

Coverity Connect API から MCP サーバーへの統合分析

概要

指定されたリポジトリ (`keides2/cov_auto.git`) と (`keides2/cov_snap.git`) は直接アクセスできませんでしたが、**本格的なCoverity Connect API実装**と包括的なMCPサーバーパターンを特定し、MCPサーバー拡張プロジェクトの優れた基盤を提供できます。この分析では、成熟した認証パターン、堅牢なAPI統合アプローチ、およびCoverity Connect機能をMCPツールに変換するための明確な実装パスが明らかになりました。

リポジトリ分析結果

対象リポジトリの状況

`keides2`ユーザーの両リポジトリ (`cov_auto`と`cov_snap`) はプライベートまたは公開アクセスから削除されています。しかし、包括的なCoverity Connect API統合パターンを実証する**優れた代替実装**を見つけました。

発見されたCoverity Connect実装の代替案

主要参考実装: `jjarboe/python-coverity`

- **アーキテクチャ:** 構成、不具合、管理用の個別モジュールを持つマルチサービスSOAPクライアント
- **認証:** セキュア/非セキュア接続サポート付きSOAP UsernameTokenによるユーザー名/パスワード
- **API対応範囲:** Coverity Connect Webサービス (v5.0-6.5.1) の包括的対応
- **主要機能:** データオブジェクト用ファクトリパターン、URL生成、Base64復号化、日時処理

モダンな代替案: `synopsys-sig/coverity-common-api`

- **デュアルプロトコル対応:** SOAPとREST APIの両方の実装
- **生成クラス:** 構造化API対話用のリクエスト/レスポンスクラス
- **エンタープライズ対応:** 本番環境用の包括的API対応

MCPサーバーアーキテクチャ分析

中核MCPフレームワークコンポーネント

トランスポート層オプション:

- **STDIO:** ローカル開発とテスト用
- **HTTP+SSE:** リモートデプロイメントと本番使用用
- **WebSocket:** リアルタイム通信ニーズ用

プロトコル基盤:

- **JSON-RPC 2.0:** ベースメッセージングプロトコル
- **OAuth 2.1:** PKCEサポート付きモダン認証
- **構造化スキーマ:** パラメータ用JSON Schema検証

FastMCPフレームワークの利点

- **デコレータベースAPI:** `@mcp.tool()`、`@mcp.resource()`、`@mcp.prompt()`
- **自動スキーマ生成:** Pythonタイプヒントから
- **組み込みエラーハンドリング:** 構造化エラーレスポンス
- **環境設定:** 広範囲な設定管理

具体的なMCP拡張推奨事項

推奨アーキテクチャパターン

python

```

from fastmcp import FastMCP
import requests
from typing import Optional, Dict, List

# MCPサーバーの初期化
mcp = FastMCP(
    name="CoverityConnectServer",
    dependencies=["requests", "pydantic>=2.0.0"],
    include_tags={"projects", "defects", "scans", "admin"},
    exclude_tags={"internal", "deprecated"}
)

class CoverityClient:
    """分析されたパターンに基づく Coverity Connect APIクライアント"""

    def __init__(self, host: str, port: int, username: str, password: str, secure: bool = True):
        self.base_url = f"{'https' if secure else 'http'}://{host}:{port}"
        self.auth = (username, password)
        self.session = requests.Session()

    def connect(self):
        """Coverity Connectへの接続確立"""
        # 認証とセッション設定
        pass

    def query_defects(self, project_id: str, filters: Optional[Dict] = None) -> List[Dict]:
        """オプションフィルタ付きの不具合クエリ"""
        # 分析されたパターンに基づく実装
        pass

```

中核MCPツール実装

1. プロジェクト管理ツール

python

```

@mcp.tool(tags={"projects", "read"})
def get_coverity_projects() -> Dict:
    """全Coverityプロジェクトを取得"""
    client = CoverityClient.from_env()
    projects = client.get_projects()
    return {
        "content": [{
            "type": "text",
            "text": f"{len(projects)}個のプロジェクトが見つかりました: {[p['name'] for p in projects]}"
        }]
    }

@mcp.tool(tags={"projects", "write"})
def create_coverity_project(
    name: str,
    description: str = None,
    template_id: Optional[int] = None
) -> Dict:
    """新しいCoverityプロジェクトを作成"""
    client = CoverityClient.from_env()
    result = client.create_project(name, description, template_id)
    return {
        "content": [{
            "type": "text",
            "text": f"プロジェクト'{name}'をID: {result['project_id']}で作成しました"
        }]
    }

```

2. 不具合分析ツール

python

```
@mcp.tool(tags={"defects", "read"})
```

```
def get_project_defects(
```

```
    project_id: str,
```

```
    status: str = "open",
```

```
    severity: Optional[str] = None,
```

```
    limit: int = 100
```

```
) -> Dict:
```

```
    """フィルタリング付きで特定プロジェクトの不具合を取得"""
```

```
    client = CoverityClient.from_env()
```

```
    filters = {"status": status}
```

```
    if severity:
```

```
        filters["severity"] = severity
```

```
    defects = client.query_defects(project_id, filters)[:limit]
```

```
    return {
```

```
        "content": [{
```

```
            "type": "text",
```

```
            "text": f"プロジェクト {project_id} で {len(defects)} 個の {status} 不具合が見つかりました"
```

```
        ], {
```

```
            "type": "text",
```

```
            "text": "\n".join([f"- CID {d['cid']}: {d['checker']} in {d['file']}" for d in defects])
```

```
        ]
```

```
    }
```

```
@mcp.tool(tags={"defects", "read"})
```

```
def get_defect_details(project_id: str, cid: int) -> Dict:
```

```
    """特定の不具合の詳細情報を取得"""
```

```
    client = CoverityClient.from_env()
```

```
    defect = client.get_defect_details(project_id, cid)
```

```
    return {
```

```
        "content": [{
```

```
            "type": "text",
```

```
            "text": f"""
```

```
不具合CID {cid}の詳細:
```

```
- チェッカー: {defect['checker']}
```

```
- カテゴリ: {defect['category']}
```

```
- 影響度: {defect['impact']}
```

```
- ファイル: {defect['file']}:{defect['line']}
```

```
- ステータス: {defect['status']}
```

```
- 分類: {defect['classification']}
```

```
        """
```

```
    }  
}
```

3. スキャン管理ツール

python

```
@mcp.tool(tags={"scans", "read"})  
def get_scan_history(stream_id: str, limit: int = 10) -> Dict:  
    """ストリームのスキャン履歴を取得"""  
    client = CoverityClient.from_env()  
    scans = client.get_scan_history(stream_id, limit)  
  
    return {  
        "content": [{  
            "type": "text",  
            "text": f"ストリーム{stream_id}の最新{len(scans)}回のスキャン:\n" +  
                "\n".join([f"- {s['date']}: {s['status']} ({s['defect_count']}個の不具合)" for s in scans])  
        }]  
    }  
  
@mcp.tool(tags={"scans", "write"})  
def trigger_scan(stream_id: str, build_id: Optional[str] = None) -> Dict:  
    """ストリームの新しいスキャンをトリガー"""  
    client = CoverityClient.from_env()  
    result = client.trigger_scan(stream_id, build_id)  
  
    return {  
        "content": [{  
            "type": "text",  
            "text": f"ストリーム{stream_id}のスキャンをトリガーしました。スキャンID: {result['scan_id']}"  
        }]  
    }  
}
```

設定管理

環境設定:

python

```
# 環境変数設定
```

```
COVERITY_URL = os.getenv("COVERITY_URL")
```

```
COVERITY_USERNAME = os.getenv("COVERITY_USERNAME")
```

```
COVERITY_PASSWORD = os.getenv("COVERITY_PASSWORD")
```

```
COVERITY_TOOLSETS = os.getenv("COVERITY_TOOLSETS", "projects,defects,scans").split(",")
```

```
class CoverityClient:
```

```
    @classmethod
```

```
    def from_env(cls):
```

```
        """環境変数からクライアントを作成"""
```

```
        return cls(
```

```
            host=COVERITY_URL.split("://")[1].split(":")[0],
```

```
            port=int(COVERITY_URL.split(":")[-1]),
```

```
            username=COVERITY_USERNAME,
```

```
            password=COVERITY_PASSWORD
```

```
        )
```

MCPサーバー設定:

```
json
```

```
{
  "mcpServers": {
    "coverity": {
      "command": "uvx",
      "args": ["coverity-connect-mcp-server"],
      "env": {
        "COVERITY_URL": "${env:COVERITY_URL}",
        "COVERITY_USERNAME": "${env:COVERITY_USERNAME}",
        "COVERITY_PASSWORD": "${env:COVERITY_PASSWORD}",
        "COVERITY_TOOLSETS": "projects,defects,scans"
      }
    }
  }
}
```

認証実装

分析されたパターンに基づく複数メソッド認証の実装:

```
python
```

```
class CoverityAuthenticator:
    """複数の認証方法进行处理"""

    def __init__(self, auth_method: str = "basic"):
        self.auth_method = auth_method

    def get_auth_headers(self) -> Dict[str, str]:
        """メソッドに基づく認証ヘッダーを取得"""
        if self.auth_method == "basic":
            credentials = base64.b64encode(f"{username}:{password}".encode()).decode()
            return {"Authorization": f"Basic {credentials}"}
        elif self.auth_method == "token":
            return {"Authorization": f"Bearer {api_token}"}
        elif self.auth_method == "oauth":
            return {"Authorization": f"Bearer {oauth_token}"}

    def refresh_token(self):
        """必要に応じて認証トークンを更新"""
        pass
```

エラーハンドリングとログ記録

MCPパターンに基づく包括的なエラーハンドリング:

```
python
```



```

import logging
from fastmcp import FastMCP

# ログ設定
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

@mcp.tool(tags={"defects", "read"})
def get_defect_details(project_id: str, cid: int) -> Dict:
    """適切なエラーハンドリング付きで不具合詳細を取得"""
    try:
        client = CoverityClient.from_env()
        defect = client.get_defect_details(project_id, cid)
        return {
            "content": [{
                "type": "text",
                "text": f"不具合{cid}の詳細を正常に取得しました"
            }]
        }
    except requests.exceptions.RequestException as e:
        logger.error(f"APIリクエストが失敗しました: {e}")
        return {
            "isError": True,
            "content": [{
                "type": "text",
                "text": f"エラー: 不具合詳細の取得に失敗しました - {str(e)}"
            }]
        }
    except Exception as e:
        logger.error(f"予期しないエラー: {e}")
        return {
            "isError": True,
            "content": [{
                "type": "text",
                "text": f"エラー: 予期しないエラーが発生しました - {str(e)}"
            }]
        }

```

実装ロードマップ

フェーズ1: 中核インフラストラクチャ

1. FastMCPフレームワークの設定 - 適切な環境設定付き
2. CoverityClientクラスの実装 - 分析されたパターンに基づく
3. 認証システムの作成 - 複数メソッドサポート

4. 基本エラーハンドリング - ログ記録の確立

フェーズ2: 必須ツール

1. **プロジェクト管理ツール**: プロジェクトの一覧表示、作成、更新
2. **不具合クエリツール**: 不具合の検索、フィルタリング、取得
3. **基本スキャンツール**: スキャン履歴とステータスの取得

フェーズ3: 高度な機能

1. **スキャン管理**: スキャンのトリガー、進行状況の監視
2. **ユーザー管理**: 権限とアクセス制御の処理
3. **レポートツール**: 不具合レポートと分析の生成
4. **ストリーム管理**: ストリーム操作の処理

フェーズ4: 本番対応

1. **パフォーマンス最適化**: 接続プール、キャッシュ
2. **セキュリティ強化**: 安全な認証情報管理
3. **監視とアラート**: ヘルスチェックと通知
4. **ドキュメント**: 包括的APIドキュメント

主要な技術的利点

Coverity Connect APIをMCPツールに変換することで提供される利点:

- **標準化されたインターフェース**: 全Coverity操作における一貫したツール署名
- **型安全性**: JSON Schemaによる自動検証
- **エラーハンドリング**: 構造化されたエラーレスポンスとログ記録
- **認証**: 統一された認証情報管理
- **スケーラビリティ**: ローカルとリモート両方のデプロイメントサポート
- **統合**: シームレスなAIアシスタント統合

この実装アプローチは、既存のCoverity Connect実装のベストプラクティスを活用しながら、AIアシスタント統合のための最新のMCPサーバー機能を提供します。

次のステップ

あなたの具体的な実装をより良く支援するために、以下の情報をお教えいただけませんか？

1. **既存実装の主要機能**: `cov_auto`と`cov_snap`で何を自動化していますか？
2. **使用APIエンドポイント**: どのCoverity Connect APIを使用していますか？
3. **認証方式**: Basic認証、APIキー、OAuth等のどれを使用していますか？

4. **データ処理:** どのような分析やレポートを生成していますか？

これらの情報があれば、あなたの既存実装に特化したMCPサーバー拡張をより具体的に提案できます。