Dynamic Mitigation of RESTful Service Failures Using LLMs

Sébastien Salva and Jarod Sue
LIMOS - UMR CNRS 6158, Clermont Auvergne University, UCA, France

Keywords: RESTful APIs, Security, Software Healing, LLM.

Abstract: This paper presents a novel self-healing approach for RESTful services, leveraging the capabilities of large language models (LLMs) to generate source code that implement fine-grained mitigations. The proposed solution introduces 18 healing operators tailored for RESTful services, accommodating both grey-box and black-box perspectives. These operators implement a dual-mitigation strategy. The first mitigation employs encapsulation techniques, enabling dynamic service adaptation by generating supplementary source code without modifying the original implementation. If the primary mitigation fails, a fallback mitigation is applied to maintain service continuity. We investigate the potential of LLMs to perform the first mitigation of these healing operators by means of chains of prompts we specifically designed for these tasks. Furthermore, we introduce a novel metric that integrates test-passing correctness and LLM confidence, providing a rigorous evaluation framework for the effectiveness of the mitigations performed by LLMs. Preliminary experiments using four healing operators on 15 RESTful services with various and multiple vulnerabilities demonstrate the approach feasibility and adaptability across both grey-box and black-box perspectives.

1 INTRODUCTION

The rise of service-oriented architectures and the widespread adoption of RESTful services have revolutionised modern software system design, offering scalability, modularity, and ease of integration. However, ensuring the reliability and security of these services remains a persistent challenge, as the complexity of service compositions increases. This challenge is exacerbated by the growing reliance on large language models (LLMs) for generating and maintaining service code, which introduces new risks of vulnerabilities and unpredictable behaviours. In such environments, the need for robust techniques to address failures dynamically is more critical than ever. To address this need, (self) dynamic repairing or healing approaches may be considered: repairing localises and patches bugs in the source code, while healing applies mitigations to maintain software availability whenever possible. Repairing approaches are promising, but at the moment, they are not always accurate, as they may generate code that contains bugs (Jesse et al., 2023) or code with vulnerabilities (Schuster et al., 2021).

Existing software healing approaches (Tosi et al., 2007; Dinkel et al., 2007; Subramanian et al., 2008; Vizcarrondo et al., 2012; Wang, 2019; Rajagopalan and Jamjoom, 2015a; Magableh and Almiani, 2020) offer valuable strategies to mitigate the impact of bugs or vulnerabilities by maintaining the availability and functionality of services during failures. These techniques, however, often rely on relatively basic solutions, such as service restarts, rollbacks (Subramanian et al., 2008; Wang, 2019; Rajagopalan and Jamjoom, 2015b), or manual interventions (Tosi et al., 2007; Dinkel et al., 2007; Subramanian et al., 2008; Vizcarrondo et al., 2012; Rajagopalan and Jamjoom, 2015a), which can be labour-intensive and are not always scalable in dynamic and highly interconnected systems. Given the increasing complexity of RESTful service architectures, there is a pressing need for innovative, fine-grained healing methods that can dynamically adapt to diverse failure scenarios while minimising service disruptions.

This paper proposes a healing approach to quickly enhance the reliability and security of RESTful services deployed in production environments. Compared to previous healing approaches, we propose algorithms and healing operators aimed to perform fine-grained mitigations. To this end, our healing operators rely on two kinds of mitigations. The first aims to heal a faulty service precisely using encapsulation techniques and source code generation. The first mitigation generates source code, but unlike repair ap-

Salva, S. and Sue, J.
Dynamic Mitigation of RESTful Service Failures Using LLMs.
DOI: 10.5220/0013460700003964
In *Proceedings of the 20th International Conference on Software Technologies (ICSOFT 2025)*, pages 27-38
ISBN: 978-989-758-757-3; ISSN: 2184-2833

proaches that change original source code to fix an error, this mitigation produces additional source code that implements an encapsulation technique to dynamically adapt the service behaviour with precision either considering a grey or black-box perspective. If the first mitigation fails, a simpler fallback mitigation, as one of those listed previously, is applied. The choice between black-box and grey-box perspectives is primarily driven by the availability of source code. With the black-box perspective, the service is treated as an opaque entity, enabling mitigation without requiring access to internal structures, making it suitable for proprietary or legacy systems. In contrast, a grey-box perspective leverages partial knowledge of the system, allowing for more precise mitigations that may also be more resource-efficient.

Depending on the nature of the observed errors and the structure of the original service source code, implementing mitigations using encapsulation techniques can be complex and time-consuming. In this paper, we also explore the potential of leveraging generative AI, specifically LLMs, to simplify and generate source codes that implement mitigations. Recent studies indeed show that LLMs are effective at generating source code for tasks of easy to medium difficulty (Austin et al., 2021; Helander et al., 2024). The first mitigation of a healing operator, is hence performed by an LLM called with chains of prompts, which break down a healing process into successive tasks. These prompt chains were developed following a strict protocol to ensure that healing operators are both reproducible and effective. We also provide a metric to evaluate the success of a mitigation, which is based upon two main concepts: test passing correctness and LLM confidence. The former assesses how test suites capture defects, the latter quantifies the model internal reliability in the correctness of its output. It helps detect hallucinations. Furthermore, this paper presents extensive experiments on 4 RESTful service compositions (15 services) to demonstrate the practical benefits and limitations of this approach. The results show that our approach achieves a 85% success rate in healing services with LLM based mitigations. In summary, the major contributions of this paper are:

- the definition of 18 healing operators specialised for RESTful services and composed of two mitigations. The first one denoted m_1 uses an encapsulation technique considering either a black-box or a grey-box perspective. The second mitigation m_2 based upon older techniques is considered as a fallback solution;
- the design of two algorithms that materialise the testing and healing processes of RESTful services;
- the study of the capabilities of LLMs, regarded as general generators of source code, to perform m_1. We provide generic chains of prompts to perform the mitigations, i.e. prompts composed of variables;
- the definition of a metric to evaluate the successful application of m_1. This metric is based upon test passing correctness and LLM confidence;
- the evaluation of 4 healing operators on 4 RESTful service compositions having 4 different vulnerabilities, using two LLMs, by considering both grey and black-box perspectives. We evaluate the effectiveness of our operators to heal service methods having one vulnerability, and service methods having several vulnerabilities at the same time. We also investigate how LLM confidence aligns with test passing correctness.

The paper is organised as follows: Section 2 reviews the related work. In Section 3, we present the context of this study and introduce key definitions. Section 4 details the algorithms for vulnerability detection and service healing. The design of mitigations using LLMs, along with the evaluation metric, are covered in Section 5. Section 6 presents our evaluation results. Finally, Section 7 summarises our contributions and outlines some perspectives for future work.

2 RELATED WORK

Several surveys were proposed in the literature to classify the features, evaluate the performances and list the limitations of self healing systems. Frei et al. proposed a terminology and taxonomy for self-healing and self-repair, considering all areas of engineering (Frei et al., 2013). Schneider et al. focused on healing systems and proposed a comprehensive overview materialised as a tabular showing some properties of healing systems such as the detection criteria, or the recovery techniques (Schneider et al., 2015). Ghahremani et al. evaluated the performances of self-healing systems and provided another classification of different input types for such systems and analysed the limitations of each input type (Ghahremani and Giese, 2020).

A significant portion of healing systems, such as those in (Tosi et al., 2007; Dinkel et al., 2007; Subramanian et al., 2008; Vizcarrondo et al., 2012), focuses on web services and service compositions, which are ubiquitous in the industry. Given their prevalence and the critical need for resilience, it is unsurprising that

numerous software healing solutions specifically targeting web services have been proposed in the literature. In the previous surveys, the techniques listed to heal services are limited in number and mostly correspond to: reconfigure the available resources (isolation, reroute messages to another instance, service restart, data restoration), replace it with an older version, add Quality of Service (QOS) metadata, limit access to some URLs with rule policies, reorganise the service composition if the service descriptions are provided or perform manual intervention. The surveys concluded that the characteristics of the recovery techniques are not sophisticated, and thus the validity of the outcome is often not clear. This is mostly due to the limitations of the healing approaches, which do not change the system source code. Some more recent papers about service or micro-service healing presented more sophisticated techniques, such as machine learning to detect anomalies. But the chosen recovery techniques yet correspond to managing the versions of services without human intervention (Wang, 2019; Rajagopalan and Jamjoom, 2015a; Magableh and Almiani, 2020).

Compared to the previous approaches, we study the possibility of using LLMs for improving the healing process. Overall, AI has been used in various software engineering activities for decades. Allamanis et al. proposed a survey of machine learning methods applied to source code in (Allamanis et al., 2018). Many learning methods have been proposed for program repair, e.g.,(Fan et al., 2023; Jiang et al., 2023; Yasunaga and Liang, 2020; Chen et al., 2021; Maniatis and Tarlow, 2023) These approaches offer promising results but they still may generate code that contains bugs (Jesse et al., 2023) or code with known and risky vulnerabilities (Schuster et al., 2021). Recently, LLMs have been more and more used for programming. For example, Kanade et al. proposed CuBERT, a specialised transformer model for code understanding (Kanade et al., 2020). Other papers showed that LLMs such as ChatGPT can solve easy and medium programming problems (Austin et al., 2021; Helander et al., 2024). They can also correctly audit applications to detect vulnerabilities and explain the results (Ma et al., 2024).

This paper studies and leverages the ability of LLMs called with specialised chains of prompts to generate source code for healing services with fine granularity i.e. at the method level instead of the service level as the previous healing approaches. But if the result provided by the LLM is unsuccessful to mitigate a vulnerability, our algorithms apply one of the classical recovery techniques cited previously as a failing mitigation.

3 CONTEXT AND ASSUMPTIONS

Let S be a countable set of RESTful services, and T be a countable set of test cases. A service $s \in S$ is deployed in an environment that supports testing from two perspectives: black-box and grey-box testing. With the former, the service s is accessible exclusively through HTTP requests and responses, referred to as communication events. To determine the success or failure of a test, these events can be read (assuming no encryption). The events also include parameter assignments, which allow for the identification of their source and destination. In the case of grey-box testing, the service s can still be experimented using events, but the project's source files are also accessible.

To simplify fault localisation, which is not the primary focus of this paper, we assume that a service $s \in S$ can be tested in isolation. This assumption offers several practical benefits. First, fewer resources are required, as each test case is executed on a single service instance at a time. Testing s in isolation removes dependencies, preventing test execution from being blocked by faults in dependent services and speeding up the process. Additionally, testing in isolation reduces verdict inconsistencies caused by environmental factors, such as network traffic or container management. Finally, isolation testing explicitly simplifies fault localisation.

We consider having a test suite denoted $T(s) \subset T$ for each service $s \in S$. $T(s)$ contains test cases encoding functional scenarios, which will be used here for regression testing. A service s passes a test case $t \in T(s)$, denoted s passes t if and only if all possible test executions result in a pass verdict. This implies that the service may need to be tested multiple times to cover all possible scenarios encoded in a single test case and to explore any nondeterministic behaviours of the service. The notation s fails t indicates that s does not pass the test case.

We also assume the existence of additional test suites to assess non-functional aspects of the services. In this work, we focus on security and robustness, although other aspects, such as performance, could be explored in future research. We denote V as the set of countable vulnerabilities, and $T_v(s) \subset T$ as the test suite used to detect whether the service s has the vulnerability $v \in V$. $T_v(s)$ includes test cases that simulate various attack scenarios, or experiment s with unexpected events for robustness. Given a test case $t \in T_v(s)$, if s fails t, then we say that s has the vulnerability v. At this point, we consider having the test suites $T_{v_1}(s), \ldots, T_{v_m}(s)$ to check whether s has the vulnerabilities $v_1, \ldots, v_m$.

However, testing services in isolation requires writing specific test cases that interact with mocks, which simulate the real requisite services called by s. Additionally, writing test cases and mocks that simulate attacks is often considered as a challenging task. To alleviate this difficulty, we proposed test case and mock generation algorithms, along with supporting tools in (Salva and Blot, 2020; Salva and Sue, 2023; Salva and Sue, 2024).

4 VULNERABILITY DETECTION AND SERVICE HEALING

We propose in this paper a proactive solution for detecting and subsequently healing vulnerabilities in services. This approach involves periodically evaluating a Quality of service (QoS), and when QoS is low, healing services. The QoS can be measured at each service deployment, at specific time intervals, or when server errors are detected (e.g., receipt of events with an HTTP status 500). We propose to define the QoS measurement as the ratio of passing tests. We present a first algorithm, Algorithm 1, to experiment each service, detect vulnerabilities and localise them on the service interfaces. If vulnerabilities are detected, a second healing algorithm is then invoked. Both algorithms are presented subsequently.

4.1 Vulnerability Detection and Localisation

In our context of service healing, a vulnerability localisation refers to a URL (in black-box mode) or a method (in grey-box mode) that invokes specific source code containing a vulnerability. For a service s and a test case $t \in T_v(s)$ such that s fails t, Algorithm 1 infers the localisation, denoted $l(s)$, of the vulnerability v, which are then passed to the healing algorithm.

Definition 1. *Let $\mathcal{X}$ be a variable set and $\mathcal{V}$ a value set. We write $x := *$ the assignment of the variable x with an arbitrary element of $\mathcal{V}$.*

A vulnerability localisation $l(s)$ for the service $s \in S$ is the tuple $(URL, Verb, Method, Package)$ composed of variables in $\mathcal{X}$, which may be assigned to values in $\mathcal{V}$. L stands for the set of vulnerability localisations.

The variables of a localisation refer to static information extracted from the source code or the interface of a RESTful service. *Verb* represents an HTTP verb (e.g., GET), *Method* refers to a method name, and *Package* is assigned to a string that defines a namespace organising classes. It is noteworthy that the variables of a localisation could be easily expanded to support additional vulnerability types or faults. The variables of $l(s)$ should be assigned to values just after the failure of a test case, either directly (extraction from the test case of the URL and HTTP verb) or collected from the source code of the service s if the grey-box mode is used. This extraction from the source code can be performed with automatic tools, including LLMs. We also denote $T(l(s)) \subseteq T(s)$ as the test suite that exercises the service s at the localisation $l(s)$. For RESTful services, this subset can be determined by scanning the URLs and verbs found in the test cases of $T(s)$.

Algorithm 1: Service Testing and Vulnerability Localisation.

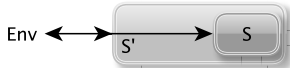
```
   input : Service s, T(s), T_{v_1}(s), ..., T_{v_k}(s),
           p ∈ {black_box, grey_box}
   output: Vl
1  Vl := ∅;
2  foreach t ∈ T(s) do
3      if s fails t then
4          Alert ;
5          Replace s with an older and trusted version;
6          Deploy s;
7  foreach t ∈ T_{v_i}(s) with 1 ≤ i ≤ k do
8      if s fails t then
9          Locate the vulnerability v_i and infer l(s);
10         {l_1(s), ..., l_n(s)} := complete(l(s), p);
11         Vl := Vl ∪ {(v_i, l_1(s)), ..., (v_i, l_k(s))};
```

Algorithm 1 takes a service $s \in S$, some test suites $T(s)$, $T_{v_1}(s), \ldots, T_{v_k}(s)$ and a testing perspective p. It returns a set Vl composed of pairs of the form (vulnerability, localisation). The service s is firstly experimented with $T(s)$ to detect potential regressions. If the service does not pass a test case of $T(s)$, an alert is raised, furthermore the service is directly healed by replacing it with an older and trusted version. Other actions might be possible, e.g., the service composition reorganisation (Subramanian et al., 2008), or a manual intervention. If the service does not pass a test case $t \in T_{v_i}(s)$, a localisation $l(s)$ is inferred either from t or from the source code project of the service. Subsequently, the procedure *complete* is invoked to assign values to the variables of $l(s)$. We consider that the URL variable is always assigned to a value as a failing test case always refers to a URL called to experiment a RESTful service. If the VERB variable is not assigned to a value, then 4 localisations are built for the HTTP verbs GET, POST, PUT, DELETE. With the grey-box perspective only, the procedure scans the project source files to get additional information (package, method names). A URL

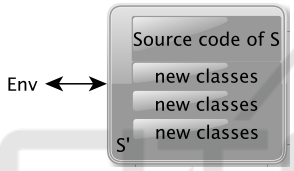
and HTTP verb can only be linked to one method, given in one package. The resulting pairs $(v_i, l_j(s))$, expressing that s has the vulnerability v_i tied to a fault location $l_j(s)$, are added to the set Vl.

4.2 Service Recovery

A service $s \in S$, having a vulnerability v, is healed with an operator denoted H_v. This healing operator is composed of two mitigation functions m_1 and m_2 that aim at returning a new service s'. The first mitigation attempts to produce a new service s' aiming to minimise the effects of the vulnerability v or by eliminating v from s at the localisation $l(s)$. This mitigation, which uses an encapsulation technique, is performed by calling an LLM with a chain of prompts denoted p_v.



(a) Black-box perspective.



(b) Grey-box perspective.

Figure 1: Healing process by encapsulation: in black-box, a new service s' calls the faulty service s to modify its behaviours; in grey-box, new classes are included in the original source code of s. With both perspectives, the original source code remains unmodified.

Figure 1 illustrates the intuition behind the encapsulation techniques. With the black-box perspective, m_1 builds a new service s' that calls/encapsulates the original service s, filtering both incoming and outgoing messages. With the grey-box perspective, m_1 enhances the service functionality without modifying the original source code by introducing new classes. These classes implement various encapsulation techniques, such as introspection, HTTP filters, or aspect-oriented programming, to heal the service s. This approach allows engineers or automated methods to repair the service later by modifying the original source code. If the mitigation function m_1 fails to heal the service s, then m_2 is invoked. This function implements a fallback mitigation, which is simpler to apply.

Definition 2 (Healing Operator H_v)**.** *A healing operator H_v for the vulnerability v is the tuple (m_1, m_2) such that :*

- *$m_1 : S \times L \times p \rightarrow S$ is a mitigation function that takes a service s, a localisation $l(s)$, a perspective p (black or grey-box) and produces a service $s' = m_1(s, l(s), p)$;*
- *m_2: $S \rightarrow S$ is an mitigation function that produces a service $s' = m2(s)$.*

We designed healing operators to heal 18 vulnerabilities. The operator list is presented in (Sue and Salva, 2025) with a tabular form that gives: the general weakness, the vulnerability targeted by a given CAPEC[1] attack, the generic mitigations m_1 and m_2 for both testing perspectives. m_1 is materialised by prompt chains while m_2 refers to a classical mitigation action.

As we wish the mitigation succeeds in healing RESTful services, we applied the solutions proposed and evaluated in the papers (Tosi et al., 2007; Dinkel et al., 2007; Subramanian et al., 2008; Vizcarrondo et al., 2012). As stated in Section 2, for most of the healing operators, the mitigation m_2 comes down to "reverting back to a previous and trusted version". For some operators, m_2 corresponds to "sanitising the data", which applies an input validator on the input data to filter out potential attacks.

To ensure that the mitigation m_2 can effectively heal the functionality of RESTful services, we applied the solutions proposed and evaluated in the prior papers (Tosi et al., 2007; Dinkel et al., 2007; Subramanian et al., 2008; Vizcarrondo et al., 2012). As discussed in Section 2, for most healing operators, the mitigation m_2 mostly comes down to 'reverting to a previous and trusted version'. For certain operators, m_2 corresponds to 'sanitizing the data,' which entails applying an input validator to filter out potentially malicious inputs. sFigure 3 illustrates an example for the operator designed to heal a service vulnerable to the attack CAPEC 274 (HTTP verb tampering). For readability reasons, we only show the two generic prompt chains used for the mitigation m_1. The prompt chain construction is detailed in the next section.

Algorithm 2 implements a simple iterative healing process. It takes the service $s \in S$, the two test suites $T(s) \in T$ and $T_v(s) \in T$ along with the subset $\{(v, l_1(s)), \ldots, (v, l_n(s))\}$ returned by Algorithm 1. Algorithm 2 covers each localisation and tries to heal the service with the mitigation m_1 of the healing operator H_v. If the mitigation is applied with success, it returns a new service s', which replaces s. m_1 is made up of a generic chain of prompts specifically written to mitigate v. m_1 calls a prompt generator to instantiate the generic prompts, i.e to assign every of its variables to a concrete value. This instantiation is performed by two ways. Either a variable also belongs to a localisation $l(s)$ given as input to m_1. In this

[1] https://capec.mitre.org/

Algorithm 2: Service Healing.

input : Service s, $T(s)$, $T_v(s)$, $\{(v,l_1(s)),\ldots,(v,l_n(s))\} \subseteq Vl$, $perspective = greybox/blacbox$
output: New Service s

1 $i := 1$;
2 **while** $i <= n$ **do**
3 $s' := m_1(s,l_i(s),perspective)$ with $H_v = (m_1,m_2)$;
// m_1 calls a prompt generator and build a prompt chain p_v
// m_1 calls an LLM with p_v to get source code
// m_1 aggregate files, if several files with the same name are available
4 **if** $success(m_1(s,l_i(s),perspective),T(l(s)),T_v(l(s)))$ **then**
5 Replace s by s';
6 **else**
7 $s'' := m_2(s)$;
8 Replace s by s'';
9 Deploy s;

case, the assignment is directly performed. Or, in the grey-box perspective, further information is collected from the service source code project. Then an LLM is called with the prompt chain p_v to generate source code and to build a new service. The success of the mitigation is evaluated with the function *success* that takes the new service, and test suites (line 4). If the application of m_1 at a localisation is unsuccessful, the mitigation function m_2 is then called (line 7). At the end of this algorithm, the service s is replaced by another service s', which is finally deployed.

The design of prompt chains, the execution of the mitigation function m_1 and the concept of success are detailed in the next section.

5 MITIGATION FUNCTIONS WITH LLM

A mitigation function m_1 generates source code by means of an LLM queried with a chain of prompts. This section describes how prompt chains have been designed, the application of the mitigation function and the evaluation of its success.

5.1 Writing of the Mitigation Function M_1

We have written prompt chains for the mitigations m_1 of the healing operators by means of a strict protocol based upon prompt engineering techniques to optimise the generation of correct healing corrections in a reproducible manner. We followed these supervised and incremental steps to write prompt chains

1) Act as an expert in software security
Service information
Task:
constraints 1
constraints 2
Format: Don't include any explanations in your responses Give me the code wrapped in <code> tags. Give me 3 different responses:
2) Give me an agreement score on the 3 responses expressing that the code is "error free" and "meets the demand" between 0 and 100 with 0 the code is incorrect and 100 the code is "error free" and meets the demand
3) Give me all the modification required to make my project working for the response X

Figure 2: Generic form of chain of prompts.

specialised to mitigate security or robustness vulnerabilities:

1. construction of a first prompt defining a context, a list of actions required to heal a service for a given vulnerability v and asking 5 different solutions (no generation of code here);
2. writing of a prompt chain for calling a LLM to generate source code : we manually evaluated the previous solutions to order them from correct to incorrect using the CAPEC base; from the three correct and best solutions, we wrote a first prompt chain;
3. cycle of refinement on the prompt chain: we applied the prompt chain on case studies and incrementally optimised it by manually evaluated the correctness of the source codes;
4. cycle of refinement to increase reproducibility: we applied multiple times the prompt chain on the same case study and assessed reproducibility by means of a source code similarity measure applied on the generated source codes. The prompt chain is kept when the similarity score exceeds 70%;
5. writing of the generic prompt chain p_v made up of variables to replace the specific information related to the case studies.

Figure 2 summarises the three initial generic forms of prompts we use to write prompt chains at step 2. Those are completed in accordance with a vulnerability v and may be split into several prompts while the refinement process. The reason of asking three different responses is related to LLM confidence assessment, which is discussed below.

A generic prompt, which contains variables, is instantiated by assigning every of its variables to a value, by means of a prompt generator. This instantiation can occur in two ways. If a variable also belongs to a localisation $l(s)$ provided as input to m_1, the value

Weakness	Successful Attack	First_Black_Box_Mitigation	First_Grey_Box_Mitigation
CWE-302: Authentication Bypass by Assumed-Immutable Data	CAPEC-274 HTTP Verb Tampering	{Act as an expert in cybersecurity, I have a "TYPE" web service with the route "ROUTE" with the verb "VERB". The service is vulnerable to verb tampering. Task: Create a new "TYPE" web service that : is called "with the same route" to heal the vulnerability using "TECH" and encapsulates the vulnerable service without touching its source code. Don't include any explanations in your responses Give me the code wrapped in <code> tags: Give me 3 different responses.} {Give me an agreement score on the 3 responses expressing that the code is error free and meets the demand between 0 and 100 with 0 the code is incorrect and 100 the code is error free and meets the demand} {Complete the new service having the best score so that it can redirect any other request to the first vulnerable service}	{Act as an expert in cybersecurity, I have a "TYPE" web service with the route "ROUTE" with the verb "VERB". Its method is METHOD The package is named PACKAGE the class is named CLASS. The service is vulnerable to verb tampering. Task: Update the service source code by using a "TECH" and VERSION to heal the service without modifying the original class". Don't include any explanations in your responses Give me the code wrapped in <code> tags: Give me 3 different responses.} {Give me an agreement score on the 3 responses expressing that the code is error free and meets the demand between 0 and 100 with 0 the code is incorrect and 100 the code is error free and meets the demand}{provide me with all the files needed to implement the solution you gave the best score}

Figure 3: Prompt examples to heal the vulnerability targeted by CAPEC-274.

is directly retrieved from $l(s)$. Alternatively, with the grey-box perspective, additional information is extracted from the service's source code project, such as the project name, class names, and other relevant metadata. In the example shown in Figure 3, variables are represented by words in uppercase. The variables VERB and ROUTE are assigned to values retrieved from $l(s)$. The variables TYPE and VERSION, which indicate the framework type, are either hardcoded or extracted from the source code project. Similarly, the variable TECH, which denotes the encapsulation approach (e.g., filter, aspect programming, etc.), is also either hardcoded or determined based on project-specific information.

5.2 Reliability Evaluation

Algorithm 2 uses the boolean expression *success* : $S \times T \times T_v \times P \rightarrow \{true, false\}$ to evaluate whether the mitigation function m_1 has healed a service s correctly. This evaluation is performed using both test passing correctness and LLM confidence to compute a reliability score, denoted $(corr_1, corr_2, conf)$.

Test passing correctness, which exclusively relies on test suites, is measured by two indicators $corr_1$ and $corr_2$, the ratio of passing tests in $T(l(s))$ and the ratio of passing tests in $T_v(l(s))$. Both are obtained after the experimentation of the new service s' generated by the mitigation function m_1. $corr_1$ helps identify potential regressions in the new service s', and $corr_2$ allows to check whether the vulnerability v has been mitigated at the localisation $l(s)$.

The LLM confidence indicator $conf$ measures how much trust we should put into the source code the LLM has generated. A higher confidence score should mean a higher likelihood of being correct (Austin et al., 2021). This indicator can be obtained by using intrinsic measures based on conditional probability, i.e. the probability derived from the response generated by the LLM, or by using self-reflective measures, i.e. metrics that evaluate the LLM's own confidence within its responses. Recent works showed that the second approach often returns better-calibrated confidence scores (Tian et al., 2023).

The LLM confidence evaluation (and the reduction of overconfidence) can be improved by using different kinds of techniques (Xiong et al., 2024), e.g., ask for several responses, ask to different LLMs, ask for several explanations and evaluate them. This is why we request the LLM for three source code generations in our prompt chains. Hence, every time we apply a mitigation function m_1, we obtain 3 responses and 3 tuples $(corr_1, corr_2, conf)$.

Next, we apply Reciprocal Rank Fusion (RRF), which is a rank aggregation approach that combines rankings from multiple sources into a single ranking. The results of our experimentations suggest to apply RFF only on the test passing scores $corr_1$ and $corr_2$. If the best tuple $(corr_1, corr_2, conf)$ has scores higher than 90%, then *success* returns true else false. This 90% threshold will be analysed during the evaluation.

6 EXPERIMENTAL RESULTS

We investigated the capabilities of our algorithms through the following questions:

- RQ1: What is the effectiveness of the mitigations m_1 with service methods having one vulnerability?
- RQ2: How does effectiveness evolve when service methods have several vulnerabilities?
- RQ3: How well LLM confidence aligns with test passing correctness?

This study was conducted with 4 healing operators denoted H_{verb}, H_{xss}, H_{path}, H_{token}, which aims at healing services vulnerable to Verb Tampering (CAPEC-274), XSS injection (CAPEC-86), Path Traversal (CAPEC-126) and Token Impersonation (CAPEC-633). For readability, the vulnerabilities are denoted v_{verb}, v_{xss}, v_{path}, v_{token}. We considered the following RESTful service compositions :

- C1: Piggy metric[2] is a financial advisor application composed of 3 micro-services specialised in account management, statistics generation and notification management;
- C2: eShopOnContainers[3], implementing an e-commerce web site using a services-based architecture (5 services);
- C3: a loan approval process implemented with 4 services developed by third year computer science undergraduate students;
- C4: a composition of 3 services used to implement an online shop (stock and client management, purchase, etc.) developed by students.

We wrote between 7 and 60 conformance test cases per RESTful service. We checked that the services pass the test cases. As all the services have deterministic behaviours, we did need to experiment them multiple times. For each RESTful service, we developed between 7 and 60 conformance test cases. We then checked that the services successfully passed them. Since all services exhibit deterministic behaviour, repeated test execution was not required. We then generated security tests with our approach presented in (Salva and Sue, 2024) that builds test cases to try detect the fours vulnerabilities listed previously. We obtained between 6 and 75 security test cases per service. We then intentionally injected vulnerabilities on the 15 RESTful services, as non-vulnerable services cannot be healed and are therefore not relevant for this evaluation. We generated mutants from the RESTful service source codes by applying the tool PITest [4] completed by our own source code mutations injecting modifications. We kept the 200 mutants that have at least one failing security test case.

To perform this evaluation, we implemented a prototype tool that applies the mitigation m_1 of the 4 healing operators on RESTful services by calling LLMs. We considered two open-source LLMs, Mistral's Codestral-22B, which is an open-weight small language model explicitly designed for code generation tasks, and Meta's Llama3.1-70B, a larger model capable of generating code, and natural language about code, from both code and natural language prompts. This prototype tool heals services with our base of prompt chains, it compiles, redeploys the healed services, and experiments them with test suites. It also returns a reliability evaluation score as presented in Section 5.2. All the services, tools, prompt chains are available in (Sue and Salva, 2025).

[2]https://github.com/sqshq/piggymetrics

[3]https://github.com/dotnet/eShop

[4]https://github.com/hcoles/pitest

Table 1: Precision@1 detailed for both perspectives and both LLMs.

Perspective	LLM	Precision@1
Black-box	Codestral-22B	0.87
	Llama3.1-70B	0.87
Grey-box	Codestral-22B	0.78
	Llama3.1-70B	0.87

6.1 RQ1: What Is the Effectiveness of the Mitigation M_1 with Service Methods Having One Vulnerability?

Setup: to investigate RQ1, we applied the mitigation m_1 of the healing operators on the mutants and we computed the reliability evaluation scores $(corr1, corr2, conf)$ for all healed services, with both black-box and grey-box perspectives. In this analysis, we considered the three source codes generated by the LLMs for each vulnerability localisation, not just the best one. From the tuples $(corr1, corr2, conf)$, we also calculated Precision@1, which represents the precision of the best solution returned by the LLMs. Precision@1 reflects the proportion of corrections that were truly effective in healing the mutants. To determine Precision@1, we manually reviewed the source codes of the best solutions provided by the LLMs to verify whether the vulnerability was truly fixed.

Results: Figure 4 depicts violin diagrams showing for every vulnerability: the distributions, medians, quartiles along with densities of the scores $corr1$, $corr2$ and $conf$ obtained for all the healed services. Additionally, Table 1 gives Precision@1 for each perspective and for each LLM.

The violin diagrams show that the ratios of passing tests $corr1$ and $corr2$ are most of the time at 100% (distribution : 92% of ratios of passing tests above 90% for $corr1$, 82% of ratios of passing tests above 90% for $corr2$) for all the vulnerabilities and all the obtained healed services (not only the best solution). This shows that our approach is effective in repairing these vulnerabilities with our prompt chains and LLMs. The diagrams also indicate, through $corr1$, that there are some differences among healing operators and vulnerabilities. For H_{path} and H_{verb}, the ra-

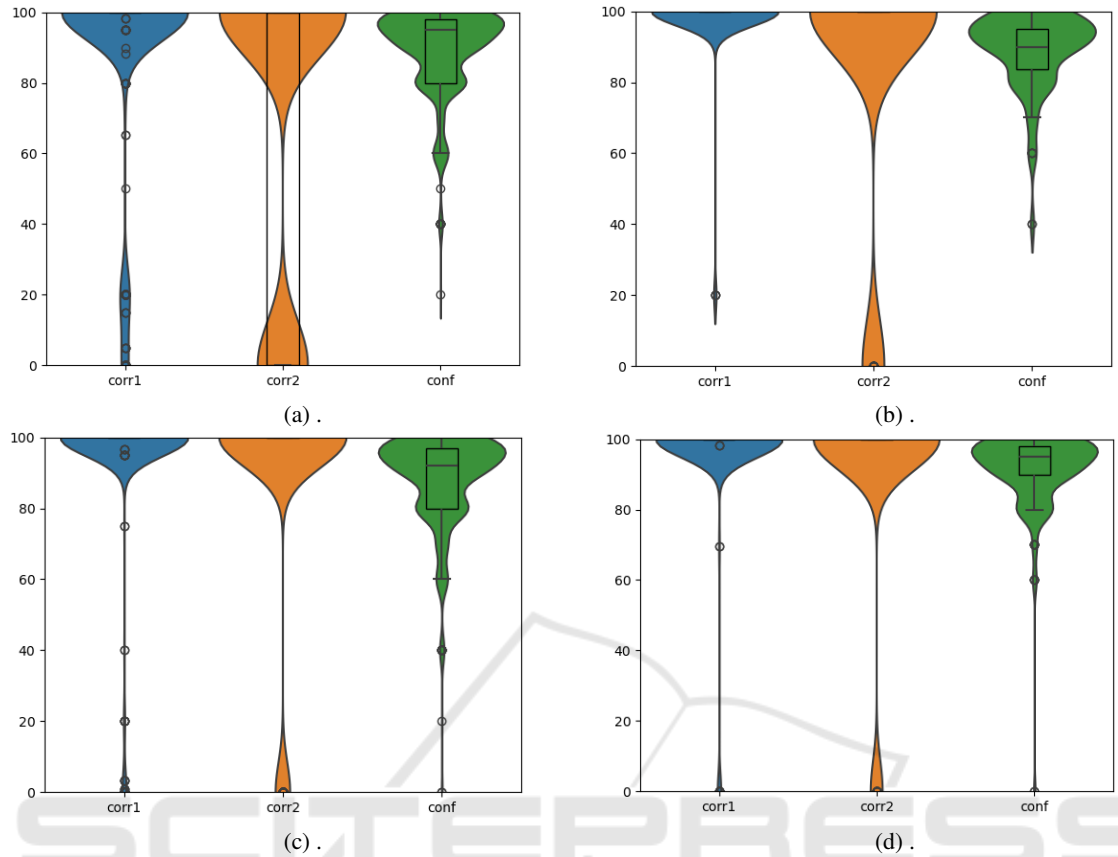


Figure 4: Violin diagrams showing the distributions, medians, quartiles along with densities of the scores $corr1$, $corr2$ and $conf$ obtained after having applied the healing operators H_{verb} (4a), H_{xss} (4b), H_{path} (4c), H_{token} (4d on all the mutants).

tio of passing tests is sometimes below 100% and can drop as low as 20%. These results suggest that the healing process in these cases compromises the initial functional behaviours of the RESTful services. This highlights the importance of computing $corr1$ to assess the impact of healing on service functionalities. In contrast, the ratios of passing security tests, represented by $corr2$, are consistently either 100% or 0%, which shows whether the targeted vulnerability has been repaired or not.

The effectiveness of the approach is manually verified using Precision@1, which achieves an overall value of 85%. This indicates that 85% of the top-ranked healed services are really healed. Table 1 demonstrates that there is no significant difference between the two perspectives when using Llama3.1-70B, whereas Codestral shows reduced effectiveness under the grey-box perspective. When comparing the two LLMs, their effectiveness is comparable within the black-box perspective and lower for Codestral otherwise. This finding is noteworthy as it highlights that a locally executable LLM can perform competitively compared to a larger LLM. The observations performed on the violin diagrams and with Precision@1 tend to suggest that the 90% threshold considered in the boolean expression *success*, to state whether a healed service is truly healed, could be slightly augmented to increase Precision@1. Further experimentation may be needed to establish a general threshold.

Figure 4 also highlights the high confidence levels, with $conf$, provided by the LLMs on their generated source codes, with values predominantly ranging between 80% and 100%. Confidence is slightly higher for H_{xss} and H_{token}. The violin diagrams show that LLM confidence does not contradict test passing correctness. However, whether LLM confidence alone can be reliably trusted by users will be addressed in RQ3.

6.2 RQ2: How Does Effectiveness Evolve when Service Methods Have Several Vulnerabilities?

Setup: RQ2 studies whether the healing operators and their mitigations m_1 can be effectively applied several times on the same service method having

several vulnerabilities. We took back the composition C1, which includes 3 services and a total of 12 methods, and manually injected the 4 vulnerabilities considered in this evaluation into each method. We then considered the 1152 code projects generated by iteratively applying permutations of the four healing operators, by querying two LLMs (Llama3.1 and Codestral), under either a black-box or grey-box perspective. With the latter perspective, the generated source code of the same class was incrementally pasted, without manual intervention. Following the application of every mitigation of the healing operators, we calculated the percentage of healed services achieving a test-passing success rate above 90% ($corr_1 > 90\% \wedge corr_2 > 90\%$).

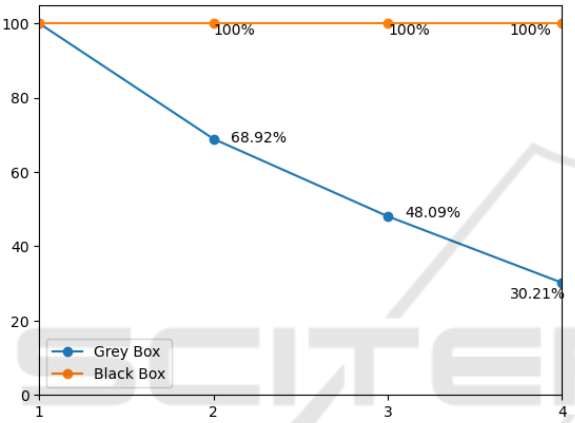


Figure 5: Ratio of projects with a test-passing success rate above 90% after the iterative application of mitigations.

Results: Figure 5 illustrates two curves corresponding to the effectiveness of the healing process for both perspectives. In the black-box approach, the healing process—defined as generating a new service that encapsulates the vulnerable one—maintains a consistent success ratio throughout. This result aligns with our expectations since this healing approach does not introduce an iterative increase in difficulty for the LLMs. Each newly generated service addresses a single vulnerability and calls a vulnerable older service. However, this result masks a significant limitation: this solution cannot be applied indefinitely due to its reliance on creating and deploying an increasing number of new services, which becomes resource-intensive. With the grey-box perspective, we observe a noticeable decline in the ratio of healed services achieving a test-passing success rate above 90% as the number of mitigations increases. This ratio drops by up to 30%. A key contributing factor is that the resulting source codes often include overlapping references to the same objects or filters. These redundancies prevent seamless integration of multiple mitigations, necessitating manual intervention. These issues could potentially be resolved in the near future with more efficient LLMs.

Table 2: Spearman's rank correlation coefficient between $corr1, corr2$ and $conf$ for each correction type.

	Codestral-22B		Lllama3.1-70B	
	Black-box	Grey-box	Black-box	Grey-box
corr1/conf	-0.006	-0.171	-0.001	0.070
corr2/conf	-0.033	-0.163	0.044	0.071

6.3 RQ3: How Well the Confidence Aligns with Test Passing Correctness?

Setup: We investigated RQ3 by taking back the tuples $(corr1, corr2, conf)$ for all the healed services obtained after applying the healing operators on the mutants with black-box and grey-box perspectives. We calculated the Spearman's rank correlation coefficients between test-passing correctness and LLM confidence, segmented by vulnerability, LLM, and testing perspective. The Spearman's rank correlation coefficient measures the strength and direction of a monotonic relationship between two variables, and returns a value between -1 and 1: the coefficient is equal to 0 when both variables are uncorrelated, to 1 when there is a perfect positive monotonic correlation (as one variable increases, the other always increases), and to -1 when there is a perfect negative monotonic correlation.

Results: Table 2 presents correlation computations between $corr1$ and $conf$, as well as $corr2$ and $conf$, with both perspectives and both LLMs. In all cases, the results indicate that the correlations are negligible, as the monotonic relationships are too weak. We observed multiple times in our results that the main reason of weak alignment is the tendency of LLMs to assign overconfident scores to incorrect or incomplete source code (which should be completed by the user).

Consequently, LLM confidence cannot be used as a reliable indicator of healing effectiveness. Testing stages hence remain essential. Nevertheless, a low confidence score can be a useful indicator for quickly identifying potential issues in code generation. A low score may indicate insufficient knowledge of the appropriate mitigation to apply, an ambiguous prompt, or the high complexity of the vulnerability being addressed. In any case, the resulting service should be rejected.

6.4 Threats to Validity

We now address potential external and internal threats to the validity of our evaluation. We identified 3 external and 2 internal threats.

Regarding external threats, the first one concerns the choice of our case studies. To limit the impact of this threat, we evaluated our tool on RESTful services covering different domains, two of these case studies were implemented by beginner developers, the other two case studies involved microservice compositions that have been extensively evaluated in previous research. While the inclusion of varied case studies strengthens our evaluation, these case studies do not represent all possible types of service compositions. As a result, we avoid drawing overly broad conclusions from our results. Another external threat relates to the test suites used during the evaluation. We generated conformance test cases to ensure that every service method was tested by at least two test cases, and we utilised the security testing approach from (Salva and Sue, 2024) to create security-specific test cases. The latter demonstrated an increase in branch coverage by 20% and consistently detected vulnerabilities in the same services used in the experiments. However, the test suites may still lack comprehensive coverage for all possible service behaviours or vulnerabilities, limiting their general applicability to other contexts. Additionally, we used mutation of service methods to inject vulnerabilities. To mitigate the risk that these mutations do not reflect realistic vulnerabilities, we employed the well-known, open-source tool Pitest, supplemented with custom mutations to ensure every method contained at least one vulnerability. We also conducted manual checks on the source code to verify the accuracy of the mutations. Despite these measures, the generated mutations may not capture the full range of vulnerabilities encountered in real-world scenarios.

Other factors that may threaten internal validity relate to the design and execution of the prompt chains used in our experiments. We followed a strict protocol when designing the prompt chains and applied them consistently across several services. However, a prompt chain may be interpreted differently by another LLM. In our experiments, we queried LLMs using chains of prompts while keeping additional model parameters, such as temperature, top_k top_p to their default values. These parameters could have been fine-tuned for each LLM to potentially improve code generation performance.

7 CONCLUSION

This paper has proposed the design and evaluation of a RESTful service healing approach, which introduces fine-grained mitigation techniques to enhance reliability and security of RESTful services. We propose 18 healing operators, each incorporating two mitigation strategies: m_1 utilises encapsulation techniques under grey-box or black-box perspectives, generating additional source code for dynamic service adaptation without altering the original source code. If m_1 fails, a fallback mitigation m_2 is applied. The study also explores the capabilities of LLMs as general-purpose source code generators for m_1, and define a novel metric to evaluate their success. An evaluation with 4 operators on RESTful services having different vulnerabilities demonstrates the effectiveness of m_1 in both grey-box and black-box contexts, addressing single and multiple vulnerabilities.

This evaluation underscores the challenge of aligning LLM confidence with test-passing success, demonstrating that LLMs alone cannot be fully trusted for automated healing of RESTful services. However, when sufficiently large test suites are available, our test-passing-based metric proves effective for assessing healing success, achieving an overall Precision@1 of 85%, though there is room for improvement. A compelling research direction would be to explore methods for evaluating healing success with minimal or no test suites. This could involve developing alternative metrics or leveraging LLMs to generate test cases, which would then require their own rigorous evaluation.

REFERENCES

Allamanis, M., Barr, E. T., Devanbu, P., and Sutton, C. (2018). A survey of machine learning for big code and naturalness. *ACM Comput. Surv.*, 51(4).

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., and Sutton, C. (2021). Program synthesis with large language models.

Chen, Z., Kommrusch, S., Tufano, M., Pouchet, L., Poshyvanyk, D., and Monperrus, M. (2021). Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering*, 47(09):1943–1959.

Dinkel, M., Stesny, S., and Baumgarten, U. (2007). Interactive self-healing for black-box components in distributed embedded environments. In *Communication in Distributed Systems - 15. ITG/GI Symposium*, pages 1–12.

Fan, Z., Gao, X., Mirchev, M., Roychoudhury, A., and Tan,

S. H. (2023). Automated repair of programs from large language models.

Frei, R., McWilliam, R., Derrick, B., Purvis, A., Tiwari, A., and Di Marzo Serugendo, G. (2013). Self-healing and self-repairing technologies. *International Journal of Advanced Manufacturing Technology*, 69(5):1033–1061.

Ghahremani, S. and Giese, H. (2020). Evaluation of self-healing systems: An analysis of the state-of-the-art and required improvements. *Computers*, 9(1).

Helander, V., Ekedahl, H., Bucaioni, A., and Nguyen, T. P. (2024). Programming with chatgpt: How far can we go? *Machine Learning with Applications*, 1:1–34.

Jesse, K., Ahmed, T., Devanbu, P. T., and Morgan, E. (2023). Large language models and simple, stupid bugs. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pages 563–575.

Jiang, N., Liu, K., Lutellier, T., and Tan, L. (2023). Impact of code language models on automated program repair.

Kanade, A., Maniatis, P., Balakrishnan, G., and Shi, K. (2020). Learning and evaluating contextual embedding of source code. In III, H. D. and Singh, A., editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 5110–5121. PMLR.

Ma, W., Wu, D., Sun, Y., Wang, T., Liu, S., Zhang, J., Xue, Y., and Liu, Y. (2024). Combining fine-tuning and llm-based agents for intuitive smart contract auditing with justifications.

Magableh, B. and Almiani, M. (2020). A self healing microservices architecture: A case study in docker swarm cluster. In Barolli, L., Takizawa, M., Xhafa, F., and Enokido, T., editors, *Advanced Information Networking and Applications*, pages 846–858, Cham. Springer International Publishing.

Maniatis, P. and Tarlow, D. (2023). Large sequence models for software development activities.

Rajagopalan, S. and Jamjoom, H. (2015a). App–Bisect: Autonomous healing for Microservice-Based apps. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, Santa Clara, CA. USENIX Association.

Rajagopalan, S. and Jamjoom, H. (2015b). App–Bisect: Autonomous healing for Microservice-Based apps. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, Santa Clara, CA. USENIX Association.

Salva, S. and Blot, E. (2020). Cktail: Model learning of communicating systems. In *Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering - Volume 1: ENASE,*, pages 27–38. INSTICC, SciTePress.

Salva, S. and Sue, J. (2023). Automated test case generation for service composition from event logs. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023 - Workshops, Luxembourg, September 11-15, 2023*, pages 127–134. IEEE.

Salva, S. and Sue, J. (2024). Security testing of restful apis with test case mutation. In Kaindl, H., Mannion, M., and Maciaszek, L. A., editors, *Proceedings of the 19th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE 2024, Angers, France, April 28-29, 2024*, pages 582–589. SCITEPRESS.

Schneider, C., Barker, A., and Dobson, S. (2015). A survey of self-healing systems frameworks. *Software: Practice and Experience*, 45(10):1375–1398.

Schuster, R., Song, C., Tromer, E., and Shmatikov, V. (2021). You autocomplete me: Poisoning vulnerabilities in neural code completion. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1559–1575. USENIX Association.

Subramanian, S., Thiran, P., Narendra, N. C., Mostefaoui, G. K., and Maamar, Z. (2008). On the enhancement of bpel engines for self-healing composite web services. In *2008 International Symposium on Applications and the Internet*, pages 33–39.

Sue, J. and Salva, S. (2025). Dynamic mitigation of restful service failures using llms, companion site. https://github.com/JarodSue/Fine-grained-Restful-Service-Healing-using-LLMs.

Tian, K., Mitchell, E., Zhou, A., Sharma, A., Rafailov, R., Yao, H., Finn, C., and Manning, C. D. (2023). Just ask for calibration: Strategies for eliciting calibrated confidence scores from language models fine-tuned with human feedback.

Tosi, D., Denaro, G., and Pezze, M. (2007). Shiws: A self-healing integrator for web services. In *29th International Conference on Software Engineering (ICSE07 Companion)*, pages 55–56, Los Alamitos, CA, USA. IEEE Computer Society.

Vizcarrondo, J., Aguilar, J., Exposito, E., and Subias, A. (2012). Armiscom: Autonomic reflective middleware for management service composition. In *2012 Global Information Infrastructure and Networking Symposium (GIIS)*, pages 1–8.

Wang, Y. (2019). Towards service discovery and autonomic version management in self-healing microservices architecture. In *Proceedings of the 13th European Conference on Software Architecture - Volume 2*, ECSA '19, page 63–66, New York, NY, USA. Association for Computing Machinery.

Xiong, M., Hu, Z., Lu, X., LI, Y., Fu, J., He, J., and Hooi, B. (2024). Can LLMs express their uncertainty? an empirical evaluation of confidence elicitation in LLMs. In *The Twelfth International Conference on Learning Representations*.

Yasunaga, M. and Liang, P. (2020). Graph-based, self-supervised program repair from diagnostic feedback. In *Proceedings of the 37th International Conference on Machine Learning*, ICML'20. JMLR.org.