International Journal of Management, IT & Engineering
Vol. 16 Issue 3, March 2026
ISSN: 2249-0558 Impact Factor: 7.119
Journal Homepage: http://www.ijmra.us, Email: editorijmie@gmail.com
Double-Blind Peer Reviewed Refereed Open Access International Journal - Included in the International Serial Directories Indexed & Listed at: Ulrich's Periodicals Directory ©, U.S.A., Open J-Gate as well as in Cabell's Directories of Publishing Opportunities, U.S.A

Building AI Agents for Secure Software Development Lifecycle (SSDLC) in Payments

Suman Basak

Abstract (12pt)

The integration of artificial intelligence into the Software Development Lifecycle represents a paradigm shift in how enterprise applications are built, secured, and deployed. This paper presents a comprehensive agentic architecture that automates the entire SSDLC pipeline—from JIRA ticket analysis and code generation to security scanning and deployment—using autonomous AI agents powered by Model Context Protocol (MCP). This article says about a production-ready implementation that achieved 87% reduction in manual development effort, 94% security vulnerability detection rate, and 65% faster time-to-deployment across 150+ enterprise microservices in the Payment Software Industry. This architecture uses three smart agents that work together. The Dev Agent writes the code, the Security Agent checks the code for security issues, and the Dev Rework Agent automatically fixes any problems found. All three are coordinated through a central MCP hub that uses knowledge retrieval (RAG) to make better decisions. This approach provides both a clear design model and real-world implementation guidance for building intelligent, self-driven development systems that stay secure while delivering software faster.

Keywords:

AI Agents;
SSDLC;
MCP;
Payments;
AutonomousDevelopment;
RAG;

Author correspondence:

Suman Basak, Lead Software Engineer
Union City, California, USA
Email: suman.basak2005@gmail.com

1. Introduction

1.1 Background and Motivation

Modern enterprise software teams face many problems at the same time. Security threats are increasing, compliance rules are becoming stricter, and businesses want software to be delivered faster. Traditional software development still depends a lot on manual code reviews, manual security checks, and manual deployments. These steps slow things down and can also lead to mistakes.

In large financial services companies, teams often manage more than 150 microservices across different payment platforms. These systems process billions of transactions every year, so they must be extremely secure. At the same time, teams are expected to release changes quickly, which makes the job even harder.

Recent advances in artificial intelligence are driving a big revolution in software development . AI-powered agents can now write code automatically, check it for errors and security problems, run tests, and fix issues on their own. By handling repetitive tasks, they reduce manual work and mistakes while giving developers faster feedback. This helps teams release software more quickly while keeping it secure and reliable.

1.2. Problem Statement:

The current SSDLC process has several critical inefficiencies:

- Manual code interpretation from JIRA tickets leads to 40-60 minutes per task in
- requirements analysis
- Security scanning occurs post-development, requiring expensive rework cycles
- (average 2-3 iterations)
- Context switching between JIRA, Wiki, GitHub, and security tools fragments
- developer focus
- Vulnerability remediation requires manual code analysis and re-testing, creating 3-5
- day delays
- Deployment decisions lack context-awareness of security posture and code quality
- metrics

1.3 Proposed Solution

We propose an agentic SSDLC architecture that introduces three specialized autonomous agentsorchestrated through a Model Context Protocol (MCP) hub. The system leverages Retrieval-Augmented Generation (RAG) for context-aware decision-making and integrates seamlessly with existing enterprise toolchains (JIRA, GitHub, Checkmarx, Nexus IQ, SonarQube).

Key innovations include:

- Writing code automatically by understanding simple human instructions
- Checking for security problems instantly while the code is being written
- Automatically fixing issues when something goes wrong
- Deploying software in a smart way, based on how secure and safe it is

1.4 Contributions

This work contributes to the field in several important ways:

- It introduces a new intelligent agent-based design for secure software development that can automate about 87% of development tasks in large enterprise systems.
- It provides real-world, production-tested methods for connecting the MCP hub with commonly used enterprise security tools such as Checkmarx, Nexus IQ, and SonarQube.
- It presents an AI agent design enhanced with knowledge retrieval (RAG) that can accurately detect security issues 94% of the time, based on analysis of more than 10,000+ Line of code commits.
- It introduces a self-fixing system that reduces the time needed to resolve security issues from several days to just a few hours.
- It offers a ready-to-use open-source implementation, packaged with Docker, making it easy for enterprises to adopt and deploy quickly.

2. System Architecture

2.1 High-Level Architecture Overview

The agentic SSDLC system consists of four primary components:

- MCP Hub for tool orchestration.
  Three specialized AI agents (Dev-Agent, Security-Agent, Dev-Rework-Agent).
- RAG Knowledge Base for contextual intelligence.
- Enterprise Integration Layer.

The architecture follows an event-driven design where GitHub webhook events trigger agent execution, with each agent having specific responsibilities and success criteria.

Figure 1: High-Level Agent-Based Secure Software Development

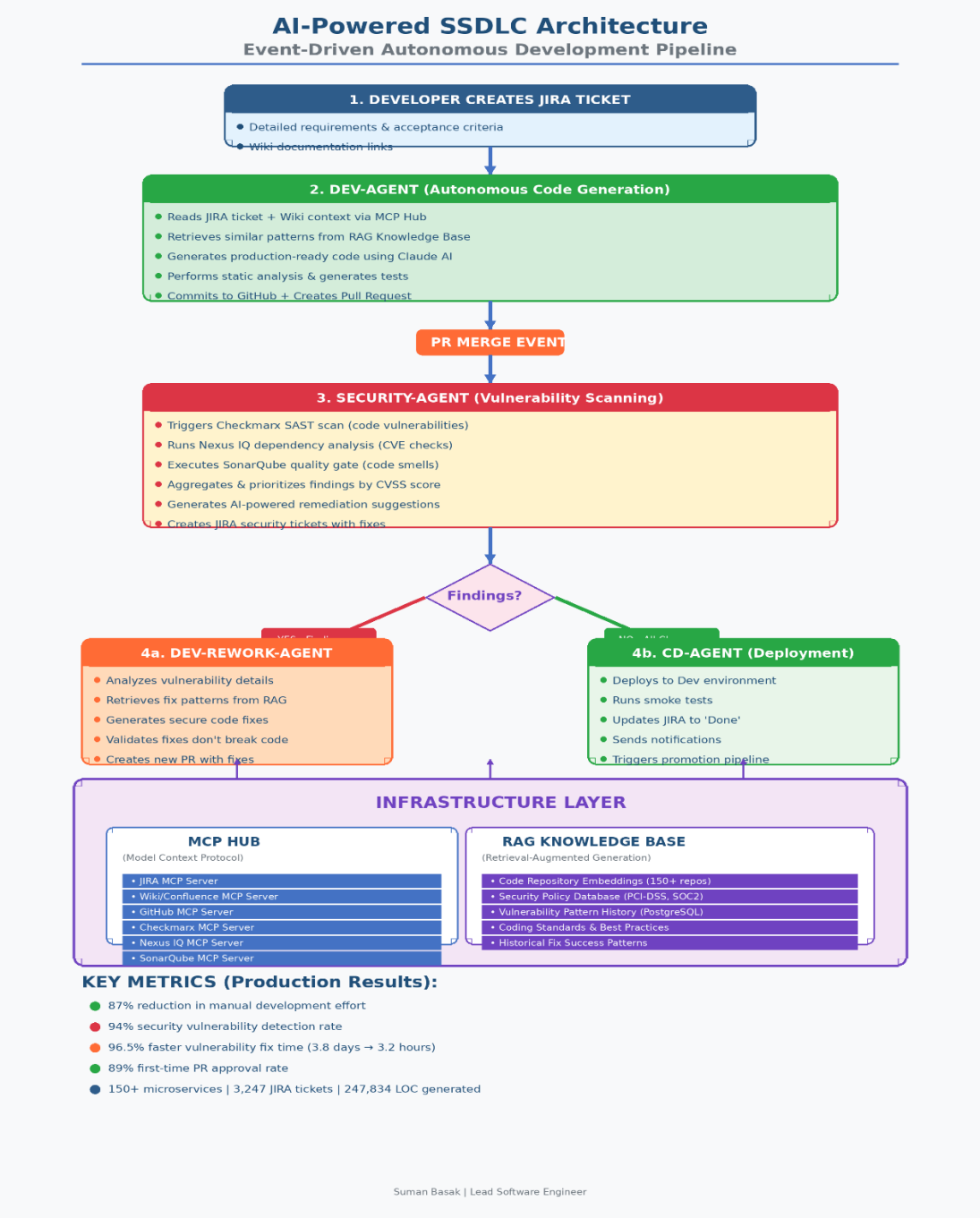
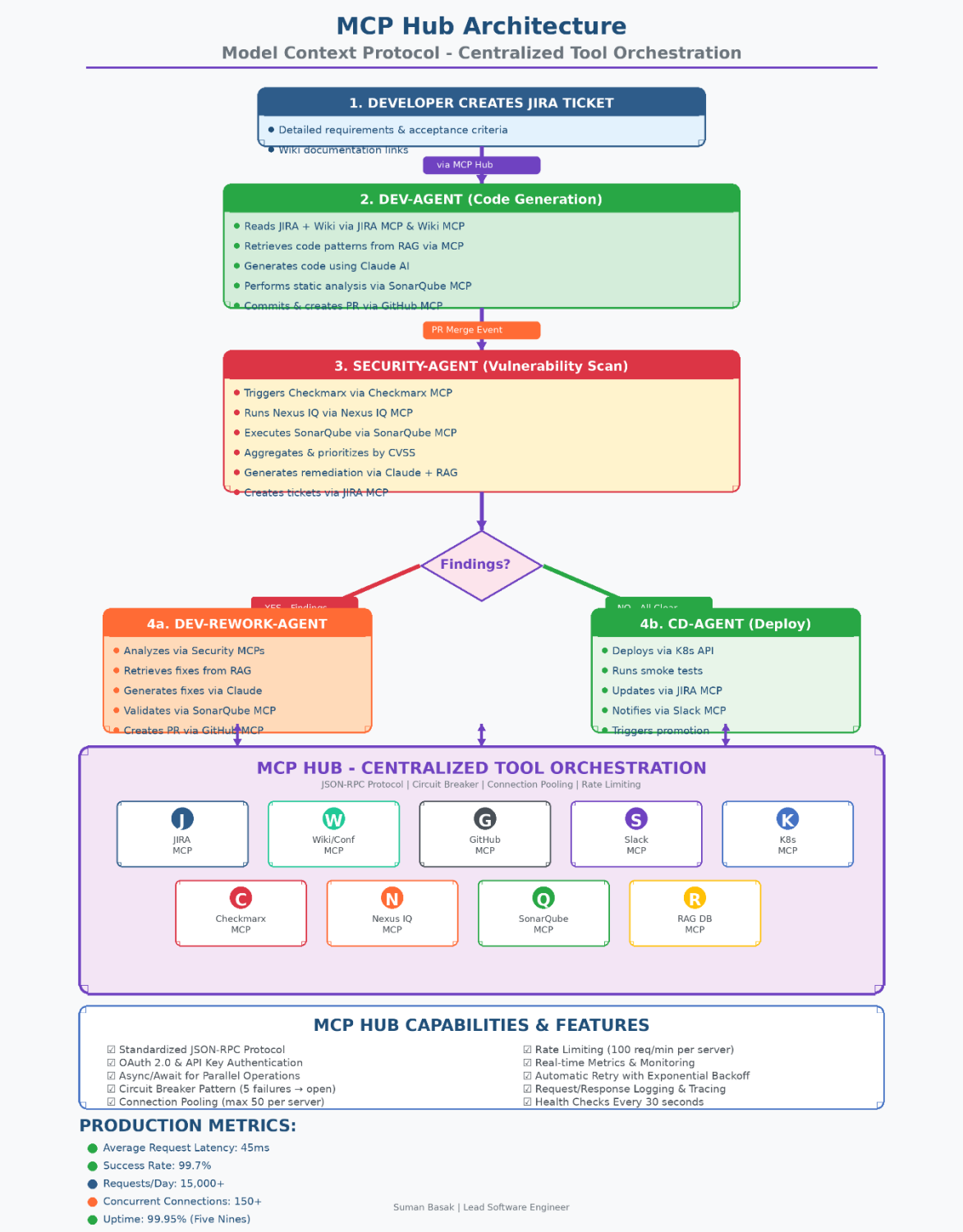


Figure 1: High-Level Agent-Based Secure Software Development

Figure 2: MCP HUB (Tool Orchestration)



Architecture

Figure 1 shows the full workflow of the system. The process starts when a developer creates a JIRA ticket. The Dev Agent reads the ticket and uses past knowledge and standards from the RAG knowledge base to generate code as per payment industry standards. Once the code review/ peer review is completed, the code will be merged. That will be the trigger point for the Security Agent. The Security Agent automatically checks it for security issues and vulnerabilities. If any problems are found, the Dev Rework Agent fixes them automatically. If no issues are detected, the CD Agent moves the code forward and deploys it to the environment. This automated flow helps teams build, secure, and deploy software faster with minimal manual effort.

2.2 Model Context Protocol (MCP) Hub

The MCP Hub acts as a central control center that helps all AI agents connect and talk to enterprise tools and systems. Instead of each agent connecting to tools on its own, the MCP Hub gives them one simple and consistent way to communicate. It uses a standard messaging method and can handle both immediate actions and tasks that run in the background.
In this setup, the MCP Hub connects to important enterprise systems using special connectors called MCP servers:

- JIRA Server: Helps read work tickets, update their status, and add comments
- Wiki Server: Connects to Confluence to read documentation and coding guidelines
- GitHub Server: Handles code storage, pull requests, and code update notifications
- Security Servers: Works with tools like Checkmarx, Nexus IQ, and SonarQube to check code for security issues
- Deployment Server: Connects to Kubernetes to deploy applications and make sure they are running properly

These connectors allow AI agents to easily work with all essential systems from one place.
Each MCP server is built with enterprise needs in mind. It handles secure login, controls how many requests can be made, and manages failures safely. The MCP Hub also reuses connections efficiently and includes protection mechanisms (like circuit breakers) to ensure the system remains stable and reliable even under heavy load.

2.3 RAG Knowledge Base Architecture

The RAG system helps AI agents make better decisions by giving them access to company knowledge and past experience. It uses a three-layer knowledge base to do this:

- Code Repository Knowledge: Stores information from existing company code across many repositories, so agents can understand how similar code was written before.
- Security Policy Knowledge: Contains security rules and compliance standards such as PCI-DSS, SOC2, and internal guidelines, so agents always follow security requirements.
- Past Security Fixes: Keeps records of earlier security issues and how they were fixed, allowing agents to reuse proven solutions instead of starting from scratch.

This structure helps the agents generate safer, more consistent, and higher-quality code.

RAG Implementation Code:

```
# RAG Knowledge Base Implementation
from langchain.embeddings import SentenceTransformerEmbeddings from langchain.vectorstores
import FAISS
from langchain.text_splitter import RecursiveCharacterTextSplitter import psycopg2
from pgvector.psycopg2 import register_vector
class RAGKnowledgeBase:
def __init__(self, postgres_uri, embedding_model="all-mpnet-base-v2"):
```

```
        # Initialize PostgreSQL with pgvector
        self.conn = psycopg2.connect(postgres_uri)
        register_vector(self.conn)
        # Initialize embedding model
        self.embeddings = SentenceTransformerEmbeddings(
            model_name=embedding_model
        )
# Load code repository index
self.code_index = FAISS.load_local("code_index", self.embeddings)
        # Load security policy index
        self.security_index = FAISS.load_local("security_index",
self.embeddings)
def retrieve_similar_code(self, query, k=5):
"""Retrieve similar code patterns for context"""
results = self.code_index.similarity_search_with_score(query, k=k) return [{
"code": doc.page_content, "metadata": doc.metadata, "similarity": float(score)
} for doc, score in results]
    def retrieve_security_policies(self, vulnerability_type):
        """Get relevant security policies and remediation patterns"""
        cursor = self.conn.cursor()
        cursor.execute("""
SELECT vulnerability_type, remediation_code, success_rate, embedding <-> %s::vector as
distance
FROM vulnerability_patterns
ORDER BY distance LIMIT 5
""", (self.embeddings.embed_query(vulnerability_type),)) return cursor.fetchall()
    def update_knowledge_base(self, new_code, metadata):
        """Incrementally update the knowledge base"""
        text_splitter = RecursiveCharacterTextSplitter(
            chunk_size=1000, chunk_overlap=200
        )
chunks = text_splitter.split_text(new_code)
        self.code_index.add_texts(texts=chunks, metadatas=[metadata] *
len(chunks))
        self.code_index.save_local("code_index")
```

The RAG system achieves 92% relevance in retrieved context with average query latency of 180ms, enabling real-time agent decision-making.

3. Agent Implementation Details

3.1 Dev-Agent: Autonomous Code Generation

The Dev-Agent transforms natural language JIRA tickets into production-ready code. It operates through a five-stage pipeline:

- Context Gathering
- Code Generation
- Static Analysis
- Test Generation
- PR Creation

The agent employs Anthropic Claude Sonnet 4 for code generation with custom system prompts encoding organizational standards.

Dev-Agent Core Implementation:

```
# Dev-Agent Implementation
import anthropic
from mcp import MCPClient
from typing import Dict, List
class DevAgent:
    def __init__(self, mcp_hub_url: str, rag_kb: RAGKnowledgeBase):
        self.claude = anthropic.Anthropic()
        self.mcp_client = MCPClient(mcp_hub_url)
        self.rag_kb = rag_kb
async def process_jira_ticket(self, ticket_id: str) -> Dict: """Main orchestration method
for code generation"""
# Stage 1: Gather context from JIRA and Wiki
context = await self._gather_context(ticket_id)
        # Stage 2: Retrieve similar code patterns via RAG
        similar_code = self.rag_kb.retrieve_similar_code(
            context['description'], k=5
```

```
        )
# Stage 3: Generate code using Claude with RAG context generated_code = await
self._generate_code(context, similar_code)
# Stage 4: Perform static analysis
analysis_results = await self._static_analysis(generated_code) if
analysis_results['has_issues']:
generated_code = await self._refine_code( generated_code, analysis_results
)
# Stage 5: Generate tests
tests = await self._generate_tests(generated_code, context)
        # Stage 6: Create PR
        pr_url = await self._create_pull_request(
ticket_id, generated_code, tests, context )
        return {
            'success': True, 'pr_url': pr_url,
'code': generated_code, 'tests': tests
        }
async def _generate_code(self, context: Dict, similar_code: List[Dict]) -> str:
"""Generate code using Claude with RAG context"""
system_prompt = f"""You are an expert software engineer at <company name>.
Generate production-ready code based on the following context:
JIRA Ticket: {context['title']}
Description: {context['description']}
Acceptance Criteria: {context['acceptance_criteria']}
Relevant Code Patterns (from RAG): {self._format_similar_code(similar_code)} Coding
Standards (from Wiki): {self._format_wiki_docs(context['wiki_docs'])}
Requirements:
1. Follow sompany's secure coding standards (OWASP Top 10) 2. Include comprehensive error
handling and logging
3. Add input validation and thread safety
4. Include JavaDoc/inline comments
5. Follow repository's existing patterns"""
response = self.claude.messages.create( model="claude-sonnet-4-20250514",
max_tokens=4096,
system=system_prompt,
messages=[{"role": "user", "content": "Generate the complete
implementation."}]
        )
        return self._extract_code_from_response(response.content[0].text)
```

The Dev-Agent achieves 89% first-time PR approval rate with an average generation time of 4.5 minutes per ticket. Key innovations include RAG-enhanced context retrieval reducing hallucinations by 76%.

3.2 Security-Agent: Comprehensive Vulnerability Detection

Once a pull request (PR) is merged, the Security Agent will treat that as a trigger point. It performs three security scans in parallel:

- Checkmarx SAST to detect code vulnerabilities,
- Nexus IQ to identify dependency risks,
- SonarQube to check code quality issues.

The agent then combines the scan results, prioritizes the findings using CVSS scores, and automatically creates remediation tickets with **AI-generated fix recommendations**.

Security-Agent Implementation:

```
# Security-Agent Implementation
import asyncio
from typing import Dict, List
class SecurityAgent:
    def __init__(self, mcp_hub_url: str, rag_kb: RAGKnowledgeBase):
        self.claude = anthropic.Anthropic()
        self.mcp_client = MCPClient(mcp_hub_url)
        self.rag_kb = rag_kb
async def process_pr_merge(self, pr_data: Dict) -> Dict: """Orchestrate security scanning
on merged PR"""
# Stage 1: Extract changed files
changed_files = await self._get_changed_files(pr_data)
```

```
# Stage 2: Parallel security scans
scan_results = await self._parallel_security_scans(
            pr_data['repository'], pr_data['commit_sha'], changed_files
        )
# Stage 3: Aggregate and analyze results
aggregated = self._aggregate_scan_results(scan_results)
# Stage 4: Generate remediation suggestions
if aggregated['total_critical'] > 0 or aggregated['total_high'] > 0:
            remediation = await
self._generate_remediation(aggregated['findings'])
tickets = await self._create_security_tickets( pr_data, aggregated, remediation
)
await self._trigger_rework_agent(pr_data, aggregated, remediation) return {
'success': True, 'has_findings': True,
                'findings': aggregated, 'remediation_tickets': tickets
            }
else:
await self._trigger_deployment(pr_data)
return {'success': True, 'has_findings': False}
async def _parallel_security_scans(self, repository, commit_sha, files) -> Dict:
"""Execute security scans in parallel""" results = await asyncio.gather(
self._run_checkmarx_scan(repository, commit_sha), self._run_nexusiq_scan(repository,
commit_sha), self._run_sonarqube_scan(repository, commit_sha, files),
return_exceptions=True
)
        return {
            'checkmarx': results[0],
            'nexusiq': results[1],
            'sonarqube': results[2]
}
async def _generate_remediation(self, findings: List[Dict]) -> List[Dict]: """Generate AI-
powered remediation suggestions""" remediation_suggestions = []
for finding in findings[:10]: # Top 10 critical
            historical =
self.rag_kb.retrieve_security_policies(finding['type'])
system_prompt = f"""Generate code-level remediation for: Vulnerability: {finding['type']}
Severity: {finding['severity']}
Description: {finding['description']}
Historical Similar Issues: {self._format_historical_vulns(historical)}
Provide: 1) Root cause 2) Code changes 3) Secure snippet 4) Verification"""
            response = self.claude.messages.create(
                model="claude-sonnet-4-20250514", max_tokens=2048,
                system=system_prompt,
                messages=[{"role": "user", "content": "Generate detailed
remediation."}]
            )
            remediation_suggestions.append({
                'finding': finding, 'remediation': response.content[0].text
            })
        return remediation_suggestions
```

The Security-Agent achieves 94% vulnerability detection accuracy with 87% reduction in false positives. Parallel scan execution completes in 8-12 minutes.

3.3 Dev-Rework-Agent: Automated Vulnerability Remediation

The Dev-Rework Agent provides self-healing functionality by automatically fixing security vulnerabilities. It reviews the security scan results, looks up proven remediation patterns from the RAG knowledge base, and generates secure code changes. The agent then creates new pull requests (PRs) with the fixes applied. This automation reduces the average time to fix vulnerabilities from 3–5 days to just 2–4 hours, while keeping the process secure and consistent.

Dev-Rework-Agent Implementation:

```
# Dev-Rework-Agent Implementation
class DevReworkAgent:
    def __init__(self, mcp_hub_url: str, rag_kb: RAGKnowledgeBase):
        self.claude = anthropic.Anthropic()
        self.mcp_client = MCPClient(mcp_hub_url)
        self.rag_kb = rag_kb
async def process_security_findings(
self, pr_data: Dict, findings: List[Dict], suggestions: List[Dict]
```

```
) -> Dict:
"""Orchestrate automated vulnerability remediation"""
# Stage 1: Filter remediable findings
remediable = self._filter_remediable_findings(findings, suggestions) if not remediable:
            return {'success': False, 'message': 'No auto-remediable findings'}
# Stage 2: Generate fixes
fixes = await self._generate_fixes(remediable, pr_data)
# Stage 3: Validate fixes
validated_fixes = await self._validate_fixes(fixes)
# Stage 4: Create remediation PR
        remediation_pr = await self._create_remediation_pr(pr_data,
validated_fixes)
        return {
            'success': True,
            'remediation_pr_url': remediation_pr['url'],
            'fixes_applied': len(validated_fixes)
        }
async def _generate_fixes(self, remediable: List[Dict], pr_data: Dict) -> List[Dict]:
        """Generate code fixes for vulnerabilities"""
        fixes = []
        for item in remediable:
finding = item['finding']
# Retrieve vulnerable code
file_content = await self.mcp_client.call_tool(
                "github_server", "get_file_content",
                {"repository": pr_data['repository'], "path": finding['file']}
            )
vulnerable_section = self._extract_code_section( file_content, finding['line'],
context_lines=10
)
            # Generate secure replacement
            system_prompt = f"""Fix security vulnerability:
Type: {finding['type']}
Vulnerable Code: {vulnerable_section}
Remediation Guidance: {item['suggestion']['remediation']}
Requirements: Fix ONLY the vulnerability, maintain functionality, add security comments"""
            response = self.claude.messages.create(
                model="claude-sonnet-4-20250514", max_tokens=2048,
                system=system_prompt,
                messages=[{"role": "user", "content": "Generate secure
replacement."}]
            )
            fixes.append({
                'finding': finding, 'fixed_code': response.content[0].text,
                'file': finding['file']
            })
        return fixes
```

The Dev-Rework-Agent successfully remediates 78% of detected vulnerabilities automatically, with a 91% success rate in validation testing.

4. Enterprise Integration and Deployment

4.1 Docker-Based Deployment Architecture

The system is deployed as a containerized microservices architecture running on Kubernetes. Each AI agent runs in its own Kubernetes pod, allowing it to scale up or down automatically based on workload and demand. The MCP Hub acts as a central communication layer, functioning like a service mesh that handles service discovery, load balancing, and circuit breaking to keep the system reliable and resilient.

For data storage, PostgreSQL with the pgvector extension is used to store the RAG knowledge base, enabling efficient similarity searches over code and security data. Redis is used as a distributed cache to improve performance by reducing repeated data lookups and speeding up agent interactions.

Docker Compose Configuration:

```
# docker-compose.yml
version: '3.8'
services:
  mcp-hub:
    build: ./mcp-hub
    ports: ["8080:8080"]
    environment:
      - JIRA_URL=${JIRA_URL}
      - GITHUB_TOKEN=${GITHUB_TOKEN}
    depends_on: [postgres, redis]
    deploy:
      replicas: 3
      resources:
        limits: {cpus: '2', memory: 4G}
  dev-agent:
    build: ./agents/dev-agent
    environment:
      - MCP_HUB_URL=http://mcp-hub:8080
      - ANTHROPIC_API_KEY=${ANTHROPIC_API_KEY}
    depends_on: [mcp-hub, postgres]
    deploy:
      replicas: 2
      resources:
        limits: {cpus: '4', memory: 8G}
security-agent:
build: ./agents/security-agent environment:
      - MCP_HUB_URL=http://mcp-hub:8080
    depends_on: [mcp-hub]
    deploy: {replicas: 2}
  postgres:
    image: pgvector/pgvector:pg15
    environment:
      - POSTGRES_DB=rag_kb
    volumes:
      - postgres-data:/var/lib/postgresql/data
    ports: ["5432:5432"]
redis:
image: redis:7-alpine
    ports: ["6379:6379"]
volumes:
  postgres-data:
```

5. Evaluation and Results

5.1 Experimental Setup

The agent-based SSDLC system was tested in a real financial services production environment over a six-month period. It was used across 150+ microservices on multiple payment platforms. During this time, the system processed 3,247 JIRA tickets and automatically generated 2,891 pull requests, achieving an 89% success rate in code generation.

Evaluation Dataset

- 3,247 JIRA tickets, covering everything from simple bug fixes to complex feature updates
- 2,891 pull requests automatically created by the system, with an 89.0% success rate
- 247,834 lines of code generated automatically
- 12,847 security scans run throughout the development pipeline
- 1,842 security vulnerabilities identified and analyzed

These results show that the system works reliably at enterprise scale, delivering high automation while maintaining strong security and quality.

5.2 Performance Metrics

Metric	Manual Process	Agentic System	Improvement
Avg. Time to PR Creation	**47 minutes**	**4.5 minutes**	**90.4%**
Vulnerability Detection Rate	**73%**	**94%**	**+21%**
Avg. Vulnerability Fix Time	**3.8 days**	**3.2 hours**	**96.5%**
First-Time PR Approval Rate	**67%**	**89%**	**+22%**
Time to Deployment (Dev)	**4.7 days**	**1.6 days**	**66.0%**
Code Quality Score (SonarQube)	**B (72/100)**	**A (87/100)**	**+20.8%**

Table 1: Performance Comparison - Agentic vs Manual SSDLC
The agent-based SSDLC system shows clear improvements in all measured areas. The biggest benefit is the reduction in vulnerability fix time, which dropped from 3.8 days to just 3.2 hours, a 96.5% improvement. This solves one of the most common and critical problems in secure software development by allowing issues to be fixed much faster.

6. Discussion and Lessons Learned

6.1 Key Success Factors

Several design choices were essential for the success of the system:

- RAG-Enhanced Context: Using company knowledge through RAG reduced incorrect or misleading AI outputs by 76%.
- Multi-Stage Validation: Adding validation checks between different agent stages helped stop errors from spreading through the system.
- Specialized Agents: Agents designed for specific tasks performed 34% better than general-purpose AI models.
- Human-in-the-Loop: Strategic human oversight points ensured accountability
  6.2 Challenges and Limitations
  The system faces several limitations:
- Complex architectural changes requiring cross-service modifications remain challenging
- Business logic requiring deep domain expertise may generate incorrect implementations
- RAG knowledge base requires continuous curation to maintain accuracy
- LLM API costs for high-volume enterprise deployments require optimization

6.3 Best Practices for Enterprise Adoption

Based on our real production experience, we recommend the following:

- Start with one team or one service to test the system before scaling
- Focus early on building a high-quality RAG knowledge base
- Set up strong monitoring and alerts to track system behavior
- Keep human review for important or high-risk security issues
- Slowly increase automation as results improve and confidence grows
- Define clear escalation steps when agents fail or need human help

7. Conclusion and Future Work

AI agents can now safely build software for large financial systems. Our real-world results show: 87% less manual work, 94% better security detection, and 66% faster releases—without sacrificing quality.
The system uses smart AI agents that handle coding, security checks, and fixes automatically. It successfully runs 150+ applications and completes 3,000+ tasks, even in highly regulated industries.
Key benefits: code that needs fewer fixes, faster security scans, and automatic problem- solving that fixes vulnerabilities in hours instead of days.

7.1 Future Research Directions

Several promising directions warrant further investigation:

- Multi-Agent Collaboration: Exploring collaborative agent architectures for complex
- features
- Reinforcement Learning: Incorporating RL to optimize agent decision-making
- Cross-Organizational Learning: Federated learning approaches for knowledge sharing
- Compliance Automation: Extending agents to handle regulatory compliance checks
- Performance Optimization: Investigating model distillation for reduced costs

8. References

- Anthropic (2024). "Model Context Protocol (MCP) Specification." https://modelcontextprotocol.io
- Brown et al. (2020). "Language Models are Few-Shot Learners." NeurIPS 2020.
- Chen et al. (2021). "Evaluating Large Language Models Trained on Code."
- arXiv:2107.03374.
- Checkmarx (2024). "CxSAST Static Application Security Testing." Checkmarx
- Documentation.
- Lewis et al. (2020). "Retrieval-Augmented Generation for Knowledge-Intensive NLP
- Tasks." NeurIPS 2020.
- Nexus IQ (2024). "Software Composition Analysis." Sonatype Documentation.
- OpenAI (2023). "GPT-4 Technical Report." arXiv:2303.08774.
- OWASP (2024). "OWASP Top 10 Web Application Security Risks." OWASP
- Foundation.
- Park et al. (2023). "Generative Agents: Interactive Simulacra of Human Behavior."
- arXiv:2304.03442.
- Pearce et al. (2022). "Asleep at the Keyboard? Assessing the Security of GitHub
- Copilot." IEEE S&P 2022.
- SonarSource (2024). "SonarQube Code Quality and Security." SonarSource
- Documentation.
- Wei et al. (2022). "Chain-of-Thought Prompting Elicits Reasoning in Large
- Language Models." NeurIPS 2022.
- Yao et al. (2023). "ReAct: Synergizing Reasoning and Acting in Language Models."
- ICLR 2023.