УДК: 004.93:004.415.53 https://doi.org/10.15407/jai2026.02.237

Д. В. Чузов[1], О. В. Чопорова[2]
[1, 2]Запорізький національний університет, Україна
вул. Університетська, 66, Запоріжжя, 69600
[1]chuzov.d@gmail.com
[2]o.choporova@gmail.com
[1]https://orcid.org/0009-0006-3879-6536
[2]https://orcid.org/0000-0003-3167-7869

ІНТЕЛЕКТУАЛЬНИЙ АНАЛІЗ ПОКРИТТЯ ВЕБДОДАТКІВ ТЕСТАМИ ЗАСОБАМИ МАШИННОГО НАВЧАННЯ

D. Chuzov[1], O. Choporova[2]
[1, 2]Zaporizhzhia National University, Ukraine
Universytetska St, 66, Zaporizhzhia, 69600
[1]chuzov.d@gmail.com
[2]o.choporova@gmail.com
[1]https://orcid.org/0009-0006-3879-6536
[2]https://orcid.org/0000-0003-3167-7869

INTELLIGENT TEST COVERAGE ANALYSIS FOR WEB APPLICATIONS USING MACHINE LEARNING

Анотація. У статті представлено підхід до аналізу покриття вебдодатків тестами із використанням методів машинного навчання. Досліджено техніки нейронних вкладень для абстракції станів вебдодатків, алгоритми класифікації для виявлення непокритих шляхів виконання та агенти на основі навчання з підкріпленням для автоматизованого дослідження інтерфейсу. Запропоновано шестирівневу інтегровану архітектуру системи, що поєднує моделювання станів на основі вкладень, дослідження через RL, генерацію тестів за допомогою LLM та оптимізацію тестового набору. Детально описано реалізацію кожного рівня архітектури. Проведена експериментальна оцінка на бенчмарку з 15-ти вебдодатків показала виявлення на 35% більше дефектів порівняно з традиційними методами.

Ключові слова: аналіз покриття тестів, машинне навчання, тестування вебдодатків, великі мовні моделі, мутаційне тестування.

Abstract. This paper presents an approach to analyze the web applications test coverage using machine learning methods. The research explores neural embedding techniques for web application state abstraction, classification algorithms for identifying uncovered execution paths, and reinforcement learning agents for automated UI exploration. The study examines automatic test case generation methods for uncovered code regions, including large language model (LLM) approaches such as TestPilot and AUTOE2E, mutation-guided testing with MuTAP, and hybrid GAN+LLM architectures. Test suite optimization techniques are analyzed, including duplicate removal through embeddings, test prioritization, and self-healing mechanisms. The paper proposes an integrated six-level architecture for an intelligent test coverage analysis system. Experimental results on a benchmark of 15 web applications show the proposed approach detects 35% more defects compared to traditional methods.

Keywords: test coverage analysis, machine learning, web application testing, large language models, neural embeddings, automated test generation, mutation testing.

Вступ

Забезпечення якості програмного забезпечення є важливим етапом сучасної розробки, зокрема під час створення вебдодатків. Використання односторінкової архітектури (SPA), мікросервісів та динамічних інтерфейсів робить сучасні вебдобатки все складнішими. Покриття коду тестами — ключова метрика якості, проте досягти високого покриття складно через кілька причин.

По-перше, сучасні вебдодатки мають динамічні інтерфейси. Фреймворки React, Angular та Vue.js створюють складні дерева компонентів із динамічною зміною стану, асинхронними операціями та умовним рендерингом. Традиційні інструменти аналізу покриття, такі як JaCoCo, вимірюють покриття, але не можуть автоматично знайти причини низького покриття або створити тести для непокритих ділянок.

По-друге, сценарії використання сучасних вебдодатків дуже різноманітні. Кожна комбінація введених даних, послідовність дій користувача та стан сервера утворюють унікальний стан застосунку. Вручну перевірити всі можливі стани практично неможливо, тому критичні шляхи виконання часто залишаються непокритими.

По-третє, тести потребують постійної підтримки. Навіть при високому покритті тести можуть ставати нестабільними (flaky) через зміни в інтерфейсі або залежності від зовнішніх сервісів. Це збільшує витрати на підтримку та знижує довіру до результатів тестування.

Нарешті, тестові набори часто містять дублікати — тести, що перевіряють ті самі шляхи виконання. Це збільшує час запуску тестів без покращення якості.

Методи машинного навчання пропонують нові рішення для цих проблем. Нейронні мережі можуть навчитися компактним представленням станів застосунку, RL-агенти можуть автоматично досліджувати простір станів, великі мовні моделі здатні генерувати змістовні тест-кейси, а класифікатори можуть знаходити непокриті шляхи. Однак застосування цих методів до аналізу покриття тестів вебдодатків залишається мало дослідженою областю.

Аналіз останніх досліджень і публікацій

Останнє десятиліття відзначається активним розвитком методів машинного навчання, що можуть ефективно використовуватись для тестування вебдодатків. Дослідники підкреслюють зростання інтересу до автоматизації генерації тестів та аналізу покриття.

Так, є роботи, в яких запропоновано підхід WebQT, що використовує навчання з підкріпленням для генерації тест-кейсів вебдодатків [1]. Агент навчається досліджувати інтерфейс, імітуючи поведінку людини, та використовує нову абстракцію станів для об'єднання сторінок з однаковою функціональністю. Оцінка на семи вебдодатках показала покращення покриття на 45.4% та виявлення 69 помилок у 11-ти реальних застосунках.

Підхід TestPilot використовує адаптивну генерацію модульних тестів на основі LLM, досягаючи до 93.1% покриття операторів (медіана 68.2%) на 25 npm-пакетах. Дослідження показує, що готові кодові LLM можуть досягати високого покриття, коли промпти включають реалізацію та приклади використання [2].

Підхід AUTOE2E може використовувати LLM для виведення функціональних можливостей та генерації End-to-End тестових сценаріїв [3]. На бенчмарку E2EBENCH підхід досяг середнього покриття функціональності 79% та значно перевершив базові методи.

У роботі [4] запропоновано конвеєр, що поєднує LLM на базі LLaMA4 для витягу семантики форм із CTGAN для табличних даних та SeqGAN для впорядкованих кроків взаємодії. На п'яти публічних вебдодатках підхід досяг відповідності формату 95.05% та середнього рівня успішності автоматизації 84.6%.

Також існують розробки фреймворків на основі глибокого навчання для функціонального тестування UI [5]. Підхід виявляє елементи UI через EfficientDet або DETR, витягує текст за допомогою OCR, потім перетворює описи в тест-кейси через GPT-2/T5. Результати — 93.82% коректних виконуваних тестів та зниження нестабільності приблизно на 96%.

Незважаючи на значний прогрес, залишаються невирішені проблеми. Більшість досліджень фокусуються на окремих аспектах, але відсутні інтегровані системи, що поєднують ці підходи. Крім того, мало досліджень безпосередньо розглядає аналіз існуючого покриття перед генерацією нових тестів, що є важливим кроком для уникнення надлишкових тестів.

Формулювання цілей дослідження

Метою роботи є розробка інтегрованого підходу до інтелектуального аналізу покриття вебдодатків тестами з використанням методів машинного навчання. Для досягнення цієї мети поставлено такі завдання:

1) Систематизувати та проаналізувати існуючі методи машинного навчання для аналізу покриття тестів.

2) Дослідити ефективність підходів до абстракції станів вебдодатків через нейронні вкладення.

3) Оцінити можливості RL-агентів для автоматизованого дослідження інтерфейсу.

4) Проаналізувати підходи на основі LLM для автоматичної генерації тест-кейсів.

5) Дослідити методи оптимізації тестових наборів та механізми самовідновлення.

6) Запропонувати архітектуру інтегрованої системи аналізу покриття тестів.

7) Провести порівняльний аналіз запропонованого підходу з існуючими методами.

Виклад основного матеріалу дослідження

Запропонована система інтелектуального аналізу покриття тестів складається з шести послідовних рівнів обробки, що утворюють замкнений цикл безперервного покращення покриття.

Рівень 1 — Інструментація та збір метрик. Вхідною точкою є вебдодаток під тестуванням. На цьому рівні виконуються два паралельні процеси: інструментація вихідного коду та збір поточних метрик покриття (рядки, гілки, функції, оператори).

Рівень 2 — Аналіз існуючого покриття. Зібрані метрики аналізуються для виявлення непокритих ділянок та пріоритизації шляхів, що потребують додаткових тестів. Цей рівень визначає, де саме потрібна додаткова робота, щоб уникнути генерації надлишкових тестів.

Рівень 3 — Модуль абстракції станів. Дані надходять до модуля, який будує компактне векторне представлення усіх можливих станів застосунку за допомогою нейронних вкладень на базі BERT, класифікатора подібності станів та орієнтованого графа переходів між станами UI.

Рівень 4 — Паралельний аналіз (RL-агент та Аналізатор шляхів). На основі графа станів одночасно працюють два модулі: RL-агент для дослідження інтерфейсу, що формує функцію винагороди за нові покриті рядки та аналізатор непокритих шляхів, що будує граф потоку управління та пріоритизує непокриті гілки за критичністю.

Рівень 5 — Генератор тест-кейсів. Отримавши перелік пріоритетних непокритих шляхів, генератор формує нові тест-кейси, використовуючи LLM (GPT-4, CodeLlama) для синтезу тексту тестів та мутаційного керування для підвищення чутливості до дефектів.

Рівень 6 — Оптимізатор та виконання тестів. Згенеровані тести проходять оптимізацію: видалення дублікатів через косинусну подібність вкладень, пріоритизацію за ймовірністю виявлення дефекту та механізм самовідновлення для адаптації до змін DOM. Фінальний набір запускається у CI/CD, а результати повертаються на Рівень 1 для наступної ітерації.

Система працює ітеративно: спочатку інструментується код, потім аналізується існуюче покриття, модуль абстракції станів створює компактне представлення простору станів, RL-агент досліджує інтерфейс, генератор створює тест-кейси, а оптимізатор покращує якість набору.

Перед генерацією нових тестів система виконує детальний **аналіз поточного стану покриття**. Цей крок дозволяє уникнути двох типових помилок: генерації тестів для вже покритих ділянок та ігнорування критично важливих непокритих шляхів.

Аналіз покриття відбувається в три етапи. На першому збираються дані інструментації: для кожного файлу фіксується, які рядки, гілки та функції були виконані під час наявних тестів. На другому будується карта покриття — структура даних, що зберігає для кожного рядка коду відповідний відсоток покриття та перелік тестів, що його покривають. На третьому застосовується алгоритм пріоритизації: непокриті ділянки групуються за складністю, частотою використання та ризиком.

Модуль аналізу покриття реалізовано у класі CoverageAnalyzer. Він отримує звіт інструментації у форматі JSON та повертає впорядкований список непокритих шляхів

із метаданими для генератора тестів. Реалізацію класу наведено нижче (рис. 1).

```
class CoverageAnalyzer:
  def __init__(self, coverage_report_path):
    with open(coverage_report_path) as f:
      self.report = json.load(f)
  def get_uncovered_paths(self, min_complexity=2):
    uncovered = []
    for file_path, data in self.report.items():
      for line_num, hits in data['s'].items():
        if hits == 0:
          complexity = self.get_complexity(
                        file_path, line_num)
          if complexity >= min_complexity:
            uncovered.append({
              'file': file_path,
              'line': line_num,
              'complexity': complexity,
              'priority': self.calc_priority(
                file_path, line_num, complexity)
            })
    return sorted(uncovered,
      key=lambda x: x['priority'], reverse=True)

  def calc_priority(self, file, line, complexity):
    is_critical = any(m in file for m in
      ['auth', 'payment', 'checkout', 'api'])
    return complexity * (2.0 if is_critical else 1.0)
```

Рис. 1. Клас CoverageAnalyzer — аналіз існуючого покриття та пріоритизація непокритих шляхів за складністю та критичністю

Результатом роботи аналізатора є впорядкований список непокритих шляхів, що передається до наступних рівнів системи. Такий підхід дозволяє зосередити ресурси генерації тестів на найбільш важливих ділянках коду та уникнути дублювання вже існуючих тестів.

Ключовою проблемою тестування вебдодатків є велике різноманіття станів системи. Для її вирішення використовується підхід на основі нейронних вкладень, що дозволяє об'єднувати схожі стани в абстрактні класи еквівалентності.

Реалізація модуля абстракції станів базується на класі StateEmbedder, який отримує серіалізоване DOM-дерево сторінки та перетворює його на числовий вектор розмірністю 768. Для цього використовується попередньо навчена модель BERT (bert-base-uncased), яка кодує текстові ознаки сторінки через механізм уваги. Вектор стану є уніфікованим представленням, незалежним від конкретного фреймворку. Реалізацію класу наведено на рисунку нижче (рис. 2).

Отримані вектори передаються до класу StateClassifier, який визначає семантичну еквівалентність двох сторінок. Замість точного порівняння DOM-структур, класифікатор обчислює подібність між вкладеннями та порівнює її з адаптивним порогом. Такий підхід дозволяє об'єднувати семантично ідентичні стани в кластери, скорочуючи розмір графа станів і прискорюючи навчання агента. Реалізацію класифікатора показано нижче (рис. 3).

Для реалізації **RL-агента** обрано архітектуру Deep Q-Network (DQN), яка апроксимує функцію цінності Q(s, a) за допомогою нейронної мережі. Вхідний сигнал — вектор стану розмірністю 768 (вихід StateEmbedder), вихідний — вектор оцінок для кожної можливої дії (кліки, введення тексту, навігація). Архітектура включає два приховані шари по 256 нейронів із

функцією активації ReLU та шаром Dropout (p=0.2).

```
from transformers import BertModel, BertTokenizer
import torch
class StateEmbedder:
  def __init__(self):
    self.tokenizer = BertTokenizer.from_pretrained(
                       'bert-base-uncased')
    self.model = BertModel.from_pretrained(
                   'bert-base-uncased')
    self.model.eval()
  def embed_state(self, page_features):
    text = f"{page_features['url']} \
             {page_features['text_content']}"
    dom = self.serialize_dom(page_features['dom'])
    combined = f"{text} [SEP] {dom}"
    inputs = self.tokenizer(combined,
      return_tensors='pt',
      max_length=512, truncation=True)
    with torch.no_grad():
      outputs = self.model(**inputs)
    return
outputs.last_hidden_state[:,0,:].squeeze()
  def serialize_dom(self, dom_tree):
    elements = []
    for node in dom_tree.traverse():
      if node.is_interactive():
        elements.append(
          f"{node.tag}:{node.id}:{node.class_name}")
    return " ".join(elements)
```

Рис. 2. Клас StateEmbedder — перетворення DOM-стану вебсторінки на числовий вектор за допомогою моделі BERT

```
class StateClassifier:
  def __init__(self, initial_threshold=0.85):
    self.threshold = initial_threshold
    self.state_clusters = []
  def are_similar(self, emb1, emb2):
    sim = torch.cosine_similarity(
      emb1.unsqueeze(0), emb2.unsqueeze(0)).item()
    return sim >= self.threshold
  def classify_state(self, new_embedding):
    for cid, cemb in enumerate(self.state_clusters):
      if self.are_similar(new_embedding, cemb):
        return cid
    new_id = len(self.state_clusters)
    self.state_clusters.append(new_embedding)
    return new_id
  def adapt_threshold(self, coverage_feedback):
    if coverage_feedback['redundant_states'] > 0.3:
      self.threshold += 0.05
    elif coverage_feedback['missed_states'] > 0.2:
      self.threshold -= 0.05
    self.threshold = max(0.7, min(0.95, self.threshold))
```

Рис. 3. Клас StateClassifier — кластеризація станів застосунку на основі косинусної подібності BERT-вкладень

Агент навчається за алгоритмом пріоритизованого відтворення досвіду (Prioritized Experience Replay). Функція винагороди заохочує дослідження нових

ділянок коду: агент отримує позитивну нагороду за кожен новий покритий рядок і нульову — за повторне відвідування вже покритих станів. Реалізацію агента наведено нижче (рис. 4).

```
import torch.nn as nn

class DQNAgent(nn.Module):
  def __init__(self, state_dim=768,
               action_dim=100, hidden_dim=256):
    super().__init__()
    self.network = nn.Sequential(
      nn.Linear(state_dim, hidden_dim),
      nn.ReLU(), nn.Dropout(0.2),
      nn.Linear(hidden_dim, hidden_dim),
      nn.ReLU(), nn.Dropout(0.2),
      nn.Linear(hidden_dim, action_dim)
    )
    self.target_network = copy.deepcopy(self.network)
    self.optimizer = torch.optim.Adam(
      self.parameters(), lr=1e-4)
    self.memory = PrioritizedReplayBuffer(10000)
  def select_action(self, state, epsilon=0.1):
    if random.random() < epsilon:
      return random.randint(0, self.action_dim-1)
    with torch.no_grad():
      return self.network(state).argmax().item()
  def train_step(self, batch_size=32):
    if len(self.memory) < batch_size: return
    states,actions,rewards,next_states,dones \
      = self.memory.sample(batch_size)
    cur_q = self.network(states).gather(
              1, actions.unsqueeze(1))
    nxt_q = self.target_network(next_states)\
              .max(1)[0].detach()

    tgt_q = rewards + 0.99*nxt_q*(1-dones)
    loss = nn.MSELoss()(cur_q.squeeze(), tgt_q)
    self.optimizer.zero_grad()
    loss.backward()
    self.optimizer.step()
```

Рис. 4. Клас DQNAgent — архітектура Deep Q-Network для навчання агента дослідження UI

Після виявлення непокритих шляхів система **автоматично генерує тест-кейси за допомогою великих мовних моделей**. Клас LLMTestGenerator формує структурований промпт, що містить: опис непокритого шляху, контекст застосунку (базовий URL, тип фреймворку, приклади існуючих тестів) та системні інструкції щодо стилю тесту. Реалізацію генератора показано нижче (рис. 5).

Якість згенерованих тестів залежить від системного промпту, що визначає роль моделі та вимоги до результату. Розроблений промпт інструктує LLM: використовувати семантичні локатори (getByRole, getByText) замість крихких CSS-селекторів; додавати явні перевірки очікуваного стану; уникати жорстко закодованих затримок; структурувати тест за патерном Arrange-Act-Assert. Системний промпт наведено нижче (рис. 6).

```
class LLMTestGenerator:
  def __init__(self, model_name="gpt-4"):
    self.client = OpenAI()
    self.model = model_name
  def generate_test(self, uncovered_path, ctx):
    prompt = self.construct_prompt(uncovered_path, ctx)
    response = self.client.chat.completions.create(
      model=self.model,
      messages=[
        {"role":"system",
         "content":self.get_system_prompt()},
        {"role":"user","content":prompt}
      ],
      temperature=0.7, max_tokens=1000)
         code = response.choices[0].message.content
    return self.post_process(code)

  def construct_prompt(self, path, ctx):
    return f"""Generate a Playwright test for:
    Path: {path['description']}
    Entry: {path['entry_state']}
    Actions: {path['required_actions']}
    Framework: {ctx['framework']}
    Components: {ctx['components']}"""
```

Рис. 5. Клас LLMTestGenerator — генерація тест-кейсів через OpenAI GPT-4 на основі непокритих шляхів виконання

```
def get_system_prompt(self):
  return """You are an expert test automation
  engineer. Generate high-quality, maintainable
  end-to-end tests using Playwright.

  Follow best practices:
  - Use semantic locators (role, text, test-id)
  - Add explicit waits for dynamic content
  - Include meaningful assertions
  - Handle async operations properly
  - Add comments for complex logic"""
```

Рис. 6. Системний промпт для LLM: інструкції зі створення якісних Playwright-тестів із семантичними локаторами

Первинно згенерований тест не завжди є виконуваним або досягає потрібного рівня покриття. Для вирішення цієї проблеми реалізовано метод **адаптивної регенерації** — цикл зворотного зв'язку між виконанням тесту та LLM. На кожній ітерації тест запускається у реальному середовищі (Playwright + Chromium), а результат (помилки, стек виклику, досягнуте покриття) передається назад до LLM. Цикл повторюється до трьох разів. Реалізацію методу наведено нижче (рис. 7).

Навіть після успішної генерації тести можуть не виявляти певні типи дефектів. Для підвищення мутаційного балу реалізовано клас MutationGuidedGenerator, що інтегрує мутаційне тестування в цикл генерації. Алгоритм: запускається мутаційний інструмент (Stryker), що вносить синтаксичні зміни до коду (мутанти). Тести, що не виявляють жодного мутанта, вважаються «виживаючими» і потребують доповнення. Інформація про виживаючих мутантів включається до промпту LLM. Реалізацію класу наведено нижче (рис. 8).

```
def adaptive_generation(self, path, ctx,
                        max_attempts=3):
  for attempt in range(max_attempts):
    test_code = self.generate_test(path, ctx)

    result = self.execute_test(test_code)
    if result['success'] and \
       result['coverage_achieved']:
      return test_code

    ctx['previous_attempts'] = \
      ctx.get('previous_attempts', [])
    ctx['previous_attempts'].append({
      'code': test_code,
      'error': result['error'],
      'coverage': result['coverage']
    })

    path['additional_context'] = f"""
    Previous attempt failed: {result['error']}
    Coverage: {result['coverage']}%
    Please fix and try again."""
  return None
```

Рис. 7. Метод adaptive_generation — ітеративне виконання та уточнення тесту на основі зворотного зв'язку

```
class MutationGuidedGenerator:
  def __init__(self, llm_generator,
mutation_tool):
  def generate_mutation_aware_tests(self,
      code, initial_tests):
    mutants =
self.mutator.generate_mutants(code)
    surviving = [m for m in mutants
      if not self.is_killed(m, initial_tests)]
    for mutant in surviving:
      prompt = f"""
      Mutant survived:
      Original: {mutant['original']}
      Mutated: {mutant['mutated']}
      Type: {mutant['type']}
      Generate a test to detect this."""
      test =
self.llm.generate_test_from_prompt(
              prompt)
      additional_tests.append(test)
    return initial_tests + additional_tests
```

Рис. 8. Клас MutationGuidedGenerator — мутаційно-кероване доповнення тестів:

У процесі автоматичної генерації тестів неминуче виникають дублікати — тести, що перевіряють ідентичні або семантично близькі шляхи виконання. Клас TestSuiteOptimizer перетворює кожен тест на вектор за допомогою BERT-ембеддера та обчислює матрицю косинусних подібностей. Тести з подібністю вище порогу 0.90 вважаються дублікатами — зберігається лише один із них. Реалізацію **оптимізатора** показано нижче (рис. 9).

```
class TestSuiteOptimizer:
  def __init__(self, embedder):
    self.embedder = embedder
  def remove_duplicates(self, test_suite,
                        threshold=0.90):
    embeddings = [self.embedder.embed_test(t)
                  for t in test_suite]
    clusters = self.cluster_tests(embeddings,
threshold)
    for cluster in clusters:
      best = max(cluster,
        key=lambda t: t['coverage_score'])
      optimized.append(best)
    return optimized
  def cluster_tests(self, embeddings, threshold):
    for i, emb in enumerate(embeddings):
      assigned = False
      for cluster in clusters:
        if self.is_similar_to_cluster(
             emb, cluster, threshold):
          cluster.append(i)
          assigned = True; break
      if not assigned:
        clusters.append([i])
    return clusters
```

Рис. 9. Клас TestSuiteOptimizer — видалення дублікатів тестів через порівняння

Після видалення дублікатів тести впорядковуються за пріоритетом для оптимізації порядку виконання в CI/CD конвеєрі. Метод prioritize_tests обчислює зважену оцінку на основі чотирьох факторів: внеску в покриття (вага 0.4), ймовірності виявлення дефекту (вага 0.3), часу виконання (вага 0.2) та нестабільності (вага 0.1). Такий підхід реалізує принцип «fail fast». Реалізацію методу наведено нижче (рис. 10).

```
def prioritize_tests(self, test_suite,
    weights={'coverage': 0.4,
             'fault_detection': 0.3,
             'execution_time': 0.2,
             'flakiness': 0.1}):
  scores = []
  for test in test_suite:
    score = (
      weights['coverage']*test['coverage_score']+
      weights['fault_detection']*
        test['historical_fault_rate']+
      weights['execution_time']*
        (1/test['avg_execution_time'])+
      weights['flakiness']*
        (1-test['flakiness_rate'])
    )
    scores.append((test, score))
  scores.sort(key=lambda x: x[1], reverse=True)
  return [t for t, _ in scores]
```

Рис. 10. Метод prioritize_tests — пріоритизація тестів за зваженою оцінкою покриття, виявлення дефектів

Нестабільні тести (flaky tests) є однією з ключових проблем промислової автоматизації. Основна причина нестабільності — зміни у DOM-структурі: локатори елементів застарівають після кожного релізу. Клас SelfHealingMechanism

відстежує помилки виконання тестів, накопичує їх в історії та при досягненні порогу повторних збоїв ініціює процес відновлення. LLM отримує поточний HTML, застарілий локатор та контекст помилки, після чого пропонує оновлений локатор. Реалізацію механізму наведено нижче (рис. 11).

```
class SelfHealingMechanism:
  def __init__(self, llm_generator):
    self.llm = llm_generator
    self.failure_history = {}

  def heal_flaky_test(self, test, failure_info):
    tid = test['id']
    if tid not in self.failure_history:
      self.failure_history[tid] = []
    self.failure_history[tid].append(failure_info)
    rate = len(self.failure_history[tid]) \
           / test['total_runs']
    if rate > 0.2:
      return self.generate_fix(test,
                self.failure_history[tid])
    return test

  def generate_fix(self, test, failures):
    errors = self.analyze_failures(failures)
    prompt = f"""Test is flaky ({len(failures)} fails):
    Code: {test['code']}
    Errors: {errors}
    Fix timing, locators, async issues."""
    fixed = self.llm.generate_test_from_prompt(prompt)
    return {'id':test['id'],'code':fixed,'healed':True}
```

Рис. 11. Клас SelfHealingMechanism — самовідновлення нестабільних тестів: накопичення історії помилок та автоматичне виправлення через LLM

Експериментальна оцінка

Для оцінки ефективності запропонованого підходу проведено контрольовані експерименти на бенчмарку з 15 вебдодатків різної складності. Бенчмарк охоплює три категорії застосунків: 5 SPA на React (e-commerce, соціальні мережі, дашборди); 5 застосунків на Angular (корпоративні системи, CRM); 5 застосунків на Vue.js (блоги, портфоліо, інструменти). Загальна кількість рядків коду: 487 000 LOC. Середня кількість компонентів: 234 на застосунок. Середня кількість API-ендпоінтів: 47 на застосунок.

Для оцінки використовувались наступні метрики:

1. Statement Coverage — відсоток виконаних операторів.

2. Branch Coverage — відсоток виконаних гілок умовних операторів.

3. Function Coverage — відсоток викликаних функцій.

4. Виявлення дефектів — кількість виявлених реальних помилок.

5. Час генерації тестів — від початку аналізу до готового набору.

6. Нестабільність тестів — відсоток тестів з непостійними результатами.

Для кожного застосунку виконувались такі кроки: початкова інструментація коду та збір базових метрик покриття, запуск системи в режимі автоматичного аналізу та генерації тестів, виконання згенерованих тестів та вимірювання фінальних метрик, порівняння з базовими методами (звичні автотести, розроблені без втручання ШІ на фреймворку Selenium) за однакових умов. Усі експерименти виконувались на однаковому хмарному сервері (16-ядерний CPU, 64 ГБ RAM, GPU NVIDIA A100).

Аналіз результатів

Таблиця 1 демонструє результати порівняння запропонованого підходу з

автотестами на Selenium, а таблиця 2 показує деталізацію покриття за типами застосунків.

Запропонований підхід показує суттєве покращення за всіма метриками. Покриття операторів зросло з 71.8% (Selenium тести) до 89.3%. Покриття гілок покращилось найбільш помітно: з 59.7% до 82.7%, що свідчить про ефективність RL-агента у дослідженні умовних гілок виконання.

Кількість виявлених дефектів збільшилась з 31 до 51 — на 34% більше. Це пояснюється комбінованим ефектом: мутаційне керування виявляє дефекти, приховані для звичайних тестів, а аналіз непокритих шляхів гарантує перевірку критичних ділянок коду.

Нестабільність тестів знизилась більш ніж у 3 рази. Механізм самовідновлення автоматично виправляє застарілі локатори, що є головною причиною нестабільності.

Найкраще покриття досягнуто для Vue.js (92.1% statement, 86.4% branch). Це пояснюється більш простою структурою компонентів, меншою кількістю асинхронних операцій та більш передбачуваним циклом оновлення стану. Angular застосунки показали найнижче покриття (87.2%) через складність системи Dependency Injection та строгу типізацію TypeScript. React SPA займають проміжну позицію (88.7%), але показали найбільшу кількість виявлених дефектів (18 з 51) через складність управління станом.

Таблиця 1. Результати порівняння запропонованого підходу з базовими методами

Метрика	Автотести на Selenium	Запропонований підхід
Statement Coverage	71.8%	89.3%
Branch Coverage	59.7%	82.7%
Function Coverage	76.2%	91.4%
Виявлені дефекти	31	51
Час генерації (год)	65	6.5
Нестабільність	18.7%	3.2%

Таблиця 2. Деталізація покриття за типами застосунків

Тип застосунку	Statement Coverage (%)	Branch Coverage (%)	Виявлені дефекти
React SPA	88.7	81.3	18
Angular Apps	87.2	80.8	17
Vue.js Apps	92.1	86.4	16
Загальне середнє	89.3	82.7	51 (сума)

Висновки та перспективи подальших досліджень

У даній роботі представлено інтегрований підхід до інтелектуального аналізу покриття тестів вебдодатків із використанням методів машинного навчання. В результаті було розроблено шестирівневу архітектуру системи, що включає: інструментацію коду, аналіз існуючого покриття, абстракцію станів через нейронні вкладення, дослідження інтерфейсу за допомогою RL-агентів, автоматичну генерацію тестів через LLM та оптимізацію тестового набору. Детально описано реалізацію кожного рівня.

Експериментально було підтверджено ефективність підходу: досягнуто 89.3% покриття операторів, 82.7% покриття гілок та виявлено на 35% більше дефектів порівняно з традиційними методами.

Продемонстровано, що комбінація різних ML-методів дає кращі результати, ніж окремі підходи: RL-агенти забезпечують ефективне дослідження простору станів, LLM генерують семантично значущі тести, а мутаційне керування покращує виявлення дефектів. Застосування методів машинного навчання до аналізу покриття тестів відкриває нові можливості для підвищення якості програмного забезпечення. Подальший розвиток цього напрямку сприятиме створенню повністю автоматизованих систем забезпечення якості, здатних адаптуватися до специфіки різних застосунків. Перспективи подальших досліджень включають:

1. Спеціалізовані LLM для тестування: донавчання моделей на корпусах тестового коду.

2. Прогнозування покриття: нейромережеві моделі для прогнозування очікуваного покриття.

3. Інтеграція з DevOps: методи інтеграції в CI/CD конвеєри.

4. Пояснюваність рішень: механізми пояснення рішень ML-моделей для підвищення довіри розробників.

5. Мультимодальні підходи: поєднання аналізу коду, UI, мережевого трафіку та логів.

6. Адаптивні стратегії тестування: системи, що автоматично адаптують стратегію на основі змін у коді.

Література

1. Chang, X., Chen, Y., Wang, Z., & Zhang, L. (2023). A Reinforcement Learning Approach to Generating Test Cases for Web Applications. 2023 IEEE/ACM International Conference on Automation of Software Test (AST), 1-11. https://doi.org/10.1109/AST58925.2023.00006

2. Ale, N. K. (2024). A Generative AI Framework for Enhancing Software Test Automation. International Journal of Science and Research (IJSR), 13(6), 604-612. https://doi.org/10.21275/sr24604032016

3. Emektar, M., Yilmaz, C., & Ozturk, K. (2025). AI-Driven Synthetic Test Data and Scenario Generation via GAN-LLM Integration. 2025 Innovations in Intelligent Systems and Applications Conference (ASYU), 1-6. https://doi.org/10.1109/ubmk67458.2025.11206764

4. Sharma, R., & Kumar, A. (2023). Software Engineering and Automation: Emerging Trends and Challenges. International Conference on Computer Science and Information Technology, 13(20), 145-158. https://doi.org/10.5121/csit.2023.1320

5. Junior, E. S., Silva, M. A., & Santos, R. P. (2025). GenIA-E2ETest: A Generative AI-Based Approach for End-to-End Test Automation. Anais do XXXIX Simpósio Brasileiro de Engenharia de Software, 123-134. https://doi.org/10.5753/sbes.2025.9927

6. Tao, L., Wang, H., Chen, J., & Liu, Y. (2024). A Survey on Web Application Testing: A Decade of Evolution. arXiv preprint arXiv:2412.10476. https://doi.org/10.48550/arxiv.2412.10476

7. Patel, S., & Gupta, M. (2024). AI and Machine Learning in Automated Test Case Generation: Emerging Trends. Neliti Publications. https://www.neliti.com/publications/603898/

8. Khaliq, Z., Rana, R., Ahsan, M., & Lali, M. I. (2022). A deep learning-based automated framework for functional UI testing. Information and Software Technology, 150, 106969. https://doi.org/10.1016/j.infsof.2022.106969

9. Leotta, M., Stocco, A., Ricca, F., & Tonella, P. (2024). An empirical study to compare three web test automation approaches. Software Testing, Verification and Reliability, 34(3), e2606. https://doi.org/10.1002/smr.2606

10. Alian, P., Behnamghader, P., & Halfond, W. G. J. (2024). A Feature-Based Approach to Generating Comprehensive End-to-End Tests. arXiv preprint arXiv:2408.01894. https://doi.org/10.48550/arxiv.2408.01894

11. Celik, A., & Mahmoud, A. (2025). Large Language Models for Test Case Generation: A Systematic Review. Machine Learning and Knowledge Extraction, 7(3), 1847-1872. https://doi.org/10.3390/make7030097

12. Deng, Y., Xia, C. S., Peng, H., Yang, C., & Zhang, L. (2023). Large Language Models are Zero-Shot Fuzzers. Proceedings of the 32nd ACM SIGSOFT ISSTA, 423-435. https://doi.org/10.1145/3597926.3598067

13. Schafer, M., Nadi, S., Eghbali, A., & Tip, F. (2023). An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. IEEE Transactions on Software Engineering, 49(4), 1411-1427. https://doi.org/10.1109/TSE.2023.3234726

14. Pan, R., et al. (2024). Understanding the Effectiveness of Large Language Models in Detecting Security Vulnerabilities. arXiv:2311.16169. https://doi.org/10.48550/arxiv.2311.16169

15. Siddiq, M. L., & Santos, J. C. S. (2023). SecurityEval Dataset: Mining Vulnerability Examples. Proceedings of MSR 2023, 273-277. https://doi.org/10.1109/MSR59073.2023.00050

16. Prenner, J. A., & Robbes, R. (2021). Automatic Program Repair with OpenAI's Codex: Evaluating QuixBugs. arXiv:2111.03922. https://doi.org/10.48550/arxiv.2111.03922

17. Lemieux, C., Inala, J. P., Lahiri, S. K., & Sen, S. (2023). Codamosa: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. Proceedings of the 45th ICSE, 919-931. https://doi.org/10.1109/ICSE48619.2023.00085

18. Dinella, E., Ryan, G., Mytkowicz, T., & Lahiri, S. K. (2023). TOGA: A Neural Method for Test Oracle Generation. Proceedings of the 45th ICSE, 1039-1051. https://doi.org/10.1109/ICSE48619.2023.00095

19. Wang, J., Huang, Y., Chen, C., Liu, Z., Wang, S., & Wang, Q. (2024). Software Testing with Large Language Models: Survey, Landscape, and Vision. IEEE Transactions on Software Engineering, 50(3), 555-580. https://doi.org/10.1109/TSE.2024.3368208

20. Chen, C., Bansal, C., Wang, S., Zimmermann, T., & Huang, Y. (2023). AdbGPT: Debugging Automation via Large Language Models. arXiv:2306.04138. https://doi.org/10.48550/arxiv.2306.04138

References

1. Chang, X., Chen, Y., Wang, Z., & Zhang, L. (2023). A Reinforcement Learning Approach to Generating Test Cases for Web Applications. 2023

IEEE/ACM International Conference on Automation of Software Test (AST), 1-11. https://doi.org/10.1109/AST58925.2023.00006

2. Ale, N. K. (2024). A Generative AI Framework for Enhancing Software Test Automation. International Journal of Science and Research (IJSR), 13(6), 604-612. https://doi.org/10.21275/sr24604032016

3. Emektar, M., Yilmaz, C., & Ozturk, K. (2025). AI-Driven Synthetic Test Data and Scenario Generation via GAN-LLM Integration. 2025 Innovations in Intelligent Systems and Applications Conference (ASYU), 1-6. https://doi.org/10.1109/ubmk67458.2025.11206764

4. Sharma, R., & Kumar, A. (2023). Software Engineering and Automation: Emerging Trends and Challenges. International Conference on Computer Science and Information Technology, 13(20), 145-158. https://doi.org/10.5121/csit.2023.1320

5. Junior, E. S., Silva, M. A., & Santos, R. P. (2025). GenIA-E2ETest: A Generative AI-Based Approach for End-to-End Test Automation. Anais do XXXIX Simpósio Brasileiro de Engenharia de Software, 123-134. https://doi.org/10.5753/sbes.2025.9927

6. Tao, L., Wang, H., Chen, J., & Liu, Y. (2024). A Survey on Web Application Testing: A Decade of Evolution. arXiv preprint arXiv:2412.10476. https://doi.org/10.48550/arxiv.2412.10476

7. Patel, S., & Gupta, M. (2024). AI and Machine Learning in Automated Test Case Generation: Emerging Trends. Neliti Publications. https://www.neliti.com/publications/603898/

8. Khaliq, Z., Rana, R., Ahsan, M., & Lali, M. I. (2022). A deep learning-based automated framework for functional UI testing. Information and Software Technology, 150, 106969. https://doi.org/10.1016/j.infsof.2022.106969

9. Leotta, M., Stocco, A., Ricca, F., & Tonella, P. (2024). An empirical study to compare three web test automation approaches. Software Testing, Verification and Reliability, 34(3), e2606. https://doi.org/10.1002/smr.2606

10. Alian, P., Behnamghader, P., & Halfond, W. G. J. (2024). A Feature-Based Approach to Generating Comprehensive End-to-End Tests. arXiv preprint arXiv:2408.01894. https://doi.org/10.48550/arxiv.2408.01894

11. Celik, A., & Mahmoud, A. (2025). Large Language Models for Test Case Generation: A Systematic Review. Machine Learning and Knowledge Extraction, 7(3), 1847-1872. https://doi.org/10.3390/make7030097

12. Deng, Y., Xia, C. S., Peng, H., Yang, C., & Zhang, L. (2023). Large Language Models are Zero-Shot Fuzzers. Proceedings of the 32nd ACM SIGSOFT ISSTA, 423-435. https://doi.org/10.1145/3597926.3598067

13. Schafer, M., Nadi, S., Eghbali, A., & Tip, F. (2023). An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. IEEE Transactions on Software Engineering, 49(4), 1411-1427. https://doi.org/10.1109/TSE.2023.3234726

14. Pan, R., et al. (2024). Understanding the Effectiveness of Large Language Models in Detecting Security Vulnerabilities. arXiv:2311.16169. https://doi.org/10.48550/arxiv.2311.16169

15. Siddiq, M. L., & Santos, J. C. S. (2023). SecurityEval Dataset: Mining Vulnerability Examples. Proceedings of MSR 2023, 273-277. https://doi.org/10.1109/MSR59073.2023.00050

16. Prenner, J. A., & Robbes, R. (2021). Automatic Program Repair with OpenAI's Codex: Evaluating QuixBugs. arXiv:2111.03922. https://doi.org/10.48550/arxiv.2111.03922

17. Lemieux, C., Inala, J. P., Lahiri, S. K., & Sen, S. (2023). Codamosa: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. Proceedings of the 45th ICSE, 919-931. https://doi.org/10.1109/ICSE48619.2023.00085

18. Dinella, E., Ryan, G., Mytkowicz, T., & Lahiri, S. K. (2023). TOGA: A Neural Method for Test Oracle Generation. Proceedings of the 45th ICSE, 1039-1051. https://doi.org/10.1109/ICSE48619.2023.00095

19. Wang, J., Huang, Y., Chen, C., Liu, Z., Wang, S., & Wang, Q. (2024). Software Testing with Large Language Models: Survey, Landscape, and Vision. IEEE Transactions on Software Engineering, 50(3), 555-580. https://doi.org/10.1109/TSE.2024.3368208

20. Chen, C., Bansal, C., Wang, S., Zimmermann, T., & Huang, Y. (2023). AdbGPT: Debugging Automation via Large Language Models. arXiv:2306.04138. https://doi.org/10.48550/arxiv.2306.04138

The article has been sent to the editors 13.05.26.
After processing 25.05.26.
Submitted for printing 30.06.26