Autonomous Legacy Web Application Upgrades Using a Multi-Agent System

Valtteri Ala-Salmi, Zeeshan Rasheed, Abdul Malik Sami, Zheying Zhang, Kai-Kristian Kemell, Jussi Rasku, Shahbaz Siddeeq, Mika Saari and Pekka Abrahamsson

Faculty of Information Technology and Communication Science, Tampere University, Finland
{*valtteri.ala-salmi, zeeshan.rasheed, malik.sami, zheying.zhang, kai-kristian.kemell, jussi.rasku, shahbaz.siddeeq, mika.saari, pekka.abrahamsson*}*@tuni.fi*

Keywords: Artificial Intelligence, Large Language Models, CakePHP, OpenAI, Multi-Agent System, Web Framework.

Abstract: The use of Large Language Models (LLMs) for autonomously generating code has become a topic of interest in emerging technologies. As the technology improves, new possibilities for LLMs use in programming continue to expand such as code refactoring, security enhancements, and legacy application upgrades. Nowadays, a large number of web applications on the internet are outdated, raising challenges related to security and reliability. Many companies continue to use these applications because upgrading to the latest technologies is often a complex and costly task. To this end, we proposed LLM based multi-agent system that autonomously upgrade the legacy web application into latest version. The proposed multi-agent system distributes tasks across multiple phases and updates all files to the latest version. To evaluate the proposed multi-agent system, we utilized Zero-Shot Learning (ZSL) and One-Shot Learning (OSL) prompts, providing the same instructions for both. The evaluation process was conducted by updating a number of view files in the application and counting the amount and type of errors in the resulting files. In more complex tasks, the amount of succeeded requirements was counted. The prompts were run with the proposed system and with the LLM as a standalone. The process was repeated multiple times to take the stochastic nature of LLM's into account. The result indicates that the proposed system is able to keep context of the updating process across various tasks and multiple agents. The system could return better solutions compared to the base model in some test cases. Based on the evaluation, the system contributes as a working foundation for future model implementations with existing code. The study also shows the capability of LLM to update small outdated files with high precision, even with basic prompts. The code is publicly available on GitHub: https://github.com/alasalm1/Multi-agent-pipeline.

1 INTRODUCTION

Generative Artificial Intelligence (GAI) has advanced rapidly in the previous years. With the network model transformer proposed in (Vaswani, 2017), the previous computational limitations related to neural networks has been passed. Compared to earlier architectures like Recurrent Neural Networks (RNNs), transformers are capable of being parallelized, allowing tokens to be processed simultaneously in a self-attention mechanism (Vaswani, 2017). This allows efficient scalable computation of generative models in a graphical/tensor processing units, leading their possible size to grow greatly.

Generative models performance improves when their size is increased as demonstrated in language models by (Kaplan et al., 2020). Large Language Models (LLMs) with advanced transformer architectures, such as Generative Pre-trained Transformers (GPTs) (Radford and Narasimhan, 2018) and Bidirectional Encoder Representations from Transformers (BERTs) (Devlin et al., 2018), have emerged as sizable and influential models in natural language processing.

With LLM's improving as larger and better trained models are introduced, the question arises as to whether the models could be used to update existing applications. The internet hosts a huge number of websites that contain deprecated components, which may affect their functionality and compatibility with modern standards. According to (Demir et al., 2021), 95% of the analyzed 5.6 million web applications had at least one deprecated component. Updating an application takes notable resources which grows with

Ala-Salmi, V., Rasheed, Z., Sami, A. M., Zhang, Z., Kemell, K.-K., Rasku, J., Siddeeq, S., Saari, M. and Abrahamsson, P.
Autonomous Legacy Web Application Upgrades Using a Multi-Agent System.
DOI: 10.5220/0013397200003928

In *Proceedings of the 20th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2025)*, pages 185-196
ISBN: 978-989-758-742-9; ISSN: 2184-4895

the technical debt and eventually it becomes deprecated. The application owners are often stuck with the application as it is crucial to the business and updating is not possible without a big investment (Ali et al., 2020). Challenges arise in safety (Smyth, 2023; Sami et al., 2024), usability (Ali et al., 2020), (Rasheed et al., 2024b), and compatibility (Antal et al., 2016) of the web applications due to companies' hesitation to undertake complex and potentially cost-ineffective operations.

In this study, we propose a multi-agent system to conduct a complex set of operations in phases for updating legacy application files. Each agent is assigned specific tasks, working collaboratively to accomplish the overall objective. For instance, a verification agent gives feedback for every executed implementation phase to ensure that phase's validity. The results of the proposed system were then tested using Zero-Shot Learning (ZSL) and One-Shot Learning (OSL) prompts (Brown et al., 2020) to assess the system's suitability for the process and the overall capability of current LLMs to implement the update.

We conducted the comparison and evaluation of the system by using it to update six view files' web framework compatibility in a legacy web application. We used A ZSL prompt in comparison with five files and an OSL prompt with two files against the system. We counted the same error once during the evaluation, giving a suitable metric to evaluate the performance of the operation. In more complex tasks, we counted the completed requirements to evaluate problem-solving in more challenging scenarios. The results show that the system updated a file with 0.46 different errors higher when compared against the best prompt implementation, on average. The results regarding requirements were varied with one task completed better by the system and one task completed with a lower performance than the compared prompt. The evaluation results are publicly available for validating the study (Tampere University and Rasheed, 2025).

Below our contribution can be summarized as follows:

- The proposed system is designed to autonomously update legacy web applications to the latest version using a multi-agent system.
- The system was evaluated using an existing legacy web application and updating view files belonging to it. Then depending on the task, errors or fulfilled requirements were counted. The system had 0.406 more errors on average and varying performance in the requirements depending on the task.
- The system was compared to the standalone prompts giving perspective of the systems functionality and overall capability of LLM to implement code updating tasks.
- We publicly released the evaluation results dataset to access all the collected data for validating our study (Tampere University and Rasheed, 2025).

The rest of the paper is organized as follows: Section 2 presents a background study on code generation using LLMs. Section 3 explains the methodology of this paper, followed by the results in Section 4. Section 5 discusses the implications and future directions of the results, and the study concludes in Section 6.

2 BACKGROUND

2.1 Code Generation Using AI Agents

Lately, various studies have implemented different agent systems to improve code generation in LLM's (Rasheed et al., 2023), (Rasheed et al., 2024c). This subsection investigates different implementations and their observed benefits compared to baseline code generation using LLM.

The ability of using a self-feedback loop to improve code output has been observed in Madaan *et al.* (Madaan et al., 2024) with implementation of SELF-REFINE. It consists of a base prompt, a feedback prompt and a refiner prompt in which feedback system gives feedback of the base result and refiner improves it iteratively. SELF-REFINE had 8.7% improvement in GPT-4 model in code optimization (Madaan et al., 2024).

In another implementation called Reflexion studied in Shinn *et al.* (Shinn et al., 2024) an agent was connected to an evaluator which gave numeric evaluation of the agent output in the given operation. This was then processed in a self-reflection system which added textual reflective analysis in the agents memory to be used in the next outputs. Reflexion could perform in the HumanEval benchmark, a dataset containing Python programming, with the result of 91.0% (Shinn et al., 2024).

One way of improvement has been testing the produced code to get a real feedback whether the produced code works. In Huang *et al.* (Huang et al., 2023) system called AgentCoder was proposed. In the system one agent was tasked to produce code, the second one was tasked to invent test cases to the code and the last agent executed the test cases and provided results from the test to other agents. A feedback loop based on a real testing data achieved 32.7% better compared to ZSL prompt of GPT-4 (Huang et al.,

2023). In HumanEval it obtained similar result as Reflexion, resulting in 91.5% score (Huang et al., 2023).

Another implementation based on testing was created in Zhong *et al.* (Zhong et al., 2024) named Large Language Debugger (LDB). In the system an agent generated program was divided into control blocks and then individual test cases were created and executed to each block by the agent-model. With LDB connected to Reflexion framework the resulting HumanEval percentage was measured in 95.1% (Zhong et al., 2024).

Based on these studies, both self reflection of an agent and testing of the produced code did result in a remarkable rise of quality in code output in LLMs. The combination of both in the same system did give the highest result, making them possibly mutually reinforcing in code generation quality.

2.2 Challenges in LLM Generated Code

The code generated by LLM has challenges that have been recognized in the literature. During background study, following challenges were found in three studies focusing on recognizing them:

Challenges in code generation using LLM

- The quality of the produced code decreases with the length of the code and the difficulty of the task (Liu et al., 2024b; Dou et al., 2024; Chong et al., 2024).
- LLM does not always consider the requirements of the user, or those which are needed for reliable and safe code (Liu et al., 2024b; Dou et al., 2024; Chong et al., 2024).
- LLM can find problems in the code that do not exist when asked to give feedback from it (Liu et al., 2024b; Chong et al., 2024).

The quality decrease of the LLM code by length and difficulty was found in two studies. In (Liu et al., 2024b) LLM's ability to solve different Python and Java coding tasks was tested. Based on the study findings the probability of returned code working correctly decreases with harder code tasks and the length of the produced code. In another study (Dou et al., 2024), different programming tasks were generated by different LLMs and evaluated by syntax, runtime, and LLM's functionality errors. The study found an increased failure rate with more code lines, code complexity, and required API calls to produce code. In (Chong et al., 2024) the security of LLM-generated code was evaluated. The findings show that creating a memory buffer correctly in complex tasks, for example, the multiplication of two floats had a much lower success rate of 1.5% than in an easier task like subtracting a float from an integer with a success rate of 50.1% (Chong et al., 2024).

LLM also has problems producing reliable, safe code and occasionally failing the user request (Rasheed et al., 2024d), (Sami et al., 2025). In Liu *et al.* errors found in the code were mostly related to wrong outputs (27%) and badly styled and lowly maintainable code (47%) while runtime errors were much lower (4%) (Liu et al., 2024b). The badly styled and lowly maintainable code included categories like a redundant modifier, ambiguously named variables, and too many local variables in a function or method. Dou *et al.* found that the LLM's functionality is the most notable reason for bugs in the code, particularly misunderstanding of the provided problem and logical errors in the code (Dou et al., 2024). This included, for example, failed corner case checking, undefined conditional branches, and a complete misunderstanding of the provided problem. Chong *et al.* compared LLM code to human-generated code in 220 files. The study found that while LLM generates fewer lines of code, the code lacks defensive programming that exists in the human-written code (Chong et al., 2024). Additionally, an SHA generation algorithm was provided by LLM as a faulty version while AES and MD5 succeeded, making the completion of similar tasks unreliable (Chong et al., 2024).

The studies found that while a feedback loop can improve code generation it can also cause additional errors to the code. Liu *et al* found that giving feedback improves produced code up to 60% but with a possibility of additional errors added to the code (Liu et al., 2024b). Chong *et al.* found that the LLM can generate new security problems in the code by a feedback loop and not just remove them, especially if the file does not contain problems in the first place.

When compared to the studies of multi-agent systems, it seems that observed challenges can be improved with the help of a multi-agent system. The observed improvement of self-feedback (Liu et al., 2024b; Chong et al., 2024), has been implemented in (Madaan et al., 2024; Shinn et al., 2024; Huang et al., 2023; Zhong et al., 2024), resulting in improvement in the code evaluation metrics despite the risks of additional errors. Additional improvement using test results in (Huang et al., 2023; Zhong et al., 2024), can also be seen as a way to solve LLM's problems in functionality as observed in (Dou et al., 2024). As multi-agent systems have been shown to add value to code generation in LLM by being able to address some of its challenges, the proposed multi-agent pipeline is expected to add value in updating code in software.

2.3 Challenges in Upgrading Legacy Applications

Legacy systems are outdated implementations by used technologies and programming languages with hardware, software, and other parts of the system being possibly obsolete (Sommerville, 2016). When it comes to strategies for dealing with legacy systems the possibilities are disposing of the system, keeping the system as such, and re-engineering or replacing components in the legacy system (Sommerville, 2016). In this study, the focus is on re-engineering and replacing components on the application side with the help of a multi-agent system. Below is a summary of found challenges in legacy applications based on case studies:

Challenges in updating legacy applications

- The programmers can have knowledge gaps in either old or new technologies of a legacy application (De Marco et al., 2018; Fritzsch et al., 2019).
- Identifying updated components' input, output, and code functionality is a demanding, time taking task in legacy applications (De Marco et al., 2018; Vesić and Laković, 2023).
- Breaking the updating process down into smaller parts and successfully managing them is a challenge in legacy applications (De Marco et al., 2018; Fritzsch et al., 2019).

First, programmers might have knowledge gaps in technology either in the original legacy application or the new version. As mentioned by (De Marco et al., 2018), a legacy mainframe application was migrated to Linux servers with changes to the database and a transition in programming language from COBOL to Java. With feature development, the paper mentions that the COBOL team has challenges with Java programming language and vice versa, causing a knowledge gap between the teams (De Marco et al., 2018). In Fritzsch *et al*. 14 legacy applications in different stages of migration to microservices were analysed (Fritzsch et al., 2019). When it came to the recognized challenges in the migration, the lack of expertise was the shared first cause as knowledge of microservice architecture was not high enough with the developers (Fritzsch et al., 2019).

Second, a component of a legacy application has a challenging task to correctly recognize correct input, output, and code functionality. In De Marco *et al*. the testing phase of the migration caused one-year delay for the project. The reason for this included a lack of high-level tests that made recognizing input, output, and inner functionalities of a component a time-taking task (De Marco et al., 2018). Additionally, obsolete code took time to correctly identify its functionality inside a component (De Marco et al., 2018). In (Vesić and Laković, 2023) a framework for legacy system evaluation was presented with an analyse of an existing legacy system. During the analysis the studied information system of water and sewerage disposal company showed multiple problems. The system lacked proper documentation, lack of personnel and people with knowledge of the whole system, and poor software architecture (Vesić and Laković, 2023). These problems shows that if the software side were updated, identifying of the component's behavior would be a complex task as in the (De Marco et al., 2018).

Lastly, breaking the legacy system update into smaller tasks and managing them is a recognised challenge. In Fritzsch *et al*. decomposition of the application was the second shared first reason for technical challenges in studied projects (Fritzsch et al., 2019). In De Marco *et al*. the decomposited work packets were tried to be used to successfully forecast the project duration but failed due to differences in batch-orientation (De Marco et al., 2018).

As stated in section 1, at least one component is considered obsolete in 95% of web applications (Demir et al., 2021), making the problem relevant in the industry. The recognized challenges found in updating legacy systems could be solved with the help of an LLM. As LLM can be trained to have knowledge of various technologies (Brown et al., 2020), it could help with the knowledge gap of project developers. Additionally, LLM could analyse application components and save time by understanding the component's functionality. LLM could be used to divide and manage tasks of the project at different levels. Combined with the findings of the previous subsections, a multi-agent system that could provide solution to these challenges is a relevant option to conduct upgrades in legacy applications.

3 RESEARCH METHOD

3.1 Research Questions

In this study we propose following two Research Questions (RQs):

> ***RQ1.*** *How to utilize multi-agent system to update legacy project into latest by refactoring deprecated code?*

The aim of RQ1 is to test the capability of a multi-agent system to autonomously upgrade legacy web

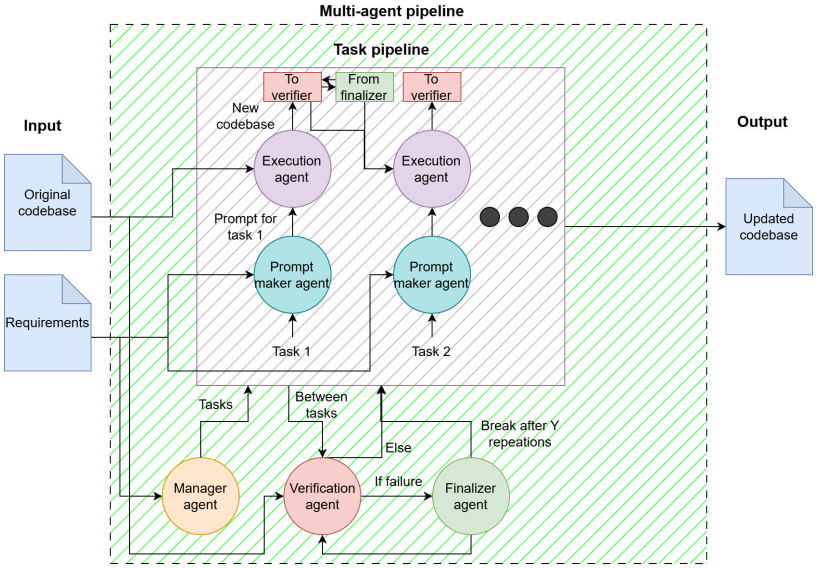

Figure 1: Proposed system: Multi-agent pipeline for updating existing code.

applications. This objective focuses on assessing how effectively the system can modify and enhance outdated functionalities by refactoring deprecated code without manual intervention.

> ***RQ2.*** *How to validate the proposed multi-agent system?*

The aim of RQ2 is to define a suitable metric to validate a multi-agent system meant for autonomously update deprecated code. This objective focuses on creating a metric and then using it to validate the proposed system.

3.2 Proposed Multi-Agent System

The proposed multi-agent system is referenced from the CodePori multi-agent system described in (Rasheed et al., 2024a). In the CodePori system, software operating in Python code are generated based on a given project description instructing the system, for example, creating a simple game or a face recognizer (Rasheed et al., 2024a). The outcome is generated by a six agent-framework tasked to operate different software development team roles including a manager, developers, finalizer, and a verifier.

In the proposed system illustrated in figure 1 the multi-agent system is designed to modify already existing code based on the user requirements. The system receives an original codebase and the requirements from the user as input. In the requirements the user specifies, what type of operations they want to commit to the code. For example, updating the code compatibility from version X to version Y or changing the libraries that the code uses. The system is composed of four units which are explained below:

Manager Agent: The manager agent receives the updating requirements from the user and is tasked to write them into manageable operations in a chronological execution order. The tasks are written in abstract level, which are later defined by the task pipeline where the list of tasks is send. The manager agent is asked once before sending the tasks to ensure that the tasks are in chronological order and overall related to the requirements.

Task Pipeline: The task pipeline consists of prompt makers and execution agents illustrated in figure 1 inside multi-agent pipeline. For every task the prompt maker agent creates an OSL prompt which is executed by execution agent. This creates a pipeline of OSL prompts that are executed sequentially to the given code. After every task the code is send for the verification agent for reviewing the completion of the task. When every task is completed in the pipeline it will give the updated code as an output for the user.

Verification Agent: The role of the verification agent is to ensure that a task has been completed. It will analyse the new version of the code and estimate whether it satisfies the requirements. If the verification agent accepts the task, it will return the code for the new task in the pipeline. If there is something left in the task, it will send the code to the finalizer agent.

Finalizer Agent: The finalizer agent makes changes to the updated code if the verification agent notices that the tasked operation has not been completed successfully. After executing changes, the finalizer agent returns the modified code back to verifi-

cation agent for the next analyze. If the feedback loop between the verification agent and the finalizer agent exceeds a certain amount of interactions, the finalizer agent will send the code back to the task pipeline. This might happen, for example, if the verification agent starts to hallucinate and find problems that does not exist.

Expected improvements of the system compared to ZSL/OSL prompts when updating existing code are based on the following features:

1. *Self-division:* The division of code update into smaller tasks avoids particularly long prompts. As observed in (Liu et al., 2024a) information in long prompts is hard to access by LLM's especially in the middle of the prompt. As the whole task prompt contains instructions, the instructions in the middle will not be necessary completed, causing task to partially fail.
2. *Self-feedback:* The output of every task is analysed on the verifier agent and then possibly improved by the finalizer agent iteratively, generating a self-feedback loop. As found in the background studies, self-feedback improves results in code generation (Liu et al., 2024b; Chong et al., 2024). As updating the code to a new syntax and improving it's functionality keeps code logically same, the improvement is expected in the output.
3. *Self-instructive:* Manually writing complex prompts to instruct LLM for updating existing code is a hard task for people that are not familiar with prompt engineering. In (Zamfirescu-Pereira et al., 2023) non expert prompt writers were studied, which found problems with over generalization and human interactive style of writing prompts resulting in bad performance. With self-instruction an user needs only to write the needed task without adding detailed instructions or examples.

3.3 Evaluation Subjects and Goal

The evaluated legacy web application is built on CakePHP 1.2, which is a PHP based web framework with development started in 2005 (CakePHP, 2022). During this evaluation the main goal was to update files to version 4.5 of the web framework (CakePHP, 2023) with additional challenges that are related to each updated file. CakePHP 1.2 was released on 2008 (Koschuetzki, 2008) and CakePHP 4.5 on 2023 (CakePHP, 2023), making the version gap between the versions 15 years.

CakePHP is built based on Model-View-Controller (MVC) architecture (CakePHP, 2022) which is a software design pattern with controller acting as a mediator between a model and a view. Between versions 1.2 and 4.5 CakePHP has got a multiple changes in the design and syntax. For example, in CakePHP 3.0 the Object–Relational Mapping (ORM) was re-built (CakePHP, 2024a) and the required version of PHP was raised to 7.4 in the version 4.x of CakePHP (CakePHP, 2024b) from the original PHP version of 4 and 5 (CakePHP, 2022).

We conducted a comparison between the proposed system and the alternatives with a module consisting of six view files, which files B-F belonging to the functionalities of an already updated controller file in a legacy web application. The web application in this study is an electronic dictionary described in (Norri et al., 2020). The dictionary is a search and editing tool for a Postgres database, which contains informative data on medieval medical English vocabulary. The component of five files form a part of the dictionary where medieval medical variants can be searched based on different features like name, original language and wildcards (Norri et al., 2020). The files, their size and any recognized challenges are seen in the table 1.

Table 1: Evaluated files.

File	LOC	Challenges
View A	35	Changed name to access to data
View B	25	Array syntax access
View C	57	Array syntax access
View D	190	Contains JavaScript and PHP
View E	19	Ajax form to be updated to JQuery
View F	118	Four helper functions must be replaced

View A is a simple reference list that shows a reference and terms related to a certain reference identifier and works as the first test in study. Challenge related to file update is changed access to data with different name. View B is tasked to show quotes that are connected to a specific variant. View B shows searched variants that are part of a certain quote and uses the functionalities of View D and E. In both View B and C, a requirement is to take changed data format into account as the Controller in the version 4.5 uses ORM resultsets introduced in CakePHP 3 (CakePHP, 2024c). However, unlike in the normal case where reference is done by object syntax, the reference must be done by array syntax with first field-name starting in a capital letter.

View D is a dynamic list that shows variants based on their first letter and the language of origin. View C contains both PHP and JavaScript code to add challenge to updating process. Unlike in View A and B, the ORM objects are accessed in a normal nota-

tion. View E is an Ajax form used in View C which is needed to be remade with jQuery. Therefore, the architecture needs to be updated as well along with updating the CakePHP syntax. View F file is an element view file which is made to highlight a search world of the search results in View C file. Besides of the highlight functionalities, four helper functions were wanted to be replaced with modern library implementations.

3.4 Evaluation Process

We tested the proposed system against a ZSL and, when needed, against an OSL prompt to update the view files. In ZSL, the prompt is defined to not include any examples of the task, whereas a OSL prompt includes exactly one example of the given task (Brown et al., 2020). An OSL prompt is typically used when a task requires a custom example to guide the model's behavior, especially in cases where a ZSL prompt does not produce the desired results. The use of prompts as a metric to evaluate LLM techniques was explored by Ouedraogo *et al*. (Ouédraogo et al., 2024), where ZSL and OSL prompts were compared with different reasoning methods for LLMs.

In a ZSL prompt the GPT-model was asked to update a view file without any examples. An OSL prompt included instructions for updating the file with an example given. The loopback in the system was set with the maximum of two iterations. We ran the system and the compared alternatives with the ChatGPT 4o-mini model (OpenAI, 2024a).

The test was repeated ten times for every prompt/file to take the stochastic nature of the LLM's into account (Brown et al., 2020). When a view file was tested, the connected controller file was in the new version of the web framework.

The View files A, B, C, and D were evaluated by different errors in the code. The evaluation was done manually with both static and dynamic testing used to find errors from the updated file. The errors found were divided into following categories:

1. *Fatal Errors:* Errors that causes the file to not run, for example, syntax errors.
2. *Runtime Errors:* Errors that do not prevent running the file but are encountered during the usage of the file.
3. *Content Errors:* The feature works but its functionality is different than in the original file.
4. *Missing/Additional features:* The updated file is missing or have additional features not existing in the file
5. *Failed generation:* If the updated file has more than 7 different type of errors or the answer does not contain the code the generation is considered as failed and the evaluation is stopped.

The same error caused by the same mistake was counted only once to ensure that recurring syntax errors do not disproportionately affect the comparison with other types of errors. From the results we calculated Standard Deviation (SD) to measure similarity of the code generation errors across the repetitions. The duration of the request was counted along with the Lines of Code (LOC) in the updated files. The reasoning behind counting LOC is to analyse whether there is difference with the length of produced code between a OSL/ZSL prompt and the system.

For View E and F we refactored entire code based on complex requirements. The evaluation process was decided to be ranked based on the requirements it passes. For every correctly working requirement, we gave a value of 1 and 0 for incorrect ones. The evaluation was conducted manually as the error counting. The features for View E and F are following:

Requirements for View E and F update

Requirements of View E updated form:

- The form sends data and receives results as expected.
- Dropdown menus show correct suggestions.
- The send button and dropdown objects work as intended.

Requirements of View F highlighting feature:

- The highlighting works in normal case with no special characters or wildcards.
- Highlighting words with special medieval English letters.
- Highlighting works with wildcards with the highlight only to the part of word where the wildcard was placed in the search query.

Referring to jQuery file as a URL or a local file were both accepted. Also data name sent to controller was slightly different this was also disregarded from error evaluation. Besides these, no additional errors were fixed in the evaluation. The results of View F were evaluated similar to the View E along with the number of replaced functions with library implementations. Not replaced functions were used in the file with added guard to not re-declare them in the testing process, but otherwise left as they were in the output.

4 RESULTS

In this section, we present the results of our proposed system. The results of systems suitability are pro-

vided in Section 4.1, and validation in Section 4.2.

4.1 Suitability of Proposed System for Updating a Deprecated File (RQ1)

We updated View A file with a following ZSL prompt:

> Update whole cakePHP view file from version 1.2 to version 4.5. Change $mainreference to $bookReference while removing ['Reference'] between wanted objects, access it ORM style. Write $term lowercase.

The prompt was added to a framework that included the updated code and a request to only return the updated code. With ZSL the file was returned each time without errors in average of 7.7 seconds. With the system the first sentence and remaining sentences were each a one requirement. The system returned the file five times correctly with avarage of 0.5 errors in the file with avarage of 56.8 seconds.

We first attempted to update View B with a ZSL prompt. The prompt to address this problem was written in the following format:

> Update CakePHP view file from version 1.2 to version 4.5. Requirement: $quotes is an ORM\ResultSet made of arrays accessed with ['Fieldname']['fieldname'] and must be accessed so in the updated file.

The prompt returned the updated file in the average of 1.6 different errors and in average of 2.3 seconds. The prompt failed in all ten times to use ['Fieldname']['fieldname'] instead using format ['fieldname']['fieldname']. In a total of six times the prompt failed to access correctly to the ORM object resulting in a fatal error. Based on these remarks an short OSL prompt was created trying to address the resulted problems in the following format:

> Update CakePHP view file from version 1.2 to version 4.5. Requirement: $quotes are ORM\ResultSets made of arrays accessed with ['Fieldname']['fieldname'] and must be accessed so in the updated file. Example: old syntax: ['apple']['lemon'] new syntax: ['Apple']['lemon']. Use function first() instead of [0] to the ORM object.

Used in the same framework, the results showed improvement with 0.4 of average amount of errors in the file. The file was updated correctly in total of seven times out of ten. As this is a relatively low average, the prompt was kept such with the exception of the last sentence removed and tested with a more complex View C file. The results with the OSL prompt were once again impressive with average amount of errors being 0.7 with the file correctly updated four times.

The ZSL and OSL prompt were next compared with the proposed system. Based on the testing of the prompts, the requirement file was written into following format:

> Requirement1: Update whole CakePHP view file from version 1.2 to version 4.5.
>
> Requirement2: ORM Arrays must be accessed with array style syntax ['Fieldname']['fieldname'] with the first fieldname starting with a capitalized letter and the second only with lowercase letters. Use first() when referring to first member in the array.

With View C the last sentence was removed. The system returned view B with an average of 53.2 seconds and 0.6 different errors. One of the runs was a statistical outlier with the return time of 165 seconds. The system succeeded four times to return the file correctly. With View C the system returned it with average return time of 60 seconds and 1.0 different errors on average. The system returned the file correctly in three times.

View D was updated using the system and a ZSL prompt. As the ORM objects are now accessed with the normal notation, therefore, prompt could be simplified notably and ZSL was determined to be enough for the comparision. Following ZSL prompt were used in the updating process:

> Update whole cakePHP view file from version 1.2 to version 4.5. Use ORM access with $variant with direct access to name and id.

The file was returned with an average of 10.7 seconds and an average of 0.3 different errors. In total of seven times the prompt returned a fully correct file. With the system, the updating process was conducted with same prompt with added requirement identifier for each sentence. The system performed poorer than a ZSL prompt with an average of errors 1.22 in 67.8 seconds. One generation did fail with the full code not being generated and is not included in the error averages.

The values of the evaluation has been collected in the Table 2. Based on the results, the proposed system and OSL/ZSL prompt are both capable of updating an deprecated code file with a high precision. However, OSL/ZSL prompt seems to perform slightly better in the evaluated files, especially in the View A and D. Average lines of code (ALOC) between methods were similar in size.

During the evaluation we counted the different error types which are shown in figure 2. Only fatal and

Table 2: Evaluation of updated View files A, B, C and D.

File	Method	Different errors	SD	ALOC	Time (s)
View A	ZSL	0	0.000	35	7.7
View B	ZSL	1.6	0.490	22	2.3
View B	OSL	0.4	0.663	22	5
View C	OSL	0.7	0.640	57	7.4
View D	ZSL	0.3	0.459	160	10.7
View A	Syst.	0.5	0.500	36	56.8
View B	Syst.	0.6	0.490	24	53.2
View C	Syst.	1.0	0.850	60	60.4
View D	Syst.	1.22	0.786	164	70.9

runtime errors were found from the generated files. In View A errors were only encountered in the system with around same level of each errro category. Changing from ZSL to OSL lowered both the fatal and runtime errors average in View B. Using the system the number of fatal errors did decrease further, but the amount of runtime errors did increase slightly.

In View C changing from OSL to system did slightly increase the amount of both observed errors. In View D ZSL did produce only fatal errors but when switched to system the files did contain runtime errors with notably higher rate than fatal errors observed in the files using ZSL. Overall, in View B and D the system did decrease fatal errors and increase runtime errors while in View A and C both types where increased slightly.

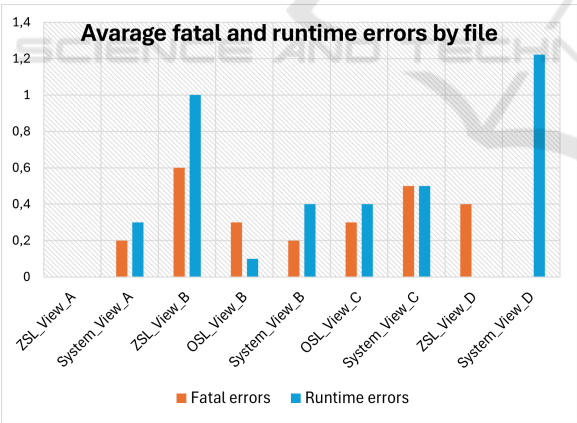

Figure 2: Found error types by method and file.

4.2 Validation of the Proposed System (RQ2)

We conducted the validation process for the harder tasks View E and F with a ZSL prompt and the system. Following ZSL prompt was made to task the update in View E:

> Update CakePHP version 1.2 ajax form into CakePHP 4.5 version with jQuery architecture including jquery-3.6.0.min file. Make the jQuery implementation fully functional version with dropdown updated every time the letter is written to it

With the system the prompt was split into two requirements by sentences. The files were evaluated based on the requirements and added to Table 3. The ZSL was able to fulfill the request with a value of 0.5 with once correctly providing completely functional form. The system had value of 0.9 with the view, twice giving the fully correct form. The system created on average twice as large code file compared to the ZSL implementation.

Table 3: Passed requirements by average of View E.

Method	Regt 1	Regt 2	Regt 3	Total	ALOC
ZSL	0.2	0.1	0.2	0.5	42
Syst. (2 tasks)	0.5	0.3	0.2	0.9	86

The ZSL prompt for View F was formed as following with the model divided into two requirements by sentences:

> Update CakePHP element view file from 1.2 to 4.5. Replace functions in the element view file by ready made PHP libraries or update them if not found

The results are collected in the table 4. The ZSL prompt had average requirement value of 1.8 and average of 3.2 replaced functions (RF) with five files passing all requirements. With the system dividing the prompt in the two tasks resulted average requirement value of 0.5 with 2.7 RF with no completely passing version. This was repeated with the system running with only one task resulting total requirement value of 1.6 and value of 1.7 in RF with four files passing the requirements fully.

Table 4: Passed requirements and replaced functions by average of View F.

Method	Regt 1	Regt 2	Regt 3	Total	RF	ALOC
ZSL	0.7	0.6	0.5	1.8	3.2	42
Syst. (1 task)	0.6	0.6	0.4	1.6	1.7	57
Syst. (2 tasks)	0.3	0	0.2	0.5	2.7	52

The results show that the system performed much weaker compared to the ZSL prompt when divided into two subtasks. When the system was run on a single task, the results closely aligned with the requirements; however, the ZSL prompt successfully replaced more functions with library alternatives. Most issues encountered during file updates were related to references to non-existent CakePHP functions, high-

lighting hallucinations commonly found in code generated by LLMs (Dou et al., 2024).

Based on these tests, the proposed system does not perform notably better compared to ZSL in more complex problem solving tasks but in some settings might perform much worse compared to a ZSL prompt. Overall, the results indicates ability of LLM to solve complex coding tasks like library replacements and transforming code to work in different library architectures. The evaluation results are publicly available for further validation (Tampere University and Rasheed, 2025).

5 DISCUSSION

During this study, we updated files belonging to an existing deprecated web application using GAI. The results shows that multi agent systems are capable of updating small deprecated web application files with a high precision with low rates of error and a low mean deviation of errors between the generations. The multi-agent system could provide completely working versions of code in every studied task that required, for example, library replacements.

The observed ability of LLM to generate not only new code but to update existing one has an potential impact for the software industry. As notable part of the industry is related for upkeeping existing applications, the possibility of automatising the upkeep process by at least partially by artificial intelligence, will result in much faster and cheaper process of the application updating. Estimations of code maintenance has been ranged, for example, at 85-90% (Erlikh, 2000) and 40-80% (Davis, 2009). Automation of maintenance can be expected to lower the percentage considerably, freeing resources of software companies.

The question also rises for LLM's capability of transforming existing code across coding languages. If the context can be kept enough, a system designed to such task could translate code across coding languages similar to human languages. This already have studies (Pan et al., 2024; Eniser et al., 2024), that shows different methods to translate code in various languages. However, the translation percentage did not in either of those studies rise above 50%, indicating that more advancements are still needed in the field to reach necessary level of quality.

We proposed a multi-agent system called multi-agent pipeline for completing a code update in sequences with a self-feedback loop. The results shows that ZSL/OSL prompt produces usually better results compared to the proposed system. There are possible multiple reasons for this:

Telephone Game: Telephone game is a play where information is passed in a chain for one player to another. The longer the game continues more distorted the original message becomes. With a chain of agents the possibility of code logic starting to cumulatively change is a possible reason for lower results. Despite the verifier agent checking the results compared to the original code, the possibility of distorted code is still existing if it goes undetected by the verifier agent. Another risk is the hallucinations which added to the codebase can create faulty code which was encountered when updating View F file.

LLM Reasoning Skills: The system used in testing 4o mini might not have the reasoning skills required to fulfill the tasks of more complex agents like the verifier or the manager agent. Testing of complex multi-agent roles needs to be repeated in a more advanced system and compare results to this study. For example, new LLM specialised for problem solving like GPT o1 (OpenAI, 2024b) could make verification agent better performing with the potential ability in problem solving.

Prompt Following: As found in (Dou et al., 2024) the LLM has challenges to understand the given task from the prompt. With the proposed system the prompt following capabilities did not have notable improvement compared to the tested short prompts as seen with the fulfilled requirements. The given prompt might be needed to be sent further processed with agents before sent to the task-pipeline to lower risk of task misunderstanding.

Based on these hypothesis for the systems underperformance, alternative versions of the system are needed to be explored to investigate their effect on the performance. Overall the system ability to update files correctly and sometimes better than the OSL/ZSL makes the system as a foundation for better refined versions in the future.

Besides of the system improvement other possible future work are recognized. Based on the background studies (Huang et al., 2023; Zhong et al., 2024), having a test bench for code evaluation can have a positive impact on the outputted code. As future work evaluating a multi-agent system with a test bench might provide way to improve the results.

5.1 Threats to Validity

This paper recognizes limitations and risks associated with the methods used in this study. First, the evaluation being limited to six files due to manual evaluation, risks incomplete comparison between methods and possible mistakes with the classification of different errors. Also validating the results in other stud-

ies is more challenging without a recognized metric like HumanEval in studies (Shinn et al., 2024; Huang et al., 2023; Zhong et al., 2024).

When it comes with the used prompts in this study, it is important to take into account that different prompts could give potentially different results. As prompt engineering with multi-agent models is not a subject in this study a differently crafted prompt could give a better result. This is also a possibility with the OSL/ZSL prompts.

There are also risks with the validity of the proposed system. A possible error in the system could give false results, for example, if one of the agents has poor instructions to operate. Lastly, it is important to note that the results of the studied files can not be generalized in other use scenarios and more extensive survey is needed to evaluate code updating abilities across different frameworks, coding languages and use cases.

6 CONCLUSION

We investicated the capabilities of LLM for updating existing code using an GPT model in a deprecated web application. The results shows that LLM's are capable of updating small files with high precision using short ZSL and OSL prompts. This study proposed an multi-agent system called multi-agent pipeline to improve code to update results of the LLM code output.

The evaluation of the system showed that while the system is capable of mostly update files like the alternative ZSL/OSL prompts, it did not offer increase for performance and, in some cases, under performed compared to the alternatives. The proposed system however offers a foundation for future multi-agent systems designed for code updating. For the future improvements multi-agent system needs to address challenges related in updating of existing code.

REFERENCES

Ali, M., Hussain, S., Ashraf, M., and Paracha, K. (2020). Addressing software related issues on legacy systems -a review. *International Journal of Scientific & Technology Research*, 9:3738–3742.

Antal, G., Havas, D., Siket, I., Beszédes, Á., Ferenc, R., and Mihalicza, J. (2016). Transforming c++ 11 code to c++ 03 to support legacy compilation environments. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 177–186. IEEE.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., andGretchen Krueger, A. H., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. (2020). Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*.

CakePHP (2022). Introduction to CakePHP. https://book.cakephp.org/1.1/en/introduction-to-cakephp.html. Accessed: Sep.19, 2024.

CakePHP (2023). CakePHP 4.5.0 Released. https://bakery.cakephp.org/2023/10/14/cakephp_450.html. Accessed: Oct.14, 2024.

CakePHP (2024a). 3.0 migration guide. https://book.cakephp.org/3/en/appendices/3-0-migration-guide.html. Accessed: Nov.30, 2024.

CakePHP (2024b). Installation. https://book.cakephp.org/4/en/installation.html. Accessed: Nov.3, 2024.

CakePHP (2024c). Retrieving Data & Results Sets. https://book.cakephp.org/3/en/orm/retrieving-data-and-resultsets.html. Accessed: Sep.27, 2024.

Chong, C. J., Yao, Z., and Neamtiu, I. (2024). Artificial-intelligence generated code considered harmful: A road map for secure and high-quality code generation. *arXiv preprint arXiv:2409.19182*.

Davis, B., editor (2009). *97 things every project manager should know: collective wisdom from the experts*. O'Reilly, 1. aufl edition.

De Marco, A., Iancu, V., and Asinofsky, I. (2018). Cobol to java and newspapers still get delivered. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 583–586. IEEE.

Demir, N., Urban, T., Wittek, K., and Pohlmann, N. (2021). Our (in)secure web: Understanding update behavior of websites and its impact on security. In Hohlfeld, O., Lutu, A., and Levin, D., editors, *Passive and Active Measurement*, volume 12671, pages 76–92. Springer International Publishing. Series Title: Lecture Notes in Computer Science.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2018). Pre-training of deep bidirectional transformers for language understanding. arxiv. *arXiv preprint arXiv:1810.04805*.

Dou, S., Jia, H., Wu, S., Zheng, H., Zhou, W., Wu, M., Chai, M., Fan, J., Huang, C., Tao, Y., et al. (2024). What's wrong with your code generated by large language models? an extensive study. *arXiv preprint arXiv:2407.06153*.

Eniser, H. F., Zhang, H., David, C., Wang, M., Christakis, M., Paulsen, B., Dodds, J., and Kroening, D. (2024). Towards translating real-world code with llms: A study of translating to rust. *arXiv preprint arXiv:2405.11514*.

Erlikh, L. (2000). Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23.

Fritzsch, J., Bogner, J., Wagner, S., and Zimmermann, A. (2019). Microservices migration in industry: Intentions, strategies, and challenges. In *2019 IEEE In-*

ternational Conference on Software Maintenance and Evolution (ICSME), pages 481–490.

Huang, D., Bu, Q., Zhang, J. M., Luck, M., and Cui, H. (2023). Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010*.

Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., and Amodei, D. (2020). Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*.

Koschuetzki, T. (2008). Extra hot: CakePHP 1.2 stable is finally released! http://debuggable.com/posts/extra-hot-cakephp-1.2-stable-is-finally-released!:4954151c-f87c-434b-abbd-4e404834cda3. Accessed: Oct.30, 2024.

Liu, N. F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F., and Liang, P. (2024a). Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173.

Liu, Y., Le-Cong, T., Widyasari, R., Tantithamthavorn, C., Li, L., Le, X.-B. D., and Lo, D. (2024b). Refining chatgpt-generated code: Characterizing and mitigating code quality issues. *ACM Transactions on Software Engineering and Methodology*, 33(5):1–26.

Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., Alon, U., Dziri, N., Prabhumoye, S., Yang, Y., et al. (2024). Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36.

Norri, J., Junkkari, M., and Poranen, T. (2020). Digitization of data for a historical medical dictionary. *Language Resources and Evaluation*, 54(3):615–643.

OpenAI (2024a). GPT-4o mini: advancing cost-efficient intelligence. https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/. Accessed: Sep.3, 2024.

OpenAI (2024b). Introducing OpenAI o1-preview. https://openai.com/index/introducing-openai-o1-preview/. Accessed: Sep.19, 2024.

Ouédraogo, W. C., Kaboré, K., Tian, H., Song, Y., Koyuncu, A., Klein, J., Lo, D., and Bissyandé, T. F. (2024). Large-scale, independent and comprehensive study of the power of llms for test case generation. *arXiv preprint arXiv:2407.00225*.

Pan, R., Ibrahimzada, A. R., Krishna, R., Sankar, D., Wassi, L. P., Merler, M., Sobolev, B., Pavuluri, R., Sinha, S., and Jabbarvand, R. (2024). Lost in translation: A study of bugs introduced by large language models while translating code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13.

Radford, A. and Narasimhan, K. (2018). Improving language understanding by generative pre-training.

Rasheed, Z., Sami, M. A., Kemell, K.-K., Waseem, M., Saari, M., Systä, K., and Abrahamsson, P. (2024a). Codepori: Large-scale system for autonomous software development using multi-agent technology. *arXiv preprint arXiv:2402.01411*.

Rasheed, Z., Sami, M. A., Rasku, J., Kemell, K.-K., Zhang, Z., Harjamaki, J., Siddeeq, S., Lahti, S., Herda, T., Nurminen, M., et al. (2024b). Timeless: A vision for the next generation of software development. *arXiv preprint arXiv:2411.08507*.

Rasheed, Z., Sami, M. A., Waseem, M., Kemell, K.-K., Wang, X., Nguyen, A., Systä, K., and Abrahamsson, P. (2024c). Ai-powered code review with llms: Early results. *arXiv preprint arXiv:2404.18496*.

Rasheed, Z., Waseem, M., Kemell, K.-K., Xiaofeng, W., Duc, A. N., Systä, K., and Abrahamsson, P. (2023). Autonomous agents in software development: A vision paper. *arXiv preprint arXiv:2311.18440*.

Rasheed, Z., Waseem, M., Systä, K., and Abrahamsson, P. (2024d). Large language model evaluation via multi AI agents: Preliminary results. In *ICLR 2024 Workshop on Large Language Model (LLM) Agents*.

Sami, M. A., Waseem, M., Zhang, Z., Rasheed, Z., Systä, K., and Abrahamsson, P. (2024). Early results of an ai multiagent system for requirements elicitation and analysis. In *International Conference on Product-Focused Software Process Improvement*, pages 307–316. Springer.

Sami, M. A., Waseem, M., Zhang, Z., Rasheed, Z., Systä, K., and Abrahamsson, P. (2025). Early results of an ai multiagent system for requirements elicitation and analysis. In Pfahl, D., Gonzalez Huerta, J., Klünder, J., and Anwar, H., editors, *Product-Focused Software Process Improvement*, pages 307–316, Cham. Springer Nature Switzerland.

Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., and Yao, S. (2024). Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36.

Smyth, S. (2023). Penetration testing and legacy systems. *arXiv preprint arXiv:2402.10217*.

Sommerville, I. (2016). *Software engineering*. Always learning. Pearson, tenth edition edition.

Tampere University and Rasheed, Z. (2025). Autonomous legacy web application upgrades using a multi-agent system. https://doi.org/10.5281/zenodo.14858713.

Vaswani, A. (2017). Attention is all you need. *Advances in Neural Information Processing Systems*.

Vesić, S. and Laković, D. (2023). A framework for evaluating legacy systems – a case study. *Kultura polisa*, 20(1):32–50.

Zamfirescu-Pereira, J., Wong, R. Y., Hartmann, B., and Yang, Q. (2023). Why johnny can't prompt: How non-AI experts try (and fail) to design LLM prompts. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, pages 1–21. ACM.

Zhong, L., Wang, Z., and Shang, J. (2024). Ldb: A large language model debugger via verifying runtime execution step-by-step. *arXiv preprint arXiv:2402.16906*.