Technical Implementation of Semantic Control in Eduba: An In-Depth Exploration

D. McDermott, J.E. Van Clief

September 22, 2024

Abstract

This sister paper provides a detailed technical exploration of the Eduba platform's implementation, focusing on the integration of Large Language Models (LLMs) with Microsoft's Semantic Kernel. Building upon the concepts presented in the previous paper, we delve into the specific code examples and architectural designs that enable the autonomous refinement of code and documents across various domains. We present practical implementations of Semantic Kernel plugins, elaborate on their functionalities, and demonstrate how they collaboratively support the system's objectives. Challenges encountered during development are discussed, along with the solutions employed to overcome them.

Contents

1 Introduction

The Eduba platform aims to revolutionize knowledge management by integrating AI-driven automation into collaborative environments. Central to this goal is the ability of the system to autonomously refine code and documents, ensuring consistency and alignment across multiple domains. This paper provides an in-depth technical examination of how Semantic Kernel facilitates this integration, with comprehensive code examples and explanations of the underlying mechanisms.

2 Overview of Eduba's Architecture

Eduba's architecture is designed to support seamless collaboration between human users and AI agents. The core components include:

- **LLM Integration**: Utilizing advanced language models to interpret and generate content.
- **Semantic Kernel Plugins**: Modular components that provide specific functionalities to the LLM.
- **Autonomous CI/CD Pipeline**: An automated pipeline that continuously integrates and deploys code changes.
- **Knowledge Management System**: A platform for managing documents, code, and other artifacts collaboratively.

The integration of these components allows Eduba to autonomously manage and refine knowledge assets while facilitating human-AI collaboration.

3 Semantic Kernel Integration

Semantic Kernel serves as the bridge between the LLM and the actionable components of the system. By developing custom plugins, we enable the LLM to perform tasks such as code editing, build execution, test automation, and documentation synchronization.

3.1 Plugin Architecture

Each plugin is implemented as a class with methods decorated by the `[SKFunction]` attribute, indicating that they are callable by the LLM. The plugins interact with the system's resources and provide the LLM with the necessary tools to execute actions.

4 Detailed Code Examples

In this section, we present code examples of key plugins, illustrating how they are implemented and how they contribute to the system's functionality.

4.1 Code Editing Plugin

The Code Editing Plugin allows the LLM to read, modify, and write code files.

Listing 1: Code Editing Plugin Implementation

```
using Microsoft.SemanticKernel;
using Microsoft.SemanticKernel.Orchestration;
using Microsoft.SemanticKernel.SkillDefinition;
using System.IO;

```

```
public class CodeEditingPlugin
{
    [SKFunction("Modify code files based on instructions.")]
    public async Task<SKContext> EditCodeAsync(
        [SKName("filePath"), SKDescription("The path to the code file to edit.")] string filePath,
        [SKName("instructions"), SKDescription("Instructions for modifying the code.")] string instructions
        SKContext context)
    {
        // Ensure the file exists
        if (!File.Exists(filePath))
        {
            context.Fail($"File not found: {filePath}");
            return context;
        }

        // Read the existing code
        string code = await File.ReadAllTextAsync(filePath);

        // Create a prompt for the LLM
        var prompt = $@"
You are a skilled C# developer. Modify the following code according to the instructions provided.

Instructions:
{instructions}

Code:
{code}

Modified Code:";

        // Invoke the LLM to get the modified code
        var result = await context.SemanticKernel.CompleteAsync(prompt);

        // Write the modified code back to the file
        await File.WriteAllTextAsync(filePath, result);

        context.Variables["Status"] = "Code modified successfully.";
        return context;
    }
}
```

Explanation:

- The plugin defines an asynchronous function `EditCodeAsync` that takes a file path and modification instructions.
- It reads the existing code from the file and constructs a prompt for the LLM.
- The LLM generates the modified code, which is then written back to the file.
- Context variables are updated to indicate the operation's success.

4.2 Build Execution Plugin

The Build Execution Plugin allows the LLM to initiate a build process and retrieve any errors.

Listing 2: Build Execution Plugin Implementation

```
using Microsoft.SemanticKernel;
using Microsoft.SemanticKernel.Orchestration;
using Microsoft.SemanticKernel.SkillDefinition;
using System.Diagnostics;

public class BuildExecutionPlugin
{
    [SKFunction("Execute the build process and return errors.")]
    public async Task<SKContext> ExecuteBuildAsync(
        [SKName("solutionPath"), SKDescription("The path to the solution file.")] string solutionPath,
        SKContext context)
    {
        // Configure the build process
        var startInfo = new ProcessStartInfo
        {
            FileName = "dotnet",
            Arguments = $"build \"{solutionPath}\"",
            RedirectStandardOutput = true,
            RedirectStandardError = true,
            UseShellExecute = false,
            CreateNoWindow = true
        };

        // Start the build process
        using var process = new Process { StartInfo = startInfo };
        process.Start();

        // Read output and errors
        string output = await process.StandardOutput.ReadToEndAsync();
        string errors = await process.StandardError.ReadToEndAsync();

        process.WaitForExit();

        // Update context with build results
        context.Variables["BuildOutput"] = output;
        context.Variables["BuildErrors"] = errors;

        return context;
    }
}
```

Explanation:

- The plugin runs the `dotnet build` command on the specified solution.
- Standard output and error streams are captured.
- Build results are stored in context variables for the LLM to analyze.

4.3 Test Execution Plugin

The Test Execution Plugin enables the LLM to run tests and analyze the results.

Listing 3: Test Execution Plugin Implementation

```
using Microsoft.SemanticKernel;
using Microsoft.SemanticKernel.Orchestration;
using Microsoft.SemanticKernel.SkillDefinition;
using System.Diagnostics;

public class TestExecutionPlugin
{
    [SKFunction("Run tests and return results.")]
    public async Task<SKContext> ExecuteTestsAsync(
        [SKName("testProjectPath"), SKDescription("The path to the test project.")] string testProjectPath
        SKContext context)
    {
        // Configure the test process
        var startInfo = new ProcessStartInfo
        {
            FileName = "dotnet",
            Arguments = $"test \"{testProjectPath}\"",
            RedirectStandardOutput = true,
            RedirectStandardError = true,
            UseShellExecute = false,
            CreateNoWindow = true
        };

        // Start the test process
        using var process = new Process { StartInfo = startInfo };
        process.Start();

        // Read output and errors
        string output = await process.StandardOutput.ReadToEndAsync();
        string errors = await process.StandardError.ReadToEndAsync();

        process.WaitForExit();

        // Update context with test results
        context.Variables["TestOutput"] = output;
        context.Variables["TestErrors"] = errors;

        return context;
    }
}
```

Explanation:

- The plugin runs the `dotnet test` command on the specified project.
- Captures the test output and any errors.
- Results are stored for further analysis by the LLM.

4.4 Merge Conflict Resolution Plugin

This plugin allows the LLM to resolve merge conflicts during code integration.

Listing 4: Merge Conflict Resolution Plugin Implementation

```
using Microsoft.SemanticKernel;
using Microsoft.SemanticKernel.Orchestration;
using Microsoft.SemanticKernel.SkillDefinition;
using System.IO;

public class MergeConflictResolutionPlugin
{
    [SKFunction("Resolve merge conflicts in a code file.")]
    public async Task<SKContext> ResolveConflictsAsync(
        [SKName("filePath"), SKDescription("The path to the conflicted code file.")] string filePath,
        SKContext context)
    {
        // Read the conflicting code
        string code = await File.ReadAllTextAsync(filePath);

        // Check for merge conflict markers
        if (!code.Contains("<<<<<<<") || !code.Contains(">>>>>>>"))
        {
            context.Fail("No merge conflicts found in the file.");
            return context;
        }

        // Create a prompt for the LLM
        var prompt = $@"
You are an expert in resolving code merge conflicts. The following code contains merge conflicts. Resolve

Conflicted Code:
{code}

Resolved Code:";

        // Invoke the LLM to resolve the conflicts
        var result = await context.SemanticKernel.CompleteAsync(prompt);

        // Write the resolved code back to the file
        await File.WriteAllTextAsync(filePath, result);

        context.Variables["Status"] = "Merge conflicts resolved successfully.";
        return context;
    }
}
```

Explanation:

- The plugin detects merge conflict markers in the code file.
- Constructs a prompt instructing the LLM to resolve the conflicts.
- The LLM generates the resolved code, which replaces the conflicted file.

4.5 Documentation Synchronization Plugin

This plugin ensures that the documentation stays consistent with the codebase.

Listing 5: Documentation Synchronization Plugin Implementation

```
using Microsoft.SemanticKernel;
using Microsoft.SemanticKernel.Orchestration;
using Microsoft.SemanticKernel.SkillDefinition;
using System.IO;

public class DocumentationSyncPlugin
{
    [SKFunction("Update documentation to align with code changes.")]
    public async Task<SKContext> SyncDocumentationAsync(
        [SKName("codeFilePath"), SKDescription("The path to the code file.")] string codeFilePath,
        [SKName("docFilePath"), SKDescription("The path to the documentation file.")] string docFilePath,
        SKContext context)
    {
        // Read the code and documentation
        string code = await File.ReadAllTextAsync(codeFilePath);
        string documentation = await File.ReadAllTextAsync(docFilePath);

        // Create a prompt for the LLM
        var prompt = $@"
You are a documentation specialist. Update the following documentation to reflect the current state of the

Code:
{code}

Current Documentation:
{documentation}

Updated Documentation:";

        // Invoke the LLM to update the documentation
        var result = await context.SemanticKernel.CompleteAsync(prompt);

        // Write the updated documentation back to the file
        await File.WriteAllTextAsync(docFilePath, result);

        context.Variables["Status"] = "Documentation synchronized successfully.";
        return context;
    }
}
```

Explanation:

- The plugin reads both the code and existing documentation.
- Uses the LLM to generate updated documentation that reflects code changes.

- Updates the documentation file with the new content.

5 Implementation Details

5.1 Orchestrating Plugin Interactions

The plugins work collaboratively to achieve the system's objectives. The orchestration involves:

- **Sequential Execution**: Plugins are executed in a specific order based on dependencies (e.g., code edits before builds).
- **Context Sharing**: Plugins share context variables, allowing them to access results from previous steps.
- **Error Handling**: If a plugin encounters an error, the system can decide to retry, skip, or escalate the issue.

5.2 Handling Errors and Exceptions

Robust error handling is crucial for an autonomous system. Strategies include:

- **Validation Checks**: Before executing actions, plugins validate inputs and preconditions.
- **Retries**: Certain operations may be retried with modified parameters if they fail initially.
- **Fallback Mechanisms**: If the LLM fails to produce a valid output, the system can revert to default behaviors or notify human operators.

6 Challenges and Solutions

6.1 Limitations of LLMs

Challenge:
LLMs may generate incorrect or suboptimal code, particularly in complex scenarios.
Solution:

- **Prompt Engineering**: Crafting precise and unambiguous prompts to guide the LLM effectively.
- **Post-Generation Validation**: Implementing automated checks to verify the correctness of the LLM's outputs.
- **Human-in-the-Loop**: Allowing human reviewers to oversee critical changes, especially in production environments.

6.2 Ensuring Code Quality and Security

Challenge:
Automatically generated code may not adhere to best practices or may introduce security vulnerabilities.
Solution:

- **Static Analysis Tools**: Integrating tools like SonarQube or Roslyn analyzers to assess code quality.
- **Security Scanners**: Employing security-focused analysis to detect potential vulnerabilities.
- **Coding Standards Enforcement**: Defining and enforcing coding standards through automated checks.

6.3 Resource Management

Challenge:
Running multiple processes (builds, tests, LLM computations) can strain system resources.
Solution:

- **Asynchronous Execution**: Utilizing async programming patterns to optimize resource usage.
- **Task Scheduling**: Prioritizing tasks based on urgency and resource availability.
- **Scalable Infrastructure**: Leveraging cloud resources to scale computational power as needed.

7 Extending to Document Management

While the examples above focus on code, similar principles apply to document management.

7.1 Document Editing Plugin

A plugin for editing documents based on collaborative inputs.

Listing 6: Document Editing Plugin Implementation

```
using Microsoft.SemanticKernel;
using Microsoft.SemanticKernel.Orchestration;
using Microsoft.SemanticKernel.SkillDefinition;
using System.IO;

public class DocumentEditingPlugin
{
    [SKFunction("Edit a document based on provided instructions.")]
    public async Task<SKContext> EditDocumentAsync(
        [SKName("docFilePath"), SKDescription("The path to the document file.")] string docFilePath,
        [SKName("instructions"), SKDescription("Instructions for editing the document.")] string instructi
        SKContext context)
    {
        // Ensure the file exists
        if (!File.Exists(docFilePath))
        {
            context.Fail($"Document not found: {docFilePath}");
            return context;
        }

        // Read the existing document
        string document = await File.ReadAllTextAsync(docFilePath);

        // Create a prompt for the LLM
        var prompt = $@"
You are an expert editor. Modify the following document according to the instructions provided.

Instructions:
{instructions}

Document:
{document}

Edited Document:";

        // Invoke the LLM to edit the document
        var result = await context.SemanticKernel.CompleteAsync(prompt);

        // Write the edited document back to the file
        await File.WriteAllTextAsync(docFilePath, result);

        context.Variables["Status"] = "Document edited successfully.";
        return context;
    }
}
```

Explanation:

- Similar to the Code Editing Plugin but tailored for document content.
- Useful for incorporating contributions from multiple collaborators.

7.2 Knowledge Synthesis Plugin

A plugin that synthesizes information from multiple documents.

Listing 7: Knowledge Synthesis Plugin Implementation

```
using Microsoft.SemanticKernel;
using Microsoft.SemanticKernel.Orchestration;
using Microsoft.SemanticKernel.SkillDefinition;
using System.Collections.Generic;
using System.IO;

public class KnowledgeSynthesisPlugin
{
    [SKFunction("Synthesize information from multiple documents.")]
    public async Task<SKContext> SynthesizeKnowledgeAsync(
        [SKName("docFilePaths"), SKDescription("List of document file paths.")] List<string> docFilePaths,
        SKContext context)
    {
        // Read all documents
        var documents = new List<string>();
        foreach (var path in docFilePaths)
        {
            if (File.Exists(path))
            {
                documents.Add(await File.ReadAllTextAsync(path));
            }
            else
            {
                context.Fail($"Document not found: {path}");
                return context;
            }
        }

        // Create a prompt for the LLM
        var prompt = $@"
You are an expert researcher. Synthesize the key information from the following documents into a cohesive

Documents:
{string.Join("\n\n---\n\n", documents)}

Summary:";

        // Invoke the LLM to generate the synthesis
        var result = await context.SemanticKernel.CompleteAsync(prompt);

        // Store the summary in context
        context.Variables["Summary"] = result;

        return context;
    }
}
```

Explanation:

- Reads multiple documents and constructs a prompt for synthesis.
- The LLM produces a summary that combines the key points.
- Useful for knowledge management and ensuring consistency across documents.

8 Security and Ethical Considerations

8.1 Access Control

Implementation:

- Plugins check user permissions before performing actions.
- Sensitive operations require elevated privileges.

8.2 Data Privacy

Implementation:

- Sensitive data is encrypted at rest and in transit.
- The LLM is restricted from accessing confidential information unless necessary.

8.3 Ethical AI Use

Considerations:

- Ensuring the LLM does not generate biased or inappropriate content.
- Implementing oversight mechanisms to monitor and correct undesirable outputs.

9 Conclusion

The Eduba platform's integration of LLMs with Semantic Kernel enables a powerful, autonomous system capable of refining code and documents across various domains. By implementing specialized plugins and robust orchestration mechanisms, the system can perform complex tasks that traditionally require human intervention. While challenges exist, thoughtful design and continuous improvement ensure that the system operates effectively and ethically.

References

- Microsoft Semantic Kernel: `https://github.com/microsoft/semantic-kernel`
- OpenAI GPT-4: `https://openai.com/product/gpt-4`
- Eduba Project Documentation: `https://eduba.example.com/docs`
- C# Documentation: `https://docs.microsoft.com/en-us/dotnet/csharp/`
- .NET Command-Line Tools: `https://docs.microsoft.com/en-us/dotnet/core/tools/`