Multi-Agent AI 開発組織が生成するコードの脆弱性：仮想組織構造が及ぼす影響の調査

及川耕一 [1]　太田瞭 [1]

概要：生成 AI 及び AI エージェント技術の発展により，Multi-Agent AI 開発組織によるソフトウェア開発が現実的となった現代において，生成されるソフトウェアやコードの品質をどう担保するかが非常に重要な課題となっている．AI エージェントが大量のコードを生成できる一方で，脆弱性を多数含むソフトウェアやコードが無数に生み出される危険性があり，これは深刻なセキュリティリスクを社会にもたらす可能性がある．そこで本研究では，仮想組織の構造に着目し，Multi-Agent AI 開発組織の構造がセキュリティ品質にどのような影響を与えるかに焦点を当てた．疑似的仮想組織を用いた実験により，組織構造がコード生成時のセキュリティ脆弱性に与える影響を分析する．複数の組織構造パターンを比較検証し，脆弱性を最小限に抑えながら効率的な開発を実現する最適な組織設計手法を検討する．この研究により，コーディングエージェントによる開発におけるセキュアな仮想組織構造構築指針を提供し，実用的なソフトウェア開発手法の社会実装への貢献を企図する．

キーワード：マルチエージェントシステム，AI 駆動ソフトウェア開発，仮想組織構造

Code Vulnerabilities Generated by Multi-Agent AI Development Organizations: Investigation of the Impact of Virtual Organizational Structures

Kouichi Oikawa[1]　Ryo Ota[1]

Abstract: With the advancement of generative AI and AI agent technologies, software development by Multi-Agent AI development organizations has become realistic in the modern era, making the quality assurance of generated software and code a critically important issue. While AI agents can generate large volumes of code, there is a danger that countless software and code containing numerous vulnerabilities could be produced, potentially posing serious security risks to society. Therefore, this study focuses on virtual organizational structures and examines how Multi-Agent AI development organizational structures affect security quality. Through experiments using pseudo-virtual organizations, we analyze the impact of organizational structures on security vulnerabilities during code generation. We compare and verify multiple organizational structure patterns to investigate optimal organizational design methods that minimize vulnerabilities while achieving efficient development. This research aims to provide guidelines for constructing secure virtual organizational structures in coding agent-based development and contribute to the social implementation of practical software development methodologies.

Keywords: Multi-Agent Systems, AI-driven Software Development, Virtual Organizational Structure

1. はじめに

1.1 研究背景

近年，大規模言語モデル（LLM）を基盤としたコーディングエージェント技術の急速な発展により，複数の AI エージェントが協調してソフトウェア開発を行うことが現実的になってきた．GitHub Copilot，ChatGPT，Claude 等の単一エージェント型の開発支援ツールから MetaGPT[1]，ChatDev[2]等の複数エージェントが役割分担して開発を進めるシステムへと進化を遂げている．これらのシステムは大量のコードを高速に生成できる一方で，セキュリティ脆弱性を含むコードが無制御に生成される危険性が指摘されている．

特に，AI エージェントが生成するコードには，SQL インジェクション，クロスサイトスクリプティング（XSS），不適切な認証・認可処理等の脆弱性が含まれる可能性があり[3]，これらが大規模に拡散した場合，深刻な社会的セキュリティリスクをもたらす恐れがある．従来の人間組織では，コードレビューやセキュリティテスト等の品質保証プロセスが確立されているが，AI エージェント組織においてこれらの機能をどのように実装すべきかは不透明である．

本研究では，マルチエージェント AI 開発組織の構造がセキュリティ品質にどのような影響を与えるかに焦点を当て，仮想組織構造（階層型，スーパーバイザー型，ネットワーク型）の違いが生成コードの脆弱性発生率に与える影響を実証的に評価する．Semgrep の静的解析に基づく脆弱性分類を用いて定量的評価を行い，セキュアな開発を実現する最適な組織設計手法を検討する．

1 Trust Base 株式会社

2. 関連研究

2.1 マルチエージェント AI システム

MetaGPT は，プロダクトマネージャー，アーキテクト，エンジニア等の役割を持つ複数の AI エージェントが協調してソフトウェア開発を行うフレームワークを提案している．ChatDev では，CEO からテスターまでの開発チーム全体を AI エージェントで構成し，ウォーターフォール型の開発プロセスを実現している．しかし，これらの研究では組織構造の違いによる影響検証が不十分であると考えられる．

2.2 組織構造と開発効率性

Conway[4]は，システムの構造は組織のコミュニケーション構造を反映すると指摘した．Brooks は，ソフトウェア開発における人員追加が必ずしも生産性向上につながらないことを示した．これらの古典的理論が AI 組織にも適用可能かは不透明である．

2.3 AI コード生成とセキュリティリスク

Pearce et al.[5]は，GitHub Copilot が生成するコードの約 40%に何らかのセキュリティ脆弱性が含まれることを報告している．Perry et al.[6]は，開発者が AI 生成コードを過度に信頼する傾向があり，セキュリティレビューが不十分になることを指摘している．これらの研究は単一エージェントに焦点を当てており，マルチエージェント環境でのセキュリティリスクは十分に検証されていない．

2.4 セキュアコーディングと組織構造

Morrison et al.[7] は，開発チームの構造とセキュリティプラクティスの採用率に相関があることを示した．セキュリティ専門家を含む階層的な組織では，脆弱性の早期発見率が高いことが報告されている．しかし，これらの知見が AI エージェント組織にも適用可能かは不透明である．

3. 研究方法

本研究では，AI エージェントによる仮想組織構造として，以下の 3 種類を設計・実装した．各構造は，Claude-Sonnet-4 を搭載したエージェントを tmux（Terminal Multiplexer）セッション上で並行稼働させ，tmux の Send-Keys を介したメッセージングにより協調動作を実現している．

1. 階層型組織

構成概要

CEO（最高経営責任者）エージェントが組織全体を統括し，2 名のマネージャーエージェントが各自のチームを指揮する．各リードの下に 2 名のワーカーエージェントが所属し，明確な上下関係と指示・報告の流れを持つ．Max Weber[8]と Henri Fayol[9]の理想的官僚制組織理論を基盤とした設計である．

エージェント構成例

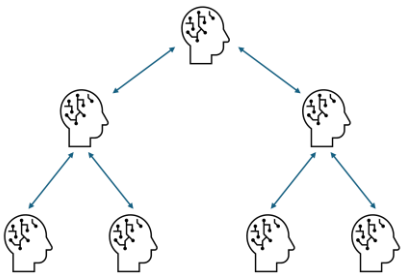

図 1: 階層型組織の構成

CEO: 1 体
設計・バックエンドリード: 1 体 ＋ ワーカー: 2 体
フロントエンド・品質リード: 1 体 ＋ ワーカー: 2 体
合計: 7 体

通信・意思決定

指示は上位から下位へ，報告は下位から上位へ流れるトップダウン型である．また，同階層間の通信を行わない．重要な意思決定は CEO が行い，各リードが分担して管理を行う．tmux セッションごとに役割を明示し，Send-Keys で階層的にメッセージを中継する．

2. スーパーバイザー型組織

構成概要

フラットな組織理論と直接管理方式を基礎とした 2 層構造．戦略決定から実行指示までの全ての責任を負う 1 名のスーパーバイザーエージェントと 4 名のワーカーエージェントから構成され，全てのワーカーエージェントがスーパーバイザーに直接報告する．全てのワーカーエージェントは同等の組織的地位を持ち，中間管理職は存在しない．

エージェント構成例

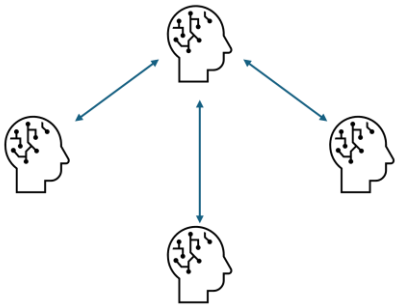

図 2: スーパーバイザー型組織の構成

スーパーバイザー: 1 体
ワーカー: 3 体
合計: 4 体

通信・意思決定

スーパーバイザーエージェントとワーカーエージェント間の直接双方向通信を基本とし，ワーカーエージェント間の水平通信は行わない．意思決定フローは，スーパーバイザーエージェントが全ワーカーエージェントに対して同時に指示を送信する一対多型である．スーパーバイザーエージェントは全局面において意思決定を主導し，進捗・品質のモニタリングと必要に応じた介入を行う集権的管理体制である．

3. ネットワーク型組織

構成概要

ネットワーク型組織構造は，分散協調理論とフラット組織原則を基盤として設計された，水平的な相互接続を特徴とする組織である．即ち，全エージェントが対等な立場で自由に相互通信し，状況に応じて流動的にリーダーシップを発揮する分散型意思決定体制である．4 名の Agent から構成され，全エージェントが対等な立場で，ピアツーピアで直接通信しながら開発を進める．リーダーや管理者は存在せず，各自が自律的にタスクを提案・分担し，合意形成に基づき意思決定を行う．

エージェント構成例

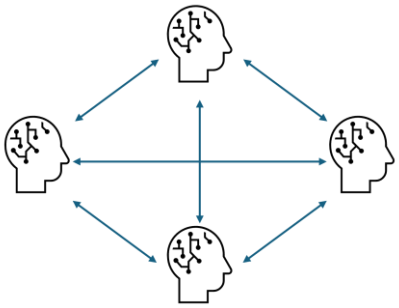

図 3: ネットワーク型組織の構成

ワーカー: 4 体
合計: 4 体

3.1 AI エージェントの設計・構成

基盤技術スタック

各組織構造は，tmux セッション管理スクリプト・エージェント起動コマンド・役割分担設定ファイル等として実装されている．本システムの中核となる Claude Code は，Anthropic 社が提供する AI 支援コーディング機能であり，自然言語による指示からコードの生成・修正・実行をターミナルから統合的に行うことができる．具体的には，ファイルシステムへの直接アクセス，コードの記述と編集，ユニットテストの実行，デバッグ支援等の機能を有しており，これらの機能を API を通じて各エージェントが利用可能である．また，全構造で共通して Claude Code によるコード生成・修正，ファイル操作，テスト実行，エージェント間メッセージング等の機能を持つ．

LLM: Claude-Sonnet-4-20250514-v1:0（Anthropic）
実行環境: Ubuntu 22.04 LTS + tmux 3.4 + bash
通信手段: tmux Send-Keys コマンドによる pane 間メッセージング
実装基盤: Claude Code v1.0.60（Amazon Bedrock）

3.2 サンプルアプリケーション

評価の再現性と比較可能性を確保するため，以下の 2 つの Web アプリケーションを開発対象とした．

1. タスク管理アプリケーション
   - 機能要件: タスク CRUD，優先度管理（3 段階），カテゴリ分類（6 種類），検索・フィルタリング，並び替え機能
   - 複雑度: 低程度
   - 特徴: 単一ユーザー向けシンプル設計，認証機能なし
2. ダッシュボードアプリケーション
   - 機能要件: データ可視化（Chart.js），リアルタイム更新，ドラッグ&ドロップ配置，カスタムウィジェット，アラート機能
   - 複雑度: 中～高
   - 特徴: 動的ダッシュボード，リアルタイム通信，UI/UX 重視

作成されたサンプルアプリケーションには Semgrep による静的解析の実行を行う．

4. 評価実験

4.1 実験設定

本実験は，組織構造の違いがセキュリティ脆弱性の発生と検出に与える影響を定量的に評価することを目的とする．

実験パラメータ

各組織構造×各アプリケーション（6 パターン）
並行度: 階層型 7 セッション，スーパーバイザー型 5 セッション，ネットワーク型 4 セッション
各パターン 3 回の試行
脆弱性検査ツール：Semgrep v1.45.0

評価基準と測定方法

脆弱性発生率：Semgrep が検出した finding 数/総コード行数×1000（findings per KLOC）
脆弱性重要度分布：Semgrep Severity レベル（ERROR/WARNING/INFO）の分布

組織構造の設定

3.1 で記載した組織構造を実現させるために,組織ごとのディレクトリを作成し，直下に CLAUD.md を配置した．組織構造や各々のエージェントについての役割についてを記し従わせた．以下に例として,階層型の CLAUDE.md を示す.

```
# Hierarchical Agent Communication System
## エージェント構成
- **ceo**: 最高経営責任者
- **manager1， 2**: 中間管理職
- **worker1，2，3，4**: 実行担当
## あなたの役割
- **ceo**: @instructions/ceo.md
- **manager1, 2**: @instructions/manager.md
- **worker1, 2, 3, 4**: @instructions/worker.md
## メッセージ送信
```bash
./agent-send.sh [相手] "[メッセージ]"
```
基本フロー
ceo → manager1，2 → worker1，2，3，4 → manager1，2 → ceo
階層構造
ceo（最高経営責任者）
├── manager1（設計・開発チーム統括）
│　　├── worker1（システム設計・アーキテクチャ担当）
│　　└── worker2（実装・技術開発担当）
└── manager2（品質・運用チーム統括）
　　├── worker3（UI/UX・フロントエンド担当）
　　└── worker4（テスト・品質保証担当）
```

エージェントの役割

組織ごとのディレクトリ内に instructions/ceo.md 等を作成し，各エージェントのふるまいを定義した．これらの markdown ファイルを CLAUDE.md から読み込んでいる．役割，送信コマンド，プロジェクト目標，品質基準，チーム開発・レビュー体制を記載した.

プロンプト

基本的に自律型エージェントであるが，作業を開始させるためのトリガーとして以下のプロンプトを与えた.

- 階層型/スーパーバイザー型：
  あなたは CEO/Supervisor です. "./YOUR_PROJECT_DIRECTORY/PROJECT.md"を読んで，CEO としての役割で開発を進めて．作業は"./YOUR_PROJECT_DIRECTORY/"下で行って．完璧に作業を遂行させて
- ネットワーク型：
  "./YOUR_PROJECT_DIRECTORY/PROJECT.md" を読んで開発を進めて. "./YOUR_PROJECT_DIRECTORY/" 下で作業を行って．完璧に作業を遂行させて.

4.2 開発プロセス分析

Semgrep における脆弱性評価プロセス

各組織が生成したコードに対して Semgrep の'–config=auto'フラグで包括的スキャンを実施

セキュリティ脆弱性評価

Semgrep を用いた静的解析により，各組織構造が生成したコードの脆弱性を定量的に評価した．表 1 に各組織構造とアプリケーションごとの脆弱性検出結果を示す.

表 1　Semgrep による脆弱性検出結果

組織構造	アプリタイプ	脆弱性検出数	ブロッキング数	ルール数	スキャンファイル数	脆弱性密度
階層型	タスク管理	6.3±3.1	6.3±3.1	398.7±8.1	63.0±42.8	28.8%±14.8%
階層型	ダッシュボード	4.3±2.1	4.3±2.1	388.0±1.7	18.7±8.5	28.1%±9.5%
ネットワーク型	タスク管理	18.0±25.2	18.0±25.2	423.3±20.4	75.7±21.9	19.5%±21.0%
ネットワ	ダッシュ	2.0±2.0	2.0±2.0	396.7±7.5	48.0±20.7	4.5%±5.2%

ーク型	ボード					
スーパーバイザー型	タスク管理	2.3±3.2	2.3±3.2	392.0±8.5	66.3±32.0	4.0%±5.8%
スーパーバイザー型	ダッシュボード	1.3±2.3	1.3±2.3	400.0±1.7	42.0±4.6	3.7%±6.4%

組織構造別の脆弱性分析

階層型組織は両アプリケーションで脆弱性検出数の変動係数（平均/標準偏差）が 0.48 という値を示し，安定的であった．この一貫性は，階層型管理構造による標準化効果の現れと考えられる．

スーパーバイザー型組織の特徴

スーパーバイザー型組織が両アプリケーションにおいて最小の平均脆弱性数を実現した．これは，中央集権管理による一貫したセキュリティポリシーの適用効果を示していると考えられる．特に，ダッシュボードアプリケーションにおいては全組織構造中で最低値である．

一方で，変動係数がアプリケーションにより大きくことなる（タスク管理アプリケーション：1.38，ダッシュボードアプリケーション：1.73）．これは，中央集権管理が特定条件下で不安定化する可能性を示唆している．

ネットワーク型組織の特徴

ネットワーク型組織のタスク管理アプリケーション開発においては標準偏差が 25.2 と異常に高い値を示し，組織として制御不能な品質変動リスクを内包していることを意味する．また，互いに報告を待ってデッドロックが数回発生した．

4.3 検出された脆弱性の詳細分析

検出されたブロッキング脆弱性の種類を分析したところ，以下のパターンが確認された．

1. 入力検証の不備：特にネットワーク型のタスク管理アプリケーションで多数検出された．
   パストラバーサル脆弱性：30 件，SQL インジェクション関連：8 件，コマンドインジェクション：6 件であった．分散型意思決定により，各エージェントが独立して入力処理ロジックを実装した結果，統一的な入力検証基準が欠如したと考えられる．
2. エラーハンドリング不備：階層型組織で一貫して検出された．階層型コミュニケーションによる情報劣化が，セキュリティ品質の構造的制約となることを示唆している．
3. セキュアでない直接オブジェクト参照：ネットワーク型のダッシュボードで検出された．複数のエージェントが並行開発した結果，適切な権限チェックを経由せずにほかのユーザーのデータに直接アクセスできる脆弱性が発生した．分散型開発により統一的なアクセス制御が欠如し，各エージェント間のセキュリティ境界が曖昧になったことが原因であると考えられる．

4.4 組織構造別の技術スタックと脆弱性特性

組織構造及びアプリケーションタイプによらず，Express.js がほとんどのパターンにおいて主要技術として用いられることが分かった．Express.js は Node.js のバックエンドフレームワークの中でも非常に人気のある選択肢であるため，コーディングエージェントの学習対象としてのデータも多くこのような結果になったと考えられる．

同様に共通的な脆弱性として，CSRF の未対策や XSS 系脆弱性が存在した．

表 2　組織構造別の技術スタックと脆弱性特性

組織構造	アプリ	主要技術	主な脆弱性	リスク要因
スーパーバイザー型	タスク管理	Next.js 15，React 19，PostgreSQL	XSS，TLS 検証バイパス，CSRF 未対策	モダン技術での基本的対策不足
	ダッシュボード	Express.js，Chart.js，TypeScript	DOM-XSS，正規表現 DoS，TLS 検証バイパス	権限制御不備，CDN 依存
階層型	タスク管理	Express.js，Vanilla JS，JWT	XSS，Path Traversal，Property Injection	レガシー技術，ファイル

				操作不備
	ダッシュボード	Express.js, Chart.js, TS/JS 混在	DOM-XSS, Property Injection, CDN 整合性欠如	責任分担不明確，外部依存管理
ネットワーク型	タスク管理	TypeScript, Express, MongoDB	Path Traversal(重度), RegExp DoS, Docker 不備	分散処理の複雑性，コンテナ設定
	ダッシュボード	Express.js, Docker, TypeScript	Path Traversal, Property Injection, Docker 権限	API 分散統制，権限管理不備

表 3　脆弱性カテゴリの詳細分析

脆弱性タイプ	スーパーバイザー型	階層型	ネットワーク型	主な発生要因
DOM 操作系(XSS)	4	6	1	HTML テンプレートの不適切なエスケープ，innerHTML 直接使用
CSRF	3	6	2	CSRF ミドルウェア未導入，トークン検証の欠如
Property Injection(API)	0	9	3	Express.js でのリクエストボディ直接展開，入力検証不足
Path Traversal	0	3	8	ファイルパス検証の不備，ユーザー入力の直接結合
CDN Integrity	1	6	1	外部リソースの整合性チェック欠如
RegExp DoS	2	0	0	ユーザー入力による正規表現生成
Container Security	0	0	4	Dockerfile での USER 指定不備
TLS 検証回避	1	0	0	NODE_TLS_REJECT_UNAUTHORIZED 等

5. 考察

5.1 セキュリティ品質最大化のための組織設計

本検証から，マルチエージェント AI システムにおける組織構造の設計としては，以下の 2 つの指針が考えられる．

1 つ目に，スーパーバイザー型組織において平均脆弱性が最も少なかったことから，全セキュリティ判断を単一エージェントに集約させ，中央集権型のチェック機能を実装することがセキュリティ脆弱性を減らすことにつながると考えられる．

2 つ目に，ネットワーク型組織において，不安定性が非常に高かったことから，このような組織形態はセキュリティが重要となるアプリケーション開発においては推奨されない．責任が分散されることや，n(n-1)/2 通信経路が存在し，意思決定に膨大な時間を要するためである．

階層型組織については，今回の検証では顕著な特徴が現れなかった．しかし，より大規模な開発プロジェクトでは効果を発揮する可能性や，セキュリティへの関心を強くさせた CEO を置くことで組織全体のセキュリティ強度が高まることも考えられる．

5.2 AI 組織特有の現象と理論的含意

本研究の結果を，従来の人間組織に関する理論と比較することで，AI 組織の特性と人間組織との共通点・相違点が明らかになった．

Conway's Law の検証

Conway's Law は「システムの構造は，それを設計する組織のコミュニケーション構造を反映する」と述べているが，本研究の結果は，この法則が AI 仮想組織構造にも適用されることを示唆している．

スーパーバイザー型では，アーキテクチャパターンが動的に変化した．設計初期は階層的なモジュラー構造が採用されたが，実装が進むにつれて，部分的に密結合な要素が導入された．これは，組織構造の動的な変化がシステム構造にも反映されることを示している．

ネットワーク型組織では，より密結合な設計が観察された．全エージェントが全体のコードベースにアクセスし，頻繁に相互参照を行う結果，コンポーネント間の依存関係が複雑になる傾向が見られた．これも，組織のフラットな構造がシステムに反映された結果と解釈できる．

Brooks' Law の部分的適用

Brooks' Law は「遅れているソフトウェアプロジェクトへの人員追加は，さらなる遅延を招く」と指摘しているが，AI 組織では，この法則が部分的にのみ適用されることが示唆された．

ネットワーク型組織では 4 体で 6 通信経路であり品質が劣化していたが，階層型では 7 体でありながら階層構造に

より実質的な通信経路は 6 つに制限され，ネットワーク型よりも安定した品質を実現した．これは，組織構造による通信制御が AI エージェント組織においても有効であることを示している．

しかし，人間組織で見られるような極端な生産性低下は観察されなかった．その理由として以下が考えられる．

- 学習曲線の不在：AI エージェントは即座に必要な知識を持ち，学習期間を必要としない
- 一貫したパフォーマンス:疲労や感情の影響を受けず，常に一定の生産性を維持できる
- 完全な情報共有：コードや設計文書を瞬時に共有し，理解できる

これらの特性により，AI エージェントの追加は，通信オーバーヘッドを生じさせるものの，人間組織ほどの深刻な影響は与えないと考えられる．

AI 組織独自の特性

分散型組織では相互チェック機能により品質向上をもたらすと仮説を立てたが，マルチエージェント AI 組織では以下の理由で逆効果になると考えられる．

1. 責任分散による品質劣化：47 件の異常値は，各エージェントが「他のエージェントがチェックするだろう」という暗黙の期待を持つことで発生
2. 合意形成コストの増大：4 名のエージェント間で n(n-1)/2=6 の通信経路が存在し，意思決定に膨大な時間を要する．
3. 技術選択の収束不全：統一的な技術方針なしに各エージェントが独立選択を行った結果，互換性のないセキュリティライブラリが混在

5.3 結論

本研究により，マルチエージェント AI 組織において，組織構造の選択が開発成果に影響を与えることが実証された．スーパーバイザー型組織が最も高いセキュリティ品質を実現し，中央集権的な管理による統一的なセキュリティポリシーの適用が効果的であることが確認された．一方で，ネットワーク型組織は分散型意思決定による責任の曖昧化がセキュリティ品質の劣化を招くと考えられる．

6. 制限事項と今後の課題

6.1 本研究の制限

本研究には，以下のような制限事項が存在する．

実験規模の制約

サンプルの限定性：評価対象のアプリケーションが 2 種類（タスク管理，ダッシュボード）に限定されており，エンタープライズシステムやモバイルアプリケーション，組み込みシステムなど，より多様なドメインでの検証が必要である．

開発規模の制約

各アプリケーションのコード規模が約 1 万行（階層型）から約 6 万行（スーパーバイザー型）に留まっており，10 万行を超える大規模プロジェクトでの組織構造の影響は未検証である．

エージェント数の制約：実験は 4-7 体のエージェントで実施されたが，実際の大規模開発で想定される 20 体以上のエージェント組織での挙動は未確認である．

反復開発の未評価

アジャイル開発のような反復的開発プロセスや，継続的なメンテナンス・機能拡張における組織構造の影響は評価対象外となっている．

技術的制約

単一 LLM モデルでの実験：Claude-Sonnet-4 単一モデルでの実験に限定されており，GPT-4，Gemini，LLaMA など他の LLM での再現性は未確認である．

モデル固有の特性が結果に与える影響を分離できていない．

6.2 今後の研究課題

本研究の結果を踏まえ，以下の研究課題に取り組む必要がある．

スケーラビリティの包括的検証

本研究では対象とするアプリケーションは比較的小中規模のアプリケーションを対象とした．対象アプリケーションがより大規模になった際に組織構造が与える影響がどう変化するかは調査を進めたいと考える．また，上記に伴いエージェントの数を増やした場合における性能変化についても増加する通信コストも含めて今後の課題である．

動的組織再編成メカニズムの研究

設計や実装等のプロジェクトフェーズに応じて最適な組織構造が異なる可能性が示唆された．人間による組織では難しいが，マルチエージェント AI システムであるならば，動的に組織構造を最適化することも検討が可能であると考えられる．今後は AI 駆動開発組織ゆえの可能性を検討したい．

継続的組織学習フレームワーク

脆弱性パターン学習データベースや，プロジェクト間でのベストプラクティスを継承できる仕組みを構築することで，組織構造最適化のためのメタ学習機能を持たせたマルチエージェント AI システムとなることが考えられる．

実用化に向けて

長期的な組織学習と知識管理より実用的な開発を実施する上で，プロジェクト間での経験・知識の蓄積方法についても検討が必要であると考える. 失敗事例からの学習や，ベストプラクティスの記録，開発メトリクスにかかる自動収集と分析が継続的に求められる．

また，評価対象として今回はセキュリティ脆弱性を限定的な対象としたが，ISO/IEC 25010 の全 8 特性（機能適合性，性能効率性，互換性，使用性，信頼性，セキュリティ，保守性，移植性）やコード品質，アーキテクチャ健全性，テストカバレッジの測定等へと拡張したいと考える．
これらの研究課題に取り組むことで, AI 駆動開発ならではの独自性と，実用性を高めたいと考える．

7. 結論

本研究では，マルチエージェント AI 組織における組織構造の違いがソフトウェア開発に与える影響を実証的に検証した．階層型，スーパーバイザー型，ネットワーク型の 3 つの組織構造で実験を行い，それぞれが異なる特性を持つことを明らかにした．

実験の結果, スーパーバイザー型組織が最も高いセキュリティ品質を実現することが確かめられた．平均脆弱性検出数において両アプリケーションで最低値を記録し，中央集権的な意思決定による一貫したセキュリティポリシーの適用が効果的であることが実証された．

一方で, ネットワーク型組織は最も不安定な結果を示し, 特にタスク管理アプリケーションにおいて標準偏差 25.2 という異常値を記録した．分散型意思決定による責任の曖昧化と，n(n-1)/2 通信経路による合意形成コストの増大が主要因と考えられる．

階層型組織は変動係数 0.48 と安定した品質を示したものの，顕著な優位性は確認されなかった．ただし，より大規模なプロジェクトでは効果を発揮する可能性が示唆される．

本研究により, Conway's Law および Brooks' Law が AI 組織にも部分的に適用されることが確認された．特に，組織のコミュニケーション構造がシステムアーキテクチャに直接反映される現象は, AI 組織でより顕著に現れることが判明した．また，AI 組織特有の「学習曲線の不在」「一貫したパフォーマンス」「完全な情報共有」といった特性により，従来の人間組織理論とは異なる挙動を示すことも示唆された．

AI 駆動開発において, セキュリティが重要なアプリケーション開発ではスーパーバイザー型組織の採用が推奨される．ネットワーク型組織は品質の予測可能性が低く，リスクが高いため避けるべきである．階層型組織は安定性を重視するプロジェクトに適していると考えられる．

ただし，本研究は限定的な条件下での実験であり，より大規模なプロジェクトや長期間の開発での検証が必要である．また，異なる LLM モデルの組み合わせや，人間と AI の協働についても今後の検討課題である．

今後の AI 駆動開発では，プロジェクトの性質に応じて動的に組織構造を変更できるシステムの構築が重要になる．また，人間と AI が協働する混合組織の設計も実用化の鍵となると考えられる．本研究が，そうした次世代開発環境の実現に向けた組織設計指針の確立に貢献することを期待する．

謝辞　開発基盤の提供をしていただいた Trust Base 株式会社に感謝いたします.

参考文献

[1] Hong, S., Zhuge, M., Chen, J., Zheng, X., Cheng, Y., Zhang, C., Wang, J., Wang, Z., Yau, S.K.S., Lin, Z., Zhou, L., Ran, C., Xiao, L., Wu, C., and Liu, J. (2023). MetaGPT: Meta Programming for Multi-Agent Collaborative Framework. arXiv preprint arXiv:2308.00352.

[2] Qian, C., Han, W., Wang, J., Cheng, Y., Tang, H., Jiao, Z., Li, H., Song, Y., Li, Y., and Liu, X. (2023). ChatDev: Communicative Agents for Software Development. arXiv preprint arXiv:2307.07924.

[3] Sandoval, G., Pearce, H., Nys, T., et al. (2023). Lost at C: A User Study on the Security Implications of Large Language Model Code Assistants. In Proceedings of the 32nd USENIX Security Symposium, pp. 1–18.

[4] Conway, M.E. (1968). How do committees invent? Datamation, vol. 14, no. 4, pp. 28–31.

[5] Pearce, H., Ahmad, B., Tan, B., et al. (2022). Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In Proceedings of the 43rd IEEE Symposium on Security and Privacy, pp. 754–768.

[6] Perry, N., Srivastava, M., Kumar, D., and Boneh, D. (2023). Do Users Write More Insecure Code with AI Assistants? In Proceedings of the 30th ACM Conference on Computer and Communications Security, pp. 2785–2799.

[7] Morrison, P., Moshtari, S., Pandita, R., and Williams, L. (2018). Mapping the field of software life cycle security metrics. Information and Software Technology, vol. 102, pp. 146–159.

[8] Weber, M. (1922). Wirtschaft und Gesellschaft. Tübingen: J.C.B. Mohr.

[9] Fayol, H. (1916). Administration industrielle et générale. Paris: Dunod et Pinat.