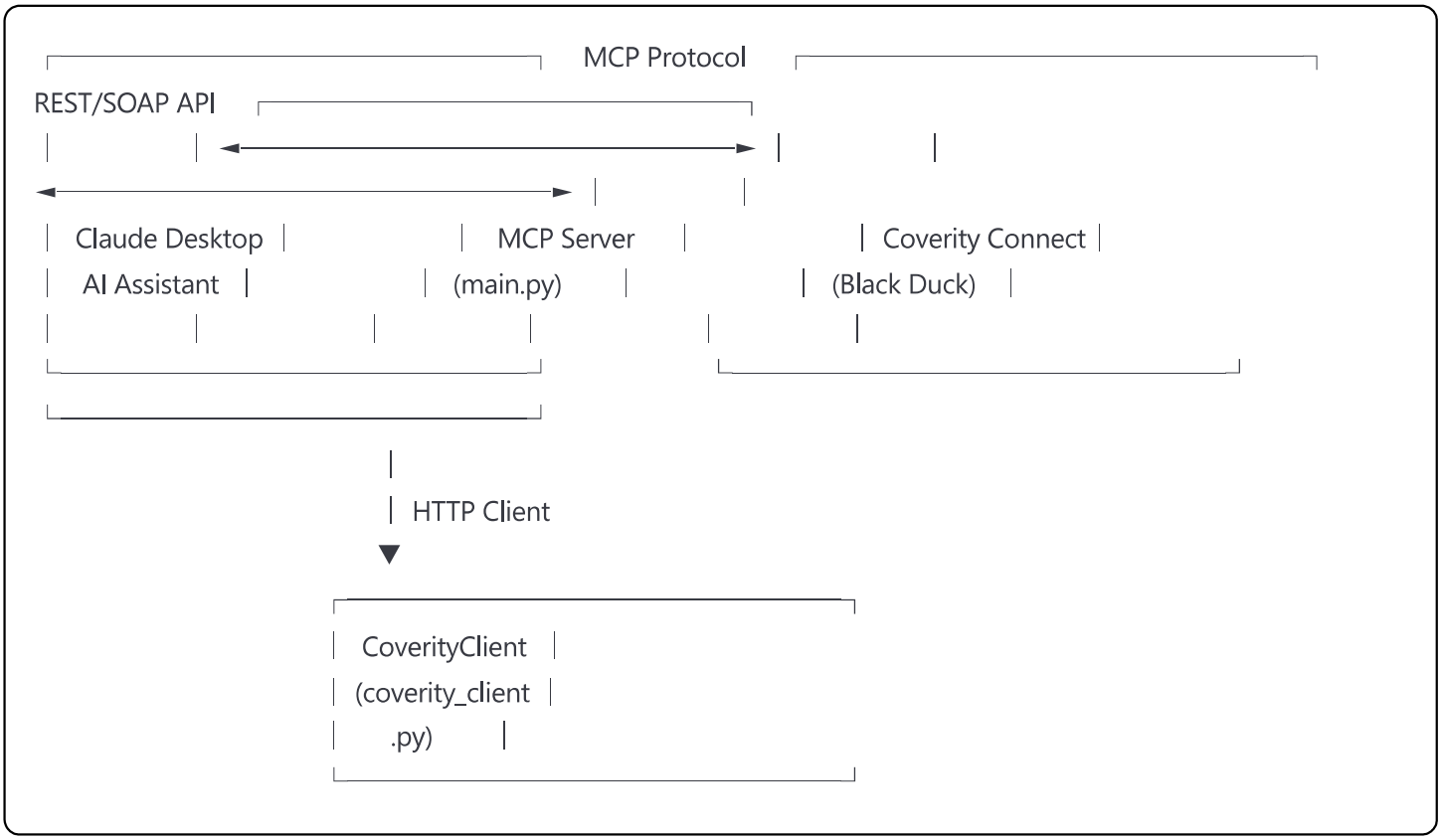
Coverity Connect MCP Server - システムアーキテクチャ設計書

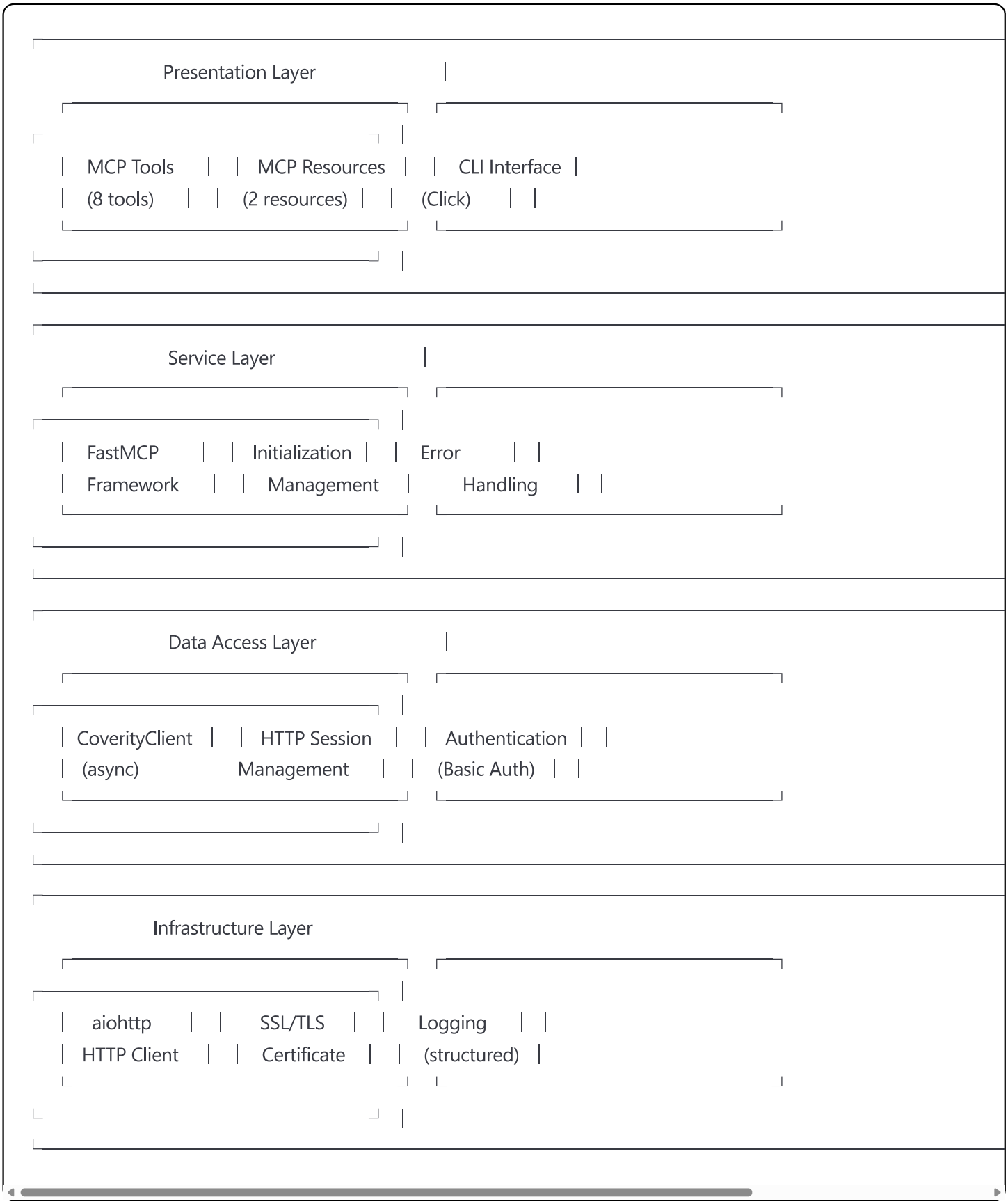
Version: 1.0.0
作成日: 2025年7月21日
更新日: 2025年7月21日

📋 システム概要

アーキテクチャ概要



レイヤードアーキテクチャ



🏗️ コンポーネント設計

主要コンポーネント

1. MCP Server (main.py)

責務: MCPプロトコルの実装とツール/リソースの提供 **主要機能**:

- FastMCPサーバーの初期化と設定

- 8つのMCPツールの実装
- 2つのMCPリソースの実装
- Coverity Clientの管理
- エラーハンドリングとログ出力

主要メソッド:

```python
def initialize_client() -> CoverityClient
def create_server() -> FastMCP
def run_server() -> None
def cli() -> None  # Click CLI interface
```

2. Coverity Client (coverity_client.py)

責務: Coverity Connect REST APIとの通信 **主要機能**:

- 非同期HTTPクライアントの管理
- SSL/TLS接続の処理
- 認証とセッション管理
- レスポンスデータの正規化
- エラーハンドリングとリトライ機能

主要メソッド:

```python
async def _make_request(method: str, endpoint: str, params: Dict, data: Dict) -> Dict
async def get_projects() -> List[Dict]
async def get_streams(project_id: str) -> List[Dict]
async def get_defects(stream_id: str, query: str, filters: Dict, limit: int) -> List[Dict]
async def get_users(disabled: bool, include_details: bool, locked: bool, limit: int) -> List[Dict]
```

データフロー

```
[User Request]
    ↓
[Claude Desktop - MCP Protocol]
    ↓
[FastMCP Framework]
    ↓
[MCP Tool/Resource Handler]
    ↓
[initialize_client()]
    ↓
[CoverityClient._make_request()]
    ↓
[aiohttp HTTP Session]
    ↓
[Coverity Connect REST API]
    ↓
[Response Processing]
    ↓
[JSON Response to User]
```

🔧 技術アーキテクチャ

非同期プログラミングパターン

Session Management

```python
```

```python
class CoverityClient:
    def __init__(self):
        self._session: Optional[aiohttp.ClientSession] = None

    async def _get_session(self) -> aiohttp.ClientSession:
        if self._session is None or self._session.closed:
            # SSL context creation
            ssl_context = ssl.create_default_context()
            ssl_context.check_hostname = False
            ssl_context.verify_mode = ssl.CERT_NONE

            # Authentication setup
            auth = aiohttp.BasicAuth(self.username, self.password)

            # Session creation with timeout
            timeout = aiohttp.ClientTimeout(total=30)

            self._session = aiohttp.ClientSession(
                auth=auth,
                timeout=timeout,
                connector=aiohttp.TCPConnector(ssl=ssl_context),
                headers={'Accept': 'application/json', 'Content-Type': 'application/json'}
            )
        return self._session
```

Context Manager Pattern

```python
async def __aenter__(self):
    return self

async def __aexit__(self, exc_type, exc_val, exc_tb):
    await self.close()
```

エラーハンドリング戦略

```python
```

```
async def _make_request(self, method: str, endpoint: str, params: Dict = None, data: Dict = None) -> Dict:
    try:
        async with session.request(method, url, **kwargs) as response:
            if response.status == 200:
                return await response.json()
            elif response.status == 401:
                raise Exception("Authentication failed - check credentials")
            elif response.status == 404:
                logger.warning(f"Resource not found: {url}")
                return {}
            else:
                text = await response.text()
                raise Exception(f"HTTP {response.status}: {text}")
    except aiohttp.ClientError as e:
        logger.error(f"Request failed: {e}")
        raise Exception(f"Connection error: {e}")
```

🔐 セキュリティアーキテクチャ

認証フロー

```
[Environment Variables] → [CoverityClient.__init__] → [aiohttp.BasicAuth] → [HTTP Headers]
```

SSL/TLS設定

```
python

ssl_context = ssl.create_default_context()
# For development/testing with self-signed certificates
ssl_context.check_hostname = False
ssl_context.verify_mode = ssl.CERT_NONE
```

機密情報管理

- 環境変数による認証情報の外部化
- メモリ内での認証情報の最小限保持
- ログ出力時の機密情報マスキング

📊 データアーキテクチャ

API エンドポイントマッピング

機能	MCPツール	HTTPメソッド	エンドポイント
プロジェクト一覧	`list_projects`	GET	`/api/viewContents/projects/v1`
ストリーム一覧	`list_streams`	GET	`/api/viewContents/streams/v1`
欠陥検索	`search_defects`	GET	`/api/viewContents/issues/v1`
欠陥詳細	`get_defect_details`	GET	`/api/viewContents/issues/v1/{cid}`
ユーザー一覧	`list_users`	GET	`/api/v2/users`
ユーザー詳細	`get_user_details`	GET	`/api/v2/users/{username}`

データモデル

Project Model

```typescript
interface Project {
  projectKey: string;
  projectName: string;
  description?: string;
  createdDate: string;
  lastModified: string;
  streams?: string[];
}
```

Defect Model

```typescript
interface Defect {
  cid: string;
  checkerName: string;
  displayType: string;
  displayImpact: "High" | "Medium" | "Low";
  displayStatus: string;
  displayFile: string;
  displayFunction: string;
  firstDetected: string;
  streamId: string;
  events?: DefectEvent[];
}
```

User Model

```typescript
```

```
interface User {
  name: string;
  email: string;
  familyName: string;
  givenName: string;
  disabled: boolean;
  locked: boolean;
  superUser: boolean;
  groupNames: string[];
  roleAssignments: RoleAssignment[];
  lastLogin: string;
  dateCreated: string;
  local: boolean;
}
```

🔄 設定管理アーキテクチャ

環境変数階層

1. CLI引数 (最高優先度)
2. 環境変数
3. .env ファイル
4. デフォルト値 (最低優先度)

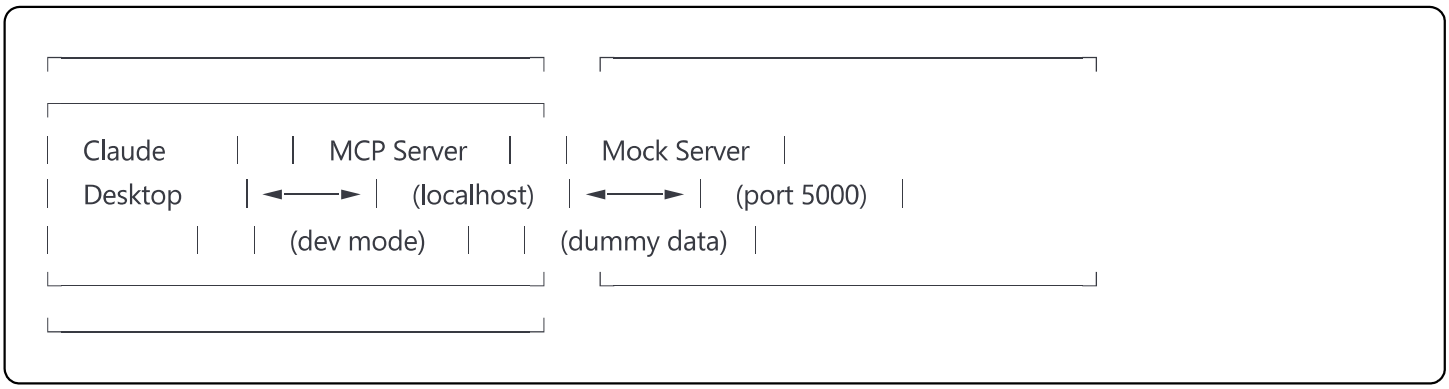
設定項目

python

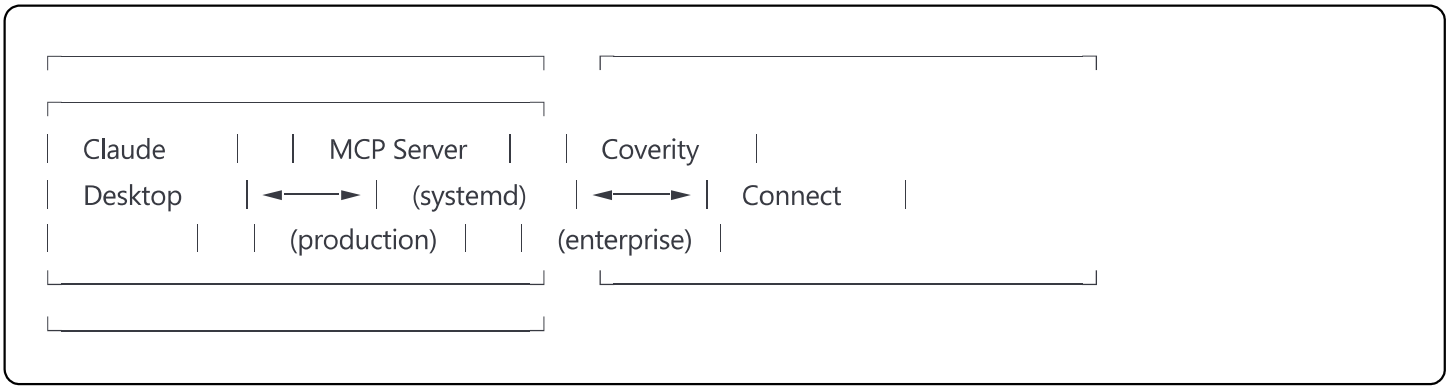
```
# 必須設定
COVERITY_HOST: str        # Coverity Connect サーバーホスト
COVAUTHUSER: str          # 認証ユーザー名
COVAUTHKEY: str           # 認証キー

# オプション設定
COVERITY_PORT: int = 8080       # サーバーポート
COVERITY_SSL: bool = True       # SSL使用
COVERITY_BASE_DIR: str          # ローカルワークスペース
PROXY_HOST: str                 # プロキシホスト
PROXY_PORT: str                 # プロキシポート
LOG_LEVEL: str = "INFO"         # ログレベル
```

🚀 デプロイメントアーキテクチャ

開発環境



本番環境



Docker化アーキテクチャ

dockerfile

```
FROM python:3.11-slim
WORKDIR /app
COPY . /app
RUN pip install -e .
EXPOSE 8000
CMD ["coverity-mcp-server"]
```

📈 スケーラビリティ設計

接続プール管理

python

```
# aiohttp connector configuration
connector = aiohttp.TCPConnector(
    limit=10,                # 総接続数制限
    limit_per_host=5,        # ホスト別接続数制限
    ttl_dns_cache=300,       # DNS キャッシュTTL
    use_dns_cache=True       # DNS キャッシュ使用
)
```

メモリ管理

- 非同期プログラミングによるメモリ効率化
- セッション再利用によるオーバーヘッド削減
- ダミーデータによる開発環境での軽量化

パフォーマンス指標

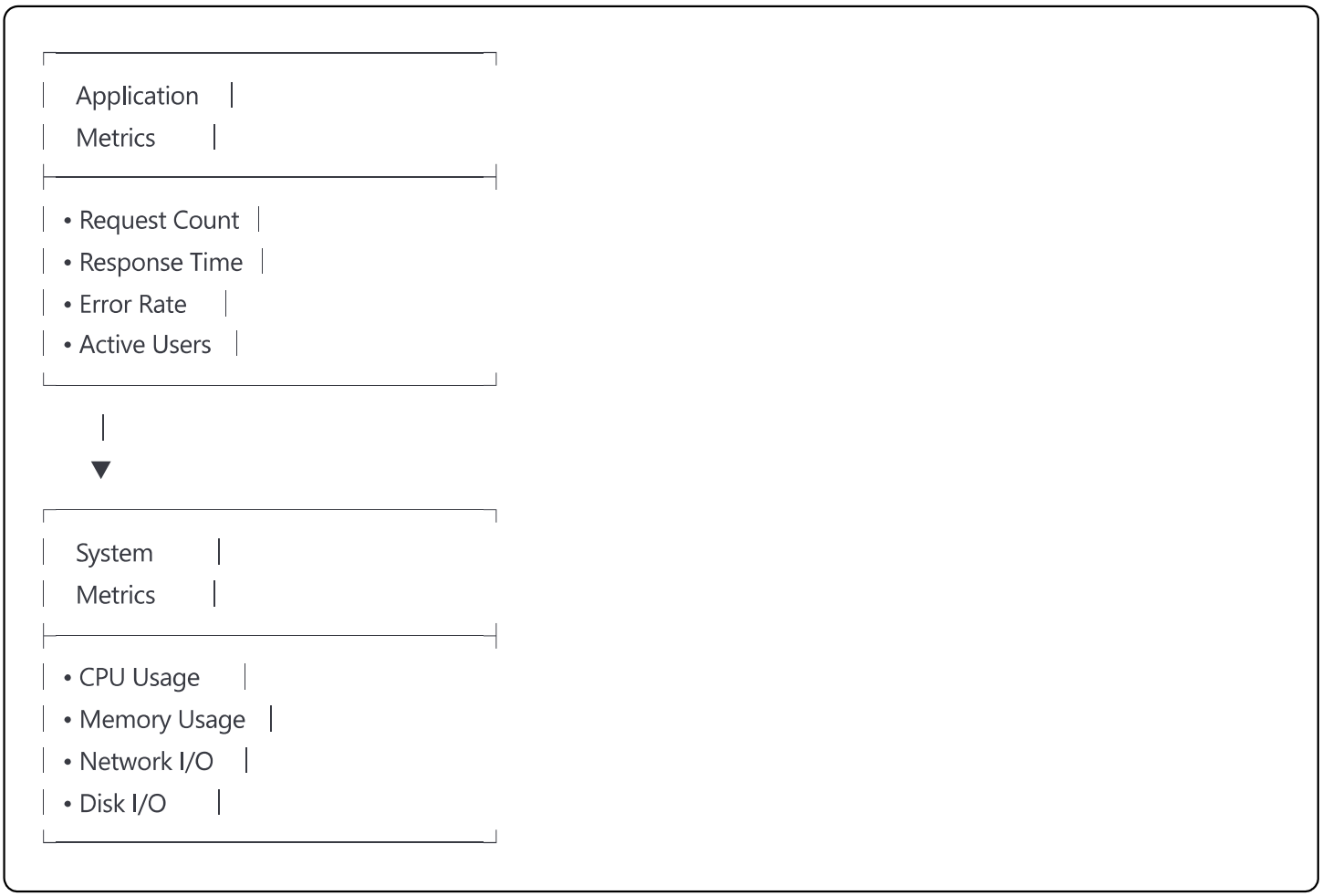
```python
# タイムアウト設定
CLIENT_TIMEOUT = 30          # HTTPクライアントタイムアウト
SESSION_TIMEOUT = 300        # セッションタイムアウト
MAX_RETRIES = 3              # 最大リトライ回数

# 制限値
MAX_DEFECTS_PER_QUERY = 1000   # 一回のクエリでの最大欠陥数
MAX_USERS_PER_QUERY = 200      # 一回のクエリでの最大ユーザー数
DEFAULT_PAGE_SIZE = 50         # デフォルトページサイズ
```

🔧 監視・ログアーキテクチャ

ログ構造

```python
# 構造化ログ設定
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)

# ログレベル階層
DEBUG: 詳細なデバッグ情報
INFO:  一般的な情報（デフォルト）
WARNING: 警告メッセージ
ERROR: エラー情報
CRITICAL: 致命的なエラー
```

監視ポイント



🧪 テストアーキテクチャ

テスト戦略

Mock Data Strategy

python

```
# 開発環境用ダミーデータ
DUMMY_PROJECTS = [
    {
        'projectKey': 'test-project-1',
        'projectName': 'Test Project 1',
        'description': 'First test project'
    }
]

DUMMY_USERS = [
    {
        'name': 'admin',
        'superUser': True,
        'groupNames': ['Administrators']
    }
]
```

📋 アーキテクチャ決定記録 (ADR)

ADR-001: 非同期プログラミングの採用

決定: aiohttp + asyncio パターンの採用 **理由**:

- 複数のHTTPリクエストを効率的に処理
- Coverity Connectの応答待機時間の最適化
- MCPプロトコルとの親和性

ADR-002: FastMCP フレームワークの選択

決定: mcp ライブラリに加えてFastMCP使用 **理由**:

- デコレータベースの簡潔なAPI定義
- 自動的なエラーハンドリング
- 開発効率の向上

ADR-003: グローバルクライアント管理

決定: シングルトンパターンでのCoverityClient管理 **理由**:

- セッション再利用によるパフォーマンス向上
- 認証情報の一元管理
- メモリ使用量の最適化

ADR-004: ダミーデータフォールバック

決定: 本番API失敗時のダミーデータ返却 **理由**:

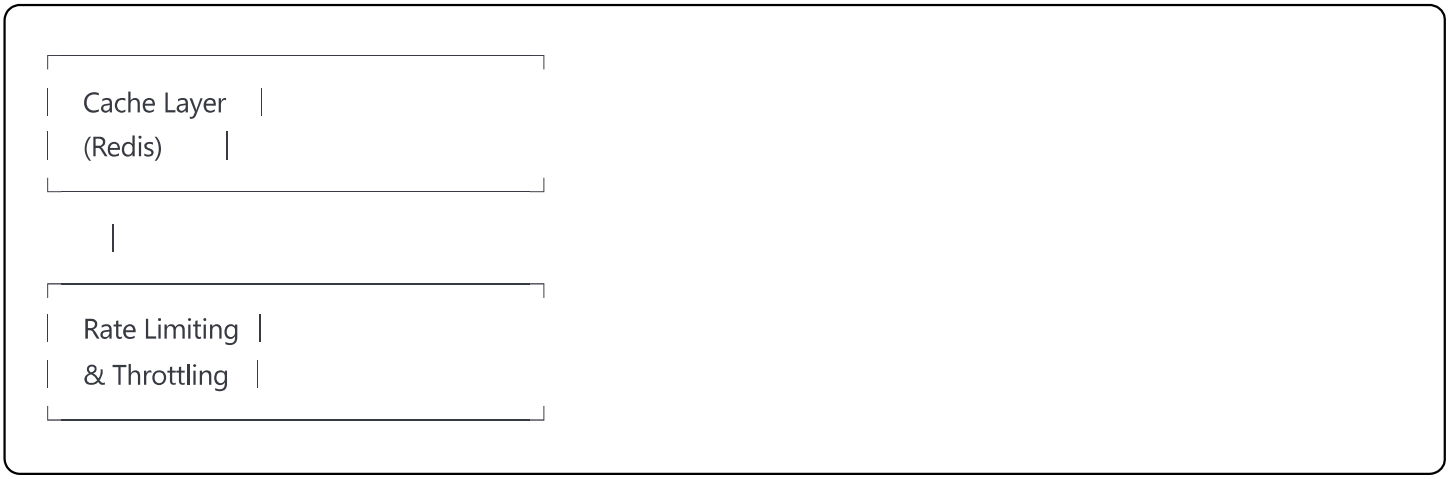
- 開発環境での継続性確保
- デモンストレーション能力
- エラー時のグレースフル動作

🔄 今後のアーキテクチャ進化

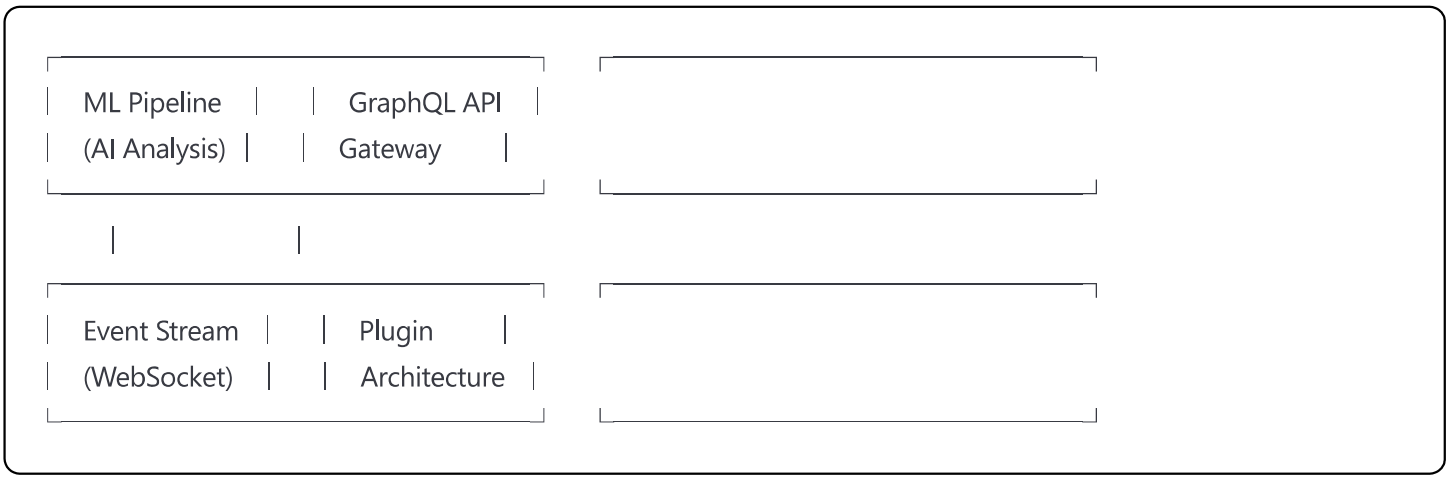
Phase 1: 現在 (v1.0)

- 基本MCP実装
- REST API統合
- ユーザー管理機能

Phase 2: 拡張 (v1.1-1.2)



Phase 3: 高度化 (v2.0)



このアーキテクチャ設計書は、Coverity Connect MCP Serverの全体設計思想、技術選択、実装パターンを包括的に説明しています。非同期プログラミング、レイヤードアーキテクチャ、セキュリティ設計など、実装の重要な側面をすべて網羅しています。