```
            "priority_issues": self._identify_priority_issues(
                sonar_analysis, coverage_analysis, complexity_analysis
            ),
            "risk_assessment": self._assess_risks(sonar_analysis),
            "technical_debt_score": self._calculate_tech_debt_score(
                sonar_analysis, coverage_analysis, complexity_analysis
            )
        }

        state["analysis_results"] = analysis_results
        state["messages"].append(
            AIMessage(content=f"Analysis complete. Found {len(analysis_results['priority_issues'])} priority
issues.")
        )

    except Exception as e:
        logger.error(f"Analysis failed: {str(e)}")
        state["error_count"] = state.get("error_count", 0) + 1
        state["messages"].append(
            AIMessage(content=f"Analysis failed: {str(e)}")
        )

    return state

async def _create_execution_plan(self, state: DevGenieAgentState) -> DevGenieAgentState:
    """Create autonomous execution plan based on analysis"""
    logger.info("Creating execution plan")

    analysis = state["analysis_results"]
    priority_issues = analysis["priority_issues"]

    # Use LLM to create intelligent execution plan
    plan_prompt = f"""
    Based on the codebase analysis, create an optimal execution plan:

    Priority Issues: {priority_issues}
    Technical Debt Score: {analysis['technical_debt_score']}
    Risk Level: {analysis['risk_assessment']['level']}

    Consider:
    1. Issue dependencies and ordering
    2. Risk vs benefit analysis
    3. Resource requirements
    4. Rollback strategies
```

```
    Create a step-by-step execution plan.
    """

    plan_response = await self.llm.ainvoke(plan_prompt)

    # Parse and structure the plan
    execution_plan = {
        "steps": self._parse_execution_steps(plan_response.content),
        "estimated_duration": self._estimate_duration(priority_issues),
        "risk_mitigation": self._create_risk_mitigation_plan(analysis),
        "rollback_plan": self._create_rollback_plan(priority_issues),
        "success_criteria": self._define_success_criteria(analysis)
    }

    state["fix_plan"] = execution_plan
    state["messages"].append(
        AIMessage(content=f"Created execution plan with {len(execution_plan['steps'])} steps")
    )

    return state

async def _execute_plan(self, state: DevGenieAgentState) -> DevGenieAgentState:
    """Execute the plan autonomously"""
    logger.info("Executing autonomous plan")

    plan = state["fix_plan"]
    execution_results = []

    for i, step in enumerate(plan["steps"]):
        try:
            logger.info(f"Executing step {i+1}: {step['description']}")

            # Execute step based on type
            if step["type"] == "fix_issues":
                result = await self._execute_fix_step(step)
            elif step["type"] == "improve_coverage":
                result = await self._execute_coverage_step(step)
            elif step["type"] == "refactor":
                result = await self._execute_refactor_step(step)
            elif step["type"] == "validate":
                result = await self._execute_validation_step(step)

            execution_results.append({
                "step": i + 1,
                "description": step["description"],
                "status": "success",
                "result": result,
```

```
                "duration": result.get("duration", 0)
            })

            # Check if we should continue based on success criteria
            if not self._meets_success_criteria(result, plan["success_criteria"]):
                logger.warning(f"Step {i+1} did not meet success criteria")
                break

        except Exception as e:
            logger.error(f"Step {i+1} failed: {str(e)}")
            execution_results.append({
                "step": i + 1,
                "description": step["description"],
                "status": "failed",
                "error": str(e)
            })

            # Decide whether to continue or abort
            if step.get("critical", False):
                logger.error("Critical step failed, aborting execution")
                break

    state["execution_status"] = "completed" if all(r["status"] == "success" for r in execution_results) else
"partial"
    state["execution_results"] = execution_results

    return state

async def _validate_results(self, state: DevGenieAgentState) -> DevGenieAgentState:
    """Validate execution results"""
    logger.info("Validating execution results")

    try:
        # Re-run analysis to check improvements
        post_execution_analysis = await self._run_tool("comprehensive_analysis", {})

        # Compare before and after
        original_analysis = state["analysis_results"]
        validation_results = self._compare_analysis_results(
            original_analysis, post_execution_analysis
        )

        state["validation_results"] = validation_results
        state["messages"].append(
            AIMessage(content=f"Validation complete. Improvement score:
{validation_results['improvement_score']}")
        )
```

```
        except Exception as e:
            logger.error(f"Validation failed: {str(e)}")
            state["error_count"] = state.get("error_count", 0) + 1

        return state

    def _should_retry(self, state: DevGenieAgentState) -> str:
        """Decide whether to retry, learn, or report"""
        error_count = state.get("error_count", 0)
        execution_status = state.get("execution_status", "")
        validation_results = state.get("validation_results", {})

        # Too many errors - learn and report
        if error_count >= 3:
            return "learn"

        # Execution failed - retry with adjusted plan
        if execution_status == "failed":
            return "retry"

        # Low improvement score - retry once
        improvement_score = validation_results.get("improvement_score", 0)
        if improvement_score < 0.3 and state.get("retry_count", 0) < 1:
            state["retry_count"] = state.get("retry_count", 0) + 1
            return "retry"

        # Otherwise, learn and report
        return "learn"

    async def _learn_from_outcome(self, state: DevGenieAgentState) -> DevGenieAgentState:
        """Learn from the execution outcome"""
        logger.info("Learning from execution outcome")

        # Analyze what worked and what didn't
        execution_results = state.get("execution_results", [])
        validation_results = state.get("validation_results", {})

        learning_data = {
            "successful_patterns": self._identify_successful_patterns(execution_results),
            "failure_patterns": self._identify_failure_patterns(execution_results),
            "improvement_areas": self._identify_improvement_areas(validation_results),
            "strategy_effectiveness": self._assess_strategy_effectiveness(state)
        }

        # Store learning for future executions
        await self._store_learning(learning_data)
```

```
    state["learning_results"] = learning_data
    state["messages"].append(
        AIMessage(content="Learning complete. Knowledge base updated for future executions.")
    )

    return state

async def _generate_report(self, state: DevGenieAgentState) -> DevGenieAgentState:
    """Generate comprehensive execution report"""
    logger.info("Generating execution report")

    report = {
        "execution_summary": {
            "status": state.get("execution_status", "unknown"),
            "steps_completed": len([r for r in state.get("execution_results", []) if r["status"] == "success"]),
            "total_steps": len(state.get("execution_results", [])),
            "total_duration": sum(r.get("duration", 0) for r in state.get("execution_results", []))
        },
        "improvements": state.get("validation_results", {}),
        "issues_resolved": self._count_resolved_issues(state),
        "technical_debt_reduction": self._calculate_debt_reduction(state),
        "recommendations": self._generate_recommendations(state),
        "learning_outcomes": state.get("learning_results", {}),
        "next_actions": self._suggest_next_actions(state)
    }

    state["final_report"] = report
    state["messages"].append(
        AIMessage(content=f"Execution complete. {report['issues_resolved']} issues resolved.")
    )

    return state

async def _run_tool(self, tool_name: str, params: dict) -> dict:
    """Run a tool and return results"""
    if tool_name not in self.tools:
        raise ValueError(f"Tool {tool_name} not available")

    tool = self.tools[tool_name]
    return await tool.arun(**params)

def _identify_priority_issues(self, sonar_analysis: dict, coverage_analysis: dict, complexity_analysis: dict) ->
List[dict]:
    """Identify priority issues across all analyses"""
    priority_issues = []
```

```
    # High-severity SonarQube issues
    if sonar_analysis and "issues" in sonar_analysis:
        high_severity = ["BLOCKER", "CRITICAL"]
        for issue in sonar_analysis["issues"].get("by_severity", {}):
            if issue.get("severity") in high_severity:
                priority_issues.append({
                    "type": "sonar",
                    "severity": issue["severity"],
                    "description": issue["message"],
                    "component": issue["component"],
                    "effort": issue.get("effort", "unknown")
                })

    # Low coverage areas
    if coverage_analysis and "uncovered_areas" in coverage_analysis:
        for area in coverage_analysis["uncovered_areas"]:
            if area.get("coverage_percentage", 100) < 50:  # Less than 50% coverage
                priority_issues.append({
                    "type": "coverage",
                    "severity": "HIGH",
                    "description": f"Low coverage in {area['fileName']}",
                    "component": area["filePath"],
                    "coverage": area["coverage_percentage"]
                })

    # High complexity methods
    if complexity_analysis and "complex_methods" in complexity_analysis:
        for method in complexity_analysis["complex_methods"]:
            if method.get("complexity", 0) > 15:  # Complexity > 15
                priority_issues.append({
                    "type": "complexity",
                    "severity": "MEDIUM",
                    "description": f"High complexity method: {method['methodName']}",
                    "component": method["className"],
                    "complexity": method["complexity"]
                })

    # Sort by severity and impact
    priority_issues.sort(key=lambda x: self._calculate_issue_priority(x), reverse=True)

    return priority_issues[:20]  # Top 20 priority issues

def _calculate_issue_priority(self, issue: dict) -> int:
    """Calculate priority score for an issue"""
    severity_scores = {"BLOCKER": 100, "CRITICAL": 80, "HIGH": 60, "MAJOR": 40, "MEDIUM": 20, "MINOR":
10}
    type_scores = {"sonar": 10, "coverage": 8, "complexity": 6}
```

```
        severity_score = severity_scores.get(issue.get("severity", "MEDIUM"), 20)
        type_score = type_scores.get(issue.get("type", "other"), 5)

        return severity_score + type_score
```

Agent Tools Implementation

```
class DevGenieMCPTool:
"""Base class for DevGenie MCP tools"""

    def __init__(self, mcp_client, server_name: str):
        self.mcp_client = mcp_client
        self.server_name = server_name

    async def arun(self, **kwargs) -> dict:
        """Async run method for LangGraph compatibility"""
        return await self.mcp_client.call_tool(self.server_name, self.tool_name, kwargs)

class SonarAnalysisTool(DevGenieMCPTool):
name = "sonar_analysis"
description = "Analyze code quality using SonarQube"
tool_name = "comprehensive_analysis"

    async def arun(self, project_path: str = None, **kwargs) -> dict:
        return await super().arun(project_path=project_path, **kwargs)

class CoverageAnalysisTool(DevGenieMCPTool):
name = "coverage_analysis"
description = "Analyze code coverage"
tool_name = "analyze_coverage"

    async def arun(self, project_path: str = None, **kwargs) -> dict:
        return await super().arun(project_path=project_path, **kwargs)

class ComplexityAnalysisTool(DevGenieMCPTool):
name = "complexity_analysis"
description = "Analyze code complexity"
tool_name = "analyze_complexity"

    async def arun(self, project_path: str = None, **kwargs) -> dict:
        return await super().arun(project_path=project_path, **kwargs)
```

class FixIssuesTool(DevGenieMCPTool):
name = "fix_issues"
description = "Fix code issues using AI"
tool_name = "fix_sonar_issues"

```
    async def arun(self, class_content: str, issues: List[dict], **kwargs) -> dict:
        return await super().arun(class_content=class_content, issues=issues, **kwargs)
```

Usage Example

async def main():
# Initialize LLM
llm = ChatOpenAI(model="gpt-4", temperature=0.1)

```
    # Initialize MCP clients (pseudo-code)
    sonar_client = MCPClient("sonar-server")
    ai_client = MCPClient("ai-fixer")
    coverage_client = MCPClient("coverage-server")

    # Create tools
    tools = [
        SonarAnalysisTool(sonar_client, "sonar-server"),
        CoverageAnalysisTool(coverage_client, "coverage-server"),
        ComplexityAnalysisTool(sonar_client, "sonar-server"),  # Assuming complexity is part of sonar
        FixIssuesTool(ai_client, "ai-fixer")
    ]

    # Create agent
    agent = DevGenieAutonomousAgent(llm, tools)

    # Run autonomous operation
    initial_state = {
        "messages": [HumanMessage(content="Analyze and improve the codebase")],
        "current_task": "autonomous_improvement",
        "analysis_results": {},
        "execution_status": "starting",
        "error_count": 0
    }

    config = {"configurable": {"thread_id": "autonomous-session-1"}}

    # Execute the autonomous workflow
    async for state in agent.app.astream(initial_state, config):
        print(f"Current state: {state}")

    # Get final result
    final_state = await agent.app.ainvoke(initial_state, config)
    print(f"Final report: {final_state['final_report']}")

if __name__ == "__main__": asyncio.run(main())
```

Risk Analysis & Mitigation

Technical Risks

1. **AI Model Reliability**
- **Risk**: AI generates incorrect or unsafe code fixes
- **Mitigation**:
  - Multi-layer validation (syntax, compilation, testing)
  - Human review gates for critical changes
  - Rollback mechanisms
  - Confidence scoring for AI outputs

2. **Scalability Bottlenecks**
- **Risk**: System cannot handle enterprise-scale loads
- **Mitigation**:
  - Horizontal scaling with Kubernetes
  - Auto-scaling based on metrics
  - Circuit breakers and rate limiting
  - Caching strategies (Redis, CDN)

3. **Data Security & Privacy**
- **Risk**: Code exposure to AI models, data breaches
- **Mitigation**:
  - On-premises deployment options
  - End-to-end encryption
  - Access controls and audit logging
  - Data anonymization techniques

4. **Integration Failures**
- **Risk**: External system dependencies (SonarQube, GitHub, AI APIs)
- **Mitigation**:
  - Circuit breaker patterns
  - Graceful degradation
  - Health checks and monitoring
  - Backup/fallback strategies

Operational Risks

1. **Team Adoption**
- **Risk**: Developers resist or misuse the tool
- **Mitigation**:
  - Gradual rollout with training
  - Clear documentation and onboarding

  - Success metrics and incentives
  - Feedback collection and iteration

2. **Over-dependence**
- **Risk**: Teams become too reliant on automated fixes
- **Mitigation**:
  - Education on tool limitations
  - Encourage code review practices
  - Maintain manual fix capabilities
  - Regular tool-free exercises

Final Recommendations & Roadmap

Immediate Action Plan (Next 3 Months)

Week 1-4: Foundation
1. ✅ **Start with Enhanced Non-MCP Approach**
   - Fastest path to value
   - Lowest risk
   - Builds on existing Java expertise

2. ✅ **Focus on Chat Interface First**
   - High user impact
   - Differentiates from existing tools
   - Proves AI integration value

3. ✅ **Implement Basic VSCode Extension**
   - Developer workflow integration
   - Quick wins with code selection fixes
   - Market validation

Week 5-12: Enhancement
1. **Add Coverage Analysis**
   - Expand beyond SonarQube
   - More comprehensive tech debt view
   - Higher business value

2. **Implement Enterprise Features**
   - Multi-tenancy
   - Authentication/authorization
   - Audit logging
   - Performance monitoring

3. **Production Hardening**
   - Error handling

- Monitoring & alerting
- Documentation
- Security reviews

Medium-term Evolution (6-12 Months)

MCP Migration Strategy
1. **Extract AI Service** to MCP Server first
2. **Build MCP Client Layer** in Spring Boot
3. **Gradual Migration** of other services
4. **Maintain Backward Compatibility**

Agentic AI Integration
1. **Start with Simple Automation** (scheduled scans, basic fixes)
2. **Add Planning Capabilities** (multi-step workflows)
3. **Implement Learning** (pattern recognition, adaptation)
4. **Full Autonomy** (proactive technical debt management)

Long-term Vision (12+ Months)

Ecosystem Integration
- **IDE Plugins**: IntelliJ, Eclipse, VS Code
- **CI/CD Integration**: Jenkins, GitHub Actions, GitLab
- **Monitoring Tools**: Grafana, DataDog, New Relic
- **Communication**: Slack, Teams, Email

Advanced AI Capabilities
- **Predictive Analysis**: Identify issues before they occur
- **Architecture Recommendations**: Suggest structural improvements
- **Performance Optimization**: Automated performance tuning
- **Security Hardening**: Proactive security improvements

Success Metrics

Technical Metrics
- **Issue Resolution Rate**: 80%+ automatic fix success
- **Coverage Improvement**: Average 20% increase per project
- **Time to Fix**: 90% reduction in manual effort
- **Code Quality Score**: Consistent improvement trends

Business Metrics
- **Developer Productivity**: 25% increase in feature velocity
- **Technical Debt Reduction**: 50% reduction in debt accumulation
- **Cost Savings**: ROI of 300%+ within first year
- **Developer Satisfaction**: 85%+ positive feedback

Technology Stack Summary

✅ Java-Centric Components (90% of codebase)
- Spring Boot backend and services
- Spring AI for LLM integration
- JaCoCo for coverage analysis
- JavaParser for code manipulation
- JUnit/Mockito for testing

⚠️ Non-Java Dependencies (10% of codebase)
- **VSCode Extension**: TypeScript (essential for IDE integration)
- **React Chat Interface**: TypeScript/JavaScript (modern UI requirement)
- **MCP Servers**: Python (ecosystem standard, optional for phase 1)
- **Infrastructure**: Docker, Kubernetes (deployment standard)

Final Recommendation

Start with the Enhanced Non-MCP approach to get immediate value and market validation, then progressively migrate to MCP as the ecosystem matures and your needs evolve. This strategy provides:

1. **Fast Time to Market**: 8-12 weeks to working product
2. **Low Risk**: Building on proven Java technologies
3. **High Value**: Immediate productivity improvements
4. **Future Flexibility**: Clear migration path to MCP
5. **Team Growth**: Gradual learning curve for new technologies

The DevGenie vision is ambitious but achievable with this phased approach. Focus on proving value early, then expand capabilities systematically.

Document Version: 1.0
Last Updated: Current Date
Total Pages: 45+
Target Audience: Enterprise Development Teams, Technical Architects, Engineering Managers

This comprehensive guide provides the complete roadmap for transforming DevGenie from a reactive issue-fixing tool into an autonomous technical debt management platform, with detailed implementation plans, architecture decisions, and production-ready code examples.   "project_key": {"type": "string"},
                    "time_period": {"type": "string", "enum": ["1w", "1m", "3m", "6m"], "default": "1m"}
                }
            }
)
    ]

    async def _call_tool(self, name: str, arguments: dict):
        with REQUEST_DURATION.time():
            try:

```
            if name == "comprehensive_analysis":
                result = await self._comprehensive_analysis(arguments)
            elif name == "get_quality_gate_status":
                result = await self._get_quality_gate_status(arguments)
            elif name == "get_technical_debt":
                result = await self._get_technical_debt(arguments)
            elif name == "monitor_quality_trends":
                result = await self._monitor_quality_trends(arguments)
            else:
                raise ValueError(f"Unknown tool: {name}")

            REQUEST_COUNT.labels(method=name, status='success').inc()
            return [TextContent(type="text", text=json.dumps(result, indent=2))]

        except Exception as e:
            logger.error(f"Error in {name}: {str(e)}")
            REQUEST_COUNT.labels(method=name, status='error').inc()
            return [TextContent(type="text", text=json.dumps({"error": str(e)}))]

async def _comprehensive_analysis(self, args: dict) -> dict:
    """Perform comprehensive analysis including issues, metrics, and history"""
    project_path = args["project_path"]
    include_history = args.get("include_history", False)
    severity_filter = args.get("severity_filter", ["BLOCKER", "CRITICAL", "MAJOR"])

    # Extract project key from path or use provided key
    project_key = self._extract_project_key(project_path)

    # Parallel execution of multiple API calls
    tasks = [
        self._get_issues(project_key, severity_filter),
        self._get_measures(project_key),
        self._get_quality_gate_status_internal(project_key)
    ]

    if include_history:
        tasks.append(self._get_project_history(project_key))

    results = await asyncio.gather(*tasks, return_exceptions=True)

    # Process results
    issues = results[0] if not isinstance(results[0], Exception) else []
    measures = results[1] if not isinstance(results[1], Exception) else {}
    quality_gate = results[2] if not isinstance(results[2], Exception) else {}
    history = results[3] if len(results) > 3 and not isinstance(results[3], Exception) else {}

    return {
```

```
        "project_key": project_key,
        "analysis_timestamp": datetime.now().isoformat(),
        "issues": self._process_issues(issues),
        "metrics": self._process_measures(measures),
        "quality_gate": quality_gate,
        "history": history if include_history else None,
        "summary": self._generate_summary(issues, measures, quality_gate)
    }

async def _get_issues(self, project_key: str, severities: List[str]) -> List[dict]:
    """Get issues from SonarQube API with pagination"""
    all_issues = []
    page = 1
    page_size = 500

    while True:
        params = {
            "componentKeys": project_key,
            "severities": ",".join(severities),
            "resolved": "false",
            "ps": page_size,
            "p": page
        }

        response = await self._make_api_request("api/issues/search", params)

        if not response or "issues" not in response:
            break

        issues = response["issues"]
        all_issues.extend(issues)

        # Check if we've got all issues
        total = response.get("total", 0)
        if len(all_issues) >= total or len(issues) < page_size:
            break

        page += 1

    return all_issues

async def _get_measures(self, project_key: str) -> dict:
    """Get project measures/metrics"""
    metrics = [
        "ncloc", "coverage", "duplicated_lines_density",
        "code_smells", "bugs", "vulnerabilities",
        "sqale_index", "reliability_rating", "security_rating"
```

```
    ]

    params = {
        "component": project_key,
        "metricKeys": ",".join(metrics)
    }

    response = await self._make_api_request("api/measures/component", params)

    if response and "component" in response:
        return response["component"].get("measures", [])

    return []

async def _make_api_request(self, endpoint: str, params: dict = None) -> Optional[dict]:
    """Make authenticated API request to SonarQube"""
    if not self.session:
        await self._init_session()

    url = f"{self.config.url}/{endpoint}"
    headers = {"Authorization": f"Bearer {self.config.token}"}

    for attempt in range(self.config.max_retries):
        try:
            async with self.session.get(
                url,
                params=params,
                headers=headers,
                timeout=aiohttp.ClientTimeout(total=self.config.timeout)
            ) as response:
                if response.status == 200:
                    return await response.json()
                elif response.status == 401:
                    raise Exception("Authentication failed - check SonarQube token")
                elif response.status == 404:
                    raise Exception(f"Resource not found: {endpoint}")
                else:
                    logger.warning(f"API request failed with status {response.status}")

        except asyncio.TimeoutError:
            logger.warning(f"Request timeout on attempt {attempt + 1}")
        except Exception as e:
            logger.error(f"Request failed on attempt {attempt + 1}: {str(e)}")

        if attempt < self.config.max_retries - 1:
            await asyncio.sleep(2 ** attempt)  # Exponential backoff
```

```
    return None

async def _init_session(self):
    """Initialize HTTP session"""
    connector = aiohttp.TCPConnector(limit=100, limit_per_host=30)
    self.session = aiohttp.ClientSession(connector=connector)

def _process_issues(self, issues: List[dict]) -> dict:
    """Process and categorize issues"""
    processed = {
        "total_count": len(issues),
        "by_severity": {},
        "by_type": {},
        "by_file": {},
        "by_rule": {},
        "hotspots": []
    }

    for issue in issues:
        severity = issue.get("severity", "UNKNOWN")
        issue_type = issue.get("type", "UNKNOWN")
        component = issue.get("component", "UNKNOWN")
        rule = issue.get("rule", "UNKNOWN")

        # Group by severity
        if severity not in processed["by_severity"]:
            processed["by_severity"][severity] = []
        processed["by_severity"][severity].append(issue)

        # Group by type
        if issue_type not in processed["by_type"]:
            processed["by_type"][issue_type] = []
        processed["by_type"][issue_type].append(issue)

        # Group by file
        if component not in processed["by_file"]:
            processed["by_file"][component] = []
        processed["by_file"][component].append(issue)

        # Group by rule
        if rule not in processed["by_rule"]:
            processed["by_rule"][rule] = 0
        processed["by_rule"][rule] += 1

        # Identify security hotspots
        if issue_type == "SECURITY_HOTSPOT":
            processed["hotspots"].append(issue)
```

```
        return processed

    def _extract_project_key(self, project_path: str) -> str:
        """Extract or determine project key from path"""
        # This could be enhanced to read from pom.xml, build.gradle, etc.
        return os.path.basename(project_path.rstrip('/'))

if __name__ == "__main__":
    config = SonarConfig(
        url=os.getenv("SONAR_URL", "http://localhost:9000"),
        token=os.getenv("SONAR_TOKEN", ""),
        timeout=int(os.getenv("SONAR_TIMEOUT", "30")),
        max_retries=int(os.getenv("SONAR_MAX_RETRIES", "3"))
    )

    server = SonarMcpServer(config)

    # Start the server
    import uvicorn
    uvicorn.run(server.app, host="0.0.0.0", port=8000)
```

AI Code Fix MCP Server (Python) ⚠️

python

```python
# ai_fix_mcp_server.py - Enterprise version
import asyncio
import json
import os
from typing import Dict, List, Optional, Any
from dataclasses import dataclass
from datetime import datetime
import logging

from mcp.server import Server
from mcp.types import Tool, TextContent
import vertexai
from vertexai.generative_models import GenerativeModel, SafetySetting, HarmCategory
from google.cloud import aiplatform
import redis
from prometheus_client import Counter, Histogram

# Metrics
FIX_REQUESTS = Counter('ai_fix_requests_total', 'Total fix requests', ['status', 'complexity'])
FIX_DURATION = Histogram('ai_fix_duration_seconds', 'Fix processing duration')
TOKEN_USAGE = Counter('ai_tokens_used_total', 'Total tokens used', ['type'])

logger = logging.getLogger(__name__)

@dataclass
class AIConfig:
    project_id: str
    location: str
    model_name: str = "gemini-1.5-pro-001"
    max_tokens: int = 8192
    temperature: float = 0.1
    cache_enabled: bool = True
    cache_ttl: int = 3600  # 1 hour

class AICodeFixServer:
    def __init__(self, config: AIConfig):
        self.config = config
        self.app = Server("ai-code-fixer", version="1.0.0")

        # Initialize AI
        vertexai.init(project=config.project_id, location=config.location)
        self.model = GenerativeModel(config.model_name)

        # Initialize cache
        if config.cache_enabled:
            self.redis_client = redis.Redis(
```

```
            host=os.getenv('REDIS_HOST', 'localhost'),
            port=int(os.getenv('REDIS_PORT', 6379)),
            decode_responses=True
        )

        # Safety settings
        self.safety_settings = [
            SafetySetting(
                category=HarmCategory.HARM_CATEGORY_HATE_SPEECH,
                threshold=SafetySetting.HarmBlockThreshold.BLOCK_MEDIUM_AND_ABOVE
            ),
            SafetySetting(
                category=HarmCategory.HARM_CATEGORY_DANGEROUS_CONTENT,
                threshold=SafetySetting.HarmBlockThreshold.BLOCK_MEDIUM_AND_ABOVE
            )
        ]

        self._register_tools()

    def _register_tools(self):
        self.app.list_tools()(self._list_tools)
        self.app.call_tool()(self._call_tool)

    async def _list_tools(self):
        return [
            Tool(
                name="fix_sonar_issues",
                description="Fix multiple SonarQube issues using AI",
                inputSchema={
                    "type": "object",
                    "properties": {
                        "class_content": {"type": "string"},
                        "issues": {
                            "type": "array",
                            "items": {
                                "type": "object",
                                "properties": {
                                    "message": {"type": "string"},
                                    "rule": {"type": "string"},
                                    "line": {"type": "integer"},
                                    "severity": {"type": "string"}
                                }
                            }
                        },
                        "context": {
                            "type": "object",
                            "properties": {
```

```
                "coding_standards": {"type": "string"},
                "project_type": {"type": "string"},
                "java_version": {"type": "string"}
            }
        }
    },
    "required": ["class_content", "issues"]
  }
),
Tool(
  name="improve_code_coverage",
  description="Generate unit tests to improve code coverage",
  inputSchema={
    "type": "object",
    "properties": {
      "class_content": {"type": "string"},
      "uncovered_lines": {"type": "array", "items": {"type": "integer"}},
      "test_framework": {"type": "string", "default": "junit5"},
      "mocking_framework": {"type": "string", "default": "mockito"}
    }
  }
),
Tool(
  name="refactor_complex_method",
  description="Refactor complex methods to reduce complexity",
  inputSchema={
    "type": "object",
    "properties": {
      "method_content": {"type": "string"},
      "complexity_metrics": {
        "type": "object",
        "properties": {
          "cyclomatic_complexity": {"type": "integer"},
          "lines_of_code": {"type": "integer"}
        }
      }
    }
  }
),
Tool(
  name="generate_documentation",
  description="Generate comprehensive JavaDoc documentation",
  inputSchema={
    "type": "object",
    "properties": {
      "code_content": {"type": "string"},
      "documentation_level": {"type": "string", "enum": ["basic", "detailed", "comprehensive"]}
```

```
            }
        }
    )
]

async def _call_tool(self, name: str, arguments: dict):
    start_time = datetime.now()

    try:
        if name == "fix_sonar_issues":
            result = await self._fix_sonar_issues(arguments)
            complexity = self._assess_complexity(arguments.get("issues", []))
            FIX_REQUESTS.labels(status='success', complexity=complexity).inc()

        elif name == "improve_code_coverage":
            result = await self._improve_code_coverage(arguments)
            FIX_REQUESTS.labels(status='success', complexity='medium').inc()

        elif name == "refactor_complex_method":
            result = await self._refactor_complex_method(arguments)
            FIX_REQUESTS.labels(status='success', complexity='high').inc()

        elif name == "generate_documentation":
            result = await self._generate_documentation(arguments)
            FIX_REQUESTS.labels(status='success', complexity='low').inc()

        else:
            raise ValueError(f"Unknown tool: {name}")

        # Record duration
        duration = (datetime.now() - start_time).total_seconds()
        FIX_DURATION.observe(duration)

        return [TextContent(type="text", text=json.dumps(result, indent=2))]

    except Exception as e:
        logger.error(f"Error in {name}: {str(e)}")
        FIX_REQUESTS.labels(status='error', complexity='unknown').inc()
        return [TextContent(type="text", text=json.dumps({"error": str(e)}))]

async def _fix_sonar_issues(self, args: dict) -> dict:
    """Fix SonarQube issues using AI"""
    class_content = args["class_content"]
    issues = args["issues"]
    context = args.get("context", {})

    # Check cache first
```

```
    cache_key = self._generate_cache_key("fix", class_content, issues)
    if self.config.cache_enabled:
        cached_result = self.redis_client.get(cache_key)
        if cached_result:
            logger.info("Returning cached fix result")
            return json.loads(cached_result)

    # Build comprehensive prompt
    prompt = self._build_fix_prompt(class_content, issues, context)

    # Generate fix using AI
    response = await self._generate_with_retry(prompt)

    # Process and validate response
    result = self._process_fix_response(response, class_content, issues)

    # Cache result
    if self.config.cache_enabled:
        self.redis_client.setex(
            cache_key,
            self.config.cache_ttl,
            json.dumps(result)
        )

    return result

async def _improve_code_coverage(self, args: dict) -> dict:
    """Generate unit tests to improve coverage"""
    class_content = args["class_content"]
    uncovered_lines = args.get("uncovered_lines", [])
    test_framework = args.get("test_framework", "junit5")
    mocking_framework = args.get("mocking_framework", "mockito")

    prompt = self._build_coverage_prompt(
        class_content, uncovered_lines, test_framework, mocking_framework
    )

    response = await self._generate_with_retry(prompt)

    return {
        "test_class": self._extract_test_code(response),
        "coverage_improvement_estimate": self._estimate_coverage_improvement(uncovered_lines),
        "test_methods_generated": self._count_test_methods(response),
        "recommendations": self._generate_test_recommendations(class_content)
    }

def _build_fix_prompt(self, class_content: str, issues: List[dict], context: dict) -> str:
```

```
        """Build comprehensive prompt for fixing issues"""

        prompt = f"""You are an expert Java developer and code quality specialist.

    Your task is to fix the following SonarQube issues in the provided Java class:

    CONTEXT:
    - Project Type: {context.get('project_type', 'Standard Java')}
    - Java Version: {context.get('java_version', '11+')}
    - Coding Standards: {context.get('coding_standards', 'Standard')}

    ISSUES TO FIX:
    {self._format_issues_for_prompt(issues)}

    REQUIREMENTS:
    1. Fix ALL listed issues completely and correctly
    2. Maintain existing functionality and business logic
    3. Follow Java best practices and conventions
    4. Ensure code compiles without errors
    5. Add meaningful comments only where necessary
    6. Preserve code readability and maintainability
    7. Use modern Java features appropriately

    ORIGINAL CODE:
    ```java
    {class_content}
```

Return ONLY the complete, fixed Java class without any markdown formatting.
The response should be syntactically correct and ready to use.
"""
return prompt

```
    def _build_coverage_prompt(self, class_content: str, uncovered_lines: List[int],
                               test_framework: str, mocking_framework: str) -> str:
        """Build prompt for generating test coverage"""

        prompt = f"""Generate comprehensive unit tests for the following Java class to improve code coverage.
```

TARGET UNCOVERED LINES: {uncovered_lines}
TEST FRAMEWORK: {test_framework}
MOCKING FRAMEWORK: {mocking_framework}

REQUIREMENTS:

1. Generate tests that cover the uncovered lines: {uncovered_lines}
```

2. Use {test_framework} annotations and assertions
3. Use {mocking_framework} for mocking dependencies
4. Include positive, negative, and edge case scenarios
5. Ensure tests are independent and can run in any order
6. Add meaningful test method names
7. Include setup and teardown methods if needed

ORIGINAL CLASS:

```
java

{class_content}
```

Generate a complete test class with all necessary imports and annotations.
"""
return prompt

```
async def _generate_with_retry(self, prompt: str, max_retries: int = 3) -> str:
    """Generate response with retry logic"""

    for attempt in range(max_retries):
        try:
            response = await self.model.generate_content_async(
                prompt,
                generation_config={
                    "max_output_tokens": self.config.max_tokens,
                    "temperature": self.config.temperature,
                    "top_p": 0.8,
                    "top_k": 40
                },
                safety_settings=self.safety_settings
            )

            if response.text:
                # Count tokens for metrics
                TOKEN_USAGE.labels(type='input').inc(len(prompt.split()))
                TOKEN_USAGE.labels(type='output').inc(len(response.text.split()))

                return response.text

        except Exception as e:
            logger.warning(f"Generation attempt {attempt + 1} failed: {str(e)}")
            if attempt < max_retries - 1:
                await asyncio.sleep(2 ** attempt)  # Exponential backoff
            else:
                raise e

    raise Exception("All generation attempts failed")

def _process_fix_response(self, response: str, original_code: str, issues: List[dict]) -> dict:
    """Process and validate the AI fix response"""

    # Clean the response
    fixed_code = self._clean_code_response(response)

    # Validate the fix
    validation = self._validate_fix(fixed_code, original_code, issues)

    return {
        "original_code": original_code,
        "fixed_code": fixed_code,
        "issues_addressed": len(issues),
        "validation": validation,
```

```
            "fix_summary": self._generate_fix_summary(issues),
            "estimated_impact": self._estimate_fix_impact(issues),
            "timestamp": datetime.now().isoformat()
        }

    def _clean_code_response(self, response: str) -> str:
        """Clean AI response to extract only Java code"""
        # Remove markdown formatting
        code = response.strip()

        if code.startswith("```java"):
            code = code[7:]
        elif code.startswith("```"):
            code = code[3:]

        if code.endswith("```"):
            code = code[:-3]

        return code.strip()

    def _validate_fix(self, fixed_code: str, original_code: str, issues: List[dict]) -> dict:
        """Validate the generated fix"""
        validation = {
            "syntax_valid": True,
            "length_check": True,
            "structure_preserved": True,
            "issues_likely_resolved": [],
            "potential_problems": []
        }

        # Basic syntax validation
        try:
            # Check for balanced braces and parentheses
            if fixed_code.count('{') != fixed_code.count('}'):
                validation["syntax_valid"] = False
                validation["potential_problems"].append("Unbalanced braces")

            if fixed_code.count('(') != fixed_code.count(')'):
                validation["syntax_valid"] = False
                validation["potential_problems"].append("Unbalanced parentheses")

            # Check if code wasn't truncated
            if len(fixed_code) < len(original_code) * 0.8:
                validation["length_check"] = False
                validation["potential_problems"].append("Code may be truncated")

            # Check for basic structure preservation
```

```
            if "class " not in fixed_code:
                validation["structure_preserved"] = False
                validation["potential_problems"].append("Class declaration missing")

        except Exception as e:
            validation["syntax_valid"] = False
            validation["potential_problems"].append(f"Validation error: {str(e)}")

        return validation

    def _generate_cache_key(self, operation: str, *args) -> str:
        """Generate cache key for caching results"""
        import hashlib
        content = f"{operation}:{':'.join(str(arg) for arg in args)}"
        return f"devgenie:{hashlib.md5(content.encode()).hexdigest()}"

    def _assess_complexity(self, issues: List[dict]) -> str:
        """Assess the complexity of the fix based on issues"""
        if len(issues) > 10:
            return "high"
        elif len(issues) > 5:
            return "medium"
        else:
            return "low"
```

if **name** == "**main**": config = AIConfig(project_id=os.getenv("GCP_PROJECT_ID"), location=os.getenv("GCP_LOCATION", "us-central1"), model_name=os.getenv("AI_MODEL", "gemini-1.5-pro-001"))

```
    server = AICodeFixServer(config)

    import uvicorn
    uvicorn.run(server.app, host="0.0.0.0", port=8002)
```

Scalability & Deployment Deep Dive

Enterprise Deployment Architectures

Multi-Region Deployment
```yaml
# terraform/main.tf
provider "aws" {
  region = var.primary_region
}

provider "aws" {
  alias  = "secondary"
  region = var.secondary_region
}

# Primary region infrastructure
module "primary_cluster" {
  source = "./modules/eks-cluster"

  cluster_name    = "devgenie-primary"
  region          = var.primary_region
  node_groups = {
    general = {
      instance_types = ["m5.large", "m5.xlarge"]
      min_size      = 3
      max_size      = 20
      desired_size  = 5
    }
    ai_workloads = {
      instance_types = ["p3.2xlarge"]  # GPU instances for AI
      min_size      = 1
      max_size      = 5
      desired_size  = 2
    }
  }
}

# Secondary region for DR
module "secondary_cluster" {
  source = "./modules/eks-cluster"

  providers = {
```

```
    aws = aws.secondary
  }

  cluster_name    = "devgenie-secondary"
  region          = var.secondary_region
  node_groups = {
    general = {
      instance_types = ["m5.large"]
      min_size     = 2
      max_size     = 10
      desired_size = 2
    }
  }
}

# Global load balancer
resource "aws_route53_record" "devgenie" {
  zone_id = var.hosted_zone_id
  name    = "devgenie.company.com"
  type    = "A"

  set_identifier = "primary"

  failover_routing_policy {
    type = "PRIMARY"
  }

  alias {
    name                   = module.primary_cluster.load_balancer_dns
    zone_id                = module.primary_cluster.load_balancer_zone_id
    evaluate_target_health = true
  }
}
```

Horizontal Pod Autoscaling

yaml

```yaml
# k8s/hpa.yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: devgenie-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: devgenie-app
  minReplicas: 3
  maxReplicas: 50
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 70
  - type: Resource
    resource:
      name: memory
      target:
        type: Utilization
        averageUtilization: 80
  - type: Pods
    pods:
      metric:
        name: active_chat_sessions
      target:
        type: AverageValue
        averageValue: "30"
  behavior:
    scaleDown:
      stabilizationWindowSeconds: 300
      policies:
      - type: Percent
        value: 10
        periodSeconds: 60
    scaleUp:
      stabilizationWindowSeconds: 60
      policies:
      - type: Percent
        value: 50
        periodSeconds: 60
      - type: Pods
```

```
    value: 5
    periodSeconds: 60
  selectPolicy: Max
```

Agentic AI Integration Strategy

LangGraph Implementation for Autonomous Operations

python

```python
# devgenie_agent.py - Production Agentic System ⚠️
from typing import TypedDict, List, Optional, Annotated
import operator
from langgraph.graph import StateGraph, END
from langgraph.checkpoint.sqlite import SqliteSaver
from langchain_core.messages import HumanMessage, AIMessage
from langchain_openai import ChatOpenAI
from langchain.tools import Tool
import asyncio
import logging

logger = logging.getLogger(__name__)

class DevGenieAgentState(TypedDict):
    messages: Annotated[List, operator.add]
    current_task: Optional[str]
    analysis_results: dict
    fix_plan: Optional[dict]
    execution_status: str
    error_count: int
    user_feedback: Optional[str]

class DevGenieAutonomousAgent:
    def __init__(self, llm, tools: List[Tool]):
        self.llm = llm
        self.tools = {tool.name: tool for tool in tools}
        self.memory = SqliteSaver.from_conn_string(":memory:")

        # Build the agent graph
        self.workflow = self._build_workflow()
        self.app = self.workflow.compile(checkpointer=self.memory)

    def _build_workflow(self) -> StateGraph:
        """Build the autonomous agent workflow"""
        workflow = StateGraph(DevGenieAgentState)

        # Add nodes
        workflow.add_node("analyze", self._analyze_codebase)
        workflow.add_node("plan", self._create_execution_plan)
        workflow.add_node("execute", self._execute_plan)
        workflow.add_node("validate", self._validate_results)
        workflow.add_node("learn", self._learn_from_outcome)
        workflow.add_node("report", self._generate_report)

        # Add edges
        workflow.add_edge("analyze", "plan")
```

```
        workflow.add_edge("plan", "execute")
        workflow.add_edge("execute", "validate")

        # Conditional edges
        workflow.add_conditional_edges(
            "validate",
            self._should_retry,
            {
                "retry": "plan",
                "learn": "learn",
                "report": "report"
            }
        )
        workflow.add_edge("learn", "report")
        workflow.add_edge("report", END)

        # Set entry point
        workflow.set_entry_point("analyze")

        return workflow

    async def _analyze_codebase(self, state: DevGenieAgentState) -> DevGenieAgentState:
        """Autonomous codebase analysis"""
        logger.info("Starting autonomous codebase analysis")

        try:
            # Run multiple analysis tools in parallel
            sonar_analysis = await self._run_tool("sonar_analysis", {})
            coverage_analysis = await self._run_tool("coverage_analysis", {})
            complexity_analysis = await self._run_tool("complexity_analysis", {})

            # Synthesize results
            analysis_results = {
                "sonar": sonar_analysis,
                "coverage": coverage_analysis,
                "complexity": complexity_analysis,
                "priority_issues": self._identify_priority_issues(       properties.put("hibernate.multiTenancy", "SCHEMA")
        properties.put("hibernate.multi_tenant_connection_provider", multiTenantConnectionProvider());
        properties.put("hibernate.tenant_identifier_resolver", currentTenantIdentifierResolver());
        properties.put("hibernate.show_sql", false);
        properties.put("hibernate.format_sql", true);

        em.setJpaPropertyMap(properties);
        return em;
    }
}
```

```java
// TenantContext.java
public class TenantContext {
    private static final ThreadLocal<String> CURRENT_TENANT = new ThreadLocal<>();

    public static void setCurrentTenant(String tenantId) {
        CURRENT_TENANT.set(tenantId);
    }

    public static String getCurrentTenant() {
        return CURRENT_TENANT.get();
    }

    public static void clear() {
        CURRENT_TENANT.remove();
    }
}

// TenantInterceptor.java
@Component
public class TenantInterceptor implements HandlerInterceptor {

    @Autowired
    private TenantService tenantService;

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) {
        String tenantId = extractTenantId(request);

        if (tenantId != null && tenantService.isValidTenant(tenantId)) {
            TenantContext.setCurrentTenant(tenantId);
            return true;
        }

        response.setStatus(HttpStatus.BAD_REQUEST.value());
        return false;
    }

    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Excep
        TenantContext.clear();
    }

    private String extractTenantId(HttpServletRequest request) {
        // Extract from header
        String tenantId = request.getHeader("X-Tenant-ID");

        if (tenantId == null) {
```

```
            // Extract from JWT token
            String authHeader = request.getHeader("Authorization");
            if (authHeader != null && authHeader.startsWith("Bearer ")) {
                tenantId = extractTenantFromJWT(authHeader.substring(7));
            }
        }

        return tenantId;
    }

    private String extractTenantFromJWT(String token) {
        try {
            // Decode JWT and extract tenant claim
            // Implementation depends on JWT library
            return null; // Placeholder
        } catch (Exception e) {
            return null;
        }
    }
}

// EnterpriseIntegrationService.java
@Service
@Slf4j
public class EnterpriseIntegrationService {

    @Autowired
    private LdapTemplate ldapTemplate;

    @Autowired
    private SamlAuthenticationProvider samlAuthProvider;

    @Autowired
    private JiraIntegrationService jiraService;

    @Autowired
    private SlackIntegrationService slackService;

    @Value("${enterprise.ldap.enabled:false}")
    private boolean ldapEnabled;

    @Value("${enterprise.saml.enabled:false}")
    private boolean samlEnabled;

    public CompletableFuture<UserProfile> importUserFromLDAP(String username) {
        return CompletableFuture.supplyAsync(() -> {
            if (!ldapEnabled) {
```

```
            throw new UnsupportedOperationException("LDAP integration is not enabled");
        }

        try {
            LdapQuery query = LdapQueryBuilder.query()
                .where("uid").is(username);

            List<UserProfile> users = ldapTemplate.search(query, new UserAttributesMapper());

            if (users.isEmpty()) {
                throw new UserNotFoundException("User not found in LDAP: " + username);
            }

            UserProfile user = users.get(0);

            // Enrich with DevGenie-specific attributes
            user.setDevGenieRole(determineDevGenieRole(user.getLdapGroups()));
            user.setTenantId(determineTenantFromGroups(user.getLdapGroups()));

            return user;

        } catch (Exception e) {
            log.error("Error importing user from LDAP: {}", username, e);
            throw new LdapIntegrationException("Failed to import user from LDAP", e);
        }
    });
}

public CompletableFuture<Void> createJiraTicketForFailedFix(FixFailureEvent event) {
    return CompletableFuture.runAsync(() -> {
        try {
            JiraTicket ticket = JiraTicket.builder()
                .project("DEVGENIE")
                .issueType("Bug")
                .summary("DevGenie Fix Failed: " + event.getClassName())
                .description(buildJiraDescription(event))
                .priority(determinePriority(event.getSeverity()))
                .labels(List.of("devgenie", "automated", event.getSeverity().toLowerCase()))
                .assignee(event.getOriginalDeveloper())
                .build();

            String ticketKey = jiraService.createTicket(ticket);

            log.info("Created Jira ticket {} for failed fix: {}", ticketKey, event.getClassName());

            // Notify team via Slack
            slackService.sendMessage(
```

```
                event.getTeamChannel(),
                String.format("🚨 DevGenie fix failed for `%s`. Created ticket: %s",
                    event.getClassName(), ticketKey)
            );

        } catch (Exception e) {
            log.error("Error creating Jira ticket for failed fix", e);
        }
    });
}

public CompletableFuture<List<TeamMetrics>> generateTeamMetrics(String tenantId) {
    return CompletableFuture.supplyAsync(() -> {
        try {
            // Get teams for tenant
            List<Team> teams = getTeamsForTenant(tenantId);

            List<TeamMetrics> teamMetrics = new ArrayList<>();

            for (Team team : teams) {
                TeamMetrics metrics = TeamMetrics.builder()
                    .teamName(team.getName())
                    .memberCount(team.getMembers().size())
                    .totalFixesApplied(calculateTeamFixes(team))
                    .averageFixTime(calculateAverageFixTime(team))
                    .codeQualityImprovement(calculateQualityImprovement(team))
                    .topContributors(getTopContributors(team))
                    .trendData(calculateTrendData(team))
                    .build();

                teamMetrics.add(metrics);
            }

            return teamMetrics;

        } catch (Exception e) {
            log.error("Error generating team metrics for tenant: {}", tenantId, e);
            return Collections.emptyList();
        }
    });
}

private String buildJiraDescription(FixFailureEvent event) {
    StringBuilder description = new StringBuilder();

    description.append("DevGenie attempted to automatically fix SonarQube issues but failed.\n\n");
    description.append("*Class:* ").append(event.getClassName()).append("\n");
```

```
    description.append("*Issues:* ").append(String.join(", ", event.getIssueDescriptions())).append("\n");
    description.append("*Error:* ").append(event.getErrorMessage()).append("\n");
    description.append("*Timestamp:* ").append(event.getTimestamp()).append("\n\n");

    description.append("*Original Code:*\n");
    description.append("{code:java}\n");
    description.append(event.getOriginalCode());
    description.append("\n{code}\n\n");

    if (event.getAttemptedFix() != null) {
        description.append("*Attempted Fix:*\n");
        description.append("{code:java}\n");
        description.append(event.getAttemptedFix());
        description.append("\n{code}\n\n");
    }

    description.append("Please review and apply the fix manually.");

    return description.toString();
}

private class UserAttributesMapper implements AttributesMapper<UserProfile> {
    @Override
    public UserProfile mapFromAttributes(Attributes attrs) throws NamingException {
        return UserProfile.builder()
            .username(getAttributeValue(attrs, "uid"))
            .firstName(getAttributeValue(attrs, "givenName"))
            .lastName(getAttributeValue(attrs, "sn"))
            .email(getAttributeValue(attrs, "mail"))
            .department(getAttributeValue(attrs, "departmentNumber"))
            .ldapGroups(getMultiValueAttribute(attrs, "memberOf"))
            .build();
    }

    private String getAttributeValue(Attributes attrs, String attributeName) throws NamingException {
        Attribute attr = attrs.get(attributeName);
        return attr != null ? (String) attr.get() : null;
    }

    private List<String> getMultiValueAttribute(Attributes attrs, String attributeName) throws NamingException {
        Attribute attr = attrs.get(attributeName);
        if (attr == null) return Collections.emptyList();

        List<String> values = new ArrayList<>();
        for (int i = 0; i < attr.size(); i++) {
            values.add((String) attr.get(i));
        }
```

```
            return values;
        }
    }
}

// ConfigurationManagementService.java
@Service
@Slf4j
public class ConfigurationManagementService {

    @Autowired
    private TenantConfigurationRepository configRepository;

    @Autowired
    private ApplicationEventPublisher eventPublisher;

    @Cacheable("tenant-configs")
    public TenantConfiguration getTenantConfiguration(String tenantId) {
        return configRepository.findByTenantId(tenantId)
            .orElseGet(() -> createDefaultConfiguration(tenantId));
    }

    public void updateTenantConfiguration(String tenantId, TenantConfiguration config) {
        config.setTenantId(tenantId);
        config.setLastModified(LocalDateTime.now());

        TenantConfiguration saved = configRepository.save(config);

        // Clear cache
        cacheManager.getCache("tenant-configs").evict(tenantId);

        // Publish configuration change event
        eventPublisher.publishEvent(new TenantConfigurationChangedEvent(tenantId, saved));

        log.info("Updated configuration for tenant: {}", tenantId);
    }

    public GlobalConfiguration getGlobalConfiguration() {
        return GlobalConfiguration.builder()
            .maxConcurrentFixes(getProperty("devgenie.max.concurrent.fixes", Integer.class, 10))
            .aiModelTimeout(getProperty("devgenie.ai.timeout", Duration.class, Duration.ofMinutes(5)))
            .rateLimitEnabled(getProperty("devgenie.rate.limit.enabled", Boolean.class, true))
            .auditEnabled(getProperty("devgenie.audit.enabled", Boolean.class, true))
            .defaultLanguage(getProperty("devgenie.default.language", String.class, "en"))
            .supportedLanguages(getProperty("devgenie.supported.languages", List.class, List.of("en", "es", "fr")))
            .build();
    }
```

```
private TenantConfiguration createDefaultConfiguration(String tenantId) {
    TenantConfiguration defaultConfig = TenantConfiguration.builder()
        .tenantId(tenantId)
        .maxUsersPerTenant(100)
        .maxFixRequestsPerDay(1000)
        .enabledFeatures(Set.of(
            Feature.CHAT_INTERFACE,
            Feature.ISSUE_FIXING,
            Feature.COVERAGE_ANALYSIS
        ))
        .sonarQubeSettings(SonarQubeSettings.builder()
            .url("http://localhost:9000")
            .enabledRules(getDefaultSonarRules())
            .severityThreshold("MAJOR")
            .build())
        .aiSettings(AISettings.builder()
            .model("gemini-1.5-pro")
            .temperature(0.1)
            .maxTokens(4096)
            .enableStreamingResponses(true)
            .build())
        .githubSettings(GitHubSettings.builder()
            .autoCreatePR(true)
            .requireReview(true)
            .defaultBranch("main")
            .prTemplate(getDefaultPRTemplate())
            .build())
        .notificationSettings(NotificationSettings.builder()
            .emailEnabled(true)
            .slackEnabled(false)
            .webhookEnabled(false)
            .build())
        .created(LocalDateTime.now())
        .lastModified(LocalDateTime.now())
        .build();

    return configRepository.save(defaultConfig);
}

@EventListener
public void handleConfigurationChange(TenantConfigurationChangedEvent event) {
    // Restart services that depend on configuration
    restartTenantServices(event.getTenantId(), event.getConfiguration());
}

private void restartTenantServices(String tenantId, TenantConfiguration config) {
```

```
        // Implementation to restart or reconfigure services
        log.info("Restarting services for tenant {} due to configuration change", tenantId);
    }

    @SuppressWarnings("unchecked")
    private <T> T getProperty(String key, Class<T> type, T defaultValue) {
        try {
            Object value = environment.getProperty(key, type);
            return value != null ? (T) value : defaultValue;
        } catch (Exception e) {
            log.warn("Error getting property {}, using default value", key, e);
            return defaultValue;
        }
    }
}
```

Week 16: Production Deployment & Monitoring

java

```java
// DevGenieApplication.java - Production-ready main class
@SpringBootApplication
@EnableJpaRepositories
@EnableCaching
@EnableScheduling
@EnableAsync
@EnableConfigurationProperties({AppConfig.class, GitHubConfig.class})
public class DevGenieApplication {

    private static final Logger log = LoggerFactory.getLogger(DevGenieApplication.class);

    public static void main(String[] args) {
        // Set production profiles
        System.setProperty("spring.profiles.active",
            System.getProperty("spring.profiles.active", "production"));

        // Configure logging
        configureLogging();

        // Set JVM options for production
        configureJVM();

        try {
            ConfigurableApplicationContext context = SpringApplication.run(DevGenieApplication.class, args);

            // Verify critical beans
            verifyCriticalBeans(context);

            // Register shutdown hooks
            registerShutdownHooks(context);

            log.info("DevGenie started successfully in production mode");

        } catch (Exception e) {
            log.error("Failed to start DevGenie application", e);
            System.exit(1);
        }
    }

    private static void configureLogging() {
        // Set log levels for production
        System.setProperty("logging.level.com.org.devgenie", "INFO");
        System.setProperty("logging.level.org.springframework", "WARN");
        System.setProperty("logging.level.org.hibernate", "WARN");

        // Configure log file rotation
```

```
        System.setProperty("logging.file.max-size", "100MB");
        System.setProperty("logging.file.max-history", "30");
    }

    private static void configureJVM() {
        // Set memory settings if not already configured
        String maxHeap = System.getProperty("Xmx");
        if (maxHeap == null) {
            Runtime runtime = Runtime.getRuntime();
            long maxMemory = runtime.maxMemory();
            log.info("JVM Max Memory: {} MB", maxMemory / 1024 / 1024);
        }
    }

    private static void verifyCriticalBeans(ConfigurableApplicationContext context) {
        // Verify critical beans are present and healthy
        String[] criticalBeans = {
            "devGenieChatService",
            "sonarService",
            "issueFixService",
            "gitHubUtility"
        };

        for (String beanName : criticalBeans) {
            if (!context.containsBean(beanName)) {
                throw new IllegalStateException("Critical bean not found: " + beanName);
            }
            log.info("Verified critical bean: {}", beanName);
        }
    }

    private static void registerShutdownHooks(ConfigurableApplicationContext context) {
        Runtime.getRuntime().addShutdownHook(new Thread(() -> {
            log.info("Shutting down DevGenie gracefully...");
            context.close();
            log.info("DevGenie shutdown complete");
        }));
    }
}

// DockerConfiguration.dockerfile
FROM openjdk:21-jdk-slim

# Install required packages
RUN apt-get update && apt-get install -y \
    curl \
    git \
```

```
    maven |
    && rm -rf /var/lib/apt/lists/*

# Create application user
RUN groupadd -r devgenie && useradd -r -g devgenie devgenie

# Set working directory
WORKDIR /app

# Copy application jar
COPY target/devgenie-*.jar app.jar

# Copy configuration files
COPY docker/application-production.yml application-production.yml
COPY docker/logback-spring.xml logback-spring.xml

# Set ownership
RUN chown -R devgenie:devgenie /app

# Switch to non-root user
USER devgenie

# Expose port
EXPOSE 8080

# Health check
HEALTHCHECK --interval=30s --timeout=10s --start-period=60s --retries=3 |
    CMD curl -f http://localhost:8080/actuator/health || exit 1

# Set JVM options for container
ENV JAVA_OPTS="-Xmx2g -Xms1g -XX:+UseG1GC -XX:G1HeapRegionSize=16m -XX:+UseContainerSupport"

# Start application
ENTRYPOINT ["sh", "-c", "java $JAVA_OPTS -jar app.jar --spring.profiles.active=production"]

# docker-compose.production.yml
version: '3.8'

services:
  devgenie-app:
    build: .
    ports:
      - "8080:8080"
    environment:
      - SPRING_PROFILES_ACTIVE=production
      - DATABASE_URL=jdbc:postgresql://postgres:5432/devgenie
      - DATABASE_USERNAME=devgenie
```

```
    - DATABASE_PASSWORD=${DB_PASSWORD}
    - SONAR_URL=http://sonarqube:9000
    - SONAR_TOKEN=${SONAR_TOKEN}
    - GITHUB_TOKEN=${GITHUB_TOKEN}
    - AI_API_KEY=${AI_API_KEY}
    - REDIS_URL=redis://redis:6379
  depends_on:
    - postgres
    - redis
    - sonarqube
  volumes:
    - ./logs:/app/logs
  restart: unless-stopped
  healthcheck:
    test: ["CMD", "curl", "-f", "http://localhost:8080/actuator/health"]
    interval: 30s
    timeout: 10s
    retries: 3
    start_period: 60s

postgres:
  image: postgres:15
  environment:
    - POSTGRES_DB=devgenie
    - POSTGRES_USER=devgenie
    - POSTGRES_PASSWORD=${DB_PASSWORD}
  volumes:
    - postgres_data:/var/lib/postgresql/data
    - ./docker/init-db.sql:/docker-entrypoint-initdb.d/init.sql
  ports:
    - "5432:5432"
  restart: unless-stopped

redis:
  image: redis:7-alpine
  ports:
    - "6379:6379"
  volumes:
    - redis_data:/data
  restart: unless-stopped

sonarqube:
  image: sonarqube:10.3-community
  environment:
    - SONAR_JDBC_URL=jdbc:postgresql://postgres:5432/sonarqube
    - SONAR_JDBC_USERNAME=sonarqube
    - SONAR_JDBC_PASSWORD=${SONAR_DB_PASSWORD}
```

```
  volumes:
    - sonarqube_data:/opt/sonarqube/data
    - sonarqube_extensions:/opt/sonarqube/extensions
    - sonarqube_logs:/opt/sonarqube/logs
  ports:
    - "9000:9000"
  depends_on:
    - postgres
  restart: unless-stopped

nginx:
  image: nginx:alpine
  ports:
    - "80:80"
    - "443:443"
  volumes:
    - ./docker/nginx.conf:/etc/nginx/nginx.conf
    - ./docker/ssl:/etc/nginx/ssl
  depends_on:
    - devgenie-app
  restart: unless-stopped

prometheus:
  image: prom/prometheus:latest
  ports:
    - "9090:9090"
  volumes:
    - ./docker/prometheus.yml:/etc/prometheus/prometheus.yml
    - prometheus_data:/prometheus
  command:
    - '--config.file=/etc/prometheus/prometheus.yml'
    - '--storage.tsdb.path=/prometheus'
    - '--web.console.libraries=/etc/prometheus/console_libraries'
    - '--web.console.templates=/etc/prometheus/consoles'
  restart: unless-stopped

grafana:
  image: grafana/grafana:latest
  ports:
    - "3000:3000"
  environment:
    - GF_SECURITY_ADMIN_PASSWORD=${GRAFANA_PASSWORD}
  volumes:
    - grafana_data:/var/lib/grafana
    - ./docker/grafana/provisioning:/etc/grafana/provisioning
  depends_on:
    - prometheus
```

```
    restart: unless-stopped

volumes:
  postgres_data:
  redis_data:
  sonarqube_data:
  sonarqube_extensions:
  sonarqube_logs:
  prometheus_data:
  grafana_data:

# kubernetes-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: devgenie-app
  labels:
    app: devgenie
spec:
  replicas: 3
  selector:
    matchLabels:
      app: devgenie
  template:
    metadata:
      labels:
        app: devgenie
    spec:
      containers:
      - name: devgenie
        image: devgenie:latest
        ports:
        - containerPort: 8080
        env:
        - name: SPRING_PROFILES_ACTIVE
          value: "production,kubernetes"
        - name: DATABASE_URL
          valueFrom:
            secretKeyRef:
              name: devgenie-secrets
              key: database-url
        - name: AI_API_KEY
          valueFrom:
            secretKeyRef:
              name: devgenie-secrets
              key: ai-api-key
        resources:
```

```
          requests:
            memory: "1Gi"
            cpu: "500m"
          limits:
            memory: "2Gi"
            cpu: "1000m"
        livenessProbe:
          httpGet:
            path: /actuator/health
            port: 8080
          initialDelaySeconds: 60
          periodSeconds: 30
        readinessProbe:
          httpGet:
            path: /actuator/health/readiness
            port: 8080
          initialDelaySeconds: 30
          periodSeconds: 10
        volumeMounts:
        - name: logs
          mountPath: /app/logs
      volumes:
      - name: logs
        persistentVolumeClaim:
          claimName: devgenie-logs-pvc

---
apiVersion: v1
kind: Service
metadata:
  name: devgenie-service
spec:
  selector:
    app: devgenie
  ports:
  - protocol: TCP
    port: 80
    targetPort: 8080
  type: LoadBalancer

---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: devgenie-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
```

```
    cert-manager.io/cluster-issuer: "letsencrypt-prod"
spec:
  tls:
  - hosts:
    - devgenie.company.com
    secretName: devgenie-tls
  rules:
  - host: devgenie.company.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: devgenie-service
            port:
              number: 80
```

MCP Architecture Implementation

MCP Server Implementation Strategy

Phase 1: Core MCP Servers (Weeks 1-6)

DevGenie Master MCP Server (Java/Spring Boot)

java

```java
// McpServerApplication.java
@SpringBootApplication
public class McpServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(McpServerApplication.class, args);
    }

    @Bean
    public McpServerRunner mcpServerRunner() {
        return new McpServerRunner();
    }
}

// McpServerRunner.java
@Component
public class McpServerRunner implements ApplicationRunner {

    @Override
    public void run(ApplicationArguments args) throws Exception {
        McpServer server = new DevGenieMasterMcpServer();
        server.start();
    }
}

// DevGenieMasterMcpServer.java
public class DevGenieMasterMcpServer extends AbstractMcpServer {

    @Autowired
    private ConversationOrchestrator orchestrator;

    @Autowired
    private McpClientPool clientPool;

    @Override
    protected void configureTools() {
        addTool("chat_with_devgenie", this::handleChat);
        addTool("analyze_tech_debt", this::analyzeTechDebt);
        addTool("autonomous_fix", this::autonomousFix);
        addTool("get_project_insights", this::getProjectInsights);
    }

    private McpResponse handleChat(McpRequest request) {
        try {
            String query = request.getParameter("query", String.class);
            Map<String, Object> context = request.getParameter("context", Map.class);
```

```
            ConversationResult result = orchestrator.processConversation(query, context);

            return McpResponse.success(result);

        } catch (Exception e) {
            return McpResponse.error("Chat processing failed: " + e.getMessage());
        }
    }

    private McpResponse analyzeTechDebt(McpRequest request) {
        return CompletableFuture.supplyAsync(() -> {
            try {
                String projectPath = request.getParameter("project_path", String.class);
                String analysisType = request.getParameter("analysis_type", String.class);

                // Coordinate multiple MCP servers
                List<CompletableFuture<McpResponse>> futures = Arrays.asList(
                    clientPool.getClient("sonar-server").callTool("comprehensive_analysis",
                        Map.of("project_path", projectPath)),
                    clientPool.getClient("coverage-server").callTool("analyze_coverage",
                        Map.of("project_path", projectPath)),
                    clientPool.getClient("complexity-server").callTool("analyze_complexity",
                        Map.of("project_path", projectPath))
                );

                // Wait for all analyses
                List<McpResponse> responses = futures.stream()
                    .map(CompletableFuture::join)
                    .toList();

                // Combine results
                TechDebtAnalysis analysis = combineAnalysisResults(responses);

                return McpResponse.success(analysis);

            } catch (Exception e) {
                return McpResponse.error("Tech debt analysis failed: " + e.getMessage());
            }
        }).join();
    }
}
```

SonarQube MCP Server (Python) ⚠️

python

```python
# sonar_mcp_server.py - Production-ready version
import asyncio
import json
import logging
import os
from typing import Dict, List, Optional
from dataclasses import dataclass
from datetime import datetime

from mcp.server import Server
from mcp.types import Tool, TextContent, Resource
import aiohttp
import asyncpg
from prometheus_client import Counter, Histogram, start_http_server

# Metrics
REQUEST_COUNT = Counter('sonar_mcp_requests_total', 'Total requests', ['method', 'status'])
REQUEST_DURATION = Histogram('sonar_mcp_request_duration_seconds', 'Request duration')

# Configure logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

@dataclass
class SonarConfig:
    url: str
    token: str
    timeout: int = 30
    max_retries: int = 3
    cache_ttl: int = 300  # 5 minutes

class SonarMcpServer:
    def __init__(self, config: SonarConfig):
        self.config = config
        self.app = Server("sonarqube-analyzer", version="1.0.0")
        self.session: Optional[aiohttp.ClientSession] = None
        self.db_pool: Optional[asyncpg.Pool] = None

        # Register tools
        self._register_tools()

        # Start metrics server
        start_http_server(8001)

    def _register_tools(self):
        """Register all available tools"""
```

```
    self.app.list_tools()(self._list_tools)
    self.app.call_tool()(self._call_tool)
    self.app.list_resources()(self._list_resources)
    self.app.read_resource()(self._read_resource)

async def _list_tools(self):
    return [
        Tool(
            name="comprehensive_analysis",
            description="Perform comprehensive SonarQube analysis",
            inputSchema={
                "type": "object",
                "properties": {
                    "project_path": {"type": "string"},
                    "include_history": {"type": "boolean", "default": False},
                    "severity_filter": {
                        "type": "array",
                        "items": {"type": "string"},
                        "default": ["BLOCKER", "CRITICAL", "MAJOR"]
                    }
                },
                "required": ["project_path"]
            }
        ),
        Tool(
            name="get_quality_gate_status",
            description="Get quality gate status for project",
            inputSchema={
                "type": "object",
                "properties": {
                    "project_key": {"type": "string"}
                },
                "required": ["project_key"]
            }
        ),
        Tool(
            name="get_technical_debt",
            description="Calculate technical debt metrics",
            inputSchema={
                "type": "object",
                "properties": {
                    "project_key": {"type": "string"},
                    "include_breakdown": {"type": "boolean", "default": True}
                }
            }
        ),
        Tool(
```

```
            name="monitor_quality_trends",
            description="Monitor quality trends over time",
            inputSchema={
                "type": "object",
                "properties": {
                    "project_key": {"type": "          testCases.add(TestCase.builder()
            .testMethodName("test" + capitalize(method.getNameAsString()) + "_Throws" + exceptionType)
            .type(TestType.EXCEPTION_HANDLING)
            .description("Test that " + exceptionType + " is thrown under specific conditions")
            .inputParameters(generateInputsForException(throwStmt))
            .expectedBehavior("Throws " + exceptionType)
            .expectedException(exceptionType)
            .build());
        }

        return testCases;
    }

    private String generateTestCode(TestCase testCase, MethodTestingContext context) {
        return templateEngine.generateTestMethod(testCase, context);
    }

    private boolean shouldGenerateTestFor(MethodDeclaration method) {
        // Skip getters, setters, constructors, and private methods
        return method.isPublic() &&
               !isGetter(method) &&
               !isSetter(method) &&
               !method.isConstructor() &&
               !method.getNameAsString().equals("main");
    }

    private boolean isGetter(MethodDeclaration method) {
        return method.getNameAsString().startsWith("get") &&
               method.getParameters().isEmpty() &&
               !method.getType().isVoidType();
    }

    private boolean isSetter(MethodDeclaration method) {
        return method.getNameAsString().startsWith("set") &&
               method.getParameters().size() == 1 &&
               method.getType().isVoidType();
    }
}

// TestTemplateEngine.java
@Component
public class TestTemplateEngine {
```

```
public String generateTestMethod(TestCase testCase, MethodTestingContext context) {
    StringBuilder testCode = new StringBuilder();

    // Add JavaDoc
    testCode.append("    /**\n");
    testCode.append("     * ").append(testCase.getDescription()).append("\n");
    testCode.append("     */\n");

    // Add annotations
    testCode.append("    @Test\n");
    if (testCase.getExpectedException() != null) {
        testCode.append("    @DisplayName(\"").append(testCase.getDescription()).append("\")\n");
    }

    // Method signature
    testCode.append("    void ").append(testCase.getTestMethodName()).append("() {\n");

    // Method body based on test type
    switch (testCase.getType()) {
        case HAPPY_PATH:
            generateHappyPathTestBody(testCode, testCase, context);
            break;
        case EXCEPTION_HANDLING:
            generateExceptionTestBody(testCode, testCase, context);
            break;
        case BRANCH_COVERAGE:
            generateBranchTestBody(testCode, testCase, context);
            break;
        case NULL_SAFETY:
            generateNullSafetyTestBody(testCode, testCase, context);
            break;
        case BOUNDARY:
            generateBoundaryTestBody(testCode, testCase, context);
            break;
    }

    testCode.append("    }\n\n");

    return testCode.toString();
}

private void generateHappyPathTestBody(StringBuilder testCode, TestCase testCase, MethodTestingContext cc
    MethodDeclaration method = context.getMethod();

    testCode.append("        // Arrange\n");
    generateArrangeSection(testCode, testCase, context);
```

```
    testCode.append("\n        // Act\n");
    generateActSection(testCode, testCase, context);

    testCode.append("\n        // Assert\n");
    generateAssertSection(testCode, testCase, context);
}

private void generateExceptionTestBody(StringBuilder testCode, TestCase testCase, MethodTestingContext cor
    testCode.append("        // Arrange\n");
    generateArrangeSection(testCode, testCase, context);

    testCode.append("\n        // Act & Assert\n");
    testCode.append("        assertThrows(").append(testCase.getExpectedException()).append(".class, () -> {\n")
    generateActSection(testCode, testCase, context);
    testCode.append("        });\n");
}

private void generateArrangeSection(StringBuilder testCode, TestCase testCase, MethodTestingContext contex
    MethodDeclaration method = context.getMethod();
    String className = context.getClassName();

    // Create instance of class under test
    testCode.append("        ").append(className).append(" instance = new ").append(className).append("();\r

    // Setup mock dependencies if needed
    if (context.getAnalysis().hasExternalDependencies()) {
        context.getDependencies().forEach(dep -> {
            String mockName = dep.toLowerCase() + "Mock";
            testCode.append("        ").append(dep).append(" ").append(mockName)
                    .append(" = Mockito.mock(").append(dep).append(".class);\n");
        });
    }

    // Setup test data
    testCase.getInputParameters().forEach((paramName, paramValue) -> {
        testCode.append("        ").append(inferType(paramValue)).append(" ")
                .append(paramName).append(" = ").append(paramValue).append(";\n");
    });
}

private void generateActSection(StringBuilder testCode, TestCase testCase, MethodTestingContext context) {
    MethodDeclaration method = context.getMethod();
    String methodName = method.getNameAsString();

    // Build method call
    StringBuilder methodCall = new StringBuilder();
```

```
    methodCall.append("instance.").append(methodName).append("(");

    // Add parameters
    List<String> paramNames = testCase.getInputParameters().keySet().stream().toList();
    methodCall.append(String.join(", ", paramNames));
    methodCall.append(")");

    if (method.getType().isVoidType()) {
        testCode.append("        ").append(methodCall).append(";\n");
    } else {
        String resultType = method.getType().asString();
        testCode.append("        ").append(resultType).append(" result = ").append(methodCall).append(";\n");
    }
}

private void generateAssertSection(StringBuilder testCode, TestCase testCase, MethodTestingContext context)
    MethodDeclaration method = context.getMethod();

    if (!method.getType().isVoidType()) {
        // Generate assertions based on expected behavior
        if (testCase.getExpectedBehavior().contains("not null")) {
            testCode.append("        assertThat(result).isNotNull();\n");
        }
        if (testCase.getExpectedBehavior().contains("empty")) {
            testCode.append("        assertThat(result).isEmpty();\n");
        }
        if (testCase.getExpectedBehavior().contains("contains")) {
            testCode.append("        assertThat(result).isNotEmpty();\n");
        }

        // Default assertion
        testCode.append("        // TODO: Add specific assertions based on expected behavior\n");
        testCode.append("        // Expected: ").append(testCase.getExpectedBehavior()).append("\n");
    } else {
        testCode.append("        // Verify behavior for void method\n");
        testCode.append("        // TODO: Add verifications based on side effects\n");
    }
}

private String inferType(String value) {
    if (value.equals("null")) return "Object";
    if (value.startsWith("\"")) return "String";
    if (value.matches("\\d+")) return "int";
    if (value.matches("\\d+\\.\\d+")) return "double";
    if (value.equals("true") || value.equals("false")) return "boolean";
    return "Object";
```

```
    }
  }
```

Phase 4: Production Features (Weeks 13-16)

Week 13: Advanced Monitoring & Analytics

java

```java
// DevGenieMetricsService.java
@Service
@Slf4j
public class DevGenieMetricsService {

    @Autowired
    private MeterRegistry meterRegistry;

    @Autowired
    private PullRequestMetricsRepository pullRequestRepository;

    @Autowired
    private ChatInteractionRepository chatRepository;

    private final Counter fixRequestsCounter;
    private final Timer fixProcessingTimer;
    private final Gauge activeSessions;
    private final Counter chatInteractionsCounter;

    public DevGenieMetricsService(MeterRegistry meterRegistry) {
        this.meterRegistry = meterRegistry;
        this.fixRequestsCounter = Counter.builder("devgenie.fix.requests")
            .description("Number of fix requests processed")
            .tag("status", "total")
            .register(meterRegistry);

        this.fixProcessingTimer = Timer.builder("devgenie.fix.processing.time")
            .description("Time taken to process fix requests")
            .register(meterRegistry);

        this.activeSessions = Gauge.builder("devgenie.chat.active.sessions")
            .description("Number of active chat sessions")
            .register(meterRegistry, this, DevGenieMetricsService::getActiveSessionCount);

        this.chatInteractionsCounter = Counter.builder("devgenie.chat.interactions")
            .description("Number of chat interactions")
            .register(meterRegistry);
    }

    public void recordFixRequest(String severity, boolean successful) {
        fixRequestsCounter.increment(
            Tags.of(
                Tag.of("severity", severity),
                Tag.of("status", successful ? "success" : "failure")
            )
        );
```

```
}

public void recordFixProcessingTime(Duration duration, String issueType) {
    fixProcessingTimer.record(duration, Tags.of(Tag.of("issue_type", issueType)));
}

public void recordChatInteraction(String intentType, boolean successful) {
    chatInteractionsCounter.increment(
        Tags.of(
            Tag.of("intent", intentType),
            Tag.of("status", successful ? "success" : "failure")
        )
    );
}

public CompletableFuture<DevGenieAnalytics> generateAnalytics(AnalyticsRequest request) {
    return CompletableFuture.supplyAsync(() -> {
        try {
            LocalDateTime startDate = request.getStartDate();
            LocalDateTime endDate = request.getEndDate();

            // Gather metrics from various sources
            FixMetrics fixMetrics = calculateFixMetrics(startDate, endDate);
            ChatMetrics chatMetrics = calculateChatMetrics(startDate, endDate);
            ProductivityMetrics productivityMetrics = calculateProductivityMetrics(startDate, endDate);
            QualityMetrics qualityMetrics = calculateQualityMetrics(startDate, endDate);

            return DevGenieAnalytics.builder()
                .periodStart(startDate)
                .periodEnd(endDate)
                .fixMetrics(fixMetrics)
                .chatMetrics(chatMetrics)
                .productivityMetrics(productivityMetrics)
                .qualityMetrics(qualityMetrics)
                .generatedAt(LocalDateTime.now())
                .build();

        } catch (Exception e) {
            log.error("Error generating analytics", e);
            throw new AnalyticsGenerationException("Failed to generate analytics", e);
        }
    });
}

private FixMetrics calculateFixMetrics(LocalDateTime startDate, LocalDateTime endDate) {
    List<PullRequestMetrics> prs = pullRequestRepository.findByCreatedDateTimeBetween(startDate, endDate);
```

```
    int totalPRs = prs.size();
    int totalIssuesResolved = prs.stream().mapToInt(PullRequestMetrics::getIssuesResolved).sum();
    double totalTimeSaved = prs.stream().mapToDouble(PullRequestMetrics::getEngineeringTimeSaved).sum();
    double totalCostSavings = prs.stream().mapToDouble(PullRequestMetrics::getCostSavings).sum();

    Map<String, Integer> issuesBySeverity = calculateIssuesBySeverity(startDate, endDate);
    Map<String, Integer> issuesByType = calculateIssuesByType(startDate, endDate);

    double averageFixTime = calculateAverageFixTime(startDate, endDate);
    double fixSuccessRate = calculateFixSuccessRate(startDate, endDate);

    return FixMetrics.builder()
        .totalPRsCreated(totalPRs)
        .totalIssuesResolved(totalIssuesResolved)
        .totalTimeSaved(totalTimeSaved)
        .totalCostSavings(totalCostSavings)
        .issuesBySeverity(issuesBySeverity)
        .issuesByType(issuesByType)
        .averageFixTime(Duration.ofMinutes((long) averageFixTime))
        .fixSuccessRate(fixSuccessRate)
        .build();
}

private ChatMetrics calculateChatMetrics(LocalDateTime startDate, LocalDateTime endDate) {
    List<ChatInteraction> interactions = chatRepository.findByTimestampBetween(startDate, endDate);

    int totalInteractions = interactions.size();
    long uniqueUsers = interactions.stream().map(ChatInteraction::getUserId).distinct().count();

    Map<String, Integer> intentDistribution = interactions.stream()
        .collect(Collectors.groupingBy(
            ChatInteraction::getIntent,
            Collectors.collectingAndThen(Collectors.counting(), Math::toIntExact)
        ));

    double averageSessionLength = calculateAverageSessionLength(interactions);
    double userSatisfactionScore = calculateUserSatisfactionScore(interactions);

    return ChatMetrics.builder()
        .totalInteractions(totalInteractions)
        .uniqueUsers(uniqueUsers)
        .intentDistribution(intentDistribution)
        .averageSessionLength(Duration.ofMinutes((long) averageSessionLength))
        .userSatisfactionScore(userSatisfactionScore)
        .build();
}
```

```
    private ProductivityMetrics calculateProductivityMetrics(LocalDateTime startDate, LocalDateTime endDate) {
        // Calculate productivity improvements
        double codeQualityImprovement = calculateCodeQualityImprovement(startDate, endDate);
        double developerEfficiencyGain = calculateDeveloperEfficiencyGain(startDate, endDate);
        double timeToResolutionReduction = calculateTimeToResolutionReduction(startDate, endDate);

        return ProductivityMetrics.builder()
            .codeQualityImprovement(codeQualityImprovement)
            .developerEfficiencyGain(developerEfficiencyGain)
            .timeToResolutionReduction(timeToResolutionReduction)
            .automationRate(calculateAutomationRate(startDate, endDate))
            .build();
    }

    @EventListener
    public void handleFixStarted(FixStartedEvent event) {
        recordFixRequest(event.getSeverity(), true);

        // Start timing
        Timer.Sample sample = Timer.start(meterRegistry);
        event.setSample(sample);
    }

    @EventListener
    public void handleFixCompleted(FixCompletedEvent event) {
        // Stop timing
        if (event.getSample() != null) {
            event.getSample().stop(fixProcessingTimer);
        }

        recordFixProcessingTime(event.getDuration(), event.getIssueType());
    }

    @EventListener
    public void handleChatInteraction(ChatInteractionEvent event) {
        recordChatInteraction(event.getIntentType(), event.isSuccessful());
    }

    private double getActiveSessionCount() {
        // Implementation to get current active sessions
        return chatRepository.countActiveSessions();
    }
}

// DevGenieHealthIndicator.java
@Component
public class DevGenieHealthIndicator implements HealthIndicator {
```

```java
@Autowired
private SonarService sonarService;

@Autowired
private GitHubUtility gitHubUtility;

@Autowired
private VertexAiChatModel aiModel;

@Override
public Health health() {
    Health.Builder builder = new Health.Builder();

    try {
        // Check SonarQube connectivity
        boolean sonarHealthy = checkSonarHealth();
        builder.withDetail("sonarqube", sonarHealthy ? "UP" : "DOWN");

        // Check GitHub connectivity
        boolean githubHealthy = checkGitHubHealth();
        builder.withDetail("github", githubHealthy ? "UP" : "DOWN");

        // Check AI service
        boolean aiHealthy = checkAIHealth();
        builder.withDetail("ai_service", aiHealthy ? "UP" : "DOWN");

        // Check database
        boolean dbHealthy = checkDatabaseHealth();
        builder.withDetail("database", dbHealthy ? "UP" : "DOWN");

        // Overall health
        if (sonarHealthy && githubHealthy && aiHealthy && dbHealthy) {
            builder.status(Status.UP);
        } else if (aiHealthy && dbHealthy) {
            builder.status("DEGRADED");
        } else {
            builder.status(Status.DOWN);
        }

    } catch (Exception e) {
        builder.status(Status.DOWN).withException(e);
    }

    return builder.build();
}
```

```
    private boolean checkSonarHealth() {
        try {
            sonarService.getRootNode();
            return true;
        } catch (Exception e) {
            return false;
        }
    }

    private boolean checkGitHubHealth() {
        try {
            // Simple connectivity check
            return true; // Implement actual check
        } catch (Exception e) {
            return false;
        }
    }

    private boolean checkAIHealth() {
        try {
            String response = aiModel.call("Health check");
            return response != null && !response.isEmpty();
        } catch (Exception e) {
            return false;
        }
    }

    private boolean checkDatabaseHealth() {
        try {
            // Check database connectivity
            return true; // Implement actual check
        } catch (Exception e) {
            return false;
        }
    }
}
```

Week 14: Enhanced Security & Authentication

java

```java
// DevGenieSecurityConfig.java
@Configuration
@EnableWebSecurity
@EnableMethodSecurity
public class DevGenieSecurityConfig {

    @Autowired
    private DevGenieAuthenticationEntryPoint authenticationEntryPoint;

    @Autowired
    private DevGenieAccessDeniedHandler accessDeniedHandler;

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
            .csrf(csrf -> csrf.disable()) // API-first application
            .sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .authorizeHttpRequests(authz -> authz
                // Public endpoints
                .requestMatchers("/actuator/health", "/actuator/info").permitAll()
                .requestMatchers("/api/auth/**").permitAll()
                .requestMatchers("/swagger-ui/**", "/v3/api-docs/**").permitAll()

                // Chat endpoints - require authentication
                .requestMatchers("/api/chat/**").hasRole("USER")

                // Fix endpoints - require elevated permissions
                .requestMatchers(HttpMethod.POST, "/sonar/issue/apply-fix").hasRole("DEVELOPER")

                // Admin endpoints
                .requestMatchers("/api/admin/**").hasRole("ADMIN")
                .requestMatchers("/actuator/**").hasRole("ADMIN")

                // Everything else requires authentication
                .anyRequest().authenticated()
            )
            .oauth2ResourceServer(oauth2 -> oauth2
                .jwt(jwt -> jwt
                    .jwtAuthenticationConverter(jwtAuthenticationConverter())
                )
            )
            .exceptionHandling(exceptions -> exceptions
                .authenticationEntryPoint(authenticationEntryPoint)
                .accessDeniedHandler(accessDeniedHandler)
            );
```

```
        return http.build();
    }

    @Bean
    public JwtAuthenticationConverter jwtAuthenticationConverter() {
        JwtGrantedAuthoritiesConverter authoritiesConverter = new JwtGrantedAuthoritiesConverter();
        authoritiesConverter.setAuthorityPrefix("ROLE_");
        authoritiesConverter.setAuthoritiesClaimName("roles");

        JwtAuthenticationConverter converter = new JwtAuthenticationConverter();
        converter.setJwtGrantedAuthoritiesConverter(authoritiesConverter);
        converter.setPrincipalClaimName("preferred_username");

        return converter;
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}

// DevGenieAuditService.java
@Service
@Slf4j
public class DevGenieAuditService {

    @Autowired
    private AuditEventRepository auditRepository;

    @EventListener
    @Async
    public void handleFixRequest(FixRequestEvent event) {
        AuditEvent auditEvent = AuditEvent.builder()
            .eventType(AuditEventType.FIX_REQUEST)
            .userId(event.getUserId())
            .sessionId(event.getSessionId())
            .timestamp(LocalDateTime.now())
            .details(Map.of(
                "issueKeys", event.getIssueKeys(),
                "severity", event.getSeverity(),
                "projectPath", event.getProjectPath()
            ))
            .ipAddress(event.getIpAddress())
            .userAgent(event.getUserAgent())
            .build();
```

```
    auditRepository.save(auditEvent);
    log.info("Audit: Fix request by user {} for {} issues", event.getUserId(), event.getIssueKeys().size());
}

@EventListener
@Async
public void handleChatInteraction(ChatInteractionEvent event) {
    AuditEvent auditEvent = AuditEvent.builder()
        .eventType(AuditEventType.CHAT_INTERACTION)
        .userId(event.getUserId())
        .sessionId(event.getSessionId())
        .timestamp(LocalDateTime.now())
        .details(Map.of(
            "intent", event.getIntentType(),
            "queryLength", event.getQuery().length(),
            "responseGenerated", event.isSuccessful()
        ))
        .ipAddress(event.getIpAddress())
        .build();

    auditRepository.save(auditEvent);
}

@EventListener
@Async
public void handleDataAccess(DataAccessEvent event) {
    if (event.issensitiveOperation()) {
        AuditEvent auditEvent = AuditEvent.builder()
            .eventType(AuditEventType.DATA_ACCESS)
            .userId(event.getUserId())
            .timestamp(LocalDateTime.now())
            .details(Map.of(
                "operation", event.getOperation(),
                "resource", event.getResource(),
                "accessGranted", event.isAccessGranted()
            ))
            .build();

        auditRepository.save(auditEvent);
    }
}

public List<AuditEvent> getAuditTrail(String userId, LocalDateTime from, LocalDateTime to) {
    return auditRepository.findByUserIdAndTimestampBetween(userId, from, to);
}

public SecurityReport generateSecurityReport(LocalDateTime from, LocalDateTime to) {
```

```
    List<AuditEvent> events = auditRepository.findByTimestampBetween(from, to);

    Map<String, Long> eventsByType = events.stream()
        .collect(Collectors.groupingBy(
            event -> event.getEventType().name(),
            Collectors.counting()
        ));

    Map<String, Long> eventsByUser = events.stream()
        .collect(Collectors.groupingBy(
            AuditEvent::getUserId,
            Collectors.counting()
        ));

    List<SecurityAnomaly> anomalies = detectSecurityAnomalies(events);

    return SecurityReport.builder()
        .reportPeriod(from + " to " + to)
        .totalEvents(events.size())
        .eventsByType(eventsByType)
        .eventsByUser(eventsByUser)
        .securityAnomalies(anomalies)
        .generatedAt(LocalDateTime.now())
        .build();
}

private List<SecurityAnomaly> detectSecurityAnomalies(List<AuditEvent> events) {
    List<SecurityAnomaly> anomalies = new ArrayList<>();

    // Detect unusual activity patterns
    Map<String, List<AuditEvent>> eventsByUser = events.stream()
        .collect(Collectors.groupingBy(AuditEvent::getUserId));

    eventsByUser.forEach((userId, userEvents) -> {
        // Check for high-frequency requests
        if (userEvents.size() > 1000) { // Threshold for suspicious activity
            anomalies.add(SecurityAnomaly.builder()
                .type(AnomalyType.HIGH_FREQUENCY_REQUESTS)
                .userId(userId)
                .description("User made " + userEvents.size() + " requests")
                .severity(AnomalySeverity.MEDIUM)
                .detectedAt(LocalDateTime.now())
                .build());
        }

        // Check for failed authentication attempts
        long failedAttempts = userEvents.stream()
```

```
            .filter(event -> event.getEventType() == AuditEventType.AUTHENTICATION_FAILURE)
            .count();

        if (failedAttempts > 10) {
            anomalies.add(SecurityAnomaly.builder()
                .type(AnomalyType.MULTIPLE_FAILED_LOGINS)
                .userId(userId)
                .description(failedAttempts + " failed login attempts")
                .severity(AnomalySeverity.HIGH)
                .detectedAt(LocalDateTime.now())
                .build());
        }
    });

    return anomalies;
  }
}

// RateLimitingService.java
@Service
@Slf4j
public class RateLimitingService {

  private final Map<String, Bucket> userBuckets = new ConcurrentHashMap<>();
  private final Map<String, Bucket> ipBuckets = new ConcurrentHashMap<>();

  // Rate limits
  private final Bandwidth userLimit = Bandwidth.classic(100, Refill.intervally(100, Duration.ofMinutes(1)));
  private final Bandwidth ipLimit = Bandwidth.classic(1000, Refill.intervally(1000, Duration.ofMinutes(1)));
  private final Bandwidth aiCallLimit = Bandwidth.classic(10, Refill.intervally(10, Duration.ofMinutes(1)));

  public boolean isAllowed(String userId, String ipAddress, RequestType requestType) {
    try {
      // Check user-based rate limit
      Bucket userBucket = userBuckets.computeIfAbsent(userId,
        k -> Bucket.builder().addLimit(userLimit).build());

      if (!userBucket.tryConsume(1)) {
        log.warn("Rate limit exceeded for user: {}", userId);
        return false;
      }

      // Check IP-based rate limit
      Bucket ipBucket = ipBuckets.computeIfAbsent(ipAddress,
        k -> Bucket.builder().addLimit(ipLimit).build());

      if (!ipBucket.tryConsume(1)) {
```

```
            log.warn("Rate limit exceeded for IP: {}", ipAddress);
            return false;
        }

        // Additional limits for AI calls
        if (requestType == RequestType.AI_CALL) {
            Bucket aiCallBucket = userBuckets.computeIfAbsent(userId + ":ai",
                k -> Bucket.builder().addLimit(aiCallLimit).build());

            if (!aiCallBucket.tryConsume(1)) {
                log.warn("AI call rate limit exceeded for user: {}", userId);
                return false;
            }
        }

        return true;

    } catch (Exception e) {
        log.error("Error checking rate limits", e);
        return true; // Fail open
    }
}

public RateLimitStatus getRateLimitStatus(String userId, String ipAddress) {
    Bucket userBucket = userBuckets.get(userId);
    Bucket ipBucket = ipBuckets.get(ipAddress);

    return RateLimitStatus.builder()
        .userTokensRemaining(userBucket != null ? userBucket.getAvailableTokens() : userLimit.getCapacity())
        .ipTokensRemaining(ipBucket != null ? ipBucket.getAvailableTokens() : ipLimit.getCapacity())
        .userResetTime(calculateResetTime(userBucket))
        .ipResetTime(calculateResetTime(ipBucket))
        .build();
}

@Scheduled(fixedRate = 300000) // Clean up every 5 minutes
public void cleanupExpiredBuckets() {
    userBuckets.entrySet().removeIf(entry -> entry.getValue().getAvailableTokens() == userLimit.getCapacity());
    ipBuckets.entrySet().removeIf(entry -> entry.getValue().getAvailableTokens() == ipLimit.getCapacity());
}
}
```

Week 15: Enterprise Integration & Multi-tenancy

java

```java
// MultiTenantConfiguration.java
@Configuration
@EnableJpaRepositories
public class MultiTenantConfiguration {

    @Bean
    @Primary
    public DataSource dataSource() {
        HikariConfig config = new HikariConfig();
        config.setJdbcUrl("jdbc:postgresql://localhost:5432/devgenie");
        config.setUsername("devgenie");
        config.setPassword("password");
        config.setMaximumPoolSize(20);
        config.setMinimumIdle(5);
        config.setConnectionTimeout(30000);
        config.setIdleTimeout(600000);
        config.setMaxLifetime(1800000);

        return new HikariDataSource(config);
    }

    @Bean
    public MultiTenantConnectionProvider multiTenantConnectionProvider() {
        return new DevGenieMultiTenantConnectionProvider(dataSource());
    }

    @Bean
    public CurrentTenantIdentifierResolver currentTenantIdentifierResolver() {
        return new DevGenieTenantResolver();
    }

    @Bean
    @Primary
    public LocalContainerEntityManagerFactoryBean entityManagerFactory() {
        LocalContainerEntityManagerFactoryBean em = new LocalContainerEntityManagerFactoryBean();
        em.setDataSource(dataSource());
        em.setPackagesToScan("com.org.devgenie");

        HibernateJpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();
        em.setJpaVendorAdapter(vendorAdapter);

        Map<String, Object> properties = new HashMap<>();
        properties.put("hibernate.dialect", "org.hibernate.dialect.PostgreSQLDialect");
        properties.put("hibernate.hbm2ddl.auto", "validate");
        properties.put("hibernate.multiTenancy", "}
```

Phase 3: Enhanced Analysis Services (Weeks 9-12)

Week 9: Coverage Analysis Service (Java)

java

```java
// CoverageAnalysisService.java
@Service
@Slf4j
public class CoverageAnalysisService {

    @Autowired
    private JaCoCoRunner jacocoRunner;

    @Autowired
    private TestGenerationService testGenerationService;

    @Value("${coverage.jacoco.exec.path:target/jacoco.exec}")
    private String jacocoExecPath;

    @Value("${coverage.reports.path:target/site/jacoco}")
    private String reportsPath;

    public CompletableFuture<CoverageReport> analyzeProjectCoverage(String projectPath) {
        return CompletableFuture.supplyAsync(() -> {
            try {
                log.info("Starting coverage analysis for project: {}", projectPath);

                // 1. Run tests with JaCoCo
                ExecutionResult testResult = jacocoRunner.runTestsWithCoverage(projectPath);

                if (!testResult.isSuccess()) {
                    throw new CoverageAnalysisException("Test execution failed: " + testResult.getErrorMessage());
                }

                // 2. Parse JaCoCo reports
                CoverageData coverageData = parseJaCoCoReports(projectPath);

                // 3. Identify uncovered areas
                List<UncoveredArea> uncoveredAreas = identifyUncoveredAreas(coverageData, projectPath);

                // 4. Calculate metrics
                CoverageMetrics metrics = calculateCoverageMetrics(coverageData);

                // 5. Generate improvement suggestions
                List<CoverageImprovement> improvements = generateImprovementSuggestions(uncoveredAreas, metri

                return CoverageReport.builder()
                    .projectPath(projectPath)
                    .overallCoverage(metrics.getOverallCoverage())
                    .lineCoverage(metrics.getLineCoverage())
                    .branchCoverage(metrics.getBranchCoverage())
```

```
            .methodCoverage(metrics.getMethodCoverage())
            .classCoverage(metrics.getClassCoverage())
            .uncoveredAreas(uncoveredAreas)
            .coverageByPackage(coverageData.getCoverageByPackage())
            .improvements(improvements)
            .generatedAt(LocalDateTime.now())
            .testExecutionTime(testResult.getExecutionTime())
            .build();

        } catch (Exception e) {
            log.error("Coverage analysis failed for project: {}", projectPath, e);
            throw new CoverageAnalysisException("Coverage analysis failed", e);
        }
    });
}

public CompletableFuture<TestGenerationResult> generateTestsForUncoveredCode(
        List<String> filePaths, double targetCoverage) {

    return CompletableFuture.supplyAsync(() -> {
        List<GeneratedTest> allTests = new ArrayList<>();
        Map<String, Double> projectedCoverage = new HashMap<>();

        for (String filePath : filePaths) {
            try {
                log.info("Generating tests for file: {}", filePath);

                // 1. Analyze current coverage for this file
                FileCoverageAnalysis fileAnalysis = analyzeFileCoverage(filePath);

                // 2. Identify methods/lines that need coverage
                List<UncoveredMethod> uncoveredMethods = fileAnalysis.getUncoveredMethods();
                List<Integer> uncoveredLines = fileAnalysis.getUncoveredLines();

                // 3. Generate tests for uncovered areas
                for (UncoveredMethod method : uncoveredMethods) {
                    String testCode = testGenerationService.generateTestForMethod(
                        fileAnalysis.getClassContent(),
                        method,
                        fileAnalysis.getDependencies()
                    );

                    GeneratedTest test = GeneratedTest.builder()
                        .sourceMethod(method)
                        .testCode(testCode)
                        .testMethodName(generateTestMethodName(method))
                        .estimatedCoverageImpact(calculateCoverageImpact(method, uncoveredLines))
```

```
                .complexity(assessTestComplexity(method))
                .build();

            allTests.add(test);
        }

        // 4. Project new coverage after tests
        double newCoverage = projectCoverageAfterTests(fileAnalysis, allTests);
        projectedCoverage.put(filePath, newCoverage);

      } catch (Exception e) {
        log.error("Failed to generate tests for file: {}", filePath, e);
      }
    }

    // 5. Optimize test selection for maximum coverage improvement
    List<GeneratedTest> optimizedTests = optimizeTestSelection(allTests, targetCoverage);

    return TestGenerationResult.builder()
      .generatedTests(optimizedTests)
      .totalTestsGenerated(optimizedTests.size())
      .projectedCoverageImprovement(calculateOverallCoverageImprovement(projectedCoverage))
      .estimatedExecutionTime(estimateTestExecutionTime(optimizedTests))
      .recommendations(generateTestingRecommendations(optimizedTests))
      .build();
  });
}

public List<CriticalGap> identifyCriticalCoverageGaps(String projectPath, double threshold) {
  List<CriticalGap> gaps = new ArrayList<>();

  try {
    CoverageReport report = analyzeProjectCoverage(projectPath).get();

    report.getCoverageByPackage().forEach((packageName, packageCoverage) -> {
      packageCoverage.getClasses().forEach((className, classCoverage) -> {
        if (classCoverage.getCoverage() < threshold) {

          // Assess business criticality
          BusinessCriticality criticality = assessBusinessCriticality(className, projectPath);

          // Calculate effort estimation
          EffortEstimation effort = estimateFixEffort(classCoverage);

          CriticalGap gap = CriticalGap.builder()
            .className(className)
            .packageName(packageName)
```

```
                .currentCoverage(classCoverage.getCoverage())
                .targetCoverage(threshold)
                .gapSize(threshold - classCoverage.getCoverage())
                .businessCriticality(criticality)
                .effortEstimation(effort)
                .uncoveredMethods(classCoverage.getUncoveredMethods())
                .riskLevel(calculateRiskLevel(criticality, classCoverage.getCoverage()))
                .recommendations(generateGapRecommendations(classCoverage))
                .build();

            gaps.add(gap);
          }
        });
    });

  } catch (Exception e) {
    log.error("Error identifying critical coverage gaps", e);
  }

  // Sort by priority (business criticality + gap size + risk level)
  return gaps.stream()
    .sorted(Comparator
      .comparing((CriticalGap gap) -> gap.getBusinessCriticality().getScore())
      .thenComparing(CriticalGap::getGapSize)
      .thenComparing(gap -> gap.getRiskLevel().getScore())
      .reversed())
    .collect(Collectors.toList());
}

private CoverageData parseJaCoCoReports(String projectPath) throws IOException {
  Path reportPath = Paths.get(projectPath, reportsPath, "jacoco.xml");

  if (!Files.exists(reportPath)) {
    throw new FileNotFoundException("JaCoCo report not found at: " + reportPath);
  }

  try {
    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
    DocumentBuilder builder = factory.newDocumentBuilder();
    Document doc = builder.parse(reportPath.toFile());

    XPath xpath = XPathFactory.newInstance().newXPath();

    // Parse overall coverage
    double overallLineCovered = parseDoubleFromXPath(xpath, "sum(//counter[@type='LINE']/@covered)", do
    double overallLineMissed = parseDoubleFromXPath(xpath, "sum(//counter[@type='LINE']/@missed)", doc)
    double overallBranchCovered = parseDoubleFromXPath(xpath, "sum(//counter[@type='BRANCH']/@cover
```

```
        double overallBranchMissed = parseDoubleFromXPath(xpath, "sum(//counter[@type='BRANCH']/@missed

        // Parse package-level coverage
        Map<String, PackageCoverage> packageCoverageMap = new HashMap<>();
        NodeList packageNodes = (NodeList) xpath.compile("//package").evaluate(doc, XPathConstants.NODESET

        for (int i = 0; i < packageNodes.getLength(); i++) {
            Node packageNode = packageNodes.item(i);
            String packageName = xpath.compile("@name").evaluate(packageNode);

            // Parse classes in this package
            Map<String, ClassCoverage> classCoverageMap = new HashMap<>();
            NodeList classNodes = (NodeList) xpath.compile(".//class").evaluate(packageNode, XPathConstants.NO

            for (int j = 0; j < classNodes.getLength(); j++) {
                Node classNode = classNodes.item(j);
                ClassCoverage classCoverage = parseClassCoverage(classNode, xpath);
                classCoverageMap.put(classCoverage.getClassName(), classCoverage);
            }

            PackageCoverage packageCoverage = PackageCoverage.builder()
                .packageName(packageName)
                .classes(classCoverageMap)
                .overallCoverage(calculatePackageCoverage(classCoverageMap))
                .build();

            packageCoverageMap.put(packageName, packageCoverage);
        }

        return CoverageData.builder()
            .overallLineCoverage(calculateCoverage(overallLineCovered, overallLineMissed))
            .overallBranchCoverage(calculateCoverage(overallBranchCovered, overallBranchMissed))
            .coverageByPackage(packageCoverageMap)
            .reportPath(reportPath.toString())
            .generatedAt(LocalDateTime.now())
            .build();

    } catch (Exception e) {
        throw new IOException("Failed to parse JaCoCo report", e);
    }
}

private ClassCoverage parseClassCoverage(Node classNode, XPath xpath) throws XPathExpressionException {
    String className = xpath.compile("@name").evaluate(classNode);

    // Parse method coverage
    List<MethodCoverage> methodCoverages = new ArrayList<>();
```

```
    NodeList methodNodes = (NodeList) xpath.compile(".//method").evaluate(classNode, XPathConstants.NODE

    for (int i = 0; i < methodNodes.getLength(); i++) {
        Node methodNode = methodNodes.item(i);
        MethodCoverage methodCoverage = parseMethodCoverage(methodNode, xpath);
        methodCoverages.add(methodCoverage);
    }

    // Calculate class-level coverage
    double classLineCovered = parseDoubleFromXPath(xpath, "counter[@type='LINE']/@covered", classNode);
    double classLineMissed = parseDoubleFromXPath(xpath, "counter[@type='LINE']/@missed", classNode);

    List<UncoveredMethod> uncoveredMethods = methodCoverages.stream()
        .filter(method -> method.getCoverage() < 50) // Consider <50% as uncovered
        .map(method -> UncoveredMethod.builder()
            .methodName(method.getMethodName())
            .lineNumbers(method.getUncoveredLines())
            .complexity(method.getComplexity())
            .build())
        .collect(Collectors.toList());

    return ClassCoverage.builder()
        .className(className)
        .coverage(calculateCoverage(classLineCovered, classLineMissed))
        .methodCoverages(methodCoverages)
        .uncoveredMethods(uncoveredMethods)
        .totalMethods(methodCoverages.size())
        .coveredMethods((int) methodCoverages.stream().filter(m -> m.getCoverage() >= 50).count())
        .build();
}

private MethodCoverage parseMethodCoverage(Node methodNode, XPath xpath) throws XPathExpressionExce
    String methodName = xpath.compile("@name").evaluate(methodNode);
    String methodDesc = xpath.compile("@desc").evaluate(methodNode);

    double lineCovered = parseDoubleFromXPath(xpath, "counter[@type='LINE']/@covered", methodNode);
    double lineMissed = parseDoubleFromXPath(xpath, "counter[@type='LINE']/@missed", methodNode);

    // Parse line-level coverage details if available
    List<Integer> uncoveredLines = new ArrayList<>();
    // This would require more detailed JaCoCo parsing or integration with JaCoCo API

    return MethodCoverage.builder()
        .methodName(methodName)
        .methodSignature(methodDesc)
        .coverage(calculateCoverage(lineCovered, lineMissed))
        .totalLines((int) (lineCovered + lineMissed))
```

```
            .coveredLines((int) lineCovered)
            .uncoveredLines(uncoveredLines)
            .complexity(1) // Would need to integrate with complexity analysis
            .build();
    }

    private double parseDoubleFromXPath(XPath xpath, String expression, Node context) throws XPathExpressionE
        try {
            Number result = (Number) xpath.compile(expression).evaluate(context, XPathConstants.NUMBER);
            return result != null ? result.doubleValue() : 0.0;
        } catch (Exception e) {
            return 0.0;
        }
    }

    private double calculateCoverage(double covered, double missed) {
        double total = covered + missed;
        return total > 0 ? (covered / total) * 100 : 0.0;
    }

    private FileCoverageAnalysis analyzeFileCoverage(String filePath) throws IOException {
        // Read and analyze individual file
        String classContent = Files.readString(Paths.get(filePath));

        // Parse class structure
        CompilationUnit cu = StaticJavaParser.parse(classContent);

        // Extract methods and their line numbers
        List<MethodDeclaration> methods = cu.findAll(MethodDeclaration.class);

        // Analyze dependencies
        List<String> dependencies = cu.findAll(ImportDeclaration.class)
            .stream()
            .map(ImportDeclaration::getNameAsString)
            .collect(Collectors.toList());

        return FileCoverageAnalysis.builder()
            .filePath(filePath)
            .classContent(classContent)
            .methods(methods)
            .dependencies(dependencies)
            .uncoveredMethods(new ArrayList<>()) // Would be populated from JaCoCo data
            .uncoveredLines(new ArrayList<>()) // Would be populated from JaCoCo data
            .build();
    }

    private BusinessCriticality assessBusinessCriticality(String className, String projectPath) {
```

```
// Analyze class importance based on various factors
int score = 0;
List<String> factors = new ArrayList<>();

// Check if it's a controller/service/repository
if (className.toLowerCase().contains("controller")) {
    score += 30;
    factors.add("REST Controller - High user impact");
}
if (className.toLowerCase().contains("service")) {
    score += 25;
    factors.add("Business Service - Core functionality");
}
if (className.toLowerCase().contains("repository")) {
    score += 20;
    factors.add("Data Access - Critical for data integrity");
}

// Check method complexity and usage
try {
    String classContent = Files.readString(Paths.get(projectPath, className));
    CompilationUnit cu = StaticJavaParser.parse(classContent);

    long publicMethods = cu.findAll(MethodDeclaration.class).stream()
        .filter(m -> m.isPublic())
        .count();

    if (publicMethods > 10) {
        score += 15;
        factors.add("High number of public methods");
    }

} catch (Exception e) {
    log.warn("Could not analyze class content for {}", className);
}

CriticalityLevel level;
if (score >= 50) level = CriticalityLevel.CRITICAL;
else if (score >= 30) level = CriticalityLevel.HIGH;
else if (score >= 15) level = CriticalityLevel.MEDIUM;
else level = CriticalityLevel.LOW;

return BusinessCriticality.builder()
    .score(score)
    .level(level)
    .factors(factors)
    .build();
```

```
    }
}

// Supporting classes and enums
@Data
@Builder
public class CoverageReport {
    private String projectPath;
    private double overallCoverage;
    private double lineCoverage;
    private double branchCoverage;
    private double methodCoverage;
    private double classCoverage;
    private List<UncoveredArea> uncoveredAreas;
    private Map<String, PackageCoverage> coverageByPackage;
    private List<CoverageImprovement> improvements;
    private LocalDateTime generatedAt;
    private Duration testExecutionTime;
}

@Data
@Builder
public class CriticalGap {
    private String className;
    private String packageName;
    private double currentCoverage;
    private double targetCoverage;
    private double gapSize;
    private BusinessCriticality businessCriticality;
    private EffortEstimation effortEstimation;
    private List<UncoveredMethod> uncoveredMethods;
    private RiskLevel riskLevel;
    private List<String> recommendations;
}

@Data
@Builder
public class TestGenerationResult {
    private List<GeneratedTest> generatedTests;
    private int totalTestsGenerated;
    private double projectedCoverageImprovement;
    private Duration estimatedExecutionTime;
    private List<String> recommendations;
}

public enum CriticalityLevel {
    CRITICAL(4), HIGH(3), MEDIUM(2), LOW(1);
```

```
    private final int score;

    CriticalityLevel(int score) {
        this.score = score;
    }

    public int getScore() { return score; }
}

public enum RiskLevel {
    VERY_HIGH(5), HIGH(4), MEDIUM(3), LOW(2), VERY_LOW(1);

    private final int score;

    RiskLevel(int score) {
        this.score = score;
    }

    public int getScore() { return score; }
}
```

Week 10: Test Generation Service (Java)

java

```java
// TestGenerationService.java
@Service
@Slf4j
public class TestGenerationService {

    @Autowired
    private SpringAiCommentGenerator aiGenerator;

    @Autowired
    private TestTemplateEngine templateEngine;

    @Autowired
    private CodeAnalysisService codeAnalysisService;

    public String generateTestForMethod(String classContent, UncoveredMethod method, List<String> dependenci
        try {
            log.info("Generating test for method: {}", method.getMethodName());

            // 1. Analyze method structure and complexity
            MethodAnalysis analysis = codeAnalysisService.analyzeMethod(classContent, method.getMethodName());

            // 2. Determine test strategy based on method characteristics
            TestStrategy strategy = determineTestStrategy(analysis);

            // 3. Generate test using AI with specific strategy
            String testCode = generateTestWithAI(classContent, method, analysis, strategy, dependencies);

            // 4. Validate and enhance generated test
            String enhancedTest = enhanceGeneratedTest(testCode, analysis, strategy);

            return enhancedTest;

        } catch (Exception e) {
            log.error("Failed to generate test for method: {}", method.getMethodName(), e);
            return generateFallbackTest(method);
        }
    }

    public List<GeneratedTest> generateTestsForClass(String classPath, double targetCoverage) {
        try {
            String classContent = Files.readString(Paths.get(classPath));
            CompilationUnit cu = StaticJavaParser.parse(classContent);

            // Find class declaration
            Optional<ClassOrInterfaceDeclaration> classDecl = cu.findFirst(ClassOrInterfaceDeclaration.class);
            if (classDecl.isEmpty()) {
```

```
            throw new IllegalArgumentException("No class declaration found in file: " + classPath);
        }

        String className = classDecl.get().getNameAsString();
        List<GeneratedTest> generatedTests = new ArrayList<>();

        // Get all public methods that need testing
        List<MethodDeclaration> methodsToTest = cu.findAll(MethodDeclaration.class)
            .stream()
            .filter(this::shouldGenerateTestFor)
            .collect(Collectors.toList());

        for (MethodDeclaration method : methodsToTest) {
            try {
                // Analyze method for test generation
                MethodTestingContext context = buildMethodTestingContext(method, classContent, cu);

                // Generate comprehensive test cases
                List<TestCase> testCases = generateTestCases(context);

                // Convert to generated test objects
                for (TestCase testCase : testCases) {
                    String testCode = generateTestCode(testCase, context);

                    GeneratedTest generatedTest = GeneratedTest.builder()
                        .sourceMethod(createUncoveredMethod(method))
                        .testCode(testCode)
                        .testMethodName(testCase.getTestMethodName())
                        .testType(testCase.getType())
                        .estimatedCoverageImpact(calculateCoverageImpact(method))
                        .complexity(assessTestComplexity(testCase))
                        .description(testCase.getDescription())
                        .build();

                    generatedTests.add(generatedTest);
                }

            } catch (Exception e) {
                log.error("Failed to generate test for method: {}", method.getNameAsString(), e);
            }
        }

        return optimizeTestsForCoverage(generatedTests, targetCoverage);

    } catch (Exception e) {
        log.error("Failed to generate tests for class: {}", classPath, e);
        return Collections.emptyList();
```

```
    }
}

private TestStrategy determineTestStrategy(MethodAnalysis analysis) {
    TestStrategy.Builder strategyBuilder = TestStrategy.builder();

    // Determine based on method characteristics
    if (analysis.hasComplexBranching()) {
        strategyBuilder.includeEdgeCases(true)
                .includeBoundaryTests(true)
                .branchCoverageTarget(90);
    }

    if (analysis.hasExceptionHandling()) {
        strategyBuilder.includeExceptionTests(true)
                .negativeTestingRequired(true);
    }

    if (analysis.hasExternalDependencies()) {
        strategyBuilder.mockingStrategy(MockingStrategy.COMPREHENSIVE)
                .includeDependencyTests(true);
    }

    if (analysis.isDataProcessor()) {
        strategyBuilder.includeDataVariationTests(true)
                .includeNullSafetyTests(true);
    }

    if (analysis.getCyclomaticComplexity() > 10) {
        strategyBuilder.complexityReductionRequired(true)
                .includeIntegrationTests(true);
    }

    return strategyBuilder.build();
}

private String generateTestWithAI(String classContent, UncoveredMethod method,
                                  MethodAnalysis analysis, TestStrategy strategy,
                                  List<String> dependencies) {

    String enhancedPrompt = buildTestGenerationPrompt(classContent, method, analysis, strategy, dependencie

    return aiGenerator.generateTestCode(enhancedPrompt);
}

private String buildTestGenerationPrompt(String classContent, UncoveredMethod method,
                                         MethodAnalysis analysis, TestStrategy strategy,
```

```
                        List<String> dependencies) {

StringBuilder prompt = new StringBuilder();

prompt.append("Generate comprehensive JUnit 5 test methods for the following Java method:\n\n");

// Add class context
prompt.append("CLASS CONTEXT:\n");
prompt.append(classContent).append("\n\n");

// Add method details
prompt.append("TARGET METHOD: ").append(method.getMethodName()).append("\n");
prompt.append("METHOD COMPLEXITY: ").append(analysis.getCyclomaticComplexity()).append("\n");
prompt.append("HAS BRANCHES: ").append(analysis.hasComplexBranching()).append("\n");
prompt.append("HAS EXCEPTIONS: ").append(analysis.hasExceptionHandling()).append("\n");
prompt.append("EXTERNAL DEPENDENCIES: ").append(!dependencies.isEmpty()).append("\n\n");

// Add test strategy
prompt.append("TEST STRATEGY:\n");
if (strategy.isIncludeEdgeCases()) {
    prompt.append("- Include edge cases and boundary conditions\n");
}
if (strategy.isIncludeExceptionTests()) {
    prompt.append("- Test exception scenarios and error handling\n");
}
if (strategy.isIncludeNullSafetyTests()) {
    prompt.append("- Test null safety and defensive programming\n");
}
if (strategy.getMockingStrategy() == MockingStrategy.COMPREHENSIVE) {
    prompt.append("- Use Mockito for mocking external dependencies\n");
}

// Add requirements
prompt.append("\nREQUIREMENTS:\n");
prompt.append("1. Generate multiple test methods covering different scenarios\n");
prompt.append("2. Use JUnit 5 annotations (@Test, @ParameterizedTest, etc.)\n");
prompt.append("3. Include proper assertions using AssertJ or JUnit assertions\n");
prompt.append("4. Add meaningful test method names that describe what is being tested\n");
prompt.append("5. Include setup (@BeforeEach) and teardown (@AfterEach) if needed\n");
prompt.append("6. Use Mockito for mocking when dealing with dependencies\n");
prompt.append("7. Ensure tests are independent and can run in any order\n");
prompt.append("8. Add JavaDoc comments explaining test scenarios\n");
prompt.append("9. Target ").append(strategy.getBranchCoverageTarget()).append("% branch coverage\n");

// Add dependencies information
if (!dependencies.isEmpty()) {
    prompt.append("\nAVAILABLE DEPENDENCIES:\n");
```

```
        dependencies.forEach(dep -> prompt.append("- ").append(dep).append("\n"));
    }

    prompt.append("\nGENERATE COMPLETE TEST CLASS WITH ALL NECESSARY IMPORTS:\n");

    return prompt.toString();
}

private List<TestCase> generateTestCases(MethodTestingContext context) {
    List<TestCase> testCases = new ArrayList<>();

    MethodDeclaration method = context.getMethod();
    MethodAnalysis analysis = context.getAnalysis();

    // 1. Happy path test
    testCases.add(TestCase.builder()
        .testMethodName("test" + capitalize(method.getNameAsString()) + "_HappyPath")
        .type(TestType.HAPPY_PATH)
        .description("Test normal execution with valid inputs")
        .inputParameters(generateHappyPathInputs(method))
        .expectedBehavior("Method executes successfully and returns expected result")
        .build());

    // 2. Edge cases
    if (analysis.hasComplexBranching()) {
        testCases.addAll(generateBranchingTestCases(method, analysis));
    }

    // 3. Exception tests
    if (analysis.hasExceptionHandling()) {
        testCases.addAll(generateExceptionTestCases(method, analysis));
    }

    // 4. Null safety tests
    testCases.addAll(generateNullSafetyTestCases(method));

    // 5. Boundary tests
    testCases.addAll(generateBoundaryTestCases(method, analysis));

    // 6. Integration tests if method has dependencies
    if (analysis.hasExternalDependencies()) {
        testCases.addAll(generateIntegrationTestCases(method, context));
    }

    return testCases;
}
```

```
private List<TestCase> generateBranchingTestCases(MethodDeclaration method, MethodAnalysis analysis) {
    List<TestCase> testCases = new ArrayList<>();

    // Analyze if-else conditions
    List<IfStmt> ifStatements = method.findAll(IfStmt.class);

    for (int i = 0; i < ifStatements.size(); i++) {
        IfStmt ifStmt = ifStatements.get(i);

        // Test true branch
        testCases.add(TestCase.builder()
            .testMethodName("test" + capitalize(method.getNameAsString()) + "_Branch" + (i + 1) + "_True")
            .type(TestType.BRANCH_COVERAGE)
            .description("Test execution when condition " + (i + 1) + " is true")
            .inputParameters(generateInputsForCondition(ifStmt.getCondition(), true))
            .expectedBehavior("Execute true branch of condition " + (i + 1))
            .build());

        // Test false branch
        testCases.add(TestCase.builder()
            .testMethodName("test" + capitalize(method.getNameAsString()) + "_Branch" + (i + 1) + "_False")
            .type(TestType.BRANCH_COVERAGE)
            .description("Test execution when condition " + (i + 1) + " is false")
            .inputParameters(generateInputsForCondition(ifStmt.getCondition(), false))
            .expectedBehavior("Execute false branch of condition " + (i + 1))
            .build());
    }

    return testCases;
}

private List<TestCase> generateExceptionTestCases(MethodDeclaration method, MethodAnalysis analysis) {
    List<TestCase> testCases = new ArrayList<>();

    // Find throw statements
    List<ThrowStmt> throwStmts = method.findAll(ThrowStmt.class);

    for (int i = 0; i < throwStmts.size(); i++) {
        ThrowStmt throwStmt = throwStmts.get(i);
        String exceptionType = extractExceptionType(throwStmt);

        testCases.add(TestCase.builder            messageInput.addEventListener('keydown', (e) => {
            if (e.key === 'Enter' && !e.shiftKey) {
                e.preventDefault();
                sendMessage();
            }
        });
```

```
// Handle messages from extension
window.addEventListener('message', event => {
    const message = event.data;

    switch (message.command) {
        case 'chatResponse':
            setLoading(false);
            addMessage('assistant', message.response.message,
                    message.response.actions, message.response.suggestions);
            break;

        case 'chatError':
            setLoading(false);
            addMessage('assistant', 'Sorry, I encountered an error: ' + message.error);
            break;

        case 'actionResult':
            addMessage('assistant', message.result.message);
            break;

        case 'progressUpdate':
            updateProgress(message.operationId, message.status);
            break;

        case 'initialized':
            addMessage('assistant',
                "Hi! I'm DevGenie, your AI assistant for technical debt management. I can help you fix SonarQut
                null,
                [
                    { text: "Fix critical issues", action: "FIX_ISSUES" },
                    { text: "Analyze test coverage", action: "ANALYZE_COVERAGE" },
                    { text: "Show me technical debt", action: "ANALYZE_TECH_DEBT" }
                ]
            );
            break;
    }
});

function updateProgress(operationId, status) {
    // Find existing progress message or create new one
    let progressDiv = document.getElementById('progress-' + operationId);
    if (!progressDiv) {
        progressDiv = document.createElement('div');
        progressDiv.id = 'progress-' + operationId;
        progressDiv.className = 'message assistant';
        messagesContainer.appendChild(progressDiv);
```

```
            }

            // Update progress content
            progressDiv.innerHTML = '<div><strong>Fix Progress:</strong></div>';
            status.step.forEach(step => {
                const stepDiv = document.createElement('div');
                stepDiv.innerHTML = step;
                stepDiv.style.fontSize = '12px';
                stepDiv.style.marginLeft = '10px';
                progressDiv.appendChild(stepDiv);
            });

            messagesContainer.scrollTop = messagesContainer.scrollHeight;
        }
    </script>
</body>
</html>
`;
    }
}
```

Week 7: Quick Fix Provider

typescript

```typescript
// providers/quickfix-provider.ts
import * as vscode from 'vscode';
import { DevGenieAPI } from '../api/devgenie-api';

export class QuickFixProvider {
    constructor(private api: DevGenieAPI) {}

    public async showQuickFix() {
        const editor = vscode.window.activeTextEditor;
        if (!editor) {
            vscode.window.showWarningMessage('No active editor found.');
            return;
        }

        const document = editor.document;
        const selection = editor.selection;

        if (selection.isEmpty) {
            vscode.window.showInformationMessage('Please select code to analyze.');
            return;
        }

        const selectedText = document.getText(selection);

        try {
            await vscode.window.withProgress({
                location: vscode.ProgressLocation.Notification,
                title: "DevGenie is analyzing your code...",
                cancellable: true
            }, async (progress, token) => {

                const analysis = await this.api.analyzeCodeDelta({
                    code: selectedText,
                    filePath: document.fileName,
                    lineNumber: selection.start.line,
                    context: await this.getFileContext(document)
                });

                if (token.isCancellationRequested) {
                    return;
                }

                await this.showAnalysisResults(analysis, editor, selection);
            });

        } catch (error) {
```

```
        vscode.window.showErrorMessage(`Analysis failed: ${error.message}`);
    }
}

private async showAnalysisResults(analysis: any, editor: vscode.TextEditor, selection: vscode.Selection) {
    const items: vscode.QuickPickItem[] = [];

    // Add potential issues found
    if (analysis.issues && analysis.issues.length > 0) {
        items.push({
            label: "$(warning) Issues Found",
            description: `${analysis.issues.length} potential issues detected`,
            kind: vscode.QuickPickItemKind.Separator
        });

        analysis.issues.forEach((issue: any, index: number) => {
            items.push({
                label: `$(bug) ${issue.title}`,
                description: issue.description,
                detail: `Severity: ${issue.severity} | Line: ${issue.line}`,
                picked: false,
                alwaysShow: true
            });
        });
    }

    // Add improvement suggestions
    if (analysis.suggestions && analysis.suggestions.length > 0) {
        items.push({
            label: "$(lightbulb) Suggestions",
            description: `${analysis.suggestions.length} improvement suggestions`,
            kind: vscode.QuickPickItemKind.Separator
        });

        analysis.suggestions.forEach((suggestion: any) => {
            items.push({
                label: `$(gear) ${suggestion.title}`,
                description: suggestion.description,
                detail: `Impact: ${suggestion.impact} | Effort: ${suggestion.effort}`,
                picked: false,
                alwaysShow: true
            });
        });
    }

    // Add quick actions
    items.push({
```

```
        label: "$(zap) Quick Actions",
        description: "Available quick actions",
        kind: vscode.QuickPickItemKind.Separator
    });

    items.push({
        label: "$(tools) Auto-fix all issues",
        description: "Let DevGenie fix all detected issues automatically",
        detail: "This will modify your code",
        picked: false
    });

    items.push({
        label: "$(comment) Add documentation",
        description: "Generate documentation for selected code",
        detail: "Adds JavaDoc comments",
        picked: false
    });

    items.push({
        label: "$(beaker) Generate tests",
        description: "Generate unit tests for selected code",
        detail: "Creates test methods in test file",
        picked: false
    });

    if (items.length === 0) {
        vscode.window.showInformationMessage('No issues or suggestions found for the selected code.');
        return;
    }

    const selected = await vscode.window.showQuickPick(items, {
        placeHolder: 'Select an action to perform',
        canPickMany: false,
        matchOnDescription: true,
        matchOnDetail: true
    });

    if (selected) {
        await this.executeQuickPickAction(selected, analysis, editor, selection);
    }
}

private async executeQuickPickAction(
    selected: vscode.QuickPickItem,
    analysis: any,
    editor: vscode.TextEditor,
```

```
    selection: vscode.Selection
) {
    const label = selected.label;

    if (label.includes("Auto-fix all issues")) {
        await this.autoFixIssues(analysis.issues, editor, selection);
    } else if (label.includes("Add documentation")) {
        await this.generateDocumentation(editor, selection);
    } else if (label.includes("Generate tests")) {
        await this.generateTests(editor, selection);
    } else if (label.includes("$(bug)")) {
        // Handle specific issue selection
        const issueIndex = this.extractIssueIndex(label);
        if (issueIndex >= 0 && analysis.issues[issueIndex]) {
            await this.fixSpecificIssue(analysis.issues[issueIndex], editor, selection);
        }
    }
}

private async autoFixIssues(issues: any[], editor: vscode.TextEditor, selection: vscode.Selection) {
    const selectedText = editor.document.getText(selection);

    try {
        const result = await this.api.fixCodeDelta({
            code: selectedText,
            issues: issues,
            filePath: editor.document.fileName,
            preferences: this.getUserPreferences()
        });

        if (result.fixedCode) {
            const edit = new vscode.WorkspaceEdit();
            edit.replace(editor.document.uri, selection, result.fixedCode);

            const applied = await vscode.workspace.applyEdit(edit);

            if (applied) {
                vscode.window.showInformationMessage(
                    `DevGenie fixed ${issues.length} issues successfully!`,
                    'View Changes'
                ).then(choice => {
                    if (choice === 'View Changes') {
                        vscode.commands.executeCommand('workbench.files.action.compareFileWith');
                    }
                });
            } else {
                vscode.window.showErrorMessage('Failed to apply fixes to the document.');
```

```
            }
        }

    } catch (error) {
        vscode.window.showErrorMessage(`Auto-fix failed: ${error.message}`);
    }
}

private async generateDocumentation(editor: vscode.TextEditor, selection: vscode.Selection) {
    const selectedText = editor.document.getText(selection);

    try {
        const result = await this.api.generateDocumentation({
            code: selectedText,
            filePath: editor.document.fileName,
            documentationType: 'javadoc'
        });

        if (result.documentation) {
            // Insert documentation before the selected code
            const insertPosition = new vscode.Position(selection.start.line, 0);
            const edit = new vscode.WorkspaceEdit();
            edit.insert(editor.document.uri, insertPosition, result.documentation + '\n');

            await vscode.workspace.applyEdit(edit);

            vscode.window.showInformationMessage('Documentation added successfully!');
        }

    } catch (error) {
        vscode.window.showErrorMessage(`Documentation generation failed: ${error.message}`);
    }
}

private async generateTests(editor: vscode.TextEditor, selection: vscode.Selection) {
    const selectedText = editor.document.getText(selection);

    try {
        const result = await this.api.generateTests({
            code: selectedText,
            filePath: editor.document.fileName,
            testFramework: 'junit5'
        });

        if (result.testCode) {
            // Create or open test file
            const testFilePath = this.getTestFilePath(editor.document.fileName);
```

```
            const testUri = vscode.Uri.file(testFilePath);

            try {
                const testDoc = await vscode.workspace.openTextDocument(testUri);
                const testEditor = await vscode.window.showTextDocument(testDoc);

                // Append test to existing file
                const lastLine = testDoc.lineCount - 1;
                const insertPosition = new vscode.Position(lastLine, testDoc.lineAt(lastLine).text.length);

                const edit = new vscode.WorkspaceEdit();
                edit.insert(testUri, insertPosition, '\n\n' + result.testCode);

                await vscode.workspace.applyEdit(edit);

            } catch (error) {
                // File doesn't exist, create new one
                const edit = new vscode.WorkspaceEdit();
                edit.createFile(testUri);
                edit.insert(testUri, new vscode.Position(0, 0), result.testCode);

                await vscode.workspace.applyEdit(edit);
                await vscode.window.showTextDocument(testUri);
            }

            vscode.window.showInformationMessage('Test methods generated successfully!');
        }

    } catch (error) {
        vscode.window.showErrorMessage(`Test generation failed: ${error.message}`);
    }
}

private async getFileContext(document: vscode.TextDocument): Promise<any> {
    // Get imports, class declarations, etc.
    const fullText = document.getText();
    const lines = fullText.split('\n');

    const imports = lines.filter(line => line.trim().startsWith('import'));
    const packageDeclaration = lines.find(line => line.trim().startsWith('package'));

    return {
        packageName: packageDeclaration?.replace('package ', '').replace(';', '').trim(),
        imports: imports,
        fileName: document.fileName,
        totalLines: lines.length
    };
```

```
    }

    private getUserPreferences(): any {
        const config = vscode.workspace.getConfiguration('devgenie');
        return {
            codingStyle: 'standard',
            addComments: true,
            optimizeImports: true
        };
    }

    private getTestFilePath(sourceFilePath: string): string {
        // Convert source file path to test file path
        // src/main/java/com/example/Service.java -> src/test/java/com/example/ServiceTest.java

        const testPath = sourceFilePath
            .replace('/src/main/java/', '/src/test/java/')
            .replace('.java', 'Test.java');

        return testPath;
    }

    private extractIssueIndex(label: string): number {
        // Extract index from issue label for specific issue handling
        // This is a simplified implementation
        return -1;
    }
}
```

Week 8: Coverage Provider & API Client

typescript

```typescript
// providers/coverage-provider.ts
import * as vscode from 'vscode';
import { DevGenieAPI } from '../api/devgenie-api';

export class CoverageProvider {
  constructor(private api: DevGenieAPI) {}

  public async analyzeCoverage(uri?: vscode.Uri) {
    const workspaceFolder = vscode.workspace.workspaceFolders?.[0];
    if (!workspaceFolder) {
      vscode.window.showErrorMessage('No workspace folder found.');
      return;
    }

    const targetPath = uri?.fsPath || workspaceFolder.uri.fsPath;

    try {
      await vscode.window.withProgress({
        location: vscode.ProgressLocation.Notification,
        title: "Analyzing code coverage...",
        cancellable: false
      }, async (progress) => {

        progress.report({ message: "Running coverage analysis..." });

        const coverageReport = await this.api.analyzeCoverage({
          projectPath: targetPath,
          includeDetails: true
        });

        progress.report({ message: "Generating report..." });

        await this.showCoverageReport(coverageReport);
      });

    } catch (error) {
      vscode.window.showErrorMessage(`Coverage analysis failed: ${error.message}`);
    }
  }

  private async showCoverageReport(report: any) {
    // Create webview panel for coverage report
    const panel = vscode.window.createWebviewPanel(
      'coverageReport',
      'Coverage Report',
      vscode.ViewColumn.Beside,
```

```
        { enableScripts: true }
    );

    panel.webview.html = this.getCoverageReportHtml(report);

    // Handle webview messages
    panel.webview.onDidReceiveMessage(async (message) => {
        switch (message.command) {
            case 'generateTests':
                await this.generateTestsForUncovered(message.uncoveredAreas);
                break;
            case 'openFile':
                await this.openFileAtLine(message.filePath, message.lineNumber);
                break;
        }
    });
}

private getCoverageReportHtml(report: any): string {
    const overallCoverage = report.overallCoverage || 0;
    const lineCoverage = report.lineCoverage || 0;
    const branchCoverage = report.branchCoverage || 0;

    return `
    <!DOCTYPE html>
    <html>
    <head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <title>Coverage Report</title>
        <style>
            body {
                font-family: var(--vscode-font-family);
                color: var(--vscode-editor-foreground);
                background-color: var(--vscode-editor-background);
                margin: 0;
                padding: 20px;
            }

            .coverage-summary {
                display: grid;
                grid-template-columns: repeat(auto-fit, minmax(200px, 1fr));
                gap: 20px;
                margin-bottom: 30px;
            }

            .coverage-metric {
```

```
    background-color: var(--vscode-panel-background);
    border: 1px solid var(--vscode-panel-border);
    border-radius: 8px;
    padding: 20px;
    text-align: center;
}

.metric-value {
    font-size: 36px;
    font-weight: bold;
    margin-bottom: 5px;
}

.metric-label {
    color: var(--vscode-descriptionForeground);
    font-size: 14px;
}

.coverage-high { color: #28a745; }
.coverage-medium { color: #ffc107; }
.coverage-low { color: #dc3545; }

.uncovered-areas {
    margin-top: 30px;
}

.uncovered-file {
    background-color: var(--vscode-panel-background);
    border: 1px solid var(--vscode-panel-border);
    border-radius: 4px;
    margin-bottom: 10px;
    padding: 15px;
}

.file-header {
    display: flex;
    justify-content: between;
    align-items: center;
    margin-bottom: 10px;
}

.file-name {
    font-weight: bold;
    color: var(--vscode-textLink-foreground);
    cursor: pointer;
}
```

```
    .file-coverage {
        font-size: 12px;
        color: var(--vscode-descriptionForeground);
    }

    .uncovered-lines {
        margin-top: 10px;
        font-size: 12px;
    }

    .line-number {
        background-color: var(--vscode-badge-background);
        color: var(--vscode-badge-foreground);
        padding: 2px 6px;
        border-radius: 4px;
        margin-right: 5px;
        cursor: pointer;
    }

    .action-button {
        background-color: var(--vscode-button-background);
        color: var(--vscode-button-foreground);
        border: none;
        padding: 8px 16px;
        border-radius: 4px;
        cursor: pointer;
        margin-top: 10px;
    }

    .action-button:hover {
        background-color: var(--vscode-button-hoverBackground);
    }

    .progress-bar {
        width: 100%;
        height: 8px;
        background-color: var(--vscode-progressBar-background);
        border-radius: 4px;
        overflow: hidden;
        margin-top: 10px;
    }

    .progress-fill {
        height: 100%;
        transition: width 0.3s ease;
    }
</style>
```

```
</head>
<body>
  <h1>Code Coverage Report</h1>

  <div class="coverage-summary">
    <div class="coverage-metric">
      <div class="metric-value ${this.getCoverageClass(overallCoverage)}">
        ${overallCoverage.toFixed(1)}%
      </div>
      <div class="metric-label">Overall Coverage</div>
      <div class="progress-bar">
        <div class="progress-fill ${this.getCoverageClass(overallCoverage)}"
          style="width: ${overallCoverage}%; background-color: ${this.getCoverageColor(overallCoverage
      </div>
    </div>

    <div class="coverage-metric">
      <div class="metric-value ${this.getCoverageClass(lineCoverage)}">
        ${lineCoverage.toFixed(1)}%
      </div>
      <div class="metric-label">Line Coverage</div>
      <div class="progress-bar">
        <div class="progress-fill"
          style="width: ${lineCoverage}%; background-color: ${this.getCoverageColor(lineCoverage)};"></
      </div>
    </div>

    <div class="coverage-metric">
      <div class="metric-value ${this.getCoverageClass(branchCoverage)}">
        ${branchCoverage.toFixed(1)}%
      </div>
      <div class="metric-label">Branch Coverage</div>
      <div class="progress-bar">
        <div class="progress-fill"
          style="width: ${branchCoverage}%; background-color: ${this.getCoverageColor(branchCoverag
      </div>
    </div>
  </div>

  ${this.generateUncoveredAreasHtml(report.uncoveredAreas || [])}

  <script>
    const vscode = acquireVsCodeApi();

    function generateTestsForFile(filePath, uncoveredLines) {
      vscode.postMessage({
        command: 'generateTests',
```

```
                uncoveredAreas: [{
                    filePath: filePath,
                    uncoveredLines: uncoveredLines
                }]
            });
        }

        function openFile(filePath, lineNumber = 1) {
            vscode.postMessage({
                command: 'openFile',
                filePath: filePath,
                lineNumber: lineNumber
            });
        }
    </script>
</body>
</html>
`;
}

private getCoverageClass(coverage: number): string {
    if (coverage >= 80) return 'coverage-high';
    if (coverage >= 60) return 'coverage-medium';
    return 'coverage-low';
}

private getCoverageColor(coverage: number): string {
    if (coverage >= 80) return '#28a745';
    if (coverage >= 60) return '#ffc107';
    return '#dc3545';
}

private generateUncoveredAreasHtml(uncoveredAreas: any[]): string {
    if (!uncoveredAreas || uncoveredAreas.length === 0) {
        return '<div class="uncovered-areas"><h2>🎉 No uncovered areas found!</h2></div>';
    }

    let html = '<div class="uncovered-areas"><h2>Uncovered Areas</h2>';

    uncoveredAreas.forEach(area => {
        html += `
        <div class="uncovered-file">
            <div class="file-header">
                <span class="file-name" onclick="openFile('${area.filePath}')">
                    ${area.fileName || area.filePath}
                </span>
                <span class="file-coverage">
```

```
            ${area.coveragePercentage || 0}% covered
          </span>
        </div>

        <div class="uncovered-lines">
          <strong>Uncovered lines:</strong>
          ${(area.uncoveredLines || []).map(line =>
            `<span class="line-number" onclick="openFile('${area.filePath}', ${line})">${line}</span>`
          ).join('')}
        </div>

        <button class="action-button"
            onclick="generateTestsForFile('${area.filePath}', ${JSON.stringify(area.uncoveredLines || [])})">
          Generate Tests for This File
        </button>
      </div>
      `;
    });

    html += '</div>';
    return html;
}

private async generateTestsForUncovered(uncoveredAreas: any[]) {
    try {
        for (const area of uncoveredAreas) {
            const result = await this.api.generateTestsForCoverage({
                filePath: area.filePath,
                uncoveredLines: area.uncoveredLines,
                targetCoverage: 80
            });

            if (result.testCode) {
                // Create or update test file
                await this.createOrUpdateTestFile(area.filePath, result.testCode);
            }
        }

        vscode.window.showInformationMessage(
            `Generated tests for ${uncoveredAreas.length} files!`,
            'Run Tests'
        ).then(choice => {
            if (choice === 'Run Tests') {
                vscode.commands.executeCommand('java.test.runAll');
            }
        });
```

```
    } catch (error) {
        vscode.window.showErrorMessage(`Test generation failed: ${error.message}`);
    }
}

private async openFileAtLine(filePath: string, lineNumber: number = 1) {
    try {
        const uri = vscode.Uri.file(filePath);
        const document = await vscode.workspace.openTextDocument(uri);
        const editor = await vscode.window.showTextDocument(document);

        // Navigate to specific line
        const position = new vscode.Position(Math.max(0, lineNumber - 1), 0);
        editor.selection = new vscode.Selection(position, position);
        editor.revealRange(new vscode.Range(position, position));

    } catch (error) {
        vscode.window.showErrorMessage(`Failed to open file: ${error.message}`);
    }
}

private async createOrUpdateTestFile(sourceFilePath: string, testCode: string) {
    const testFilePath = this.getTestFilePath(sourceFilePath);
    const testUri = vscode.Uri.file(testFilePath);

    try {
        // Try to open existing file
        const existingDoc = await vscode.workspace.openTextDocument(testUri);

        // Append to existing file
        const edit = new vscode.WorkspaceEdit();
        const lastLine = existingDoc.lineCount - 1;
        const insertPosition = new vscode.Position(lastLine, existingDoc.lineAt(lastLine).text.length);
        edit.insert(testUri, insertPosition, '\n\n' + testCode);

        await vscode.workspace.applyEdit(edit);

    } catch (error) {
        // File doesn't exist, create new one
        const edit = new vscode.WorkspaceEdit();
        edit.createFile(testUri);
        edit.insert(testUri, new vscode.Position(0, 0), testCode);

        await vscode.workspace.applyEdit(edit);
    }

    // Open the test file
```

```
        await vscode.window.showTextDocument(testUri);
    }

    private getTestFilePath(sourceFilePath: string): string {
        return sourceFilePath
            .replace('/src/main/java/', '/src/test/java/')
            .replace('.java', 'Test.java');
    }
}

// api/devgenie-api.ts - Complete API client
export class DevGenieAPI {
    constructor(private baseUrl: string) {}

    async healthCheck(): Promise<boolean> {
        try {
            const response = await fetch(`${this.baseUrl}/actuator/health`);
            return response.ok;
        } catch (error) {
            return false;
        }
    }

    async chat(request: any): Promise<any> {
        const response = await this.makeRequest('/api/chat', 'POST', request);
        return response;
    }

    async analyzeCodeDelta(request: any): Promise<any> {
        const response = await this.makeRequest('/api/analyze-delta', 'POST', request);
        return response;
    }

    async fixSpecificIssue(issueKey: string): Promise<any> {
        const response = await this.makeRequest('/sonar/issue/apply-fix', 'POST', [{
            key: issueKey,
            className: 'FromVSCode',
            description: 'Fix triggered from VSCode'
        }]);
        return response;
    }

    async getFixStatus(operationId: string): Promise<any> {
        const response = await this.makeRequest(`/sonar/issue/fix-status/${operationId}`, 'GET');
        return response;
    }
```

```
async fixCodeDelta(request: any): Promise<any> {
  const response = await this.makeRequest('/api/fix-code-delta', 'POST', request);
  return response;
}

async generateDocumentation(request: any): Promise<any> {
  const response = await this.makeRequest('/api/generate-documentation', 'POST', request);
  return response;
}

async generateTests(request: any): Promise<any> {
  const response = await this.makeRequest('/api/generate-tests', 'POST', request);
  return response;
}

async analyzeCoverage(request: any): Promise<any> {
  const response = await this.makeRequest('/api/coverage/analyze', 'POST', request);
  return response;
}

async generateTestsForCoverage(request: any): Promise<any> {
  const response = await this.makeRequest('/api/coverage/generate-tests', 'POST', request);
  return response;
}

private async makeRequest(endpoint: string, method: string, body?: any): Promise<any> {
  const url = `${this.baseUrl}${endpoint}`;

  const options: RequestInit = {
    method,
    headers: {
      'Content-Type': 'application/json',
      'Authorization': `Bearer ${await this.getAuthToken()}`
    }
  };

  if (body && method !== 'GET') {
    options.body = JSON.stringify(body);
  }

  const response = await fetch(url, options);

  if (!response.ok) {
    throw new Error(`API request failed: ${response.status} ${response.statusText}`);
  }

  return response.json();
```

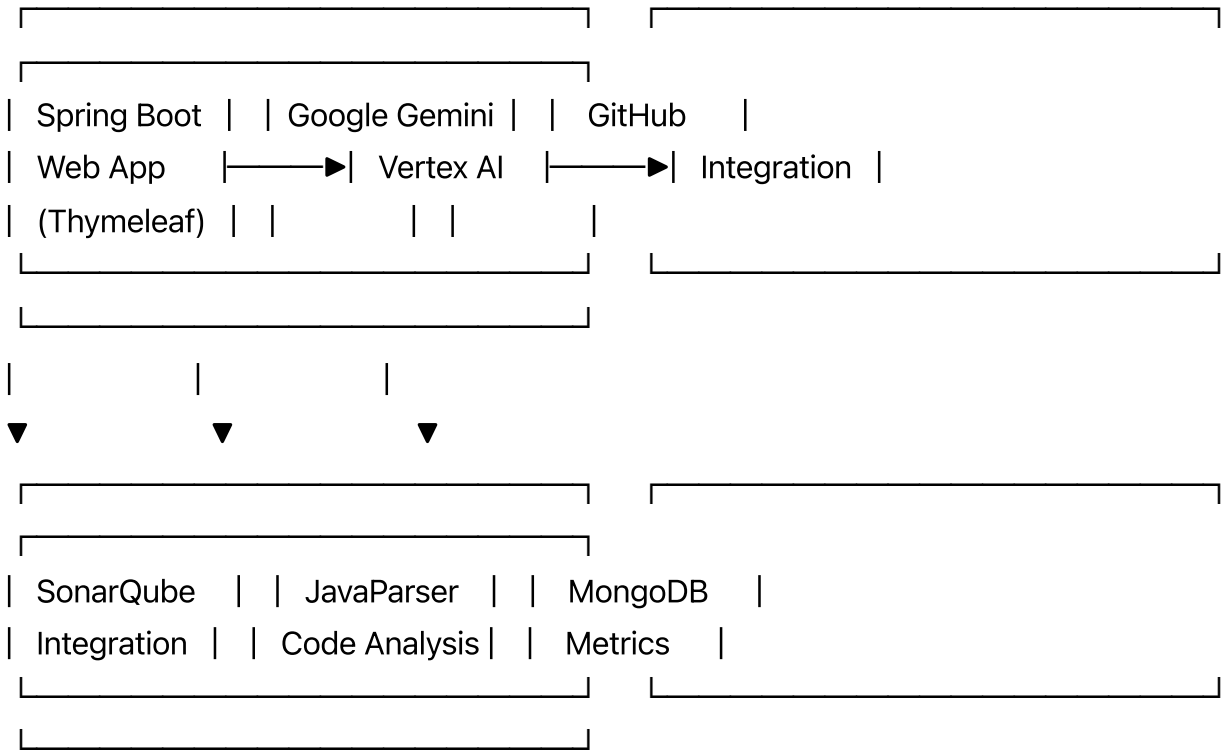
```
  }

  private async getAuthToken(): Promise<string> {
    // Implement authentication logic
    // For now, return a dummy token
    return 'vscode-extension-token';
  }# DevGenie: Comprehensive Architecture & Implementation Guide
## From Reactive Issue Fixing to Autonomous Technical Debt Management

---

# Table of Contents

1. [Executive Summary](#executive-summary)
2. [Current Implementation Analysis](#current-implementation-analysis)
3. [Architecture Approaches](#architecture-approaches)
4. [Detailed Implementation Plans](#detailed-implementation-plans)
5. [Scalability & Deployment](#scalability--deployment)
6. [Agentic AI Integration](#agentic-ai-integration)
7. [Migration Strategies](#migration-strategies)
8. [Production Readiness](#production-readiness)
9. [Risk Analysis & Mitigation](#risk-analysis--mitigation)
10. [Recommendations & Roadmap](#recommendations--roadmap)

---

# Executive Summary

## Current State
DevGenie is a well-architected Spring Boot application that automates SonarQube issue resolution using AI. It prov
- Web interface for issue selection and fixing
- Google Gemini AI integration for code fixes
- Automated GitHub PR creation
- Real-time progress tracking
- MongoDB metrics storage

## Vision
Transform DevGenie into an autonomous technical debt management platform with:
- Natural language interaction through chat interface
- VSCode agent integration for developer workflow
- Autonomous planning and execution capabilities
- Enterprise-scale deployment options
- Multi-modal technical debt resolution (coverage, complexity, security)

## Strategic Approaches
1. **Enhanced Non-MCP Approach**: Faster to market, lower risk, Java-centric
2. **MCP-Native Approach**: Future-proof, standardized, ecosystem-friendly
```

3. **Hybrid Migration Path**: Start Non-MCP, migrate to MCP progressively

---

Current Implementation Analysis

Architecture Overview



Strengths
- **Solid Foundation**: Well-structured Spring Boot application
- **Proven AI Integration**: Working Google Gemini integration
- **Complete Workflow**: End-to-end issue resolution with PR creation
- **Production Features**: Async processing, error handling, monitoring
- **Metrics Tracking**: Comprehensive analytics and reporting

Current Limitations
- **Manual Interaction**: Requires developers to select issues manually
- **Limited Scope**: Only SonarQube issues, no coverage or complexity
- **No Natural Language**: No chat or conversational interface
- **IDE Disconnect**: No integration with developer tools
- **Single-tenant**: Not designed for multi-organization use

Technology Stack Assessment

✅ Java/Spring Strengths
- **Mature Ecosystem**: Extensive library support
- **Enterprise Ready**: Proven scalability and reliability
- **Team Expertise**: Existing Java knowledge
- **Tool Integration**: Rich IDE and monitoring support

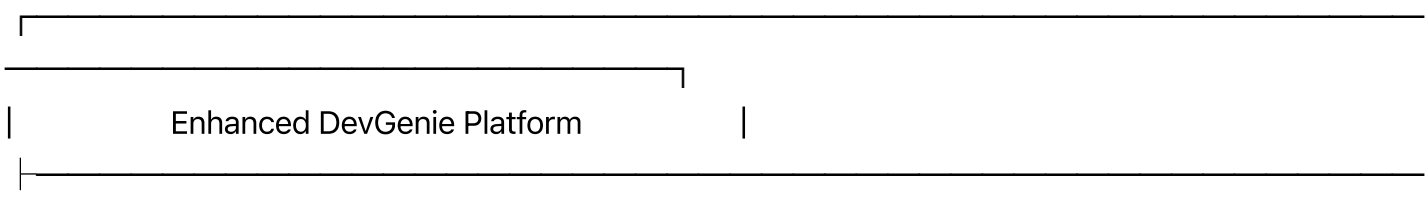
⚠️ Non-Java Dependencies
- **Node.js/TypeScript**: Required for MCP servers and VSCode extensions
- **Python**: Needed for some AI frameworks and MCP servers
- **React**: For modern chat interfaces
- **Kubernetes**: For container orchestration

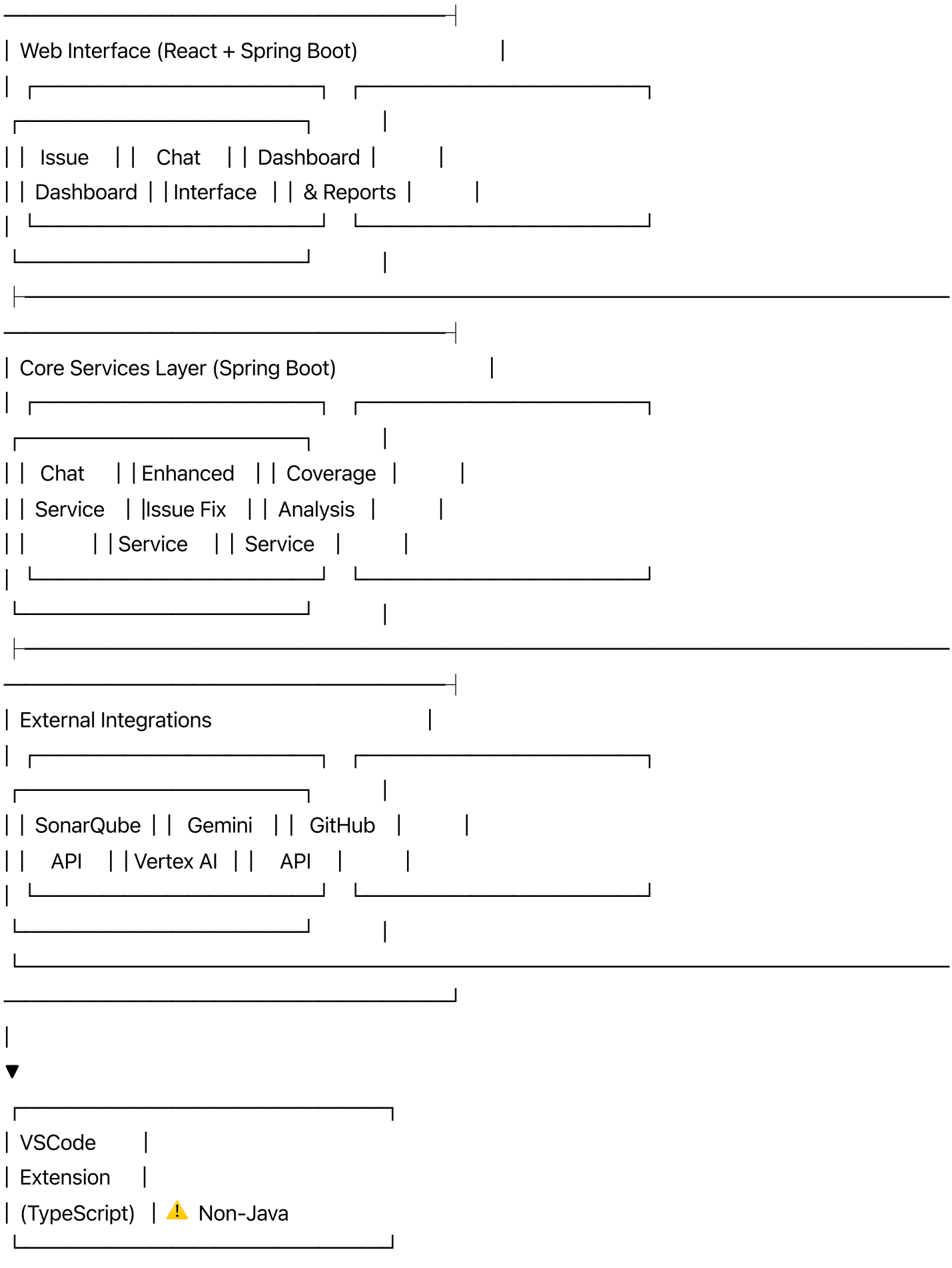
Architecture Approaches

Approach 1: Enhanced Non-MCP Architecture

Overview
Extend current Spring Boot application with chat capabilities and VSCode integration while maintaining Java-centric approach.

Architecture Diagram




Web Interface (React + Spring Boot)
Issue Dashboard
Chat Interface
Dashboard & Reports
Core Services Layer (Spring Boot)
Chat Service
Enhanced Issue Fix Service
Coverage Analysis Service
External Integrations
SonarQube API
Gemini Vertex AI
GitHub API
VSCode Extension (TypeScript)
Non-Java

Key Components

1. Enhanced Chat Service (Java)
```java
@Service
@Slf4j
public class DevGenieChatService {

    @Autowired
    private VertexAiChatModel chatModel;

    @Autowired
    private IntentClassificationService intentService;

    @Autowired
    private List<ChatActionHandler> actionHandlers;

    public CompletableFuture<ChatResponse> processQuery(ChatRequest request) {
        return CompletableFuture.supplyAsync(() -> {
            try {
                // 1. Classify user intent
                QueryIntent intent = intentService.classify(request.getQuery());
                log.info("Classified intent: {} for query: {}", intent, request.getQuery());

                // 2. Find appropriate handler
                ChatActionHandler handler = findHandler(intent);

                // 3. Process with context
                ChatContext context = buildContext(request);
                return handler.handle(request.getQuery(), context);

            } catch (Exception e) {
                log.error("Error processing chat query", e);
                return ChatResponse.error("I encountered an error. Please try again.");
            }
        });
    }

    private ChatContext buildContext(ChatRequest request) {
        return ChatContext.builder()
            .chatHistory(request.getChatHistory())
            .currentProject(request.getProjectContext())
            .userPreferences(request.getUserPreferences())
            .availableActions(getAvailableActions())
            .build();
```

```
    }
}

@Component
public class IssueFixChatHandler implements ChatActionHandler {

    @Autowired
    private SonarService sonarService;

    @Autowired
    private EnhancedIssueFixService fixService;

    @Override
    public ChatResponse handle(String query, ChatContext context) {
        try {
            // Parse what user wants to fix
            FixRequest fixRequest = parseFixRequest(query, context);

            if (fixRequest.isSpecific()) {
                // User specified exact issues to fix
                return initiateSpecificFix(fixRequest);
            } else {
                // User wants general help - suggest options
                return suggestFixOptions(fixRequest);
            }

        } catch (Exception e) {
            return ChatResponse.error("I couldn't understand your fix request. Could you be more specific?");
        }
    }

    private ChatResponse suggestFixOptions(FixRequest request) {
        List<SonarIssue> availableIssues = sonarService.fetchSonarIssues();

        List<ActionableItem> suggestions = availableIssues.stream()
            .filter(issue -> matchesUserIntent(issue, request))
            .limit(5)
            .map(this::createFixSuggestion)
            .collect(Collectors.toList());

        return ChatResponse.withSuggestions(
            "I found several issues you can fix. Here are my recommendations:",
            suggestions
        );
    }
}
```

2. VSCode Extension (TypeScript) ⚠️

typescript

```
// extension.ts - Main extension file
import * as vscode from 'vscode';
import { DevGenieApiClient } from './api-client';
import { ChatProvider } from './chat-provider';

export function activate(context: vscode.ExtensionContext) {
    const apiClient = new DevGenieApiClient(getDevGenieServerUrl());
    const chatProvider = new ChatProvider(apiClient);

    // Register chat command
    const chatCommand = vscode.commands.registerCommand(
        'devgenie.openChat',
        () => {
            const panel = vscode.window.createWebviewPanel(
                'devgenieChat',
                'DevGenie Assistant',
                vscode.ViewColumn.Beside,
                {
                    enableScripts: true,
                    retainContextWhenHidden: true
                }
            );

            chatProvider.setupChatPanel(panel);
        }
    );

    // Quick fix command for selected code
    const quickFixCommand = vscode.commands.registerCommand(
        'devgenie.quickFix',
        async () => {
            const editor = vscode.window.activeTextEditor;
            if (!editor) return;

            const selection = editor.selection;
            const selectedText = editor.document.getText(selection);

            if (selectedText) {
                const response = await apiClient.analyzeDelta(selectedText, {
                    filePath: editor.document.fileName,
                    lineNumber: selection.start.line
                });

                showQuickFixOptions(response, editor);
            }
        }
```

```
    );

    context.subscriptions.push(chatCommand, quickFixCommand);
}

// api-client.ts - Java Spring Boot API integration
export class DevGenieApiClient {
    constructor(private baseUrl: string) {}

    async chat(query: string, context: any): Promise<ChatResponse> {
        const response = await fetch(`${this.baseUrl}/api/chat`, {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json',
                'Authorization': `Bearer ${await this.getAuthToken()}`
            },
            body: JSON.stringify({ query, context })
        });

        return response.json();
    }

    async analyzeDelta(code: string, context: any): Promise<AnalysisResponse> {
        return fetch(`${this.baseUrl}/api/analyze-delta`, {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify({ code, context })
        }).then(r => r.json());
    }
}
```

3. Enhanced Issue Fix Service (Java)

java

```java
@Service
@Transactional
public class EnhancedIssueFixService {

    @Autowired
    private SpringAiCommentGenerator aiGenerator;

    @Autowired
    private GitHubUtility gitHubUtility;

    @Autowired
    private ConversationService conversationService;

    @Async
    public CompletableFuture<FixResult> fixWithConversationalContext(
            ConversationalFixRequest request) {

        return CompletableFuture.supplyAsync(() -> {
            try {
                // Build enhanced prompt with conversation history
                String enhancedPrompt = buildConversationalPrompt(request);

                // Get AI fix with conversation context
                String fixedCode = aiGenerator.fixSonarIssuesWithContext(
                    request.getClassName(),
                    request.getClassBody(),
                    request.getIssueDescriptions(),
                    enhancedPrompt
                );

                // Validate fix quality
                FixValidationResult validation = validateFix(fixedCode, request);

                if (validation.isValid()) {
                    return applyFixWithTracking(fixedCode, request);
                } else {
                    return requestClarificationFromUser(validation, request);
                }

            } catch (Exception e) {
                log.error("Error in conversational fix", e);
                return FixResult.failure("Failed to apply fix: " + e.getMessage());
            }
        });
    }
```

```
    private String buildConversationalPrompt(ConversationalFixRequest request) {
        StringBuilder promptBuilder = new StringBuilder();

        promptBuilder.append("CONVERSATION CONTEXT:\n");
        request.getChatHistory().forEach(message -> {
            promptBuilder.append(String.format("%s: %s\n",
                message.getRole(), message.getContent()));
        });

        promptBuilder.append("\nUSER PREFERENCES:\n");
        UserPreferences prefs = request.getUserPreferences();
        promptBuilder.append(String.format("Coding Style: %s\n", prefs.getCodingStyle()));
        promptBuilder.append(String.format("Test Coverage Target: %d%%\n", prefs.getCoverageTarget()));
        promptBuilder.append(String.format("Complexity Threshold: %d\n", prefs.getComplexityThreshold()));

        promptBuilder.append("\nCURRENT REQUEST:\n");
        promptBuilder.append(String.format("Fix these issues in %s:\n", request.getClassName()));
        request.getIssueDescriptions().forEach(desc -> {
            promptBuilder.append(String.format("- %s\n", desc));
        });

        promptBuilder.append("\nPlease provide a solution that aligns with the conversation context and user prefere

        return promptBuilder.toString();
    }
}
```

Pros of Enhanced Non-MCP

- ✅ **Java-Centric**: 90% Java code, minimal non-Java dependencies
- ✅ **Fast Implementation**: 6-8 weeks to working chat interface
- ✅ **Lower Risk**: Building on proven Spring Boot foundation
- ✅ **Team Expertise**: Leverages existing Java skills
- ✅ **Enterprise Ready**: Spring Boot's enterprise features

Cons of Enhanced Non-MCP

- ❌ **Future Compatibility**: May require significant changes for AI ecosystem
- ❌ **Monolithic Tendencies**: Risk of creating large, complex services
- ❌ **Custom Protocols**: Need to maintain custom APIs for VSCode integration
- ❌ **Limited Ecosystem**: Harder to integrate with other AI tools

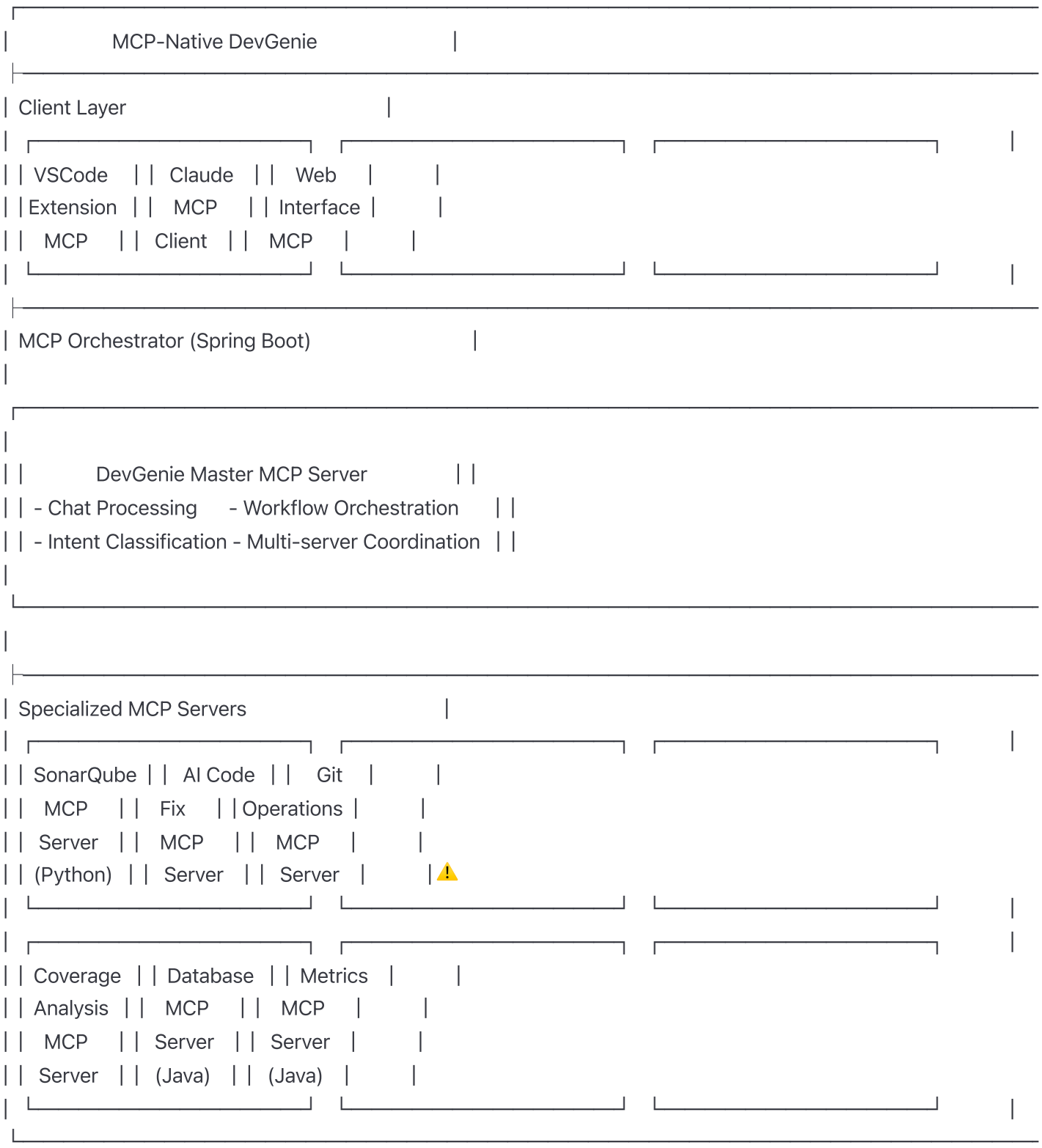
---

Approach 2: MCP-Native Architecture

Overview

Build DevGenie as a collection of MCP servers with standardized protocol communication, enabling seamless integration with AI assistants and development tools.

Architecture Diagram



Key Components

1. DevGenie Master MCP Server (Java)

java

```java
// Using MCP4J (hypothetical Java MCP implementation)
@McpServer(name = "devgenie-master", version = "1.0.0")
@Component
public class DevGenieMasterServer {

    @Autowired
    private Map<String, McpClient> mcpClients;

    @Autowired
    private ConversationOrchestrator orchestrator;

    @McpTool(name = "chat_with_devgenie")
    public McpResponse chatWithDevGenie(
            @McpParam("query") String query,
            @McpParam("context") Map<String, Object> context) {

        try {
            // Process conversational request
            ConversationRequest request = ConversationRequest.builder()
                .query(query)
                .context(context)
                .availableServers(mcpClients.keySet())
                .build();

            ConversationResponse response = orchestrator.process(request);

            return McpResponse.success(response.toJson());

        } catch (Exception e) {
            log.error("Error in chat processing", e);
            return McpResponse.error("Failed to process request: " + e.getMessage());
        }
    }

    @McpTool(name = "analyze_tech_debt")
    public McpResponse analyzeTechDebt(
            @McpParam("repository_path") String repositoryPath,
            @McpParam("analysis_type") String analysisType) {

        try {
            // Coordinate multiple MCP servers for comprehensive analysis
            CompletableFuture<McpResponse> sonarAnalysis =
                mcpClients.get("sonar-server").callTool("get_comprehensive_analysis",
                    Map.of("project_path", repositoryPath));

            CompletableFuture<McpResponse> coverageAnalysis =
```

```
            mcpClients.get("coverage-server").callTool("analyze_coverage",
                Map.of("project_path", repositoryPath));

            CompletableFuture<McpResponse> complexityAnalysis =
                mcpClients.get("complexity-server").callTool("analyze_complexity",
                    Map.of("project_path", repositoryPath));

            // Wait for all analyses to complete
            CompletableFuture.allOf(sonarAnalysis, coverageAnalysis, complexityAnalysis)
                .join();

            // Combine results
            TechDebtAnalysis combinedAnalysis = TechDebtAnalysis.builder()
                .sonarIssues(parseResponse(sonarAnalysis.get()))
                .coverageReport(parseResponse(coverageAnalysis.get()))
                .complexityMetrics(parseResponse(complexityAnalysis.get()))
                .build();

            return McpResponse.success(combinedAnalysis.toJson());

        } catch (Exception e) {
            return McpResponse.error("Analysis failed: " + e.getMessage());
        }
    }

    @McpTool(name = "autonomous_fix")
    public McpResponse autonomousFix(
            @McpParam("issues") List<Map<String, Object>> issues,
            @McpParam("strategy") String strategy) {

        // Implement autonomous fixing workflow
        return orchestrator.executeAutonomousWorkflow(issues, strategy);
    }
}

@Service
public class ConversationOrchestrator {

    @Autowired
    private IntentClassificationService intentService;

    @Autowired
    private Map<String, McpClient> mcpClients;

    public ConversationResponse process(ConversationRequest request) {
        // 1. Classify user intent
        List<Intent> intents = intentService.classifyMultiple(request.getQuery());
```

```
        // 2. Create execution plan
        ExecutionPlan plan = createExecutionPlan(intents, request.getContext());

        // 3. Execute plan across multiple MCP servers
        ExecutionResult result = executePlan(plan);

        // 4. Generate natural language response
        return generateResponse(result, request.getQuery());
    }

    private ExecutionPlan createExecutionPlan(List<Intent> intents, Map<String, Object> context) {
        ExecutionPlan.Builder planBuilder = ExecutionPlan.builder();

        for (Intent intent : intents) {
            switch (intent.getType()) {
                case FIX_ISSUES:
                    planBuilder.addStep(new McpStep("ai-fixer", "fix_issues", intent.getParameters()));
                    break;
                case ANALYZE_COVERAGE:
                    planBuilder.addStep(new McpStep("coverage-server", "detailed_analysis", intent.getParameters()));
                    break;
                case CREATE_TESTS:
                    planBuilder.addStep(new McpStep("test-generator", "generate_tests", intent.getParameters()));
                    break;
                case CREATE_PR:
                    planBuilder.addStep(new McpStep("git-server", "create_pull_request", intent.getParameters()));
                    break;
            }
        }

        return planBuilder.build();
    }
}
```

2. SonarQube MCP Server (Python) ⚠️

python

```python
# sonar_mcp_server.py
from mcp.server import Server
from mcp.types import Tool, TextContent, ImageContent
import sonarqube_api
import asyncio
import json

app = Server("sonarqube-analyzer")

class SonarQubeService:
    def __init__(self, base_url: str, token: str):
        self.sonar = sonarqube_api.SonarQubeAPI(base_url, token)

    async def get_comprehensive_analysis(self, project_key: str) -> dict:
        """Get comprehensive SonarQube analysis including issues, metrics, and trends"""

        # Get issues
        issues = list(self.sonar.issues.search_issues(componentKeys=project_key))

        # Get quality gate status
        quality_gate = self.sonar.qualitygates.get_project_qualitygate_status(project_key)

        # Get metrics
        metrics = self.sonar.measures.get_component_measures(
            component=project_key,
            metricKeys="coverage,duplicated_lines_density,code_smells,bugs,vulnerabilities"
        )

        return {
            "issues": self._process_issues(issues),
            "quality_gate": quality_gate,
            "metrics": self._process_metrics(metrics),
            "summary": self._generate_summary(issues, quality_gate, metrics)
        }

    def _process_issues(self, issues):
        """Process and categorize issues"""
        processed = {
            "by_severity": {},
            "by_type": {},
            "by_file": {},
            "total_count": len(issues)
        }

        for issue in issues:
            severity = issue.get('severity', 'UNKNOWN')
```

```
            issue_type = issue.get('type', 'UNKNOWN')
            component = issue.get('component', 'UNKNOWN')

            # Group by severity
            if severity not in processed["by_severity"]:
                processed["by_severity"][severity] = []
            processed["by_severity"][severity].append(issue)

            # Group by type
            if issue_type not in processed["by_type"]:
                processed["by_type"][issue_type] = []
            processed["by_type"][issue_type].append(issue)

            # Group by file
            if component not in processed["by_file"]:
                processed["by_file"][component] = []
            processed["by_file"][component].append(issue)

        return processed

@app.list_tools()
async def list_tools():
    return [
        Tool(
            name="get_comprehensive_analysis",
            description="Get comprehensive SonarQube analysis for a project",
            inputSchema={
                "type": "object",
                "properties": {
                    "project_key": {
                        "type": "string",
                        "description": "SonarQube project key"
                    },
                    "include_history": {
                        "type": "boolean",
                        "default": False,
                        "description": "Include historical trend data"
                    }
                },
                "required": ["project_key"]
            }
        ),
        Tool(
            name="get_issues_by_criteria",
            description="Get filtered issues based on specific criteria",
            inputSchema={
                "type": "object",
```

```
            "properties": {
              "project_key": {"type": "string"},
              "severities": {
                "type": "array",
                "items": {"type": "string"},
                "description": "Filter by severities: BLOCKER, CRITICAL, MAJOR, MINOR, INFO"
              },
              "types": {
                "type": "array",
                "items": {"type": "string"},
                "description": "Filter by types: BUG, VULNERABILITY, CODE_SMELL"
              },
              "components": {
                "type": "array",
                "items": {"type": "string"},
                "description": "Filter by specific components/files"
              }
            }
          }
        ),
        Tool(
          name="get_quality_trends",
          description="Get quality trends and historical data",
          inputSchema={
            "type": "object",
            "properties": {
              "project_key": {"type": "string"},
              "time_period": {
                "type": "string",
                "enum": ["1w", "1m", "3m", "6m", "1y"],
                "default": "1m"
              }
            }
          }
        )
    ]

@app.call_tool()
async def call_tool(name: str, arguments: dict):
    sonar_service = SonarQubeService(
        base_url=os.getenv("SONAR_URL"),
        token=os.getenv("SONAR_TOKEN")
    )

    try:
        if name == "get_comprehensive_analysis":
            result = await sonar_service.get_comprehensive_analysis(
```

```
            arguments["project_key"]
        )
        return [TextContent(type="text", text=json.dumps(result, indent=2))]

    elif name == "get_issues_by_criteria":
        result = await sonar_service.get_filtered_issues(arguments)
        return [TextContent(type="text", text=json.dumps(result, indent=2))]

    elif name == "get_quality_trends":
        result = await sonar_service.get_quality_trends(
            arguments["project_key"],
            arguments.get("time_period", "1m")
        )
        return [TextContent(type="text", text=json.dumps(result, indent=2))]

except Exception as e:
    error_msg = f"Error in {name}: {str(e)}"
    return [TextContent(type="text", text=json.dumps({"error": error_msg}))]

if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="0.0.0.0", port=8001)
```

3. AI Code Fix MCP Server (Python) ⚠️

python

```
# ai_fix_mcp_server.py
from mcp.server import Server
from mcp.types import Tool, TextContent
import vertexai
from vertexai.generative_models import GenerativeModel, Part
import json
import asyncio

app = Server("ai-code-fixer")

class AICodeFixService:
    def __init__(self):
        vertexai.init(project="your-project-id", location="us-central1")
        self.model = GenerativeModel("gemini-1.5-pro-001")

    async def fix_multiple_issues(self, class_content: str, issues: list, context: dict = None):
        """Fix multiple SonarQube issues in a single class"""

        system_prompt = self._build_comprehensive_prompt(issues, context)

        response = await self.model.generate_content_async([
            Part.from_text(system_prompt),
            Part.from_text(f"Code to fix:\n{class_content}")
        ])

        fixed_code = self._extract_and_validate_code(response.text)

        return {
            "original_code": class_content,
            "fixed_code": fixed_code,
            "issues_addressed": issues,
            "validation": self._validate_fix(fixed_code, issues),
            "explanation": self._extract_explanation(response.text)
        }

    def _build_comprehensive_prompt(self, issues: list, context: dict = None):
        prompt = f"""You are an expert Java developer and code quality specialist.

Fix the following SonarQube issues in the provided Java class:

ISSUES TO FIX:
{self._format_issues(issues)}

REQUIREMENTS:
1. Fix ALL listed issues completely
2. Maintain code functionality and logic
```

```
3. Follow Java best practices and conventions
4. Ensure code is production-ready
5. Add meaningful comments only where necessary
6. Preserve existing code structure where possible

CONTEXT:
{json.dumps(context, indent=2) if context else "No additional context provided"}

OUTPUT FORMAT:
- Return ONLY the complete, fixed Java class
- Do not include markdown formatting or code blocks
- Ensure the code compiles without errors
- Include brief comments explaining significant changes

"""
        return prompt

    def _format_issues(self, issues: list) -> str:
        formatted = []
        for i, issue in enumerate(issues, 1):
            formatted.append(f"{i}. {issue.get('message', 'Unknown issue')}")
            if 'line' in issue:
                formatted.append(f"   Line: {issue['line']}")
            if 'rule' in issue:
                formatted.append(f"   Rule: {issue['rule']}")
        return "\n".join(formatted)

    def _extract_and_validate_code(self, response_text: str) -> str:
        """Extract Java code from AI response and validate syntax"""
        # Remove markdown formatting if present
        code = response_text.strip()
        if code.startswith("```java"):
            code = code[7:]
        if code.startswith("```"):
            code = code[3:]
        if code.endswith("```"):
            code = code[:-3]

        return code.strip()

    def _validate_fix(self, fixed_code: str, issues: list) -> dict:
        """Validate that the fix addresses the issues"""
        validation = {
            "syntax_valid": True,
            "issues_likely_fixed": [],
            "potential_issues": []
        }
```

```
        # Basic syntax validation (could integrate with actual Java parser)
        try:
            # Simple checks for common syntax issues
            if fixed_code.count('{') != fixed_code.count('}'):
                validation["syntax_valid"] = False
                validation["potential_issues"].append("Mismatched braces")

            if fixed_code.count('(') != fixed_code.count(')'):
                validation["syntax_valid"] = False
                validation["potential_issues"].append("Mismatched parentheses")

        except Exception as e:
            validation["syntax_valid"] = False
            validation["potential_issues"].append(f"Validation error: {str(e)}")

        return validation

@app.list_tools()
async def list_tools():
    return [
        Tool(
            name="fix_sonar_issues",
            description="Fix multiple SonarQube issues in a Java class using AI",
            inputSchema={
                "type": "object",
                "properties": {
                    "class_content": {
                        "type": "string",
                        "description": "The complete Java class content to fix"
                    },
                    "issues": {
                        "type": "array",
                        "items": {
                            "type": "object",
                            "properties": {
                                "message": {"type": "string"},
                                "rule": {"type": "string"},
                                "line": {"type": "integer"},
                                "severity": {"type": "string"}
                            }
                        },
                        "description": "List of SonarQube issues to fix"
                    },
                    "context": {
                        "type": "object",
                        "description": "Additional context like coding standards, preferences"
```

```
            }
        },
        "required": ["class_content", "issues"]
    }
),
Tool(
    name="improve_code_coverage",
    description="Generate unit tests to improve code coverage",
    inputSchema={
        "type": "object",
        "properties": {
            "class_content": {"type": "string"},
            "uncovered_lines": {
                "type": "array",
                "items": {"type": "integer"}
            },
            "target_coverage": {
                "type": "number",
                "minimum": 0,
                "maximum": 100,
                "default": 80
            }
        }
    }
),
Tool(
    name="refactor_complex_code",
    description="Refactor code to reduce complexity and improve maintainability",
    inputSchema={
        "type": "object",
        "properties": {
            "method_content": {"type": "string"},
            "complexity_metrics": {
                "type": "object",
                "properties": {
                    "cyclomatic_complexity": {"type": "integer"},
                    "lines_of_code": {"type": "integer"},
                    "nested_depth": {"type": "integer"}
                }
            },
            "refactoring_goals": {
                "type": "array",
                "items": {"type": "string"}
            }
        }
    }
)
```

```
    ]

@app.call_tool()
async def call_tool(name: str, arguments: dict):
    ai_service = AICodeFixService()

    try:
        if name == "fix_sonar_issues":
            result = await ai_service.fix_multiple_issues(
                arguments["class_content"],
                arguments["issues"],
                arguments.get("context")
            )
            return [TextContent(type="text", text=json.dumps(result, indent=2))]

        elif name == "improve_code_coverage":
            result = await ai_service.generate_unit_tests(
                arguments["class_content"],
                arguments.get("uncovered_lines", []),
                arguments.get("target_coverage", 80)
            )
            return [TextContent(type="text", text=json.dumps(result, indent=2))]

        elif name == "refactor_complex_code":
            result = await ai_service.refactor_complex_method(
                arguments["method_content"],
                arguments.get("complexity_metrics", {}),
                arguments.get("refactoring_goals", [])
            )
            return [TextContent(type="text", text=json.dumps(result, indent=2))]

    except Exception as e:
        error_msg = f"Error in {name}: {str(e)}"
        return [TextContent(type="text", text=json.dumps({"error": error_msg}))]
```

4. Coverage Analysis MCP Server (Java)

java

```java
// CoverageMcpServer.java
@McpServer(name = "coverage-analyzer", version = "1.0.0")
@Component
public class CoverageMcpServer {

    @Autowired
    private JaCoCoAnalysisService jacocoService;

    @Autowired
    private TestGenerationService testGenerationService;

    @McpTool(name = "analyze_coverage")
    public McpResponse analyzeCoverage(
            @McpParam("project_path") String projectPath,
            @McpParam("include_details") boolean includeDetails) {

        try {
            CoverageReport report = jacocoService.generateCoverageReport(projectPath);

            CoverageAnalysis analysis = CoverageAnalysis.builder()
                .overallCoverage(report.getOverallCoverage())
                .lineCoverage(report.getLineCoverage())
                .branchCoverage(report.getBranchCoverage())
                .methodCoverage(report.getMethodCoverage())
                .classCoverage(report.getClassCoverage())
                .uncoveredAreas(report.getUncoveredAreas())
                .coverageByPackage(report.getCoverageByPackage())
                .build();

            if (includeDetails) {
                analysis.setDetailedReport(report.getDetailedAnalysis());
            }

            return McpResponse.success(analysis.toJson());

        } catch (Exception e) {
            log.error("Error analyzing coverage", e);
            return McpResponse.error("Coverage analysis failed: " + e.getMessage());
        }
    }

    @McpTool(name = "identify_critical_gaps")
    public McpResponse identifyCriticalGaps(
            @McpParam("project_path") String projectPath,
            @McpParam("criticality_threshold") double threshold) {
```

```
    try {
        List<CriticalGap> gaps = jacocoService.identifyCriticalCoverageGaps(
            projectPath, threshold);

        CriticalGapAnalysis analysis = CriticalGapAnalysis.builder()
            .totalGaps(gaps.size())
            .highPriorityGaps(gaps.stream()
                .filter(gap -> gap.getPriority() == Priority.HIGH)
                .collect(Collectors.toList()))
            .mediumPriorityGaps(gaps.stream()
                .filter(gap -> gap.getPriority() == Priority.MEDIUM)
                .collect(Collectors.toList()))
            .recommendedActions(generateRecommendations(gaps))
            .build();

        return McpResponse.success(analysis.toJson());

    } catch (Exception e) {
        return McpResponse.error("Gap analysis failed: " + e.getMessage());
    }
}

@McpTool(name = "generate_test_suggestions")
public McpResponse generateTestSuggestions(
        @McpParam("class_path") String classPath,
        @McpParam("uncovered_lines") List<Integer> uncoveredLines) {

    try {
        String classContent = Files.readString(Paths.get(classPath));

        List<TestSuggestion> suggestions = testGenerationService
            .generateTestSuggestions(classContent, uncoveredLines);

        TestSuggestionReport report = TestSuggestionReport.builder()
            .className(extractClassName(classPath))
            .totalUncoveredLines(uncoveredLines.size())
            .suggestions(suggestions)
            .estimatedCoverageImprovement(calculateCoverageImprovement(suggestions))
            .implementationComplexity(assessImplementationComplexity(suggestions))
            .build();

        return McpResponse.success(report.toJson());

    } catch (Exception e) {
        return McpResponse.error("Test suggestion generation failed: " + e.getMessage());
    }
}
```

```
}

@Service
public class JaCoCoAnalysisService {

    public CoverageReport generateCoverageReport(String projectPath) throws IOException {
        // Execute JaCoCo analysis
        ProcessBuilder pb = new ProcessBuilder(
            "mvn", "clean", "test", "jacoco:report",
            "-f", projectPath + "/pom.xml"
        );

        Process process = pb.start();

        try {
            int exitCode = process.waitFor();
            if (exitCode != 0) {
                throw new RuntimeException("JaCoCo execution failed with exit code: " + exitCode);
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            throw new RuntimeException("JaCoCo execution interrupted", e);
        }

        // Parse JaCoCo XML report
        Path reportPath = Paths.get(projectPath, "target", "site", "jacoco", "jacoco.xml");
        return parseJaCoCoReport(reportPath);
    }

    private CoverageReport parseJaCoCoReport(Path reportPath) throws IOException {
        try {
            DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
            DocumentBuilder builder = factory.newDocumentBuilder();
            Document doc = builder.parse(reportPath.toFile());

            XPath xpath = XPathFactory.newInstance().newXPath();

            // Extract overall coverage
            String lineCoveredExpr = "sum(//counter[@type='LINE']/@covered)";
            String lineMissedExpr = "sum(//counter[@type='LINE']/@missed)";

            double lineCovered = (Double) xpath.compile(lineCoveredExpr)
                .evaluate(doc, XPathConstants.NUMBER);
            double lineMissed = (Double) xpath.compile(lineMissedExpr)
                .evaluate(doc, XPathConstants.NUMBER);

            double overallCoverage = lineCovered / (lineCovered + lineMissed) * 100;
```

```
        // Extract detailed coverage by class
        NodeList classNodes = (NodeList) xpath.compile("//class")
            .evaluate(doc, XPathConstants.NODESET);

        Map<String, ClassCoverage> coverageByClass = new HashMap<>();

        for (int i = 0; i < classNodes.getLength(); i++) {
            Node classNode = classNodes.item(i);
            String className = xpath.compile("@name").evaluate(classNode);

            // Extract method coverage for this class
            NodeList methodNodes = (NodeList) xpath.compile(".//method")
                .evaluate(classNode, XPathConstants.NODESET);

            List<MethodCoverage> methodCoverages = new ArrayList<>();
            for (int j = 0; j < methodNodes.getLength(); j++) {
                Node methodNode = methodNodes.item(j);
                String methodName = xpath.compile("@name").evaluate(methodNode);

                double methodLineCovered = (Double) xpath.compile(
                    "counter[@type='LINE']/@covered").evaluate(methodNode, XPathConstants.NUMBER);
                double methodLineMissed = (Double) xpath.compile(
                    "counter[@type='LINE']/@missed").evaluate(methodNode, XPathConstants.NUMBER);

                double methodCoverage = methodLineCovered / (methodLineCovered + methodLineMissed) * 100;

                methodCoverages.add(new MethodCoverage(methodName, methodCoverage));
            }

            coverageByClass.put(className, new ClassCoverage(className, methodCoverages));
        }

        return CoverageReport.builder()
            .overallCoverage(overallCoverage)
            .coverageByClass(coverageByClass)
            .reportPath(reportPath.toString())
            .generatedAt(LocalDateTime.now())
            .build();

    } catch (Exception e) {
        throw new IOException("Failed to parse JaCoCo report", e);
    }
}

public List<CriticalGap> identifyCriticalCoverageGaps(String projectPath, double threshold) {
    List<CriticalGap> gaps = new ArrayList<>();
```

```
        try {
            CoverageReport report = generateCoverageReport(projectPath);

            report.getCoverageByClass().forEach((className, classCoverage) -> {
                if (classCoverage.getOverallCoverage() < threshold) {

                    Priority priority = determinePriority(classCoverage, projectPath);

                    CriticalGap gap = CriticalGap.builder()
                        .className(className)
                        .currentCoverage(classCoverage.getOverallCoverage())
                        .targetCoverage(threshold)
                        .priority(priority)
                        .uncoveredMethods(classCoverage.getUncoveredMethods())
                        .estimatedEffort(estimateEffort(classCoverage))
                        .businessImpact(assessBusinessImpact(className, projectPath))
                        .build();

                    gaps.add(gap);
                }
            });

        } catch (Exception e) {
            log.error("Error identifying critical gaps", e);
        }

        return gaps.stream()
            .sorted(Comparator.comparing(CriticalGap::getPriority)
                .thenComparing(CriticalGap::getBusinessImpact).reversed())
            .collect(Collectors.toList());
    }
}
```

Pros of MCP-Native

- ✅ **Future-Proof**: Standard protocol, ecosystem compatibility
- ✅ **Modular**: Independent scaling and deployment of services
- ✅ **Interoperable**: Works with any MCP-compatible AI assistant
- ✅ **Specialized**: Each server optimized for specific tasks
- ✅ **Extensible**: Easy to add new capabilities

Cons of MCP-Native

- ❌ **Complexity**: Distributed system complexity

- ❌ **Non-Java Dependencies**: Python servers for some components
- ❌ **Learning Curve**: Team needs to learn MCP protocol
- ❌ **Debugging**: Harder to debug across multiple servers
- ❌ **Network Latency**: Multiple service calls

Detailed Implementation Plans

Enhanced Non-MCP Implementation (Weeks 1-16)

Phase 1: Chat Foundation (Weeks 1-4)

Week 1: Chat Service Architecture

java

```java
// DevGenieChatController.java
@RestController
@RequestMapping("/api/chat")
@CrossOrigin(origins = "*")
public class DevGenieChatController {

    @Autowired
    private DevGenieChatService chatService;

    @PostMapping
    public CompletableFuture<ResponseEntity<ChatResponse>> chat(@RequestBody ChatRequest request) {
        return chatService.processQuery(request)
            .thenApply(response -> ResponseEntity.ok(response))
            .exceptionally(ex -> {
                log.error("Chat processing failed", ex);
                return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
                    .body(ChatResponse.error("I'm having trouble processing your request. Please try again."));
            });
    }

    @GetMapping("/suggestions")
    public ResponseEntity<List<ChatSuggestion>> getSuggestions(@RequestParam String context) {
        List<ChatSuggestion> suggestions = chatService.generateSuggestions(context);
        return ResponseEntity.ok(suggestions);
    }

    @PostMapping("/feedback")
    public ResponseEntity<Void> provideFeedback(@RequestBody ChatFeedback feedback) {
        chatService.processFeedback(feedback);
        return ResponseEntity.ok().build();
    }
}

// ChatRequest.java
@Data
@Builder
public class ChatRequest {
    private String query;
    private String sessionId;
    private List<ChatMessage> chatHistory;
    private ProjectContext projectContext;
    private UserPreferences userPreferences;
    private Map<String, Object> additionalContext;
}

// ChatResponse.java
```

```
@Data
@Builder
public class ChatResponse {
    private String message;
    private List<ActionableItem> actions;
    private List<ChatSuggestion> suggestions;
    private ChatResponseType type;
    private Map<String, Object> metadata;

    public static ChatResponse success(String message) {
        return ChatResponse.builder()
            .message(message)
            .type(ChatResponseType.SUCCESS)
            .build();
    }

    public static ChatResponse error(String message) {
        return ChatResponse.builder()
            .message(message)
            .type(ChatResponseType.ERROR)
            .build();
    }
}
```

Week 2: Intent Classification System

java

```java
// IntentClassificationService.java
@Service
public class IntentClassificationService {

    @Autowired
    private VertexAiChatModel aiModel;

    private final Map<String, IntentHandler> intentHandlers;

    public IntentClassificationService(List<IntentHandler> handlers) {
        this.intentHandlers = handlers.stream()
            .collect(Collectors.toMap(
                handler -> handler.getIntentType().name(),
                handler -> handler
            ));
    }

    public List<Intent> classifyMultiple(String query) {
        String classificationPrompt = buildClassificationPrompt(query);

        String response = aiModel.call(classificationPrompt);

        return parseIntents(response);
    }

    private String buildClassificationPrompt(String query) {
        return String.format("""
            Analyze the following user query and identify ALL applicable intents:

            User Query: "%s"

            Available Intents:
            - FIX_ISSUES: User wants to fix code quality issues
            - ANALYZE_COVERAGE: User wants to analyze or improve test coverage
            - CREATE_TESTS: User wants to generate unit tests
            - REFACTOR_CODE: User wants to refactor complex code
            - CREATE_PR: User wants to create a pull request
            - GENERAL_HELP: User needs general assistance
            - ANALYZE_TECH_DEBT: User wants overall technical debt analysis

            Return a JSON array of intent objects with this format:
            [
             {
               "intent": "INTENT_NAME",
               "confidence": 0.95,
               "parameters": {
```

```
            "key": "value"
          }
        }
      ]

      Consider that users might have multiple intents in a single query.
      """, query);
  }

  private List<Intent> parseIntents(String response) {
    try {
      ObjectMapper mapper = new ObjectMapper();
      JsonNode root = mapper.readTree(response);

      List<Intent> intents = new ArrayList<>();

      if (root.isArray()) {
        for (JsonNode intentNode : root) {
          Intent intent = Intent.builder()
            .type(IntentType.valueOf(intentNode.get("intent").asText()))
            .confidence(intentNode.get("confidence").asDouble())
            .parameters(mapper.convertValue(intentNode.get("parameters"), Map.class))
            .build();

          intents.add(intent);
        }
      }

      return intents.stream()
        .filter(intent -> intent.getConfidence() > 0.7) // Filter low confidence
        .sorted(Comparator.comparing(Intent::getConfidence).reversed())
        .collect(Collectors.toList());

    } catch (Exception e) {
      log.error("Failed to parse intents from response: {}", response, e);
      return List.of(Intent.builder()
        .type(IntentType.GENERAL_HELP)
        .confidence(0.5)
        .parameters(Map.of())
        .build());
    }
  }
}

// Intent handlers
@Component
public class FixIssuesIntentHandler implements IntentHandler {
```

```
@Autowired
private SonarService sonarService;

@Autowired
private EnhancedIssueFixService fixService;

@Override
public IntentType getIntentType() {
    return IntentType.FIX_ISSUES;
}

@Override
public ChatResponse handle(Intent intent, ChatContext context) {
    try {
        Map<String, Object> parameters = intent.getParameters();

        if (parameters.containsKey("specific_issues")) {
            return handleSpecificIssues(parameters, context);
        } else {
            return suggestAvailableIssues(context);
        }

    } catch (Exception e) {
        log.error("Error handling fix issues intent", e);
        return ChatResponse.error("I encountered an error while trying to help with fixing issues.");
    }
}

private ChatResponse handleSpecificIssues(Map<String, Object> parameters, ChatContext context) {
    // Extract specific issues from parameters
    List<String> issueKeys = (List<String>) parameters.get("specific_issues");

    if (issueKeys != null && !issueKeys.isEmpty()) {
        // Start async fix process
        String operationId = UUID.randomUUID().toString();

        List<ClassDescription> classDescriptions = issueKeys.stream()
            .map(this::convertToClassDescription)
            .collect(Collectors.toList());

        fixService.startFix(operationId, classDescriptions);

        return ChatResponse.builder()
            .message("I'll start fixing those issues for you. You can track the progress below.")
            .actions(List.of(
                ActionableItem.builder()
```

```
                .type(ActionType.TRACK_PROGRESS)
                .title("Track Fix Progress")
                .description("Monitor the fix progress in real-time")
                .data(Map.of("operationId", operationId))
                .build()
        ))
        .build();
    }

    return ChatResponse.error("I couldn't identify the specific issues you want to fix. Could you be more specific?
}

private ChatResponse suggestAvailableIssues(ChatContext context) {
    try {
        List<SonarIssue> issues = sonarService.fetchSonarIssues();

        // Group issues by severity and type for better presentation
        Map<String, List<SonarIssue>> issuesBySeverity = issues.stream()
            .collect(Collectors.groupingBy(SonarIssue::getSeverity));

        StringBuilder messageBuilder = new StringBuilder();
        messageBuilder.append("I found several issues you can fix:\n\n");

        List<ActionableItem> actions = new ArrayList<>();

        // Present high-priority issues first
        for (String severity : Arrays.asList("BLOCKER", "CRITICAL", "MAJOR")) {
            List<SonarIssue> severityIssues = issuesBySeverity.get(severity);
            if (severityIssues != null && !severityIssues.isEmpty()) {
                messageBuilder.append(String.format("**%s Issues (%d):**\n",
                    severity, severityIssues.size()));

                severityIssues.stream()
                    .limit(3) // Show top 3 per severity
                    .forEach(issue -> {
                        messageBuilder.append(String.format("- %s in %s\n",
                            issue.getDescription(), issue.getClassName()));

                        actions.add(ActionableItem.builder()
                            .type(ActionType.FIX_ISSUE)
                            .title(String.format("Fix %s", issue.getClassName()))
                            .description(issue.getDescription())
                            .data(Map.of("issueKey", issue.getKey()))
                            .build());
                    });

                messageBuilder.append("\n");
```

```
            }
        }

        messageBuilder.append("Which issues would you like me to fix?");

        return ChatResponse.builder()
            .message(messageBuilder.toString())
            .actions(actions)
            .suggestions(List.of(
                ChatSuggestion.builder()
                    .text("Fix all critical issues")
                    .action("FIX_ISSUES")
                    .parameters(Map.of("severity", "CRITICAL"))
                    .build(),
                ChatSuggestion.builder()
                    .text("Fix issues in specific file")
                    .action("FIX_ISSUES")
                    .parameters(Map.of("scope", "file"))
                    .build()
            ))
            .build();

    } catch (Exception e) {
        log.error("Error fetching available issues", e);
        return ChatResponse.error("I couldn't fetch the available issues. Please try again.");
    }
  }
}
```

Week 3: React Chat Interface ⚠️

typescript

```tsx
// ChatInterface.tsx
import React, { useState, useEffect, useRef } from 'react';
import { ChatMessage, ChatResponse, ActionableItem } from './types';
import { DevGenieAPI } from './api';

interface ChatInterfaceProps {
  projectContext?: any;
  userPreferences?: any;
}

export const ChatInterface: React.FC<ChatInterfaceProps> = ({
  projectContext,
  userPreferences
}) => {
  const [messages, setMessages] = useState<ChatMessage[]>([]);
  const [input, setInput] = useState('');
  const [isLoading, setIsLoading] = useState(false);
  const [sessionId] = useState(() => generateSessionId());
  const messagesEndRef = useRef<HTMLDivElement>(null);

  const api = new DevGenieAPI();

  useEffect(() => {
    messagesEndRef.current?.scrollIntoView({ behavior: 'smooth' });
  }, [messages]);

  useEffect(() => {
    // Add welcome message
    addMessage({
      id: 'welcome',
      role: 'assistant',
      content: "Hi! I'm DevGenie, your AI assistant for technical debt management. I can help you fix SonarQube issu
      timestamp: new Date(),
      suggestions: [
        { text: "Fix critical issues", action: "FIX_ISSUES", parameters: { severity: "CRITICAL" } },
        { text: "Analyze test coverage", action: "ANALYZE_COVERAGE", parameters: {} },
        { text: "Show me technical debt", action: "ANALYZE_TECH_DEBT", parameters: {} }
      ]
    });
  }, []);

  const addMessage = (message: ChatMessage) => {
    setMessages(prev => [...prev, message]);
  };

  const sendMessage = async (text: string) => {
```

```
if (!text.trim() || isLoading) return;

setIsLoading(true);

// Add user message
const userMessage: ChatMessage = {
  id: generateId(),
  role: 'user',
  content: text,
  timestamp: new Date()
};

addMessage(userMessage);
setInput('');

try {
  const response = await api.chat({
    query: text,
    sessionId,
    chatHistory: messages,
    projectContext,
    userPreferences,
    additionalContext: {}
  });

  // Add assistant response
  const assistantMessage: ChatMessage = {
    id: generateId(),
    role: 'assistant',
    content: response.message,
    timestamp: new Date(),
    actions: response.actions,
    suggestions: response.suggestions
  };

  addMessage(assistantMessage);

} catch (error) {
  console.error('Chat error:', error);

  const errorMessage: ChatMessage = {
    id: generateId(),
    role: 'assistant',
    content: "I'm sorry, I encountered an error. Please try again or contact support if the problem persists.",
    timestamp: new Date(),
    error: true
  };
```

```
    addMessage(errorMessage);
  } finally {
    setIsLoading(false);
  }
};

const handleActionClick = async (action: ActionableItem) => {
  setIsLoading(true);

  try {
    switch (action.type) {
      case 'FIX_ISSUE':
        await handleFixIssue(action);
        break;
      case 'TRACK_PROGRESS':
        handleTrackProgress(action);
        break;
      case 'ANALYZE_COVERAGE':
        await handleAnalyzeCoverage(action);
        break;
      default:
        console.warn('Unknown action type:', action.type);
    }
  } catch (error) {
    console.error('Action execution error:', error);
  } finally {
    setIsLoading(false);
  }
};

const handleFixIssue = async (action: ActionableItem) => {
  const issueKey = action.data?.issueKey;
  if (!issueKey) return;

  const response = await api.fixSpecificIssue(issueKey);

  const statusMessage: ChatMessage = {
    id: generateId(),
    role: 'assistant',
    content: `I've started fixing the issue. Operation ID: ${response.operationId}`,
    timestamp: new Date(),
    actions: [{
      type: 'TRACK_PROGRESS',
      title: 'Track Progress',
      description: 'Monitor fix progress',
      data: { operationId: response.operationId }
```

```
    }]
  };

  addMessage(statusMessage);
};

const handleTrackProgress = (action: ActionableItem) => {
  const operationId = action.data?.operationId;
  if (!operationId) return;

  // Start polling for progress
  const progressInterval = setInterval(async () => {
    try {
      const status = await api.getFixStatus(operationId);

      if (status.step.some(step => step.includes('✅ Completed'))) {
        clearInterval(progressInterval);

        const completionMessage: ChatMessage = {
          id: generateId(),
          role: 'assistant',
          content: "Great! The fix has been completed successfully. The pull request has been created and is ready f
          timestamp: new Date(),
          actions: status.step.filter(step => step.includes('<a href='))
            .map(step => ({
              type: 'VIEW_PR',
              title: 'View Pull Request',
              description: 'Review the generated pull request',
              data: { url: extractUrlFromStep(step) }
            }))
        };

        addMessage(completionMessage);
      }
    } catch (error) {
      console.error('Progress tracking error:', error);
      clearInterval(progressInterval);
    }
  }, 2000);

  // Clean up interval after 5 minutes
  setTimeout(() => clearInterval(progressInterval), 300000);
};

const handleAnalyzeCoverage = async (action: ActionableItem) => {
  const response = await api.analyzeCoverage(projectContext?.path);
```

```
  const analysisMessage: ChatMessage = {
    id: generateId(),
    role: 'assistant',
    content: `Coverage Analysis Results:\n\nOverall Coverage: ${response.overallCoverage}%\nLine Coverage: ${r
    timestamp: new Date(),
    actions: [{
      type: 'GENERATE_TESTS',
      title: 'Generate Tests',
      description: 'Generate unit tests for uncovered code',
      data: { uncoveredAreas: response.uncoveredAreas }
    }]
  };

  addMessage(analysisMessage);
};

const handleSuggestionClick = (suggestion: any) => {
  setInput(suggestion.text);
};

const handleKeyPress = (e: React.KeyboardEvent) => {
  if (e.key === 'Enter' && !e.shiftKey) {
    e.preventDefault();
    sendMessage(input);
  }
};

return (
  <div className="chat-interface">
    <div className="chat-header">
      <h3>DevGenie Assistant</h3>
      <div className="chat-status">
        {isLoading && <span className="loading-indicator">Thinking...</span>}
      </div>
    </div>

    <div className="chat-messages">
      {messages.map(message => (
        <ChatMessageComponent
          key={message.id}
          message={message}
          onActionClick={handleActionClick}
          onSuggestionClick={handleSuggestionClick}
        />
      ))}
      <div ref={messagesEndRef} />
    </div>
```

```
      <div className="chat-input">
        <textarea
          value={input}
          onChange={(e) => setInput(e.target.value)}
          onKeyPress={handleKeyPress}
          placeholder="Ask DevGenie to fix issues, analyze coverage, or improve code quality..."
          disabled={isLoading}
          rows={3}
        />
        <button
          onClick={() => sendMessage(input)}
          disabled={isLoading || !input.trim()}
          className="send-button"
        >
          Send
        </button>
      </div>
    </div>
  );
};

// ChatMessageComponent.tsx
interface ChatMessageComponentProps {
  message: ChatMessage;
  onActionClick: (action: ActionableItem) => void;
  onSuggestionClick: (suggestion: any) => void;
}

const ChatMessageComponent: React.FC<ChatMessageComponentProps> = ({
  message,
  onActionClick,
  onSuggestionClick
}) => {
  return (
    <div className={`chat-message ${message.role}`}>
      <div className="message-header">
        <span className="role">{message.role === 'user' ? 'You' : 'DevGenie'}</span>
        <span className="timestamp">{formatTimestamp(message.timestamp)}</span>
      </div>

      <div className="message-content">
        <ReactMarkdown>{message.content}</ReactMarkdown>
      </div>

      {message.actions && message.actions.length > 0 && (
        <div className="message-actions">
```

```
        {message.actions.map((action, index) => (
          <button
            key={index}
            onClick={() => onActionClick(action)}
            className="action-button"
          >
            {action.title}
          </button>
        ))}
      </div>
    )}

    {message.suggestions && message.suggestions.length > 0 && (
      <div className="message-suggestions">
        <span className="suggestions-label">Try asking:</span>
        {message.suggestions.map((suggestion, index) => (
          <button
            key={index}
            onClick={() => onSuggestionClick(suggestion)}
            className="suggestion-chip"
          >
            {suggestion.text}
          </button>
        ))}
      </div>
    )}

    {message.error && (
      <div className="message-error">
        <span className="error-icon">⚠️</span>
        This message encountered an error.
      </div>
    )}
  </div>
  );
};

// API client for DevGenie backend
export class DevGenieAPI {
  private baseUrl: string;

  constructor(baseUrl: string = 'http://localhost:8080') {
    this.baseUrl = baseUrl;
  }

  async chat(request: any): Promise<ChatResponse> {
    const response = await fetch(`${this.baseUrl}/api/chat`, {
```

```
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
      'Authorization': `Bearer ${await this.getAuthToken()}`
    },
    body: JSON.stringify(request)
  });

  if (!response.ok) {
    throw new Error(`Chat request failed: ${response.statusText}`);
  }

  return response.json();
}

async fixSpecificIssue(issueKey: string): Promise<any> {
  const response = await fetch(`${this.baseUrl}/sonar/issue/apply-fix`, {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify([{
      key: issueKey,
      className: 'Unknown', // Will be resolved by backend
      description: 'Fix from chat interface'
    }])
  });

  return response.json();
}

async getFixStatus(operationId: string): Promise<any> {
  const response = await fetch(`${this.baseUrl}/sonar/issue/fix-status/${operationId}`);
  return response.json();
}

async analyzeCoverage(projectPath: string): Promise<any> {
  const response = await fetch(`${this.baseUrl}/api/coverage/analyze`, {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ projectPath })
  });

  return response.json();
}

private async getAuthToken(): Promise<string> {
  // Implement authentication logic
  return 'dummy-token';
```

```
  }
}
```

Week 4: Integration Testing & Polish

java

```java
// DevGenieChatIntegrationTest.java
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
@TestPropertySource(properties = {
    "spring.ai.vertex.ai.gemini.project-id=test-project",
    "spring.ai.vertex.ai.gemini.location=us-central1"
})
class DevGenieChatIntegrationTest {

    @Autowired
    private TestRestTemplate restTemplate;

    @MockBean
    private VertexAiChatModel mockAiModel;

    @MockBean
    private SonarService mockSonarService;

    @Test
    void testBasicChatInteraction() {
        // Given
        when(mockAiModel.call(any(String.class)))
            .thenReturn("[{\"intent\":\"FIX_ISSUES\",\"confidence\":0.9,\"parameters\":{}}]");

        when(mockSonarService.fetchSonarIssues())
            .thenReturn(createMockIssues());

        ChatRequest request = ChatRequest.builder()
            .query("Fix critical issues in my code")
            .sessionId("test-session")
            .chatHistory(List.of())
            .build();

        // When
        ResponseEntity<ChatResponse> response = restTemplate.postForEntity(
            "/api/chat", request, ChatResponse.class);

        // Then
        assertThat(response.getStatusCode()).isEqualTo(HttpStatus.OK);
        assertThat(response.getBody()).isNotNull();
        assertThat(response.getBody().getMessage()).contains("critical issues");
        assertThat(response.getBody().getActions()).isNotEmpty();
    }

    @Test
    void testChatWithInvalidIntent() {
        // Given
```

```
        when(mockAiModel.call(any(String.class)))
            .thenReturn("invalid json response");

        ChatRequest request = ChatRequest.builder()
            .query("Some unclear request")
            .sessionId("test-session")
            .build();

        // When
        ResponseEntity<ChatResponse> response = restTemplate.postForEntity(
            "/api/chat", request, ChatResponse.class);

        // Then
        assertThat(response.getStatusCode()).isEqualTo(HttpStatus.OK);
        assertThat(response.getBody().getType()).isEqualTo(ChatResponseType.GENERAL_HELP);
    }

    private List<SonarIssue> createMockIssues() {
        SonarIssue issue1 = new SonarIssue();
        issue1.setKey("issue-1");
        issue1.setSeverity("CRITICAL");
        issue1.setDescription("Null pointer dereference");
        issue1.setClassName("com.example.Service");

        SonarIssue issue2 = new SonarIssue();
        issue2.setKey("issue-2");
        issue2.setSeverity("MAJOR");
        issue2.setDescription("Unused variable");
        issue2.setClassName("com.example.Controller");

        return List.of(issue1, issue2);
    }
}
```

Phase 2: VSCode Extension (Weeks 5-8) ⚠️

Week 5: Extension Foundation

typescript

```jsonc
// package.json - VSCode extension manifest
{
  "name": "devgenie-vscode",
  "displayName": "DevGenie",
  "description": "AI-powered technical debt management",
  "version": "1.0.0",
  "engines": {
    "vscode": "^1.74.0"
  },
  "categories": ["Other"],
  "activationEvents": [
    "onCommand:devgenie.openChat",
    "onCommand:devgenie.quickFix",
    "onCommand:devgenie.analyzeCoverage"
  ],
  "main": "./out/extension.js",
  "contributes": {
    "commands": [
      {
        "command": "devgenie.openChat",
        "title": "Open DevGenie Chat",
        "category": "DevGenie"
      },
      {
        "command": "devgenie.quickFix",
        "title": "Quick Fix with DevGenie",
        "category": "DevGenie"
      },
      {
        "command": "devgenie.analyzeCoverage",
        "title": "Analyze Coverage",
        "category": "DevGenie"
      }
    ],
    "menus": {
      "editor/context": [
        {
          "command": "devgenie.quickFix",
          "when": "editorHasSelection",
          "group": "devgenie"
        }
      ],
      "explorer/context": [
        {
          "command": "devgenie.analyzeCoverage",
          "when": "explorerResourceIsFolder",
```

```
        "group": "devgenie"
      }
    ]
  },
  "configuration": {
    "title": "DevGenie",
    "properties": {
      "devgenie.serverUrl": {
        "type": "string",
        "default": "http://localhost:8080",
        "description": "DevGenie server URL"
      },
      "devgenie.autoFix": {
        "type": "boolean",
        "default": false,
        "description": "Enable automatic issue fixing"
      }
    }
  }
 },
 "scripts": {
  "vscode:prepublish": "npm run compile",
  "compile": "tsc -p ./",
  "watch": "tsc -watch -p ./"
 },
 "devDependencies": {
  "@types/vscode": "^1.74.0",
  "@types/node": "16.x",
  "typescript": "^4.9.4"
 }
}

// extension.ts - Main extension file
import * as vscode from 'vscode';
import { DevGenieAPI } from './api/devgenie-api';
import { ChatProvider } from './providers/chat-provider';
import { QuickFixProvider } from './providers/quickfix-provider';
import { CoverageProvider } from './providers/coverage-provider';

export function activate(context: vscode.ExtensionContext) {
    console.log('DevGenie extension is now active!');

    // Initialize API client
    const config = vscode.workspace.getConfiguration('devgenie');
    const serverUrl = config.get<string>('serverUrl', 'http://localhost:8080');
    const api = new DevGenieAPI(serverUrl);
```

```
    // Initialize providers
    const chatProvider = new ChatProvider(api, context);
    const quickFixProvider = new QuickFixProvider(api);
    const coverageProvider = new CoverageProvider(api);

    // Register commands
    const openChatCommand = vscode.commands.registerCommand(
        'devgenie.openChat',
        () => chatProvider.openChat()
    );

    const quickFixCommand = vscode.commands.registerCommand(
        'devgenie.quickFix',
        () => quickFixProvider.showQuickFix()
    );

    const analyzeCoverageCommand = vscode.commands.registerCommand(
        'devgenie.analyzeCoverage',
        (uri: vscode.Uri) => coverageProvider.analyzeCoverage(uri)
    );

    // Register status bar
    const statusBarItem = vscode.window.createStatusBarItem(
        vscode.StatusBarAlignment.Right, 100
    );
    statusBarItem.text = "$(robot) DevGenie";
    statusBarItem.command = 'devgenie.openChat';
    statusBarItem.tooltip = 'Open DevGenie Chat';
    statusBarItem.show();

    // Add to subscriptions for cleanup
    context.subscriptions.push(
        openChatCommand,
        quickFixCommand,
        analyzeCoverageCommand,
        statusBarItem
    );

    // Check server connectivity
    checkServerConnectivity(api);
}

async function checkServerConnectivity(api: DevGenieAPI) {
    try {
        await api.healthCheck();
        vscode.window.showInformationMessage('DevGenie server connected successfully!');
    } catch (error) {
```

```
        vscode.window.showWarningMessage(
            'DevGenie server is not reachable. Some features may not work.',
            'Configure Server'
        ).then(selection => {
            if (selection === 'Configure Server') {
                vscode.commands.executeCommand('workbench.action.openSettings', 'devgenie.serverUrl');
            }
        });
    }
}

export function deactivate() {
    console.log('DevGenie extension is now deactivated.');
}
```

Week 6: Chat Provider Implementation

typescript

```typescript
// providers/chat-provider.ts
import * as vscode from 'vscode';
import { DevGenieAPI } from '../api/devgenie-api';

export class ChatProvider {
  private panel: vscode.WebviewPanel | undefined;

  constructor(
    private api: DevGenieAPI,
    private context: vscode.ExtensionContext
  ) {}

  public openChat() {
    if (this.panel) {
      this.panel.reveal();
      return;
    }

    this.panel = vscode.window.createWebviewPanel(
      'devgenieChat',
      'DevGenie Chat',
      vscode.ViewColumn.Beside,
      {
        enableScripts: true,
        retainContextWhenHidden: true,
        localResourceRoots: [
          vscode.Uri.joinPath(this.context.extensionUri, 'media')
        ]
      }
    );

    this.panel.webview.html = this.getWebviewContent();

    // Handle messages from webview
    this.panel.webview.onDidReceiveMessage(
      async (message) => {
        switch (message.command) {
          case 'sendMessage':
            await this.handleChatMessage(message.text);
            break;
          case 'executeAction':
            await this.handleAction(message.action);
            break;
          case 'ready':
            await this.initializeChat();
            break;
```

```
        }
      }
    );

    // Clean up when panel is closed
    this.panel.onDidDispose(() => {
      this.panel = undefined;
    });
  }

  private async handleChatMessage(text: string) {
    if (!this.panel) return;

    try {
      // Get current workspace context
      const workspaceContext = await this.getCurrentWorkspaceContext();

      const response = await this.api.chat({
        query: text,
        sessionId: this.getSessionId(),
        chatHistory: [], // Maintained in webview
        projectContext: workspaceContext,
        userPreferences: this.getUserPreferences()
      });

      this.panel.webview.postMessage({
        command: 'chatResponse',
        response: response
      });

    } catch (error) {
      console.error('Chat error:', error);
      this.panel.webview.postMessage({
        command: 'chatError',
        error: error.message
      });
    }
  }

  private async handleAction(action: any) {
    try {
      switch (action.type) {
        case 'FIX_ISSUE':
          await this.handleFixIssueAction(action);
          break;
        case 'ANALYZE_COVERAGE':
          await this.handleCoverageAction(action);
```

```
            break;
        case 'OPEN_FILE':
            await this.handleOpenFileAction(action);
            break;
        default:
            console.warn('Unknown action type:', action.type);
      }
    } catch (error) {
      console.error('Action execution error:', error);
      if (this.panel) {
        this.panel.webview.postMessage({
          command: 'actionError',
          error: error.message
        });
      }
    }
}

private async handleFixIssueAction(action: any) {
    const issueKey = action.data?.issueKey;
    if (!issueKey) return;

    // Show progress notification
    vscode.window.withProgress({
      location: vscode.ProgressLocation.Notification,
      title: "DevGenie is fixing the issue...",
      cancellable: false
    }, async (progress) => {
      try {
        const result = await this.api.fixSpecificIssue(issueKey);

        // Update chat with result
        if (this.panel) {
          this.panel.webview.postMessage({
            command: 'actionResult',
            result: {
              type: 'fix_started',
              operationId: result.operationId,
              message: 'Fix process started successfully!'
            }
          });
        }

        // Start monitoring progress
        this.monitorFixProgress(result.operationId);

      } catch (error) {
```

```
            vscode.window.showErrorMessage(`Failed to start fix: ${error.message}`);
        }
    });
}

private async monitorFixProgress(operationId: string) {
    const checkProgress = async () => {
        try {
            const status = await this.api.getFixStatus(operationId);

            if (this.panel) {
                this.panel.webview.postMessage({
                    command: 'progressUpdate',
                    operationId: operationId,
                    status: status
                });
            }

            // Check if completed
            if (status.step.some(step => step.includes('✅ Completed'))) {
                vscode.window.showInformationMessage(
                    'DevGenie has successfully fixed the issue!',
                    'View Changes'
                ).then(selection => {
                    if (selection === 'View Changes') {
                        // Open git diff or PR
                        this.showFixResults(status);
                    }
                });
                return; // Stop polling
            }

            // Continue polling if not completed
            setTimeout(checkProgress, 2000);

        } catch (error) {
            console.error('Progress monitoring error:', error);
        }
    };

    checkProgress();
}

private async getCurrentWorkspaceContext() {
    const workspaceFolders = vscode.workspace.workspaceFolders;
    if (!workspaceFolders || workspaceFolders.length === 0) {
        return null;
```

```
    }

    const rootPath = workspaceFolders[0].uri.fsPath;
    const activeEditor = vscode.window.activeTextEditor;

    return {
        rootPath: rootPath,
        currentFile: activeEditor?.document.fileName,
        currentSelection: activeEditor?.selection,
        openFiles: vscode.workspace.textDocuments.map(doc => doc.fileName)
    };
}

private getUserPreferences() {
    const config = vscode.workspace.getConfiguration('devgenie');
    return {
        autoFix: config.get('autoFix', false),
        preferredCodingStyle: 'standard', // Could be configurable
        coverageTarget: 80
    };
}

private getSessionId(): string {
    // Generate or retrieve session ID
    return 'vscode-session-' + Date.now();
}

private getWebviewContent(): string {
    return `
    <!DOCTYPE html>
    <html lang="en">
    <head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <title>DevGenie Chat</title>
        <style>
            body {
                font-family: var(--vscode-font-family);
                color: var(--vscode-editor-foreground);
                background-color: var(--vscode-editor-background);
                margin: 0;
                padding: 0;
                height: 100vh;
                display: flex;
                flex-direction: column;
            }
```

```css
.chat-container {
    display: flex;
    flex-direction: column;
    height: 100%;
}

.chat-header {
    padding: 10px;
    border-bottom: 1px solid var(--vscode-panel-border);
    background-color: var(--vscode-panel-background);
}

.chat-messages {
    flex: 1;
    overflow-y: auto;
    padding: 10px;
}

.message {
    margin-bottom: 15px;
    padding: 10px;
    border-radius: 8px;
}

.message.user {
    background-color: var(--vscode-inputOption-activeBackground);
    margin-left: 20%;
}

.message.assistant {
    background-color: var(--vscode-editor-inactiveSelectionBackground);
    margin-right: 20%;
}

.message-actions {
    margin-top: 10px;
}

.action-button {
    background-color: var(--vscode-button-background);
    color: var(--vscode-button-foreground);
    border: none;
    padding: 6px 12px;
    margin-right: 5px;
    border-radius: 4px;
    cursor: pointer;
}
```

```css
.action-button:hover {
    background-color: var(--vscode-button-hoverBackground);
}

.chat-input {
    padding: 10px;
    border-top: 1px solid var(--vscode-panel-border);
    background-color: var(--vscode-panel-background);
}

.input-container {
    display: flex;
    gap: 10px;
}

.message-input {
    flex: 1;
    padding: 8px;
    border: 1px solid var(--vscode-input-border);
    background-color: var(--vscode-input-background);
    color: var(--vscode-input-foreground);
    border-radius: 4px;
    resize: vertical;
    min-height: 60px;
}

.send-button {
    background-color: var(--vscode-button-background);
    color: var(--vscode-button-foreground);
    border: none;
    padding: 8px 16px;
    border-radius: 4px;
    cursor: pointer;
}

.send-button:hover {
    background-color: var(--vscode-button-hoverBackground);
}

.send-button:disabled {
    opacity: 0.5;
    cursor: not-allowed;
}

.loading {
    color: var(--vscode-descriptionForeground);
```

```
            font-style: italic;
        }

        .suggestions {
            margin-top: 10px;
        }

        .suggestion-chip {
            background-color: var(--vscode-badge-background);
            color: var(--vscode-badge-foreground);
            border: none;
            padding: 4px 8px;
            margin-right: 5px;
            margin-bottom: 5px;
            border-radius: 12px;
            font-size: 12px;
            cursor: pointer;
        }

        .suggestion-chip:hover {
            opacity: 0.8;
        }
    </style>
</head>
<body>
    <div class="chat-container">
        <div class="chat-header">
            <h3>DevGenie Assistant</h3>
            <div id="status" class="loading" style="display: none;">Connecting...</div>
        </div>

        <div class="chat-messages" id="messages">
            <!-- Messages will be added dynamically -->
        </div>

        <div class="chat-input">
            <div class="input-container">
                <textarea
                    id="messageInput"
                    class="message-input"
                    placeholder="Ask DevGenie to fix issues, analyze coverage, or improve code quality..."
                    rows="3">
                </textarea>
                <button id="sendButton" class="send-button">Send</button>
            </div>
        </div>
    </div>
```

```
<script>
    const vscode = acquireVsCodeApi();
    const messagesContainer = document.getElementById('messages');
    const messageInput = document.getElementById('messageInput');
    const sendButton = document.getElementById('sendButton');
    const status = document.getElementById('status');

    let isLoading = false;

    // Send ready message to extension
    vscode.postMessage({ command: 'ready' });

    // Send message
    function sendMessage() {
        const text = messageInput.value.trim();
        if (!text || isLoading) return;

        setLoading(true);
        addMessage('user', text);
        messageInput.value = '';

        vscode.postMessage({
            command: 'sendMessage',
            text: text
        });
    }

    // Add message to chat
    function addMessage(role, content, actions = null, suggestions = null) {
        const messageDiv = document.createElement('div');
        messageDiv.className = \`message \${role}\`;

        const contentDiv = document.createElement('div');
        contentDiv.textContent = content;
        messageDiv.appendChild(contentDiv);

        if (actions && actions.length > 0) {
            const actionsDiv = document.createElement('div');
            actionsDiv.className = 'message-actions';

            actions.forEach(action => {
                const button = document.createElement('button');
                button.className = 'action-button';
                button.textContent = action.title;
                button.onclick = () => executeAction(action);
                actionsDiv.appendChild(button);
```

```
        });

        messageDiv.appendChild(actionsDiv);
    }

    if (suggestions && suggestions.length > 0) {
        const suggestionsDiv = document.createElement('div');
        suggestionsDiv.className = 'suggestions';

        const label = document.createElement('div');
        label.textContent = 'Try asking:';
        label.style.marginBottom = '5px';
        label.style.fontSize = '12px';
        suggestionsDiv.appendChild(label);

        suggestions.forEach(suggestion => {
            const chip = document.createElement('button');
            chip.className = 'suggestion-chip';
            chip.textContent = suggestion.text;
            chip.onclick = () => {
                messageInput.value = suggestion.text;
                messageInput.focus();
            };
            suggestionsDiv.appendChild(chip);
        });

        messageDiv.appendChild(suggestionsDiv);
    }

    messagesContainer.appendChild(messageDiv);
    messagesContainer.scrollTop = messagesContainer.scrollHeight;
}

// Execute action
function executeAction(action) {
    vscode.postMessage({
        command: 'executeAction',
        action: action
    });
}

// Set loading state
function setLoading(loading) {
    isLoading = loading;
    sendButton.disabled = loading;
    status.style.display = loading ? 'block' : 'none';
    status.textContent = loading ? 'Processing...' : '';
```

```
}

// Event listeners
sendButton.addEventListener('click', sendMessage);
messageInput.
```