DevGenie: Comprehensive Architecture & Implementation Guide

From Reactive Issue Fixing to Autonomous Technical Debt Management

---

Table of Contents

Executive Summary

Current State

DevGenie is a well-architected Spring Boot application that automates SonarQube issue resolution using AI. It provides:

- 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)

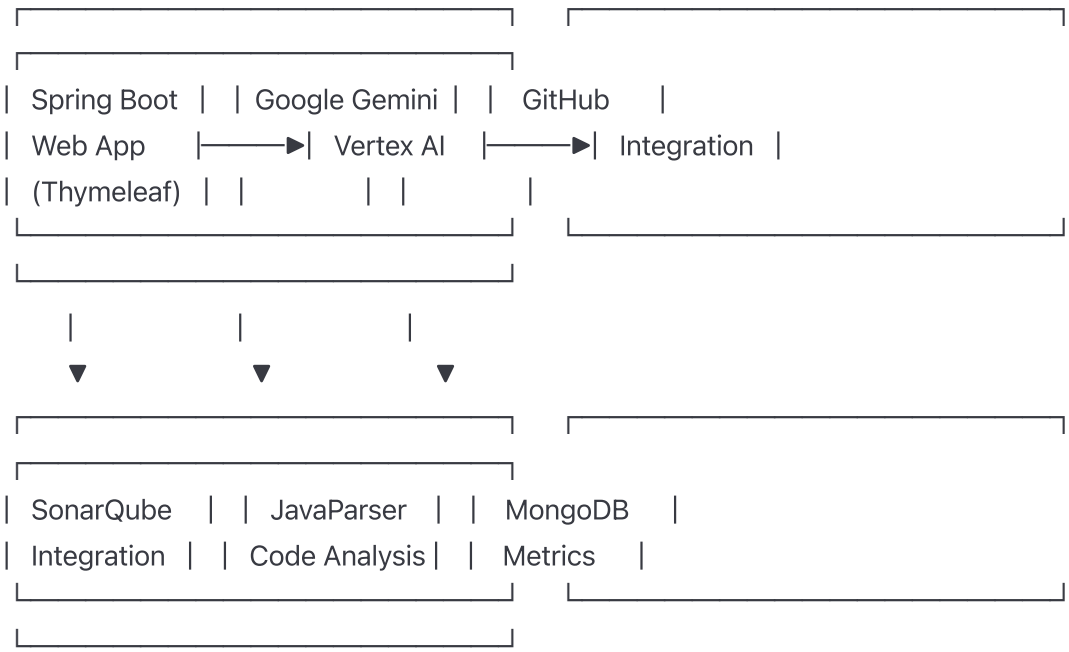
Strategic Approaches

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

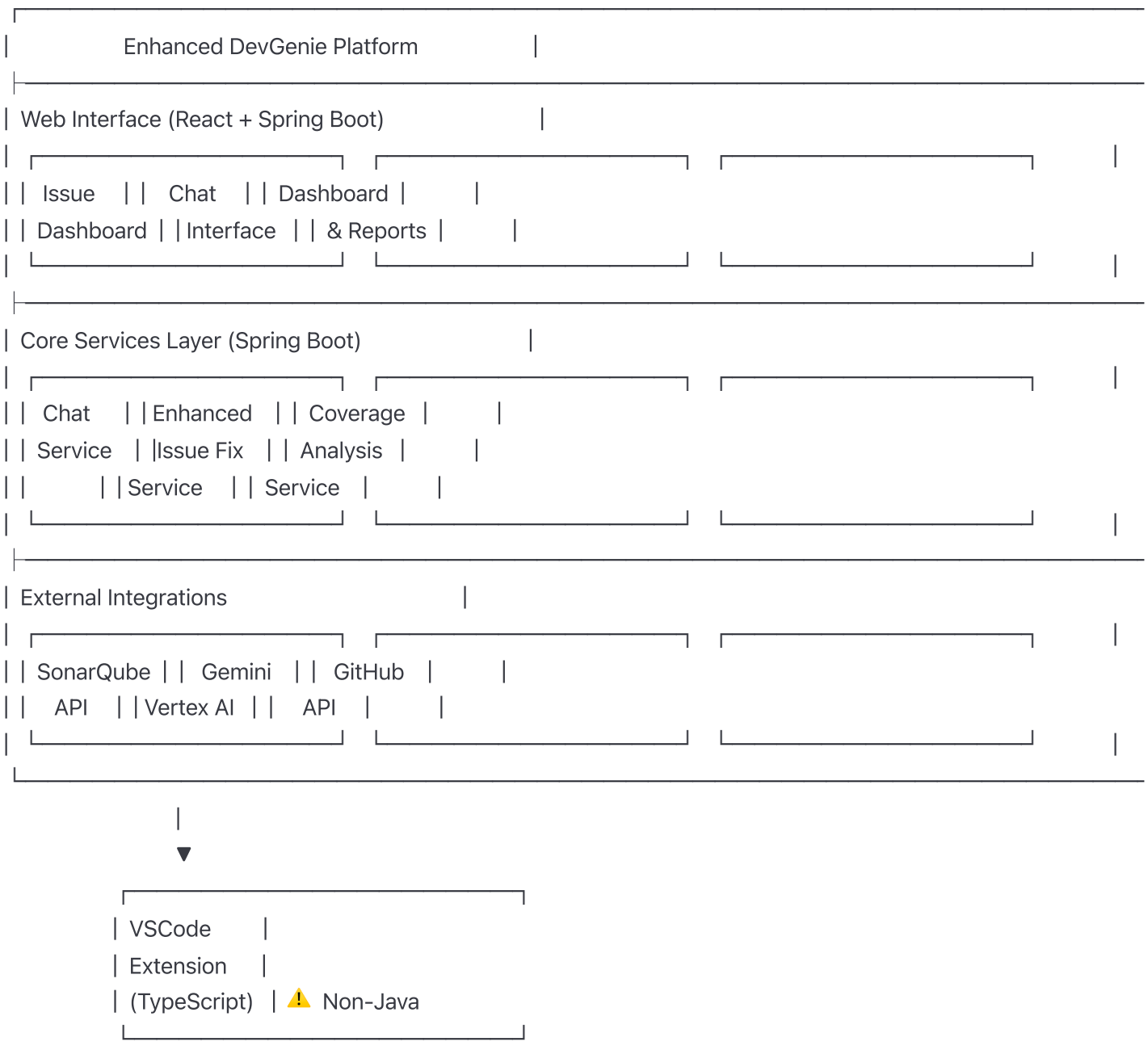
Architecture Approaches

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



Key Components

1. Enhanced Chat Service (Java)

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

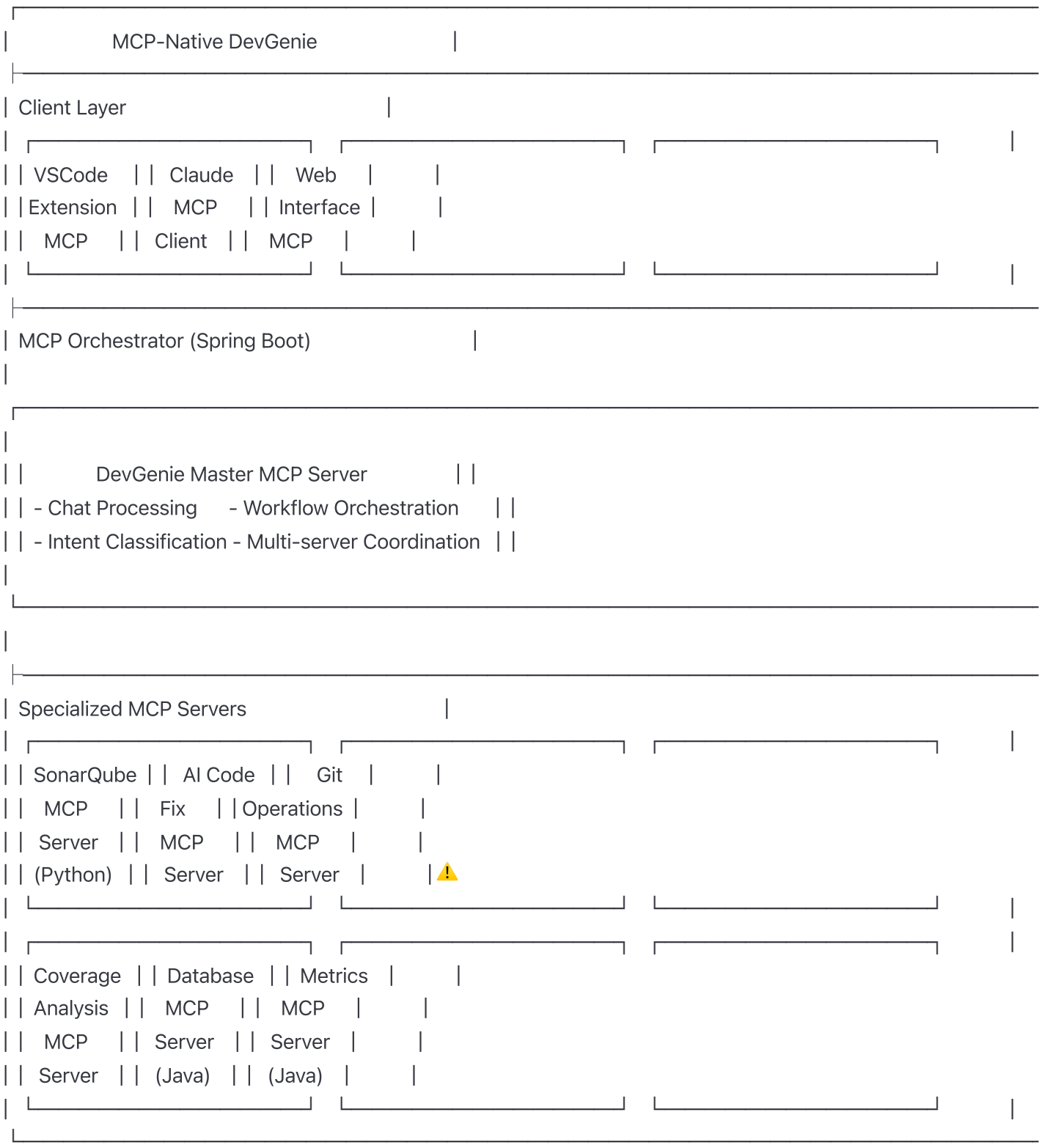
---

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