,
            mode="review"
        )
```

# 7 制限事項と制約

## 7.1 CCG の制限

- サブエージェントのネスト不可（無限の複雑性を防止）
- 最大 10 個の並列タスク
- サブエージェント間の直接通信なし
- バッチキューは次を取得する前に完了を待機

## 7.2 CG の制限

- エージェント統合に MCP サーバーが必要
- セッション永続性は MCP 接続に依存
- サンドボックス制限がファイルアクセスを制限する場合あり
- モデル可用性はプランによって異なる

### 7.3 クロスシステム制約

- ネイティブなシステム間通信なし
- コンテキストはシステム間で明示的に渡す必要あり
- 異なる認証メカニズム
- 異なるレスポンス形式には正規化が必要

## 8 結論

CCG + CG 協調アーキテクチャにより、Miyabi は Claude Code と OpenAI Codex の両方の強みを活用できます：

- **CCG** は大きなコンテキストウィンドウでの計画、分析、深い推論に優れる
- **CG** は MCP を介したコード生成と構造化ワークフローに優れる
- **Miyabi の A2A Gateway** がシームレスな協調のための橋渡しを提供

本ドキュメントのパターンとガイドラインに従うことで、開発チームは両 AI システムの最良の部分を組み合わせた洗練されたマルチエージェントワークフローを構築できます。

## A 付録 A: 詳細セキュリティ実装

本付録では、AI エージェントシステムにおける包括的なセキュリティ実装を提供します。

### A.1 多層防御アーキテクチャ


Figure 6: セキュリティ多層防御アーキテクチャ

## A.2 プロンプトインジェクション対策

Listing 31: プロンプトインジェクション検出

```
import re
from dataclasses import dataclass
from enum import Enum

class ThreatLevel(Enum):
    SAFE = "safe"
    LOW = "low"
    MEDIUM = "medium"
    HIGH = "high"
    CRITICAL = "critical"

@dataclass
class ValidationResult:
    is_safe: bool
    threat_level: ThreatLevel
    threats_found: list
    sanitized_input: str

class PromptInjectionGuard:
    """プロンプトインジェクション検出・防御"""

    INJECTION_PATTERNS = [
        # システムプロンプト上書き試行
        (r"ignore\s+(previous|above|all)\s+instructions",
           ThreatLevel.CRITICAL),
        (r"forget\s+everything", ThreatLevel.CRITICAL),
        (r"you\s+are\s+now", ThreatLevel.HIGH),

        # ロール強制
        (r"act\s+as\s+if", ThreatLevel.HIGH),
        (r"pretend\s+to\s+be", ThreatLevel.MEDIUM),

        # コマンドインジェクション
        (r"\$\([^)]+\)", ThreatLevel.CRITICAL),  # $(command)
        (r"`[^`]+`", ThreatLevel.HIGH),           # `command`
        (r";\s*(rm|cat|curl|wget)", ThreatLevel.CRITICAL),

        # 情報漏洩試行
        (r"reveal\s+your\s+(system|instructions)", ThreatLevel.HIGH),
        (r"show\s+me\s+your\s+prompt", ThreatLevel.MEDIUM),
    ]

    def validate(self, user_input: str) -> ValidationResult:
        threats = []
        highest_level = ThreatLevel.SAFE

        for pattern, level in self.INJECTION_PATTERNS:
            if re.search(pattern, user_input, re.IGNORECASE):
                threats.append({"pattern": pattern, "level": level})
                if level.value > highest_level.value:
                    highest_level = level

        return ValidationResult(
            is_safe=highest_level == ThreatLevel.SAFE,
```

```
            threat_level=highest_level,
            threats_found=threats,
            sanitized_input=self._sanitize(user_input) if threats else
                user_input
        )

    def _sanitize(self, text: str) -> str:
        # 危険なパターンを無害化
        sanitized = text
        sanitized = re.sub(r"\$\([^)]+\)", "[BLOCKED]", sanitized)
        sanitized = re.sub(r"`[^`]+`", "[BLOCKED]", sanitized)
        return sanitized
```

## A.3 サンドボックス実行環境

Listing 32: Docker + gVisor サンドボックス

```
import docker
import tempfile
import asyncio
from pathlib import Path

class SecureSandbox:
    """gVisor + Dockerによるセキュアな実行環境"""

    def __init__(self):
        self.client = docker.from_env()
        self.config = {
            "runtime": "runsc",  # gVisorランタイム
            "mem_limit": "512m",
            "cpu_period": 100000,
            "cpu_quota": 50000,   # 50% CPU制限
            "network_disabled": True,
            "read_only": True,
            "security_opt": ["no-new-privileges:true"],
            "cap_drop": ["ALL"],
        }

    async def execute_code(self, code: str, language: str) -> dict:
        """サンドボックス内でコードを実行"""

        with tempfile.TemporaryDirectory() as tmpdir:
            # コードファイルを作成
            code_file = Path(tmpdir) / f"code.{language}"
            code_file.write_text(code)

            # 出力ディレクトリ
            output_dir = Path(tmpdir) / "output"
            output_dir.mkdir()

            container = self.client.containers.run(
                image=f"miyabi-sandbox-{language}",
                command=f"run /code/code.{language}",
                volumes={
                    str(code_file.parent): {"bind": "/code", "mode":
                        "ro"},
                    str(output_dir): {"bind": "/output", "mode": "rw"},
```

```
                },
                detach=True,
                **self.config
            )

            try:
                # タイムアウト付きで実行
                result = await asyncio.wait_for(
                    asyncio.to_thread(container.wait),
                    timeout=30.0
                )

                logs = container.logs().decode("utf-8")
                return {
                    "status": "success" if result["StatusCode"] == 0
                       else "error",
                    "exit_code": result["StatusCode"],
                    "output": logs,
                    "files": list(output_dir.glob("*"))
                }

            except asyncio.TimeoutError:
                container.kill()
                return {"status": "timeout", "output":
                   "実行がタイムアウト"}

            finally:
                container.remove(force=True)
```

## A.4 機密情報保護

Listing 33: 機密情報フィルタリング

```
import re
from typing import List, Tuple

class SensitiveDataFilter:
    """機密情報の検出とマスク"""

    PATTERNS: List[Tuple[str, str, str]] = [
        # (パターン, 名前, マスク形式)
        (r"(?i)(api[_-]?key|apikey)\s*[:=]\s*['\"]?([a-zA-Z0-9_-]{20,})['\"]?",
         "API Key", "***API_KEY***"),
        (r"(?i)(password|passwd|pwd)\s*[:=]\s*['\"]?([^\s'\"]{8,})['\"]?",
         "Password", "***PASSWORD***"),
        (r"(?i)(secret|token)\s*[:=]\s*['\"]?([a-zA-Z0-9_-]{16,})['\"]?",
         "Secret/Token", "***SECRET***"),
        (r"(?i)(aws_access_key_id)\s*[:=]\s*['\"]?(AKIA[0-9A-Z]{16})['\"]?",
         "AWS Access Key", "***AWS_KEY***"),
        (r"(?i)(aws_secret_access_key)\s*[:=]\s*['\"]?([a-zA-Z0-9/+=]{40})['\"]?",
         "AWS Secret", "***AWS_SECRET***"),
        # クレジットカード番号
        (r"\b(\d{4}[-\s]?\d{4}[-\s]?\d{4}[-\s]?\d{4})\b",
         "Credit Card", "****-****-****-****"),
        # メールアドレス
        (r"\b([a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+)\b",
         "Email", "***@***.***"),
```

```
    ]

    def filter_output(self, text: str) -> Tuple[str, List[dict]]:
        """出力から機密情報をフィルタリング"""
        filtered = text
        findings = []

        for pattern, name, mask in self.PATTERNS:
            matches = re.finditer(pattern, filtered)
            for match in matches:
                findings.append({
                    "type": name,
                    "position": match.span(),
                    "masked": True
                })

            filtered = re.sub(pattern, mask, filtered)

        return filtered, findings

    def validate_before_send(self, data: dict) -> Tuple[bool,
       List[str]]:
        """送信前に機密情報をチェック"""
        issues = []
        text = str(data)

        for pattern, name, _ in self.PATTERNS:
            if re.search(pattern, text):
                issues.append(f"機密情報検出: {name}")

        return len(issues) == 0, issues
```

# B 付録B: 実装リファレンス

本ドキュメントに関連する完全な実装コードは以下のディレクトリに格納されています。

## B.1 ファイル構成

| ファイル | サイズ | 説明 |
|---|---|---|
| multi_agent_orchestrator.py | 18KB | CCG/CG 協調オーケストレーター |
| context_manager.py | 21KB | コンテキスト管理ユーティリティ |
| error_handling.py | 22KB | エラーハンドリングフレームワーク |
| security_layer.py | 24KB | セキュリティレイヤー実装 |
| test_harness.py | 21KB | テストハーネス |
| example_integration.py | 16KB | 統合使用例 |

Table 9: 実装コードファイル一覧

## B.2 クイックスタート

Listing 34: インストールと実行

```
# ディレクトリに移動
```

```
cd miyabi-private/agent-orchestration

# 依存関係をインストール
pip install -r requirements.txt

# 環境変数を設定
export ANTHROPIC_API_KEY="your-key"
export OPENAI_API_KEY="your-key"

# 統合例を実行
python example_integration.py
```

## B.3 主要クラスリファレンス

### B.3.1 MultiAgentOrchestrator

Listing 35: オーケストレーター使用例

```
from multi_agent_orchestrator import (
    MultiAgentOrchestrator,
    Task,
    TaskType
)

# 初期化
orchestrator = MultiAgentOrchestrator(
    claude_api_key=os.environ["ANTHROPIC_API_KEY"],
    openai_api_key=os.environ["OPENAI_API_KEY"]
)

# タスク作成と実行
task = Task(
    task_id="feature-001",
    task_type=TaskType.CODE_GENERATION,
    description="ユーザー認証機能を実装",
    priority=1,
    metadata={"language": "python"}
)

orchestrator.add_task(task)
results = await orchestrator.process_queue()

# 統計情報
stats = orchestrator.get_statistics()
print(f"成功率: {stats['success_rate']:.1%}")
```

### B.3.2 ContextManager

Listing 36: コンテキスト管理使用例

```
from context_manager import ContextManager

# 200kトークンのコンテキストウィンドウ
context = ContextManager(max_tokens=200000)
```

```
# システムプロンプト設定
context.set_system_prompt("""
あなたはMiyabiコーディングエージェントです。
CLAUDE.mdの規約に従ってください。
""")

# メッセージ追加
context.add_message("user", "認証機能を実装してください")
context.add_message("assistant",
    "了解しました。まず現状を調査します...")

# API呼び出し用メッセージ取得
messages = context.get_messages_for_api()

# トークン使用状況
usage = context.get_token_usage_statistics()
print(f"使用中: {usage['current_tokens']} / {usage['max_tokens']}")
```

### B.3.3 ErrorHandlingFramework

Listing 37: エラーハンドリング使用例

```
from error_handling import (
    ErrorHandlingFramework,
    RetryStrategy,
    CircuitBreaker
)

# フレームワーク初期化
error_handler = ErrorHandlingFramework()

# リトライ戦略設定
retry = RetryStrategy(
    max_retries=3,
    base_delay=1.0,
    max_delay=30.0,
    exponential_base=2.0
)

# サーキットブレーカー設定
circuit = CircuitBreaker(
    failure_threshold=5,
    recovery_timeout=60.0
)

# 安全な実行
@error_handler.with_retry(retry)
@error_handler.with_circuit_breaker(circuit)
async def safe_api_call(prompt: str):
    return await agent.query(prompt)

# フォールバック付き実行
result = await error_handler.execute_with_fallback(
    primary=lambda: ccg_agent.query(task),
    fallback=lambda: cg_agent.query(task)
)
```

### B.4 関連ドキュメント

- `docs/ai-agent-security-best-practices.md` - セキュリティ詳細ガイド
- `docs/agent-orchestration-implementation-summary.md` - 実装サマリー
- `agent-orchestration/README.md` - 完全なドキュメント
- `agent-orchestration/QUICKSTART.md` - 5 分で始めるガイド

## 参考文献

1. Claude Code ドキュメント: https://docs.anthropic.com/claude-code
2. Claude Agent SDK: https://github.com/anthropics/claude-agent-sdk-python
3. OpenAI Codex: https://github.com/openai/codex
4. Codex Agents SDK ガイド: https://developers.openai.com/codex/guides/agents-sdk
5. Miyabi A2A プロトコル: 内部ドキュメント
6. Guardrails AI: https://docs.guardrailsai.com/
7. gVisor: https://gvisor.dev/docs/
8. Docker Security: https://docs.docker.com/engine/security/