

Leipzig University of Applied Sciences

Hochschule für Technik, Wirtschaft und Kultur Leipzig

Faculty of: Computer Science and Media

Fakultät: Informatik und Medien

Bachelor-Study Programm: Media Informatics

Studiengang: Medieninformatik

Integration of the Model-Context-Protocol (MCP) with Symfony for context-sensitive code generation using large language models (LLMs)

A proof of concept

Bachelor's Thesis

By

Salem Zin Iden Naser

Responsible Academic Supervisor: Prof. Dr. rer. nat. Thomas Riechert

Company Supervisor: Dip. Lizzy Gerischer

Leipzig, 15.06.2025 – 15.09.2025

Acknowledgments

I gratefully acknowledge the open-source foundations behind this work—especially the Symfony Demo application (<https://github.com/symfony/demo>) and the MCP demo project (<https://github.com/php-llm/mcp-demo>). Their code and documentation made rigorous, reproducible experiments possible.

My sincere thanks to **Prof. Dr. Thomas Riechert** for guiding this thesis and for helping me take my first steps at the university. His steady advice at key moments made completion possible.

I also thank my second supervisor, **Dipl. Lizzy Gerischer**, a colleague whose door (and smile) were always open. Her readiness to answer questions made the research and my working life lighter and better.

To **Kim**, my study colleague in Media Informatics: thank you for being my safe space at the university and my steadfast library study buddy. Your presence turned long days into shared milestones.

To my chosen family in Leipzig—**Z, N, J, S, R**—and to the wider communities in Leipzig and Berlin: your companionship made a new life possible. Without your care, humor, and everyday solidarity, I would not have found my footing in Germany.

To my family, whom I have not seen for more than ten years: **my father, Yehya**, my first technical mentor, who taught me how to use a computer, install software, and overcome the peculiarities of Windows 98; **my brother, Samer**, my forever best friend; **my sister, Hala**, my spiritual idol. And especially **my mother, Rudina** to whom I dedicate this work. Her love is the first and the last reason for everything I was, am, and hope to become.

Abstract

This thesis investigates the practical effectiveness of the Model Context Protocol (MCP) as a standardized interface for connecting large language models (LLMs) to development tools and project data within the PHP Symfony framework. Addressing the $N \times M$ integration problem —where each AI application must implement bespoke connectors for each external tool— the work implements a complete proof of concept comprising an MCP server (tools for codebase inspection), an MCP client in Symfony, and a local inference stack using Ollama with DeepSeek-Coder 6.7B. Because the target model lacks native function calling, the study operationalizes a “context injection” strategy: the host pre-executes MCP tools (e.g., extract entities, routes, controllers) and embeds their structured outputs into the prompt. The system is evaluated on a realistic scenario —adding “like” functionality for posts and comments in the official Symfony Demo project— by comparing MCP-enhanced generation against a vanilla baseline without project context. Results show a clear quality–speed trade-off: MCP increases processing time $4.39\times$ but yields substantially more complete and architecturally appropriate code (requirements score 7/8 vs. 1/8), including correct Doctrine relationships, controller logic, and routing. Qualitative analysis indicates that structured context enables better database design and stronger adherence to framework conventions; practical validation confirms that MCP-generated code can be integrated with only syntactic updates. The thesis also outlines security considerations for MCP deployments (isolation, least privilege, input validation). Limitations include a single framework, one model, and one task scenario in a controlled environment. Contributions include an open-source MCP/Symfony implementation, a reusable evaluation framework, and empirical evidence that standardized, tool-mediated context materially improves LLM-assisted code generation despite higher latency. Future work targets broader model/framework coverage, selective context retrieval, richer MCP servers, and IDE/CI integration.

Contents

1	Introduction	6
1.1	Problem Statement	6
1.2	Research Objectives	7
1.3	Scope and Limitations	7
1.3.1	Research Scope	7
1.3.2	Research Limitations	8
2	Symfony, The Model Context Protocol (MCP) and Related Technologies	10
2.1	Symfony Framework: Architecture and Code Generation Advantages	10
2.1.1	Symfony Framework Overview	10
2.1.2	Best Practices and Standardization Benefits for LLM Code Generation	11
2.1.3	Doctrine ORM Integration and Entity Management	11
2.2	The Model Context Protocol (MCP): A "USB-C for AI Applications"	13
2.2.1	The N×M Integration Problem and MCP's Solution	14
2.2.2	MCP Protocol Architecture and Communication Model	15
2.2.3	JSON-RPC 2.0 Foundation	17
2.3	Large Language Models and Function Calling	19
2.3.1	Function Calling Fundamentals	19
2.3.2	MCP's Role in Function Calling	19
2.4	DeepSeek-Coder Model Selection	20
2.5	Ollama: Local LLM Infrastructure	20
2.5.1	Ollama Overview	20
2.5.2	Model Management and Customization	21
2.5.3	Integration with MCP Architecture	21
3	Proof of Concept Implementation	22
3.1	Implementation Goals and Architecture	22
3.1.1	Key Implementation Objectives	22
3.1.2	Target Application: Symfony Demo Project	23
3.1.3	System Architecture Overview	23
3.2	MCP Server Implementation	25
3.2.1	Server Process and Transport	25
3.2.2	Tool Registry and Capability Advertisement	26
3.2.3	Tool Execution Framework	26

3.3	MCP Client Integration	29
3.3.1	Context Collection Service	29
3.4	LLM Integration and Context Injection Strategy	30
3.4.1	Prompt Construction Strategy	30
3.4.2	LLM Bridge Service	31
3.5	Evaluation Framework	32
3.5.1	Scenario Configuration	32
3.5.2	HTTP API for Evaluation Control	33
3.5.3	Test Scenario Runner	33
3.6	Implementation Summary	34
4	Proof of Concept Results and Analysis	36
4.1	Generated Code	36
4.1.1	MCP-Enhanced Response	36
4.1.2	Vanilla Response Analysis	39
4.2	Quantitative, Qualitative and Architectural Analysis	40
4.2.1	Quantitative Performance Analysis	40
4.2.2	Qualitative Code Assessment	41
4.2.3	Architectural Quality Analysis	41
4.3	Implementation Validation	42
4.3.1	MCP Response Integration	42
4.3.2	Vanilla Response Integration	43
4.4	Key Findings and Implications	43
4.5	Limitations and Considerations	43
4.6	Summary	44
5	Security Considerations in MCP Implementation	45
5.1	Common Security Threats and Prevention	45
5.2	Development Best Practices	46
5.3	Monitoring and Compliance	47
5.4	Practical Security Checklist	47
6	Conclusion and Future Outlook	49
6.1	Summary of Findings	49
6.2	Contributions to the Field	50
6.3	Future Research Directions	50
	Figures and Tables	55

1 Introduction

1.1 Problem Statement

Connecting large language models (LLMs) to external tools and data sources presents significant integration challenges. Each model often requires its own connector for each service, creating the “ $M \times N$ ” problem—“the combinatorial difficulty of integrating M different LLMs with N different tools” [1]. This expansion of custom integrations results in high maintenance, inconsistent behavior across implementations, and fragmented workflows that inhibit productivity and scalability.

Traditional approaches to AI-tool integration lack standardization, forcing developers to create separate solutions for every combination of AI model and external service. As the ecosystem of AI applications expands and the demand for context-aware code generation increases, this integration approach becomes increasingly unsustainable. The absence of a unified protocol creates barriers to innovation, limits transaction, and increases development complexity.

The Model Context Protocol (MCP) addresses these challenges by providing “a standardized way to connect AI applications to external systems” [2]. However, despite its theoretical promise, the practical effectiveness of MCP in real-world development scenarios—particularly in structured framework environments like Symfony—remains largely unexplored. This gap between theoretical specification and practical implementation necessitates empirical validation to assess MCP’s true potential for improving context-sensitive code generation workflows.

Furthermore, the integration of MCP with web development frameworks presents unique challenges related to security, performance, and maintainability that require careful analysis. Understanding how MCP performs in practice, what benefits it provides over traditional approaches, and what trade-offs developers must consider is essential for informed adoption decisions.

1.2 Research Objectives

This thesis investigates the practical application of the Model Context Protocol (MCP) as a solution to LLM-tool integration challenges, specifically within the context of PHP web development using the Symfony framework. The research aims to bridge the gap between MCP’s theoretical capabilities and its real-world effectiveness through empirical evaluation and proof-of-concept implementation.

The primary research objectives are:

1. **Demonstrate MCP’s standardization capabilities:** To show how MCP can provide a unified approach to integrating LLMs with development tools, reducing the complexity of custom integrations and improving back-and-forth across different AI applications.
2. **Evaluate context-sensitive code generation quality:** To assess the impact of structured project context on LLM-generated code quality, and adherence to framework conventions, comparing MCP-enhanced approaches against vanilla implementations.

1.3 Scope and Limitations

1.3.1 Research Scope

This research focuses specifically on the integration of MCP with PHP-based Symfony applications for context-sensitive code generation.

The scope encompasses:

- **MCP Protocol Implementation:** Complete implementation of MCP server and client components within a Symfony framework context, following the official MCP specification and best practices.
 - **Context-Sensitive Code Generation Analysis:** Systematic comparison of MCP-enhanced versus vanilla approaches across multiple dimensions including processing time, output quality, and solution completeness.
-

- **Framework Integration Assessment:** Analysis of how MCP integrates with Symfony’s architectural patterns, including Doctrine ORM, dependency injection, routing, and security components.
- **Alternative Implementation Strategies:** Exploration of context injection techniques for models without native function-calling capabilities, demonstrating MCP’s flexibility beyond traditional tool-calling scenarios.
- **Security Considerations:** Examination of authentication, authorization, data protection, and other security aspects relevant to MCP deployment in production environments.

1.3.2 Research Limitations

Several limitations constrain the generalizability and scope of this research, which should be considered when interpreting the results:

- **Single Framework Focus:** Results are specific to Symfony and PHP development patterns. Findings may not directly translate to other web development frameworks (e.g., Django, Rails, Express.js) or programming languages with different architectural conventions.
 - **Limited Model Architecture Testing:** Evaluation focuses primarily on one LLM model (DeepSeek-Coder:6.7b) served through Ollama.
 - **Constrained Scenario Coverage:** Testing centers on a single feature implementation scenario (adding like functionality to posts and comments). This narrow focus may not represent the full spectrum of development tasks where MCP could provide benefits or reveal different performance characteristics.
 - **Non-Function Calling Model Limitation:** The use of a model without native function-calling capabilities required context injection rather than true MCP function calling, potentially not fully representing MCP’s intended usage patterns with more advanced models.
-

- **Evaluation Methodology Constraints:** Assessment focuses primarily on code quality and architectural appropriateness rather than comprehensive metrics such as long-term development productivity, maintainability over time, or integration complexity in larger development teams.
- **Controlled Environment Testing:** All testing was conducted in a controlled local development environment, which may not reflect the complexities, network latencies, and resource constraints of real-world deployment scenarios.
- **Time and Resource Boundaries:** As a bachelor’s thesis project, the research was constrained by academic timeline and resource limitations, preventing extensive longitudinal studies or large-scale evaluations.
- **Subjective Quality Assessment:** Code quality evaluation includes subjective elements that, while following structured rubrics, may not capture all aspects of code maintainability and long-term architectural robustness.

Despite these limitations, the focused approach enables deep analysis of MCP’s practical use within a specific but important domain, providing valuable insights for developers and researchers interested in context-aware code generation technologies.

2 Symphony, The Model Context Protocol (MCP) and Related Technologies

This chapter introduces the theoretical foundations underlying the proof-of-concept implementation. The chapter examines Symphony, the Model Context Protocol (MCP), function calling capabilities in Large Language Models, and the local LLM infrastructure provided by Ollama. These technologies form the technical stack for demonstrating context-sensitive code generation in Symphony applications.

2.1 Symphony Framework: Architecture and Code Generation Advantages



2.1.1 Symphony Framework Overview

Symphony is a mature PHP framework that provides structured foundations for web application development. The framework follows Model-View-Controller (MVC) architecture principles, creating separation of concerns[3] that facilitates automated code generation and analysis. This architectural approach enables predictable code patterns that are essential for effective LLM-based development assistance.

The framework is built around “very stable, high-quality codebase” with “extremely flexible and powerful object-mapping and query features” through its integration with Doctrine ORM [4]. The component-based design ensures that applications follow consistent structural patterns, making them suitable targets for automated analysis and code generation.

2.1.2 Best Practices and Standardization Benefits for LLM Code Generation

Symfony enforces development best practices through its architectural design and comprehensive documentation. This article[5] describes the best practices for developing web applications with Symfony. The framework establishes conventions that create consistency across projects, providing a structured foundation for automated analysis and code generation.

The framework’s emphasis on configuration standards, directory structure conventions, and naming patterns creates a predictable environment where LLMs can understand and extend existing codebases effectively. Symfony code is contributed by thousands of developers around the world. To make every piece of code look and feel familiar, Symfony defines some coding standards that all contributions must follow [6].

This standardization approach reduces ambiguity in code generation tasks by establishing clear patterns for common development scenarios. The comprehensive configuration system provides explicit project structure information through YAML, XML, or PHP configuration files, enabling MCP tools to extract routing patterns, service definitions, and architectural decisions that inform context-aware code generation.

2.1.3 Doctrine ORM Integration and Entity Management

Symfony’s integration with Doctrine ORM provides structured database interaction patterns essential for LLM code generation tasks. The Doctrine Project is the home to several PHP libraries primarily focused on database storage and object mapping [4]. The Object Relational Mapper (ORM) offers sophisticated entity management capabilities that complement Symfony’s architectural approach.

Doctrine’s entity mapping system uses attributes and configuration to define database relationships, constraints, and behaviors. This metadata-rich approach enables MCP tools to extract comprehensive information about data models, including entity relationships, field types, and constraints [7]. The structured nature of Doctrine entities provides clear patterns that LLMs can analyze and extend consistently.

The ORM’s emphasis on object-oriented design patterns creates predictable code structures for database interactions. Entity repositories, query builders, and relationship management follow established conventions that facilitate automated code generation. As demonstrated in Chapter 4’s evaluation, the MCP-enhanced approach successfully generated appropriate entity relationships and database constraints based on existing Doctrine patterns.

2.2 The Model Context Protocol (MCP): A "USB-C for AI Applications"

The Model Context Protocol is an open standard that enables developers to build secure, two-way connections between their data sources and AI-powered tools.[2] The protocol was announced by Anthropic in November 2024 as an open standard for connecting AI assistants to data systems such as content repositories, tools, and development environments.[8]

“Think of MCP like a USB-C port for AI applications. Just as USB-C provides a standardized way to connect your devices to various peripherals and accessories, MCP provides a standardized way to connect AI models to different data sources and tools.”[9] The protocol fundamentally transforms how AI systems interact with external resources by providing a universal communication standard. Instead of requiring customized integrations for every AI-tool combination, MCP establishes a common language that allows any MCP-compatible AI application to communicate with any MCP-compatible service. [2]

2.2.1 The N×M Integration Problem and MCP's Solution

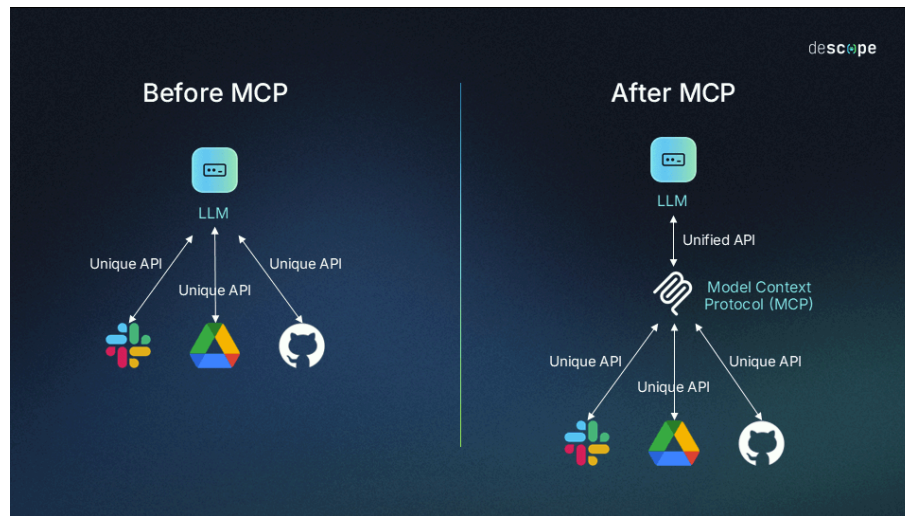


Figure 2.1: Before and after MCP: Traditional N×M integrations vs. standardized MCP approach. Source: [10]

The N×M Integration Problem: ‘M’ represents the numerous different AI applications (such as chatbots, IDE assistants, and custom agents), and ‘N’ stands for the vast array of external tools and systems (like GitHub, Slack, Asana, and databases) [11]

Traditional AI tool integration follows a pattern where each AI application must implement custom connectors for every external service it needs to access.[12] This creates an exponential integration problem.

Traditional Approach Challenges:

- **Custom Integrations Required:** Each AI tool requires unique integration code for each external service
- **Maintenance Overhead:** Updates to either AI tools or external services require modifications to custom integration code

MCP's Standardization Benefits:

- **Universal Protocol:** Single standard for all AI-tool communications
- **Reusable Components:** MCP servers can be used by any MCP-compatible client

- **Standardized Interfaces:** Consistent authentication, and data exchange patterns

Mathematical Representation:

- Traditional approach: $10 \text{ AI tools} \times 20 \text{ services} = 200 \text{ custom integrations}$
- MCP approach: $10 \text{ AI tools} + 20 \text{ MCP servers} = 30 \text{ implementations}$

2.2.2 MCP Protocol Architecture and Communication Model

MCP operates on a clear client-server architecture with three primary components:[13]

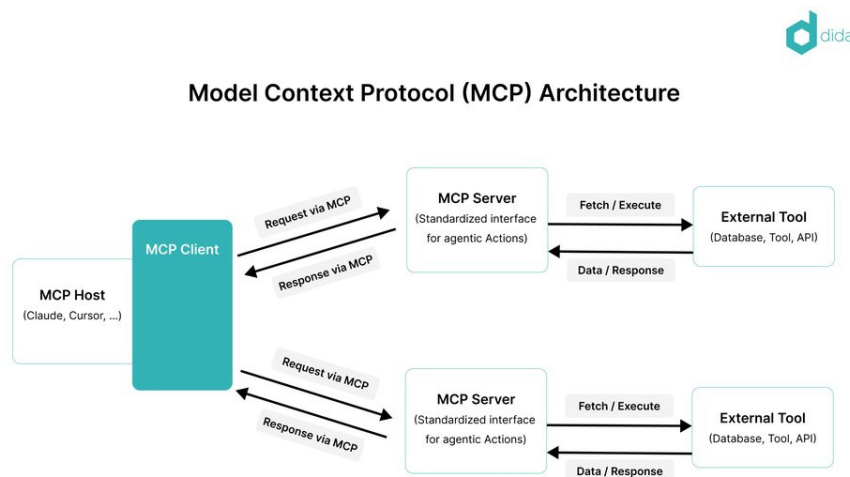


Figure 2.2: MCP Protocol Architecture showing the relationship between hosts, clients, and servers. Source: [14]

MCP Host: The AI application that coordinates and manages one or multiple MCP clients. Example:

- AI-powered IDEs (VS Code, Cursor)
- AI-assisted content creation (Claude Desktop)[12]

The host manages user interface interactions, orchestrates AI experiences, and **embeds MCP client** functionality.[12, 13]

MCP Client: A lightweight component within the host application that serves as an **intermediary** between the host/LLM and MCP servers. The client responsibilities include:

- Establishing and maintaining persistent connections with MCP servers[12, 13]
- Translating between host application requests and MCP protocol messages
- Managing message routing

MCP Server: Independent processes that provide specific functionalities through the standardized MCP. Servers can operate locally or remotely and expose:

- **Tools:** Executable functions that perform specific actions
- **Resources:** Data sources and information repositories
- **Prompts:** Template-based interaction patterns[12, 13]

The following example shows a typical MCP tool *calling* for a weather service, illustrating the JSON-RPC 2.0 request/response pattern used by the protocol.[13, 15]

```
1 {  
2   "jsonrpc": "2.0",  
3   "id": 3,  
4   "method": "tools/call",  
5   "params": {  
6     "name": "weather_current",  
7     "arguments": {  
8       "location": "San Francisco",  
9       "units": "imperial"  
10    }  
11  }  
12 }
```

Listing 2.1: Tool call Request example: weather_current function calling [13]

```
1 {  
2   "jsonrpc": "2.0",  
3   "id": 3,  
4   "result": {  
5     "content": [  

```

```
6      {
7          "type": "text",
8          "text": "Current weather in San Francisco: 68 F, partly cloudy with
light winds from the west at 8 mph. Humidity: 65%"
9      }
10  ]
11  }
12 }
```

Listing 2.2: Tool call Response example: weather_current function result [13]

Each server listens for JSON-RPC requests, exposes capabilities through structured metadata, and executes requested operations while returning standardized responses. [13]

2.2.3 JSON-RPC 2.0 Foundation

MCP builds upon JSON-RPC 2.0 for its communication protocol, providing several advantages: [15, 16]

- **Machine-readable:** Structured data exchange enables automated parsing and validation
- **Language-agnostic:** JSON-RPC implementations exist across all major programming languages
- **Efficient:** Lightweight protocol with minimal overhead
- **Standardized:** Well-established specification with clear semantics for requests, responses, and error handling

The protocol defines clear structures for capability announcement, request routing, and response formatting, ensuring consistent behavior across different implementations and environments.[15, 16]

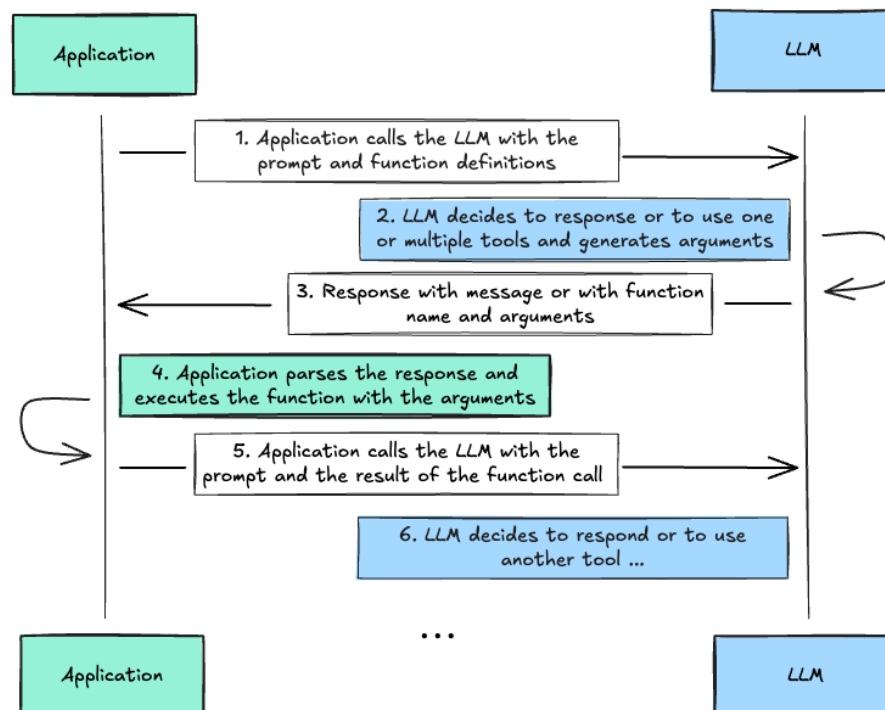


Figure 2.3: Function calling workflow without MCP showing the basic request-response pattern. Source: [17]

2.3 Large Language Models and Function Calling

2.3.1 Function Calling Fundamentals

Large Language Models (LLMs) traditionally produce text output. *Function calling* (also called *tool use*) extends this by letting a model request that the host invoke named tools with structured JSON arguments defined by a schema; the application executes the tool and returns results the model can condition on [18, 19].

Important distinction: LLMs do not execute functions themselves; they *propose* calls based on context and requirements. The application infrastructure handles the actual execution, parameter validation, and result integration.

2.3.2 MCP’s Role in Function Calling

MCP standardizes how tools, resources, and prompts are exposed and invoked, enabling hosts and models to discover and invoke capabilities in a consistent manner.[13, 20] The protocol does not modify model weights or capabilities; instead, it provides a clean, standardized interface for passing context and executing tools.

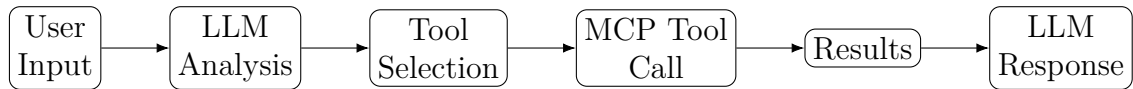


Figure 2.4: Tool-use pipeline with MCP providing standardized interfaces for the tool call stage

Implementation Considerations for Non-Native Function Calling: The proof-of-concept implementation uses a model without native function calling capabilities (DeepSeek-Coder:6.7b) [21] served via Ollama. In such cases, tools can be integrated through MCP by pre-executing MCP tools on the host and injecting their results into the prompt context, demonstrating the flexibility of the MCP approach.

2.4 DeepSeek-Coder Model Selection

This research utilizes DeepSeek-Coder:6.7b as the target language model for evaluating MCP effectiveness. DeepSeek-Coder is trained from scratch on 2T tokens, including 87% code and 13% natural language data in both English and Chinese languages, offered in model sizes of 1B, 5.7B, 6.7B and 33B [22].

The 6.7B parameter variant provides a balanced compromise between computational requirements and code generation capabilities.

Model Limitations: Several constraints influence the research design:

- **No Native Function Calling:** The base 6.7B model lacks built-in function calling capabilities, requiring context injection rather than direct MCP tool invocation[21–23]
- **Context Understanding:** The model may struggle to fully understand the context of a given prompt, potentially leading to responses that are not entirely relevant or accurate [21]
- **Complex Codebase Handling:** While DeepSeek Coder can process large amounts of code, it may struggle with very complex or nuanced codebases due to training on finite data [21]
- **Hardware Requirements:** The model requires 13.5GB VRAM with a 16K context window [24], constraining deployment options

These limitations necessitate the context injection approach demonstrated in this research, where MCP tools are pre-executed and results embedded in prompts rather than using dynamic function calling.

2.5 Ollama: Local LLM Infrastructure

2.5.1 Ollama Overview

Ollama is a lightweight runtime for executing large language models locally on macOS, Windows, and Linux.[25] It provides both a command-line interface (e.g., `ollama run`

<model>) and a local HTTP API (default at <http://localhost:11434> [26]) for chat completions, embeddings, and model management. Local execution offers benefits including privacy, offline capability, and low-latency iteration without external API dependencies.

2.5.2 Model Management and Customization

Pull & Run: Models are fetched and executed locally using commands like `ollama run deepseek-coder:6.7b`. [25] The Ollama model library documents available tags and usage patterns.

Modelfile Customization: A *Modelfile* enables declarative configuration of model defaults including system prompts, context length, temperature, and response templates. Custom model variants can be created using `ollama create`. [27]

API Integration: The REST API exposes endpoints for chat/generation, embeddings, and model lifecycle operations (list, pull, show, delete), [26] enabling programmatic integration with external applications.

2.5.3 Integration with MCP Architecture

Ollama’s local HTTP API integrates naturally with MCP-based tooling architectures. MCP servers provide context and tool execution capabilities, while Ollama supplies model inference. For models supporting native tool calling within Ollama, direct integration is possible; otherwise, the host can pre-execute MCP tools and inject results into prompts. The proof-of-concept demonstrates this latter approach, running `DeepSeek-Coder:6.7b` via Ollama and integrating external capabilities through MCP context injection rather than native function calling.

3 Proof of Concept Implementation

This chapter presents the design and implementation of a proof-of-concept system that demonstrates the integration of the Model Context Protocol (MCP) with Symfony for context-sensitive code generation using large language models. The implementation serves as a practical validation of the theoretical concepts outlined in Chapter 2, showing how MCP’s standardized architecture can enhance LLM-based code generation through structured context provision.

3.1 Implementation Goals and Architecture

Open Source Foundation: To ensure reproducibility and leverage established MCP protocol implementations, this proof-of-concept builds upon the php-llm community’s MCP demonstration framework[28]. The base framework provides validated MCP server-client communication patterns and tool execution infrastructure, enabling focus on the novel aspects of context injection for Symfony code generation rather than reimplementing protocol-level functionality.

Code availability: The complete proof-of-concept implementation (MCP server, client, evaluation framework, and experiment artifacts) is available in a read-only GitHub repository: <https://github.com/salem-zin-iden-naser/SymfonyMcpThesis>.

3.1.1 Key Implementation Objectives

- **Protocol Correctness:** Use MCP roles (host, client, server) and capability model (tools) as defined in Chapter 2
 - **Local Reproducible Stack:** Run all components locally (Symfony, MCP server, Ollama, DeepSeek Coder)
 - **Context Injection Approach:** Demonstrate context injection as a substitute for native function calling
-

- **Controlled Evaluation:** Expose HTTP endpoints to trigger scenarios and capture comparable model outputs

3.1.2 Target Application: Symfony Demo Project

The proof-of-concept uses the official Symfony Demo project¹ as the target codebase under analysis. This demo application contains typical blog entities (Post, Comment, User) and represents a realistic Symfony project structure. The repository was cloned as a sibling folder to enable MCP tools to analyze a real application without mixing repositories.

Folder Structure:

- `./my-poc/` (the MCP host, client, evaluation driver)
- `./symfony-demo/` (the Symfony Demo project to analyze)

Example path in the tool executor.

```
1 $entityDir = __DIR__ . '/../../../../../symfony-demo/src/Entity';  
2 if (!is_dir($entityDir)) {  
3     throw new \RuntimeException("Not found: $entityDir");  
4 }
```

This separation maintains clear boundaries between the proof-of-concept implementation and the target codebase while enabling MCP tools to access real project files for analysis.

3.1.3 System Architecture Overview

Figure 3.1 illustrates the complete system architecture, showing how the theoretical MCP components map to the concrete implementation and demonstrating the request flow from HTTP trigger through context collection to LLM response generation.

¹Available at: <https://github.com/symfony/demo>

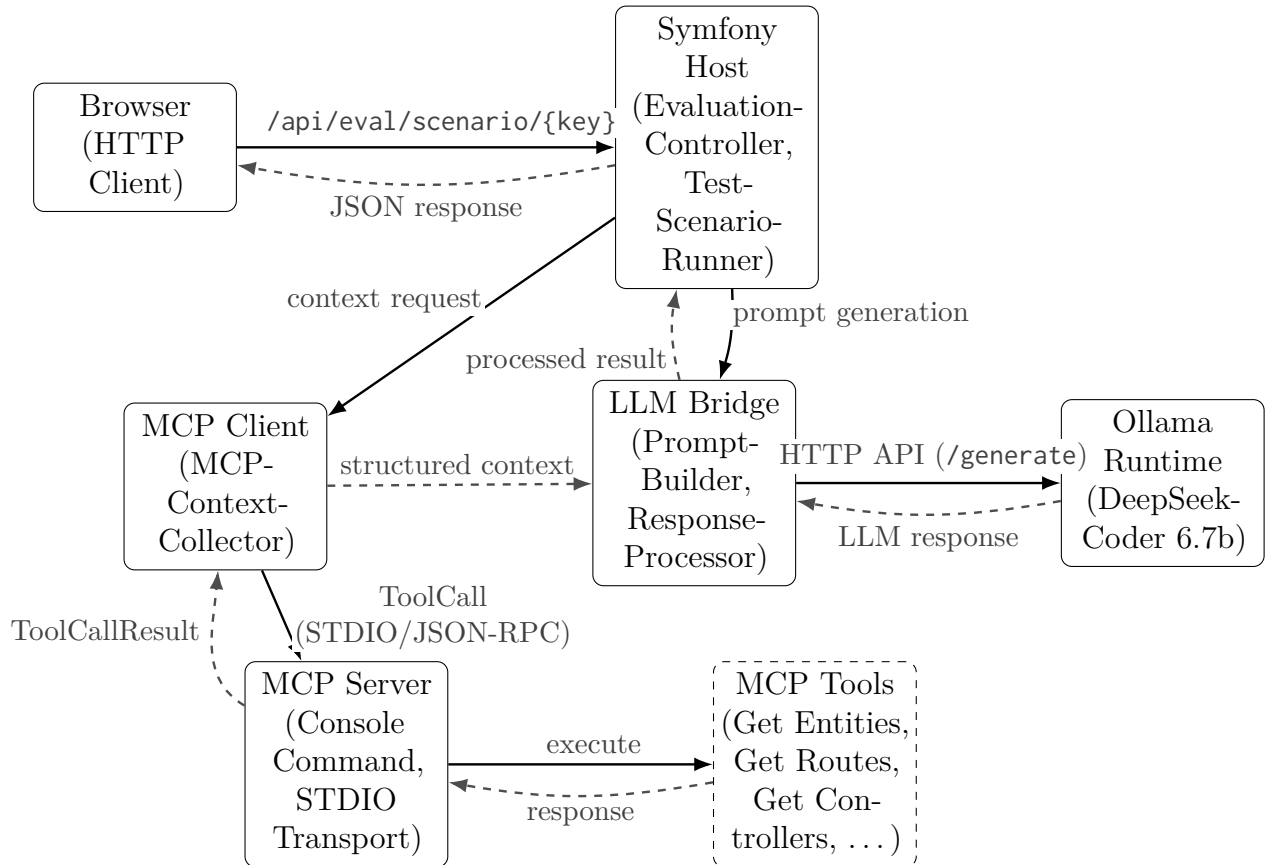


Figure 3.1: Proof-of-concept runtime architecture showing MCP integration with Symphony host coordinating context collection and LLM interaction

Operational Flow:

1. The user (browser) hits `/api/eval/scenario/add_like_feature` (Listing 3.9).
2. The `TestScenarioRunner` loads the scenario definition (Listing 3.10).
3. The host collects context via MCP: `MCPContextCollector` issues `ToolCalls` (e.g., *Get Entities*, *Get Routes*) through the server (Listings 3.1, 3.3, 3.5).
4. The `PromptBuilder` produces a contextual prompt (Listing 3.6); the bridge submits it to Ollama (Listing 3.7).
5. For comparison, a vanilla prompt leaves out the context bundle (Listing 3.6) and is sent to Ollama.
6. The `ResponseProcessor` normalizes outputs (Listing 3.7); results are returned to the controller.

3.2 MCP Server Implementation

3.2.1 Server Process and Transport

The MCP server implements the JSON-RPC 2.0 protocol specification through a Symfony Console command, providing STDIO transport for communication with the host application.

```
1 #[AsCommand('mcp', 'Starts an MCP server')]
2 class McpCommand extends Command
3 {
4     public function __construct(private readonly Server $server) { parent::__construct(); }
5
6     protected function execute(InputInterface $input, OutputInterface $output): int
7     {
8         $this->server->connect(new SymfonyConsoleTransport($input, $output)); // STDIO
9         $this->server->run();
10        return 0;
11    }
```

```
12 }
```

Listing 3.1: MCP server endpoint implementing STDIO transport

3.2.2 Tool Registry and Capability Advertisement

The server implements MCP’s capability announcement pattern through a structured tool collection, following the standardized metadata provision requirements:

```
1 class MyToolCollection implements CollectionInterface
2 {
3     public function getMetadata(): array
4     {
5         return [
6             new GetEntitiesToolMetadata(),
7             new GetFileContentToolMetadata(),
8             new ReadDirectoryToolMetadata(),
9             new SearchCodeToolMetadata(),
10            new GetRoutesToolMetadata(),
11            new GetUserRolesToolMetadata(),
12            new GetControllersToolMetadata(),
13        ];
14    }
15 }
```

Listing 3.2: Tool metadata collection advertised by the MCP server

3.2.3 Tool Execution Framework

The tool execution system routes requests to specific implementations based on standardized identifiers:

```
1 class MyToolExecutor implements ToolExecutorInterface
2 {
3     public function __construct(private readonly RouterInterface $router)
4     {}
5
6     public function call(ToolCall $input): ToolCallResult
7     {
8         try {
```

```
8         return match ($input->name) {
9             'Get Entities'           => (new GetEntitiesToolExecutor())
->call($input),
10             'Get File Content'       => (new GetFileContentToolExecutor
()->call($input),
11             'Read Directory'         => (new ReadDirectoryToolExecutor
()->call($input),
12             'Search Code'           => (new SearchCodeToolExecutor())
->call($input),
13             'Get Routes'             => (new GetRoutesToolExecutor(
$this->router))->call($input),
14             'Get User Roles'         => (new GetUserRolesToolExecutor()
)->call($input),
15             'Get Controllers'        => (new GetControllersToolExecutor
()->call($input),
16             default => throw new ToolNotFoundException($input, null, "
Unknown tool: {$input->name}"),
17         };
18     } catch (\Throwable $e) {
19         throw new ToolNotFoundException($input, $e, "Tool failed: {
$input->name}");
20     }
21 }
22 }
```

Listing 3.3: Tool dispatcher routing ToolCall requests to executors

Example Tool Implementation The `Get Entities` tool inspects Doctrine entities in the demo application and summarizes properties and relationships (useful grounding for code generation):

```
1 class GetEntitiesToolExecutor implements ToolExecutorInterface
2 {
3     public function call(ToolCall $input): ToolCallResult
4     {
5         $entityDir = __DIR__ . '/../.../.../symfony-demo/src/Entity';
6         if (!is_dir($entityDir)) {
7             throw new ToolExecutionException($input, new \RuntimeException(
            "Not found: $entityDir"));
9         }
```

```
8     }
9
10    $entitiesInfo = [];
11    foreach (scandir($entityDir) as $file) {
12        if (str_ends_with($file, '.php')) {
13            $name = pathinfo($file, PATHINFO_FILENAME);
14            $content = file_get_contents($entityDir . '/' . $file);
15            $entitiesInfo[] = [
16                'name' => $name,
17                'properties' => $this->extractProperties($content),
18                'relationships' => $this->extractRelationships($content
19            ),
20                'table_name' => $this->extractTableName($content),
21            ];
22        }
23    }
24
25    return new ToolCallResult($this->formatEntitiesInfo($entitiesInfo),
26        'text', 'text/plain', false, null);
27    }
28
29    private function extractRelationships(string $content): array
30    {
31        $relationships = [];
32
33        // OneToMany relationships
34        $pattern = '/#\[ORM\\\\OneToMany\(targetEntity:\s*(\w+):class,\s*'
35            .
36            '\s*mappedBy:\s*[\\"'](\w+)["\']*.*?\]\s*(?:#\[ORM\\\\OrderBy
37            .
38            '\s*private\s+Collection\s+\s*(\w+)/s';
39
40        if (preg_match_all($pattern, $content, $matches, PREG_SET_ORDER)) {
41            foreach ($matches as $match) {
42                $relationships[] = [
43                    'type' => 'OneToMany',
44                    'property' => $match[3],
45                    'target_entity' => $match[1],
46                    'mapped_by' => $match[2]
47                ];
48            }
49        }
50    }
51}
```

```
44         }
45     }
46
47     return $relationships;
48 }
49 }
```

Listing 3.4: GetEntitiesToolExecutor extracting Doctrine entity schema information

extractRelationships() implementation showcases how MCP tools can provide domain-specific intelligence that goes beyond generic file system access, offering structured semantic analysis of project components.

3.3 MCP Client Integration

3.3.1 Context Collection Service

The MCPContextCollector implements the MCP client functionality, serving as the interface between the Symfony host application and MCP server:

```
1 class MCPContextCollector
2 {
3     public function __construct(
4         private ToolExecutorInterface $toolExecutor,
5         private CollectionInterface $toolCollection
6     ) {}
7
8     public function collectProjectContext(string $projectPath, array
9     $toolIds = []): array
10    {
11        $context = [];
12
13        // Automatically collect all tool IDs if not explicitly specified
14        if (empty($toolIds)) {
15            $toolIds = array_map(fn($tool) => $tool->getName(),
16                               $this->toolCollection->getMetadata());
17        }
18    }
19 }
```

```
18     foreach ($toolIds as $toolName) {
19         try {
20             $toolCall = new ToolCall(
21                 id: uniqid('tool_', true),
22                 name: $toolName,
23                 arguments: ['path' => $projectPath]
24             );
25
26             $result = $this->toolExecutor->call($toolCall);
27             $context[$toolName] = json_decode($result->result, true) ??
$result->result;
28         } catch (\Throwable $e) {
29             $context[$toolName] = ['error' => $e->getMessage()];
30         }
31     }
32
33     return $context;
34 }
35 }
```

Listing 3.5: MCP client context collection service

3.4 LLM Integration and Context Injection Strategy

The host integrates an LLM through Ollama. Crucially, *tools are not invoked by the model*; instead, the host *pre-executes* tools and *injects* their results into the prompt (*context injection*). This implements the “tool use without function calling” path described in Chapter 2.

3.4.1 Prompt Construction Strategy

The implementation demonstrates context injection as an alternative to native function calling, pre-executing MCP tools and embedding results directly into LLM prompts:

```
1 class PromptBuilder
2 {
3     public function buildContextualPrompt(string $userPrompt, array
$mcpContext, string $taskType='general'): string
4     {
```

```
5     $context = json_encode($mcpContext, JSON_PRETTY_PRINT);
6     return <<<EOT
7 You are a Symfony expert.
8
9 ## Project Context
10 $context
11
12 ## User Request
13 $userPrompt
14
15 Provide a complete and clean solution.
16 EOT;
17     }
18
19     public function buildVanillaPrompt(string $userPrompt, string $taskType
20     = 'general'): string
21     {
22         return <<<EOT
23 You are a Symfony expert.
24
25 ## User Request
26 $userPrompt
27
28 Provide a complete and clean solution.
29 EOT;
30     }
```

Listing 3.6: PromptBuilder implementing context injection strategy

3.4.2 LLM Bridge Service

The LLMBridgeService implements dual-path processing, enabling direct comparison between MCP-enhanced and vanilla approaches:

```
1 class LLMBridgeService
2 {
3     private const DEFAULT_MODEL = 'deepseek-coder:6.7b';
4
5     public function __construct(
```



```
6     private OllamaClient $ollamaClient,
7     private MCPContextCollector $contextCollector,
8     private PromptBuilder $promptBuilder,
9     private ResponseProcessor $responseProcessor
10 ) {}
11
12 public function queryWithMCP(LLMRequest $req): LLMResponse
13 {
14     $context = $this->contextCollector->collectProjectContext($req->
getProjectPath(), $req->getRelevantTools());
15     $prompt = $this->promptBuilder->buildContextualPrompt($req->
getPrompt(), $context, $req->getTaskType());
16
17     $raw = $this->ollamaClient->generate($req->getModel() ?? self::
DEFAULT_MODEL, $prompt, $req->getOptions());
18     return $this->responseProcessor->processResponse($raw, true,
$context);
19 }
20
21 public function queryWithoutMCP(LLMRequest $req): LLMResponse
22 {
23     $prompt = $this->promptBuilder->buildVanillaPrompt($req->getPrompt
(), $req->getTaskType());
24     $raw = $this->ollamaClient->generate($req->getModel() ?? self::
DEFAULT_MODEL, $prompt, $req->getOptions());
25     return $this->responseProcessor->processResponse($raw, false);
26 }
27 }
```

Listing 3.7: LLM bridge service providing comparative evaluation capabilities

3.5 Evaluation Framework

3.5.1 Scenario Configuration

The evaluation system uses YAML configuration files to define test scenarios with specific tools, evaluation criteria, and expected outcomes:

```
1 scenarios:
2   add_like_feature:
```

```
3     category: feature
4     prompt: "Add the ability to like both posts and comments in a Symfony
5             app which has a post and comment entity."
6     task_type: feature
7     relevant_tools: [Get Entities, Get Routes]
8     expected_files: ["Post.php", "Comment.php", "PostController.php", "
9                     CommentController.php"]
10    evaluation_criteria: ["entity_update", "endpoint_creation", "
11                          correct_relationships"]
12    complexity_level: medium
```

Listing 3.8: Test scenario configuration demonstrating evaluation parameters

3.5.2 HTTP API for Evaluation Control

The system exposes REST endpoints for triggering evaluations and retrieving results:

```
1 #[Route('/api/eval')]
2 class EvaluationController extends AbstractController
3 {
4     public function __construct(private TestScenarioRunner $runner) {}
5
6     #[Route('/scenario/{key}', name: 'eval_key', methods: ['GET'])]
7     public function runByKey(string $key): JsonResponse
8     {
9         # test with http://localhost:8000/api/eval/scenario/
10        add_like_feature
11        ini_set('max_execution_time', 720);
12        $result = $this->runner->runByKey($key);
13        return $this->json($result);
14    }
15 }
```

Listing 3.9: Evaluation controller providing HTTP API for test execution

3.5.3 Test Scenario Runner

The scenario runner coordinates the evaluation process, executing both MCP-enhanced and vanilla approaches for comparison:

```
1 class TestScenarioRunner
2 {
3     public function __construct(
4         private LLMBridgeService $bridgeService,
5         private TestScenarioRepository $scenarioRepository,
6         private EvaluationMetricsCalculator $metricsCalculator
7     ) {}
8
9     public function runSingleScenario(TestScenario $scenario):
10     LLMTestResult
11     {
12         $request = new LLMRequest(
13             prompt: $scenario->getPrompt(),
14             projectPath: $_ENV['MCP_PROJECT_PATH'] ?? '',
15             taskType: $scenario->getTaskType(),
16             relevantTools: $scenario->getRelevantTools()
17         );
18
19         $mcpResponse = $this->bridgeService->queryWithMCP($request);
20         $vanillaResponse = $this->bridgeService->queryWithoutMCP($request);
21         $metrics = $this->metricsCalculator->calculate($scenario,
22             $mcpResponse, $vanillaResponse);
23
24         return new LLMTestResult($scenario, $mcpResponse, $vanillaResponse,
25             $metrics);
26     }
27 }
```

Listing 3.10: TestScenarioRunner coordinating comparative evaluation

3.6 Implementation Summary

The proof-of-concept successfully demonstrates the practical integration of MCP’s theoretical architecture within a Symfony application context. The system implements a complete MCP server with STDIO transport, exposes project analysis tools, integrates with Ollama for local LLM processing, and provides a controlled evaluation environment for comparing MCP-enhanced versus vanilla code generation approaches.

The architecture maintains clear separation between MCP components while enabling structured context provision to enhance LLM code generation capabilities. The implementation serves as a foundation for the empirical evaluation presented in Chapter 4.

4 Proof of Concept Results and Analysis

This chapter presents the results of evaluating the Model Context Protocol implementation described in Chapter 3. The evaluation compares code generation quality between MCP-enhanced and vanilla approaches using the "add like feature" scenario applied to the Symfony Demo project. The analysis examines both quantitative performance metrics and qualitative code assessment to determine MCP's effectiveness in improving context-sensitive code generation.

All evaluation artifacts namely the two requests (with MCP context and vanilla), their corresponding model responses, and the computed metrics are stored as JSON and available in the repository¹ at `symfony-context-mcp-agent/Results/LikePost_LikeComment.json`.

4.1 Generated Code

4.1.1 MCP-Enhanced Response

The MCP-enhanced approach had access to structured information about existing entities, their relationships, and project routing patterns. The key characteristics of the generated solution include:

Database Design:

```
1 namespace App\Entity;
2 use Doctrine\ORM\Mapping as ORM;
3
4 /** @ORM\Entity(repositoryClass="App\Repository\LikePostRepository") */
5 class LikePost
6 {
7     /** @ORM\Id() @ORM\GeneratedValue() @ORM\Column(type="integer") */
8     private $id;
9
10    /** @ORM\ManyToOne(targetEntity="App\Entity\User") */
11    private $user;
```

¹<https://github.com/salem-zin-iden-naser/SymfonyMcpThesis>

```
12
13     /** @ORM\ManyToOne(targetEntity="App\Entity\Post", inversedBy="likes")
14     */
15     private $post;
16
17     // Complete getter/setter implementation
18 }
```

Listing 4.1: MCP-generated LikePost entity with proper relationships

Entity Integration:

```
1 use Doctrine\Common\Collections\ArrayCollection;
2 use Doctrine\Common\Collections\Collection;
3
4 class Post
5 {
6     /** @ORM\OneToMany(targetEntity="App\Entity\LikePost", mappedBy="post")
7     */
8     private $likes;
9
10    public function __construct() { $this->likes = new ArrayCollection(); }
11    public function getLikes(): Collection { return $this->likes; }
12    public function addLike(LikePost $like): self { /* implementation */ }
13    public function removeLike(LikePost $like): self { /* implementation */ }
14 }
```

Listing 4.2: MCP-generated Post entity updates with relationship management

Controller Implementation:

```
1 class LikeController extends AbstractController
2 {
3     public function likePost(Post $post)
4     {
5         $user = $this->getUser();
6         if (!$user) { throw $this->createAccessDeniedException(/*...*/); }
7
8         $like = new LikePost();
9         $like->setUser($user);
10    }
```

```
10     $post->addLike($like);
11
12     $em = $this->getDoctrine()->getManager();
13     $em->persist($like);
14     $em->flush();
15
16     return $this->json(['likes_count' => count($post->getLikes())]);
17 }
18
19 public function unlikePost(Post $post) { /* complete implementation */
20 }
```

Listing 4.3: MCP-generated controller with complete like/unlike functionality

Routes Implementation:

```
1 like_post:
2   path: /post/{id}/like
3   controller: App\Controller\LikeController::likePost
4
5 unlike_post:
6   path: /post/{id}/unlike
7   controller: App\Controller\LikeController::unlikePost
8
9 # similar for comments
```

Listing 4.4: Routes (model output).

Key features of the MCP response:

- Created two separate entities: LikePost and LikeComment
 - Updated existing Post and Comment entities with proper relationships
 - Provided complete controller implementation with like/unlike methods
 - Suggested YAML routes (though not matching the project's attribute-based style)
-

4.1.2 Vanilla Response Analysis

The vanilla approach, lacking project context, generated a significantly different solution:

```
1 /** @ORM\Entity(repositoryClass="AppBundle\Repository\LikeEntityRepository") */
2 class LikeEntity
3 {
4     /** @ORM\Id() @ORM\GeneratedValue() @ORM\Column(type="integer") */
5     private $id;
6
7     /** @ORM\ManyToOne(targetEntity="AppBundle\Entity\Post") */
8     private $post;
9
10    /** @ORM\ManyToOne(targetEntity="AppBundle\Entity\Comment") */
11    private $comment;
12
13    /** @ORM\ManyToOne(targetEntity="AppBundle\Entity\User") */
14    private $user;
15 }
```

Listing 4.5: Vanilla-generated generic LikeEntity with design issues

Key features of the vanilla response:

- Created a single generic LikeEntity with optional references to posts or comments
 - Used outdated AppBundle namespace conventions
 - Provided no controller implementation or routing configuration
 - Generated incomplete solution requiring significant additional development
-

4.2 Quantitative, Qualitative and Architectural Analysis

4.2.1 Quantitative Performance Analysis

Table 4.1 presents the quantitative comparison between MCP-enhanced and vanilla approaches:

Metric	With MCP	Without MCP	Difference	Ratio
Prompt tokens (count)	1263	119	+1144	10.61×
Output tokens (count)	1656	588	+1068	2.82×
Prompt evaluation duration (s)	71.47	2.59	+68.88	27.58×
Generation duration (s)	261.35	75.22	+186.13	3.47×
Total duration (s)	341.61	77.83	+263.78	4.39×
Throughput (output tokens/s)	6.34	7.82	-1.48	0.81×
Total tokens processed	2919	707	+2212	4.13×

Table 4.1: Performance metrics comparison between MCP-enhanced and vanilla approaches

Performance Trade-offs: The quantitative analysis reveals clear trade-offs between context richness and processing efficiency:

- **Context Expansion:** The MCP approach required 10.61× more input tokens, reflecting the substantial project context provided to the model.
- **Processing Overhead:** Total processing time increased by 4.39× due to the larger context window and more complex prompt evaluation.
- **Output Completeness:** The MCP response generated 2.82× more output tokens, indicating significantly more comprehensive solutions.

Plain reading: MCP makes the prompt much bigger, so it runs slower, but it returns a longer and more complete answer.

4.2.2 Qualitative Code Assessment

Table 4.2 presents a structured assessment using a 0-2 point scale (0=missing, 1=partial, 2=complete):

Evaluation Criteria	With MCP	Without MCP
Entity updates (Post/Comment)	2	1
Endpoint creation (like/unlike)	2	0
Correct relationships (Doctrine)	2	0
Convention adherence (routing, DI)	1	0
Total Score (max 8)	7/8	1/8

Table 4.2: Requirements fulfillment comparison using structured rubric

4.2.3 Architectural Quality Analysis

Database Design Quality: The MCP approach generated two separate, focused entities (`LikePost`, `LikeComment`) with proper foreign key relationships. The vanilla approach used a single generic entity with nullable foreign keys, creating potential data integrity issues and violating database normalization principles.

Implementation Completeness: The MCP response provided end-to-end functionality including entities, relationship management, controller logic, and routing configuration. The vanilla response remained incomplete, requiring substantial additional development to achieve functional implementation.

Convention Alignment: While the MCP response used some outdated annotation syntax, it demonstrated understanding of existing project structure and attempted to follow established patterns. The vanilla response used obsolete `AppBundle` conventions, indicating no awareness of modern Symfony practices.

4.3 Implementation Validation

4.3.1 MCP Response Integration

To validate the practical applicability of generated code, the MCP response was implemented in the actual Symfony Demo project. Required modifications included:

- Updating annotation syntax to modern PHP attributes
- Converting YAML routes to attribute-based routing for consistency
- Adding database constraints to prevent duplicate likes
- Implementing proper dependency injection patterns
- Creating frontend templates for user interaction

The modified implementation demonstrates that the MCP-generated code provided a solid architectural foundation requiring primarily syntactic updates rather than structural redesign.

Example Updated Entity Implementation

```
1 #[ORM\Entity]
2 #[ORM\Table(name: 'like_post', uniqueConstraints: [
3     new ORM\UniqueConstraint(name: 'uniq_user_post_like', columns: ['
4         user_id', 'post_id'])
5 ])]
6 class LikePost
7 {
8     #[ORM\Id] #[ORM\GeneratedValue] #[ORM\Column]
9     private ?int $id = null;
10
11     #[ORM\ManyToOne(targetEntity: User::class)]
12     #[ORM\JoinColumn(nullable: false, onDelete: 'CASCADE')]
13     private ?User $user = null;
14
15     #[ORM\ManyToOne(targetEntity: Post::class, inversedBy: 'likes')]
16     #[ORM\JoinColumn(nullable: false, onDelete: 'CASCADE')]
17     private ?Post $post = null;
```

```
18 // Getter and setter methods...  
19 }
```

Listing 4.6: Production-ready LikePost entity with modern PHP attributes and constraints

4.3.2 Vanilla Response Integration

The vanilla response would have required much more extensive modifications and wouldn't have provided a working foundation. In fact, attempting to run the vanilla code wasn't practical due to its incomplete nature and outdated conventions.

4.4 Key Findings and Implications

- **Context-Driven Architecture Improvements:** The evaluation demonstrates that structured project context enables LLMs to make better architectural decisions. The MCP approach produced cleaner separation of concerns, proper relationship modeling, and more maintainable database designs compared to the generic patterns generated by the vanilla approach.
- **Solution Completeness Enhancement:** Context awareness significantly improved solution completeness. The MCP response provided comprehensive implementations spanning entities, controllers, and routing, while the vanilla response delivered only partial entity definitions requiring extensive additional development.
- **Performance vs. Quality Trade-offs:** The results highlight a clear trade-off between processing efficiency and output quality. While MCP context injection increased processing time by 4.39×, it delivered substantially more complete and architecturally sound solutions, suggesting the performance cost is justified for code generation tasks where quality takes precedence over speed.

4.5 Limitations and Considerations

- **Evaluation Scope:** This evaluation examined a single scenario with one model configuration. Broader validation across multiple scenarios, different model architectures, and various project types would strengthen the generalizability of these findings.
-

- **Subjective Assessment Factors:** The qualitative evaluation involved manual assessment of code quality, introducing potential subjective bias. Automated metrics for architectural quality and convention compliance could provide more objective evaluation frameworks.
- **Context Strategy Alternatives:** The study focused on comprehensive context injection without exploring selective context strategies or alternative prompt engineering approaches that might achieve similar benefits with reduced computational overhead.

4.6 Summary

The evaluation demonstrates that MCP’s structured context provision significantly enhances LLM-based code generation capabilities. The MCP-enhanced approach achieved a 7/8 requirements score compared to 1/8 for the vanilla approach, producing more complete, architecturally sound solutions despite requiring $4.39\times$ longer processing time.

The key insight is that providing structured project context enables more appropriate architectural decisions and comprehensive implementations. For practical code generation applications, the performance cost appears justified by the substantial quality improvements, particularly in scenarios where development efficiency and code quality outweigh generation speed considerations.

The successful integration of MCP-generated code into the target project validates the practical applicability of the approach, demonstrating that context-enhanced code generation can produce foundational implementations suitable for real-world development with minimal modifications.

5 Security Considerations in MCP Implementation

This chapter covers the essential security aspects when implementing the Model Context Protocol (MCP) with Symfony applications.

MCP allows AI models to access application data and tools. Without proper security, this can lead to unauthorized access and data breaches. MCP is designed with security in mind through process isolation and clear component boundaries, though the protocol introduces novel security challenges that require systematic analysis and enterprise-grade strategies [12, 29, 30].

5.1 Common Security Threats and Prevention

Understanding common attacks helps in building better defenses. Recent research has identified specific threat vectors unique to MCP implementations that extend beyond traditional web application security concerns [12].

Prompt Injection Attacks: Malicious users attempt to manipulate the AI by injecting harmful prompts into the context [29, 30].

Prevention:

- Validating and sanitizing all user inputs
- Using structured data formats instead of free text when possible
- Implementing content filtering for suspicious patterns

Session Hijacking: Attackers steal session IDs to impersonate legitimate users [29].

Prevention:

- Using secure, unpredictable session IDs
-

- Binding sessions to specific user information
- Implementing session timeout and rotation

Unauthorized Data Access: Compromised components might access data without proper authorization. This is particularly critical in MCP implementations where tools can access sensitive application data [30, 31].

Prevention:

- Implementing role-based access control (RBAC)
- Using the principle of least privilege
- Conducting regular security checks and access reviews

5.2 Development Best Practices

The following practices should be followed when building MCP servers with Symfony, incorporating enterprise-grade security frameworks specifically designed for MCP deployments [30].

Secure Coding Practices:

- **Input Validation:** Checking all parameters before processing [32]
- **Error Handling:** Not exposing system details in error messages
- **Logging:** Recording security events for monitoring and debugging
- **Dependencies:** Keeping all libraries and frameworks updated

Configuration Security:

- Using environment-specific configurations (dev, test, prod)
 - Disabling debug modes in production
 - Removing unused features and endpoints
 - Implementing proper secret management [33]
-

5.3 Monitoring and Compliance

Regular monitoring helps detect security issues early. MCP-specific monitoring requirements include tracking tool execution patterns and context access behaviors that may indicate security breaches [12].

Essential Monitoring:

- Authentication attempts and failures
- Unusual access patterns or data requests
- Tool execution logs with parameters
- System performance and availability
- Context injection anomalies and unauthorized tool invocations

Symfony’s logging framework integrates well with security monitoring tools [34]. Structured logging formats that security tools can easily parse should be considered.

5.4 Practical Security Checklist

The following checklist can be used to verify MCP implementation security, incorporating recommendations from recent enterprise security frameworks [30]:

- All communications using HTTPS/TLS encryption
 - Requiring authentication for all MCP servers
 - Validating user permissions before tool execution
 - Storing sensitive configuration securely
 - Validating and sanitizing all user inputs
 - Logging and monitoring security events
 - Applying regular security updates
 - Following least privilege principle for access controls
-

- Ensuring session management is secure and robust
- Ensuring error handling does not leak sensitive information
- Implementing MCP-specific threat detection for context manipulation
- Regular security assessments of tool capabilities and access patterns

By following these security guidelines, MCP implementations can be well-protected against common threats while maintaining the protocol’s powerful capabilities for context-sensitive code generation.

6 Conclusion and Future Outlook

This thesis presented a proof-of-concept implementation for integrating the Model Context Protocol (MCP) with the Symfony framework to enable context-sensitive code generation using large language models. This chapter summarizes the key findings, outlines the contributions made, and suggests promising avenues for future research.

6.1 Summary of Findings

The core objective of this work was to investigate whether providing structured project context via MCP could significantly enhance the quality of LLM-generated code. The evaluation, based on a concrete scenario of adding a "like" feature to a Symfony application, yielded clear and positive results.

- **Quality Over Speed:** The MCP-enhanced approach, while computationally more expensive ($4.39\times$ slower), produced a dramatically more complete and architecturally sound solution, scoring 7/8 on a structured rubric compared to 1/8 for the vanilla approach.
 - **Architectural Appropriateness:** With access to context, the LLM made superior design decisions, creating separate, focused entities (`LikePost`, `LikeComment`) with correct Doctrine relationships, as opposed to a single, generic, and flawed entity from the context-less model.
 - **Practical Applicability:** The code generated with MCP context served as a viable foundation for implementation, requiring only syntactic updates (e.g., converting annotations to attributes) rather than a structural redesign.
 - **Protocol Validation:** The implementation successfully demonstrated MCP's core value proposition: acting as a standardized "USB-C" interface that allows any LLM to access and reason about project-specific tools and data through a unified JSON-RPC protocol.
-

The thesis confirms that MCP effectively solves the context problem for LLMs in software development, transforming them from generic code suggesters into context-aware assistants that understand project structure, conventions, and existing codebase.

6.2 Contributions to the Field

This work provides several concrete contributions to the field of AI-assisted software engineering:

- **Practical MCP Implementation:** It offers a tangible, open-source blueprint for integrating MCP with a popular PHP framework (Symfony), serving as a reference for developers and researchers.
- **Context Injection Strategy:** It demonstrates and validates a practical "context injection" strategy for enabling tool use with LLMs that lack native function calling capabilities, broadening the applicability of MCP.
- **Empirical Evidence:** It provides empirical evidence quantifying the trade-off between generation performance and output quality when using MCP, supporting the argument that the cost is justified for complex tasks.
- **Evaluation Framework:** It establishes a reusable evaluation framework within Symfony for conducting comparative analyses of LLM-generated code, facilitating further research in this area.

This project moves beyond theoretical discussion of MCP by providing a working, evaluated prototype that underscores the protocol's potential to standardize and improve AI-assisted development workflows.

6.3 Future Research Directions

Based on the findings and limitations of this proof-of-concept, several compelling directions for future work emerge:

- **Broader Evaluation:** Expanding the research to include multiple LLMs (especially those with native function calling), diverse frameworks (Laravel, Django, Spring), and a wider set of complex development scenarios.

- **Advanced MCP Servers:** Developing more sophisticated MCP servers that provide deeper static analysis, understand architectural patterns, offer refactoring suggestions, or perform security examinations.
- **Optimized Context Strategies:** Research into intelligent context retrieval, determining which specific pieces of context are necessary for a given task to reduce token usage and latency without sacrificing quality.
- **Integration into Development Workflows:** Exploring tighter integration of MCP into developer IDEs and CI/CD pipelines for tasks like automated code review, documentation generation, and test case creation.
- **Security-Focused Analysis:** A dedicated study on the security implications of MCP, including the development of servers that can identify vulnerabilities, suggest patches, and enforce security best practices based on project context.

The integration of MCP with development frameworks marks a significant step towards more intelligent and intuitive programming assistants. This thesis lays a foundation upon which more advanced and powerful AI-powered development tools can be built.

Bibliography

- [1] Anthony Alford. *Anthropic Publishes Model Context Protocol Specification for LLM App Integration*. Dec. 24, 2024. URL: <https://www.infoq.com/news/2024/12/anthropic-model-context-protocol/> (visited on 07/20/2025).
 - [2] *Introduction*. “standardizes how applications provide context to large language models”. Model Context Protocol. 2025. URL: <https://modelcontextprotocol.io/introduction> (visited on 07/20/2025).
 - [3] *Exploring Symfony’s Code*. Symfony Documentation. URL: https://symfony.com/legacy/doc/gentle-introduction/1_4/en/02-Exploring-Symfony-s-Code (visited on 07/23/2025).
 - [4] *The Doctrine Project*. The Doctrine Project is the home to several PHP libraries primarily focused on database storage and object mapping. Doctrine Project. URL: <https://www.doctrine-project.org/> (visited on 07/23/2025).
 - [5] *The Symfony Framework Best Practices*. This article describes the best practices for developing web applications with Symfony that fit the philosophy envisioned by the original Symfony creators. Symfony Documentation. URL: https://symfony.com/doc/current/best_practices.html (visited on 07/23/2025).
 - [6] *Coding Standards*. Symfony code is contributed by thousands of developers around the world. To make every piece of code look and feel familiar, Symfony defines some coding standards that all contributions must follow. Symfony Documentation. URL: <https://symfony.com/doc/current/contributing/code/standards.html> (visited on 07/23/2025).
 - [7] *Basic Mapping*. Doctrine Object Relational Mapper Documentation. URL: <https://www.doctrine-project.org/projects/doctrine-orm/en/3.5/reference/basic-mapping.html> (visited on 07/23/2025).
 - [8] *Introducing the Model Context Protocol*. Official announcement of MCP and its purpose. Anthropic. Nov. 25, 2024. URL: <https://www.anthropic.com/news/model-context-protocol> (visited on 08/01/2025).
 - [9] *Model Context Protocol (MCP)*. Official documentation; includes the “USB-C for AI” analogy. Anthropic Docs. URL: <https://docs.anthropic.com/en/docs/mcp> (visited on 08/01/2025).
 - [10] Adeniyi Adebayo. *Understanding Model Context Protocol (MCP): The New Standard for AI Integration*. 2024. URL: <https://medium.com/@adeniyi221/understanding-model-context-protocol-mcp-the-new-standard-for-ai-integration-a588eae6fec8> (visited on 08/30/2025).
-

- [11] Yusuf Adeniyi Giwa. *The $N \times M$ Integration Problem and MCP's Solution*. The $N \times M$ Integration Problem. Medium. June 17, 2025. URL: <https://medium.com/@adeniyi221/understanding-model-context-protocol-mcp-the-new-standard-for-ai-integration-a588eae6fec8> (visited on 08/01/2025).
 - [12] Xinyi Hou et al. “Model Context Protocol (MCP): Landscape, Security Threats, and Future Research Directions”. In: *arXiv preprint arXiv:2503.23278* (2025). Comprehensive analysis of MCP architecture, security considerations, and implementation patterns. URL: <https://arxiv.org/abs/2503.23278> (visited on 08/01/2025).
 - [13] *Architecture Overview*. Model Context Protocol. 2025. URL: <https://modelcontextprotocol.io/docs/concepts/architecture> (visited on 08/01/2025).
 - [14] Dida Machine Learning. *A Practical Introduction to the Model Context Protocol (MCP)*. Dida. 2024. URL: <https://dida.do/blog/a-practical-introduction-to-the-model-context-protocol-mcp> (visited on 08/30/2025).
 - [15] *JSON-RPC 2.0 Specification*. Transport-agnostic RPC protocol used as the foundation for MCP messaging. JSON-RPC Working Group. URL: <https://www.jsonrpc.org/specification> (visited on 08/01/2025).
 - [16] *OpenRPC Specification*. Interface description standard for JSON-RPC 2.0 APIs. OpenRPC Initiative. URL: <https://spec.open-rpc.org/> (visited on 08/01/2025).
 - [17] Hugging Face. *Function Calling*. Documentation guide on function calling workflows. Hugging Face. 2024. URL: <https://huggingface.co/docs/hugs/en/guides/function-calling> (visited on 08/30/2025).
 - [18] *Function calling*. Function calling (also known as tool calling) with JSON schema-defined arguments. OpenAI. URL: <https://platform.openai.com/docs/guides/function-calling> (visited on 08/08/2025).
 - [19] *Tool use with Claude*. Official overview of Claude’s tool use: model proposes a tool call; client executes and returns results. Anthropic Documentation. URL: <https://docs.anthropic.com/en/docs/agents-and-tools/tool-use/overview> (visited on 08/08/2025).
 - [20] *Tools / OpenAI Agents SDK*. Lists Function tools and Local MCP servers as attachable tools. OpenAI. URL: <https://openai.github.io/openai-agents-js/guides/tools/> (visited on 08/08/2025).
 - [21] *Deepseek Coder 6.7b Base*. Dataloop. URL: https://dataloop.ai/library/model/deepseek-ai_deepseek-coder-67b-base/ (visited on 08/10/2025).
 - [22] *DeepSeek Coder: Let the Code Write Itself*. DeepSeek-AI. URL: <https://github.com/deepseek-ai/DeepSeek-Coder> (visited on 08/10/2025).
 - [23] *deepseek-coder-6.7b-instruct*. Official model card and capabilities overview. DeepSeek-AI on Hugging Face. URL: <https://huggingface.co/deepseek-ai/deepseek-coder-6.7b-instruct> (visited on 08/10/2025).
-

- [24] *Deepseek Coder 6.7B Instruct*. LLM Explorer. URL: <https://llm-explorer.com/model/deepseek-ai%2Fdeepseek-coder-6.7b-instruct,1W9vECdgv6BrZUP2duCq2> (visited on 08/10/2025).
 - [25] Carlos Monteiro. *How to Set Up and Run LLMs Locally Using Ollama Mac / Windows*. Set Up and Run LLMs Locally Using Ollama. Medium. URL: <https://medium.com/@ronakabhattrz/how-to-set-up-and-run-llms-locally-using-ollama-mac-windows-ec585eb128c3> (visited on 08/12/2025).
 - [26] *Ollama API Reference*. Official REST API endpoints including /api/generate and /api/chat on localhost:11434. Ollama. URL: <https://github.com/ollama/ollama/blob/main/docs/api.md> (visited on 08/12/2025).
 - [27] *Modelfile Reference*. Declarative configuration of model defaults, templates, and parameters. Ollama. URL: <https://github.com/ollama/ollama/blob/main/docs/modelfile.md> (visited on 08/21/2025).
 - [28] PHP-LLM Community. *MCP Demo Framework*. 2024. URL: <https://github.com/php-llm/mcp-demo> (visited on 06/15/2025).
 - [29] *Security Best Practices - Model Context Protocol*. Official security guidelines and best practices for MCP implementations. Model Context Protocol. June 18, 2025. URL: https://modelcontextprotocol.io/specification/2025-06-18/basic/security_best_practices (visited on 08/22/2025).
 - [30] Idan Habler Vineeth Sai Narajala. “Enterprise-Grade Security for the Model Context Protocol (MCP): Frameworks and Mitigation Strategies”. In: *arXiv preprint arXiv:2504.08623* (2025). URL: <https://arxiv.org/abs/2504.08623> (visited on 06/15/2025).
 - [31] *Model Context Protocol (MCP): Understanding security risks and controls*. Security testing and vulnerability management for MCP. Red Hat. July 25, 2025. URL: <https://www.redhat.com/en/blog/model-context-protocol-mcp-understanding-security-risks-and-controls> (visited on 08/22/2025).
 - [32] *Symfony Validation*. Symfony validation to check all parameters before processing. Symfony Documentation. URL: <https://symfony.com/doc/current/validation.html> (visited on 08/23/2025).
 - [33] *Managing Secrets in Symfony*. Symfony framework guide for secure secret management. Symfony Documentation. 2024. URL: <https://symfony.com/doc/current/configuration/secrets.html> (visited on 08/23/2025).
 - [34] *Logging*. Flexible logging capabilities that can be integrated with security information and event management (SIEM) systems. Symfony Documentation. URL: <https://symfony.com/doc/current/logging.html> (visited on 08/23/2025).
-

Figures and Tables

List of Figures

Figure 2.1: Before and after MCP: Traditional N×M integrations vs. standardized MCP approach. Source: [10]	14
Figure 2.2: MCP Protocol Architecture showing the relationship between hosts, clients, and servers. Source: [14]	15
Figure 2.3: Function calling workflow without MCP showing the basic request-response pattern. Source: [17]	18
Figure 2.4: Tool-use pipeline with MCP providing standardized interfaces for the tool call stage	19
Figure 3.1: Proof-of-concept runtime architecture showing MCP integration with Symphony host coordinating context collection and LLM interaction . . .	24

List of Tables

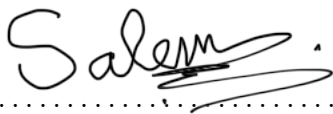
Table 4.1: Performance metrics comparison between MCP-enhanced and vanilla approaches	40
Table 4.2: Requirements fulfillment comparison using structured rubric	41

Declaration

I hereby declare that I have written this bachelor's thesis independently, without the assistance of third parties and without using any sources or aids other than those indicated. All passages taken verbatim or in substance from the sources used are individually identified as such.

This thesis has not previously been submitted to any other examination authority and has not been published.

I am aware that a false declaration will have legal consequences.



.....
Leipzig, September 15, 2025

Salem Zin Iden Naser

Born on 06.05.1996 in Kuwait, Kuwait

M.Nr: 80984

