2026
IPACT
CONFERENCE

www.ipact.kr

# 국내학술대회 논문집

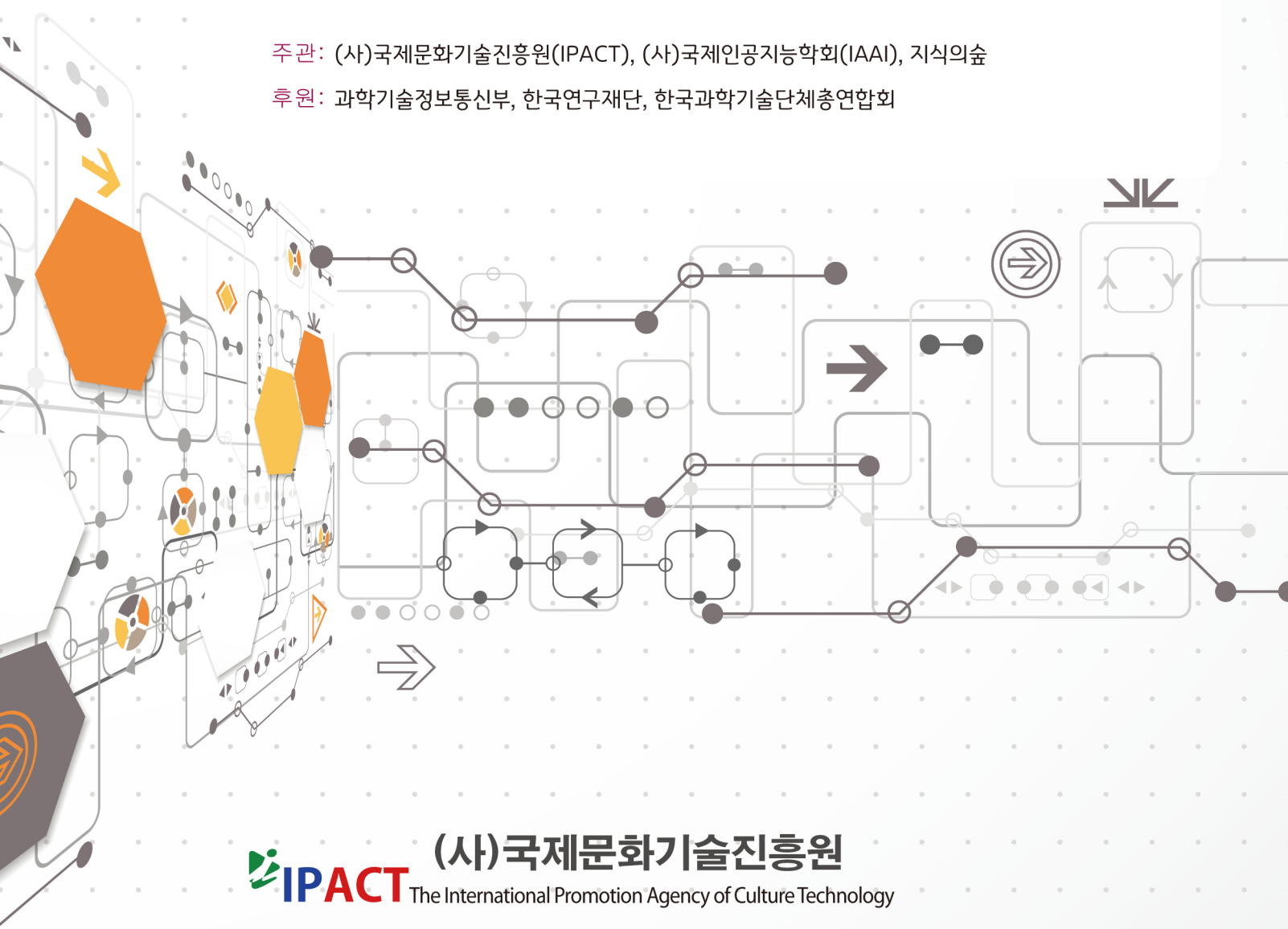
제10권 제1호

일시 | 2026.6.26(금)-27(토)

장소 | 한국과학기술회관

주관: (사)국제문화기술진흥원(IPACT), (사)국제인공지능학회(IAAI), 지식의숲

후원: 과학기술정보통신부, 한국연구재단, 한국과학기술단체총연합회



(사)국제문화기술진흥원
IPACT The International Promotion Agency of Culture Technology

# LLM & RAG 기반 저전력 파이썬 코드 최적화

## Low Power Consumption Python Code Optimization based on LLM & RAG

김재호*, 정은진**, 김영철***

Jaeho Kim*, Eunjin Jeong**, R. Young Chul Kim***

{jaehokim1005, jej031020}@g,hongik.ac.kr, bob123@hongik.ac.kr

요 약

AI 소프트웨어의 규모가 확대되고 GPU 기반 AI 생성 코드가 증가하면서, 전력 에너지 낭비가 중요한 문제로 대두되고 있다. 특히 대부분 AI 중점 Python 코드에 내재된 에너지 스멜(energy smell)은 반복 실행 환경에서 누적되는 에너지 낭비를 초래한다. 이를 해결하기 위해, LLM and RAG 기반 저전력 코드 최적화 방법을 제안한다. 이를 위해 IBM CodeNet에서 파생된 Pie-perf Python 코드 300 쌍에 에너지 소비 최소화의 시스템 프롬프트 설계 및 정의 한다. 이런 프롬프트 기반 저전력 코드 생성 및 RAPL 기반 저전력 측정한다. 이 실험 결과, 기존 코드 대비 60.2%의 에너지 절감이 가능했다. 이러한 결과는 LLM의 에너지 스멜 개선 및 저전력 코드 최적화가 기대된다. 앞으로 단순한 RAG 적용을 넘어 Context Engineering과 같은 공학적 설계를 통해 저전력 코드 최적화 연구를 확장할 수 있을 것이다.

키워드 : Green Software Engineering, Energy-Efficiency, Energy Smells

## I. 서론

소프트웨어에서 단일 함수의 실행 효율성은 거시적인 소프트웨어의 전력 소비 및 탄소 배출과 직결된다. 이러한 환경에서 데이터 사이언스의 표준으로 사용되고 백엔드 개발의 일부로 자리 잡은 Python은 높은 생산성을 제공하지만, 인터프리터 언어 특성 상 Energy Smell을 내포하고 있다 [1]. Python 코드 내에 존재하는 비효율적인 반복문, 자료구조의 선택 등은 여러 차례 반복 호출됨에 따라 막대한 에너지 낭비를 초래할 수 있다.

Green Software는 이러한 코드 수준의 낭비를 직접적으로 논하는 분야이고, 코드 성능 최적화 연구는 오래된 연구 문제이다 [2]. 그러나 기존의 코드 최적화 연구는 주로 실행 시간을 단축하는 성능 개선에 집중되었다. 또한 인간 개발자는 특정 코드 패턴이 CPU의 전력 상태와 같은 하드웨어에 미칠 영향을 인지하고 최적화하기 어렵다. 이런 어려움을 해결하기 위한 접근으로, Gottschalk et al.은 2012년 Fowler의 code smell 개념을 전력 소비 영역에 적용하여 'Energy Code Smell'이라는 용어를 처음 도입하였고 [3], 이어 2013년 Antonio Vetrò et al.은 임베디드 시스템에서 전력 소비를 실측하여 소스 코드 수정으로 전력 소모를 줄일 수 있는 코드 패턴들을 Energy Code Smell로 정의하였다 [4]. 그러나 주로 Energy Smell을 정의·탐지하거나 규칙 기반 리팩토링으로 제거하는 데 머물러, 실제 코드에 존재하는 다양한 비효율 패턴을 자동으로 수정하는 데에는 한계가 있었다.

한편 최근 대규모 언어 모델(LLM)을 활용한 코드 최적화와 리팩토링 연구가 활발하지만, 이들 연구와 Performance-Improving Edits(PIE)와 같은 대표적 데이터셋은 대부분 실행 시간 단축에 초점을 맞추고 있어 에너지 효율 자체는 충분히 다루어지지 않았다. 본 연구는 LLM을 사용하여 Python 코드에 내재된 에너지 비효율 패턴을 수정하는 코드 최적화 방법을 제안한다. 이를 위해 대규모 데이터셋인 IBM CodeNet과 그로부터 파생된 PIE Dataset에 Mehditabar et al.에 의해 제안된 Energy Smell 분류 체계의 계보를 잇되 [1], 실행 시간이 아닌 에너지를 직접 재측정하여 신뢰성을 확보하였다. 본 연구는 두 가지 핵심 요구사항들은 아래에 언급한다.

RQ1. LLM이 생성한 코드(v2)는 원본 코드(v0) 및 인간이 개선한 코드(v1)보다 더 에너지 효율적인가?

RQ2. 검색 증강 코드 생성(Retrieval Augmented Code

*홍익대학교 소프트웨어융합학과 석사과정
**홍익대학교 소프트웨어융합학과 학부생
***홍익대학교 소프트웨어융합학과 교수

Generation, RAG)을 통한 코드 샘플 참조가 독립적인 LLM과 비교하여 유의미하게 다른 결과를 보이는가?

또한 우리는 평가의 신뢰성 문제를 해결하기 위해 자체 하드웨어에서 에너지를 재측정하였다. 이를 통해 인간이 작성한 원본 코드 v0, 인간이 개선한 코드 v1, LLM으로 최적화한 코드 v2, RAG로 최적화한 코드 v3에 대하여 전력과 실행 시간을 측정하고 비교 분석한다.

실험 결과, v2는 v0 대비 88.0%의 쌍에서 에너지를 절감하고 중앙값 기준 60.2%의 에너지를 줄이는 것으로 나타났다. 또한 v2는 인간이 개선한 v1보다 낮은 에너지를 기록한 비율이 69.2%로 나타났다. RAG를 사용한 v3는 v2와 유의미한 차이를 보이지 않았으며, 오히려 41.0%의 쌍에서 v2보다 평균 239.5% 높은 에너지를 소비하는 역효과가 관찰되었다.

다시 요약하면, 첫째, Python 코드에 에너지 비효율적 패턴이 내재화되어 있음을 케이스 스터디를 통해 실증하였다. 둘째, LLM과 검색 증강 코드 생성을 활용하여 기능적 동등성을 보존하면서 에너지 효율을 개선하도록 코드를 생성하였고, 이를 통해 네 가지 경우의 코드에 대해 에너지 효율을 비교 분석하였다. 셋째, 에너지 절감을 전력 감소와 시간 단축으로 나누어 기여도를 분석하였다. 넷째, RAG 생성 코드를 분석하여 실패 원인을 규명하였다.

본 논문의 구성은 다음과 같다. 2절에서 관련 연구를 살펴본 뒤, 3절에서 Python 코드의 에너지 비효율을 실증하기 위한 케이스 스터디를 논한다. 이어 4절에서 제안하는 코드 최적화 방법을 기술한 뒤, 5절에서 실험 설계를 기술하고, 6절에서 실험 결과를 보고한다. 마지막으로 7절에서 결론 및 향후 연구 방향을 제시한다.

## II. 관련연구

### 2.1 Bad Smell과 Energy Smell

Bad Smell은 기능적 정확성에는 문제가 없으나 설계・유지보수 측면에서 잠재적 결함을 내포하는 코드 패턴이다 [5]. Energy Smell은 이 개념을 에너지 효율로 확장한 것으로, 기능적으로 동등하면서도 불필요하게 높은 에너지를 소비하는 코드 패턴을 가리킨다. Cruz and Abreu는 Android 애플리케이션을 대상으로 에너지 관련 코드 패턴을 체계적으로 분류하며 이 분야의 초기 연구 기반을 마련하였고 [6], Hasan et al.은 Java 컬렉션 클래스별 에너지 프로파일을 측정하여 동일 기능의 자료구조 선택이 에너지 소비에 유의미한 차이를 유발함을 실증하였다 [7].

Python 특화 연구는 최근 활성화되었다. Mehditabar et al. [1]은 IBM CodeNet에서 파생된 Pie-Perf의 Python 코드 쌍에 에너지・시간・메모리를 실측한 데이터셋을 공개하고, 비효율 반복문・불필요한 자료 변환・사용되지 않는 임포트 등을 포함하는 분류 체계를 제안하였다. 본 연구는 실측 데이터셋을 준거로 삼아 LLM이 Python의 에너지 비효율 패턴을 자동으로 개선할 수 있는지를 검증한다.

### 2.2 LLM 기반 코드 최적화 연구

Shypula et al. [2]은 경쟁 프로그래밍 플랫폼에서 수집한 C++ 코드 77,000여 쌍으로 PIE(Performance-Improving Edits) 데이터셋을 구축하였다. PIE는 실행 시간 단축을 최적화 기준으로 삼으며, Mehditabar et al. [1]은 그 Python 분할인 Pie-Perf에 에너지 소비를 추가 측정하여 공개하였다. 본 연구는 이 데이터셋을 평가 기반으로 활용한다.

LLM을 에너지 효율 코드 생성에 직접 적용한 연구는 공통적으로 부정적 결과를 보고한다. Cappendijk et al.은 5개 LLM을 대상으로 프롬프트 방식이 에너지 소비에 미치는 영향을 탐구하였으나, 어떤 프롬프트 전략도 LLM과 문제 유형에 걸쳐 일관된 절감을 달성하지 못하였다 [8]. Apsan et al.은 6개 LLM 생성 코드가 Green Software 전문가 코드 대비 17~30% 높은 에너지를 소비함을 확인하였으며 [9], Islam et al.은 20개 LLM 평가에서 인간 작성 코드가 최상위 LLM인 DeepSeek-v3보다 1.17배 효율적이고 특정 알고리즘 유형에서는 격차가 450배에 달함을 보였다 [10]. Solovyeva et al.과 Vartziotis et al.도 언어・모델에 걸쳐 LLM 코드가 인간 코드의 에너지 효율에 미치지 못함을 보고하였다 [11, 12]. 한편 Wu et al.이 제안한 FasterPy는 RAG와 LoRA 파인튜닝을 결합하여 Python 실행 효율을 개선하였으나 [13], 최적화 대상이 에너지가 아닌 실행 시간이라는 점에서 본 연구와 구별된다.

앞서 설명한 연구들은 대부분 Code Llama・DeepSeek-sCoder 등 2022~2023년 소형 모델을 평가 대상으로 삼았으며, 에너지 절감을 단일 수치로 보고할 뿐 전력 감소와 시간 단축이라는 두 메커니즘의 기여도를 분리하여 분석하지 않았다. 본 연구는 2025년 출시된 GPT-5.1-mini를 300쌍의 대규모 평가에 적용하고, 에너지 절감을 전력과 시간으로 분해하는 정량 분석을 수행한다.

### 2.3 소프트웨어 에너지 측정 방법론

소프트웨어 에너지 측정 방법론의 신뢰성은 실험 결과의 타당성과 직결된다. Khan et al.은 Intel RAPL(Running Average Power Limit) 인터페이스를 통한 소프트웨어 기반

에너지 측정의 정확도를 검증하였다 [14]. 스트림 벤치마크 실험에서 RAPL 패키지 전력과 벽면 소켓 전체 시스템 전력 간 상관계수가 0.99에 달함을 확인하여, RAPL이 CPU 패키지 수준의 에너지를 높은 정확도로 측정함을 입증하였다.

## III. Python 코드의 에너지 비효율 케이스 스터디

본 절은 Python 코드에서 에너지 비효율 패턴이 실재함을 실험적으로 검증한다. 에너지 스멜이 측정 가능한 차이를 만들어내는지 확인하기 위해 RAPL 기반 통제 측정 환경을 구성하였다. 모든 실험에서 Turbo Boost 비활성화, CPU governor를 performance로 고정, SMT 형제 코어 오프라인, taskset으로 단일 코어 고정, GC 비활성화, 5회 반복 측정의 동일한 조건을 적용하였다.

Case 1. for문 순회 방향에 따른 전력 비교

```
import sys

N = 10000
mode = sys.argv[1] if len(sys.argv) > 1 else "fwd"
repeats = int(sys.argv[2]) if len(sys.argv) > 2 else 5000

s = 0
if mode == "fwd":
    for _ in range(repeats):
        for i in range(1, N + 1):
            s += i
else:
    for _ in range(repeats):
        for i in range(N, 0, -1):
            s += i

print(f"mode={mode} repeats={repeats} sum={s}")
```

그림 1. for문 순회 방향 비교 코드

| 지표 | FWD | BWD | 차이 |
|---|---|---|---|
| Cycles | 13,086,538,158 | 11,347,950,802 | −13.3% |
| Instructions | 62,921,946,802 | 54,053,650,678 | −14.1% |
| IPC | 5.59 | 4.70 | −16.0% |
| Stalled-frontend | 4,536,838 (0.04%) | 6,650,147 (0.06%) | +46.6% |
| Stalled-backend | 40,456,205 (0.36%) | 40,204,658 (0.35%) | −0.6% |
| Branch-misses | 4,626,986 | 4,769,887 | +3.1% |
| Cache-misses | 2,436,399 | 2,485,092 | +2.0% |
| 실행 시간 (s) | 2.832 ± 4.22% | 2.897 ± 2.05% | +2.3% |
| 에너지 raw (J) | 194.03 ± 1.40% | 191.46 ± 1.65% | −1.3% |
| 순 에너지 (J) | 32.1 | 29.0 | −9.7% |

표 1. for문 순회 방향별 하드웨어 성능 및 에너지 측정 결과

이 케이스 스터디는 두 가지 사실을 보여 준다. 하나는 Python 코드에서 자료구조와 반복문 구성의 선택이 에너지 소비에 실질적인 영향을 미친다는 것이고, 다른 하나는 이러한 차이가 개발자에게 직관적으로 명확하지 않다는 것이다. 4절에서는 LLM으로 이러한 비효율 패턴과 에너지 스멜을 최적화하는 방법을 제안한다.

## IV. LLM & RAG 기반 저전력 코드 최적화

본 절에서는 제안하는 LLM & RAG 기반 코드 최적화에 초점을 두어 설명한다. 데이터셋부터 시스템 프롬프트 등의 하이퍼파라미터를 설명한다. 또한 RAG의 검색기를 어떻게 구성하였는지 상세히 설명한다.

### 4.1 데이터셋

본 연구는 IBM CodeNet [15]과 PIE [2] 데이터셋의 교집합에서 추출한 알고리즘 문제 풀이 Python 코드 쌍을 기반으로 한다. PIE 데이터셋은 동일 알고리즘 문제에 대해 원본 코드(v0)와 자체 개선한 코드(v1)를 쌍으로 제공하며, 문제별 테스트 케이스를 포함하여 기능적 동등성을 검증할 수 있다. 전체 3,000쌍 중 테스트 케이스가 존재하는 쌍을 대상으로 시드 42를 고정한 균등 무작위 표본 추출로 300쌍을 선정하였다.

### 4.2 LLM

LLM은 OpenAI의 GPT-5.1-mini를 선정하였다. 하이퍼파라미터로 max_completion_tokens는 16,000으로 설정하였다. temperature 및 reasoning_effort 파라미터는 별도로 지정하지 않아 OpenAI API의 기본값이 적용되었다. 시스템 프롬프트는 기능적 동등성 보존을 최우선 조건으로 명시하고, 에너지 소비 최소화를 목표로 설정하며, 단순한 실행 시간 단축과 에너지 효율을 구분하여 아래와 같이 지시하였다.

```
You are a code optimization engine. You receive one program (the "v0" program) and
produce a modified version that consumes LESS ENERGY while preserving identical behavior.

PRIORITY RULES (in order):
```

1. Behavior preservation is mandatory and overrides everything else. The patched program
MUST produce identical output for identical input, in the SAME programming language as
the input. Never alter observable semantics. Preserve the exact input parsing and output
format (spacing, newlines, rounding, edge cases).
2. Optimize for lower ENERGY consumption, not merely shorter wall-clock time.
3. If reference examples are provided in the user message, treat them ONLY as
transformation patterns: identify the technique and apply it to the target program
where applicable. Do NOT copy them verbatim. If no reference examples are provided,
optimize using your own knowledge.

You may think step by step first. Then output the FINAL answer as the COMPLETE, runnable
program inside a single ```python code block. The code block must contain the entire program
(not a diff, not a snippet). If you find no safe energy improvement, output the original
program unchanged rather than risk changing behavior. Put the code block LAST.

그림 2. GPT-5.1-mini 시스템 프롬프트

4.3 RAG

(v0, v1) 코드 쌍 전부를 OpenAI의 text-embedding-3-large 모델로 임베딩하여 FAISS 인덱스를 구축하였다. 입력 v0에 대하여 코사인 유사도 기준 상위 3개의 코드 쌍을 검색하여 원본 입력에 Few-shot으로 삽입한다. 시스템 프롬프트는 그림 2와 동일하며, problem_id를 기준으로 동일한 문제의 개선 코드는 검색에서 제외하여 데이터 누수를 방지했다.

## Ⅴ. 실험 설정

5.1 에너지 산출 공식과 측정 도구

본 연구는 각 완성된 Python 프로그램을 실행하는 전체 에너지를 측정 대상으로 삼는다. CPU 패키지는 유휴 상태에서도 전력을 소비하므로 코드가 유발하는 에너지 소비만을 분리하여 분석한다.

에너지 측정 도구로는 CPU 내장 전력 추정 인터페이스인 Running Average Power Limit(RAPL)을 채택한다. RAPL의 energy-pkg 카운터는 MSR_PKG_ENERGY_STATUS 레지스터에 CPU 패키지 전체의 에너지를 약 15.3 μJ 단위로 연속 누적하는 하드웨어 누산기로, power/energy-pkg/ 명령을 통해 측정 구간의 시작·종료 시점에 카운터를 각각 읽어 차이를 에너지량으로 환산한다 [14]. energy-pkg는 CPU 전체의 에너지를 측정하므로 운영체제, 유휴 코어의 누설 전류 등의 배경 부하가 포함된다. 이를 분리하기 위해 측정 이전에 1초 간의 유휴 구간에서 RAPL 값을 읽어 에너지를 산출하고 아래와 같은 수식에 대입하였다.

$$E_D = E_T - (P_S \times T_E) \quad (1)$$

$E_T$는 코드 실행 구간 동안 RAPL이 누산한 전체 에너지, $T_E$는 해당 구간의 실제 '시간, $P_S$는 유휴 시 전력이다. 전체 에너지 $E_T$에서 코드가 실행되지 않아도 소비되는 에너지인 $P_S \times T_E$를 빼면 코드가 유발한 에너지인 $E_D$가 남는다.

측정 대상 코드는 실행 시간이 수십 ms 수준으로 짧아, 동일 코드를 N회 반복 실행하여 실행 1회당 에너지를 산출한다. 최종 산출 식은 아래와 같다.

$$e = \frac{E_D}{N} = \frac{E_T - (P_S \times T_E)}{N} \quad (2)$$

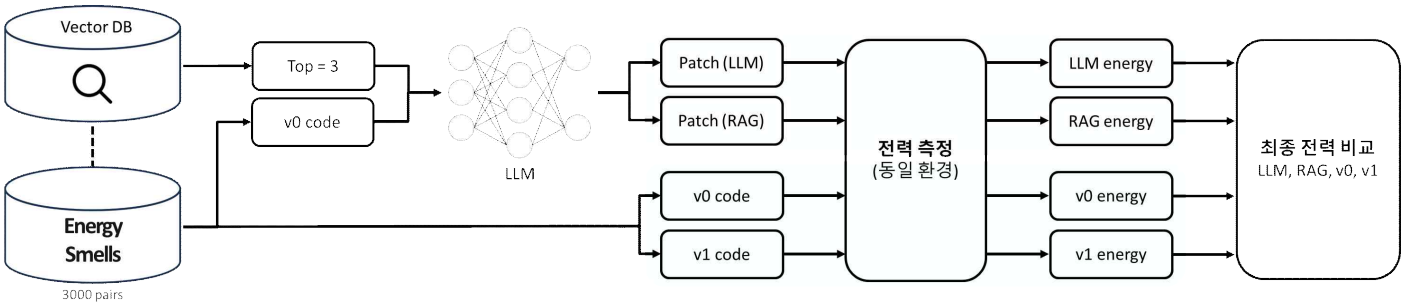



그림 3. LLM & RAG 기반 저전력 코드 최적화 전체 도면

5.2 하드웨어 조건 및 변인 통제

에너지 측정은 CPU 동작 상태에 영향을 받는다. 동일 코드라도 Turbo Boost나 DVFS에 의해 클럭 주파수가 변동하면 순 전력이 달라질 수 있다. 이를 방지하기 위해 표 2와 같이 다섯 가지 하드웨어 변인을 통제하였다. 또한 실험 환경의 주요 수치는 표 3에 요약하였다.

| 통제 항목 | 설정값 | 목적 |
|---|---|---|
| Turbo Boost | 비활성화 | 열 급등에 의한 주파수 급변 차단 |
| CPU Governor | performance | 동적 주파수 조절 방지 |
| 하이퍼스레딩 (SMT) | 형제 코어 offline | 캐시·실행 유닛 공유 간섭 제거 |
| 주파수 하한 | max값으로 고정 | 유휴 구간 클럭 하락 방지 |
| 시스템 부하 | loadavg ≤ 0.6 | 배경 프로세스 간섭 차단 |

표 2. 하드웨어 변인 통제

| 항목 | 값 |
|---|---|
| 측정 쌍 수 | 300 |
| LLM 모델 | GPT-5.1-mini |
| 임베딩 모델 | text-embedding-3-large |
| 벡터 DB / Top K | FAISS / 3 |
| 최소 측정 창 ($T_E$) | 5초 |
| 반복 횟수 | 3 |
| $P_S$ 평균 | 64.9 W |
| $p_w$ 평균 | 6.93 W |
| $\frac{p_w}{P_S}$ | 약 10.7% |

표 3. 실험 환경 요약

## VI. 실험 결과

6.1 정확성

에너지 절감이 의미를 갖기 이전에 최적화된 코드가 원본과 같이 동일한 출력을 생성해야 한다. 빠른 코드가 오답을 출력하거나 연산이 생략된 채로 프로그램의 실행이 끝나는 경우를 배제하기 위해, PIE에서 제공되는 문제 별 테스트 케이스로 각 변형의 정확성을 검증하였다. 표 4에서 보듯, LLM 생성 코드의 정확성은 인간 코드와 동등한 수준이며, 에너지 비교는 정확성 측면에서 통과된 쌍만을 대상으로 한다.

| Form | Pass / Total | Pass Rate |
|---|---|---|
| v0 (original) | 293 / 300 | 97.7% |
| v1 (human) | 291 / 300 | 97.0% |
| v2 (LLM) | 290 / 300 | 96.7% |
| v3 (RAG) | 287 / 300 | 95.7% |

표 4. 정확성 검증 결과

6.2 에너지 및 시간 절감

본 절의 비교 지표는 식 (1), (2)를 통해 산출한 실행 1회당 에너지 $e$이다. 에너지 절감은 알고리즘 복잡도 감소의 경우 크게 변화할 수 있기 때문에 산술 평균이 아닌 중앙값을 주요 통계량으로 채택했다. 표 5에 나타나있듯, v2는 283쌍 중 249쌍(88.0%)에서 원본 v0보다 낮은 에너지를 기록하였다. v0 대비 중앙값 절감률은 60.2%로, 인간 개선 코드(v1)의 절감률인 35.3%를 상회한다.

Code Llama, DeepSeek-Coder 등의 소형 모델과 오픈소스 모델을 평가한 선행 연구들이 LLM 생성 코드가 인간 코드의 에너지 효율을 상회하지 못한다고 보고한 것과 달리, 본 연구가 사용한 GPT-5.1-mini는 인간 개선을 상당 폭으로 능가한다. 이는 최신 모델이 에너지 최적화에 기여할 수 있음을 시사한다.

| 변형 | 쌍 수 | 승률 (v0 대비) | 절감률 중앙값 |
|---|---|---|---|
| v1 (Human) | 279 | 74.2% | 35.3% |
| v2 (LLM) | 283 | **88.0%** | **60.2%** |
| v3 (RAG) | 279 | 86.4% | 60.5% |

표 5. v0 대비 에너지 절감률 비교

6.3 전력, 시간 단축 비율

에너지 절감이 전력과 시간 중 어느 쪽에서 비롯되었는지 쌍 단위로 분석하였다. 정확성 검증 이후 두 값이 모두 유효한 쌍만을 분석하면, v2는 $T_S$를 중앙값 기준 54.9% 단축하는 동시에 $T_W$를 중앙값 기준 14.6% 낮추었다. 이는 에너지 절감이 속도 향상과 전력 소비 개선 모두를 동반한다는 것으로 해석된다.

| Metric | Median 비율 | 변화 |
|---|---|---|
| $P_W\,(P_{W_1}/P_{W_0})$ | 0.854 | 14.6% |
| $T_S\,(T_{S_1}/T_{S_0})$ | 0.451 | 54.9% |
| $E(E_1/E_0)$ | 0.398 | 60.2% |

표 6. 전력·시간 분해 분석 결과

6.4 RAG 효과 분석

v3(RAG)는 v2와 통계적으로 유의미한 차이를 보이지 않았으며, 41.0%의 쌍에서 v2보다 평균 239.5% 높은 에너지를 소비하였다. 이는 RAG가 에너지 최적화 지시보다 예시 패턴을 모방하는 것을 우선하면서 코드가 복잡해지는 방향으로 작용한 것으로 해석된다. 향후 단순 코드 샘플 삽입을 넘어 Context Engineering과 같은 설계 기법을 통해 RAG의 효과를 개선할 수 있을 것으로 기대된다.

## Ⅶ. 결론

본 연구는 먼저 케이스 스터디를 통해 Python 코드에 에너지 소비 관점에서의 비효율이 존재함을 실증하였다. 그 후, GPT-5.1-mini와 RAG를 이용하여 Python 코드의 저전력 코드 최적화 방법을 제안하고, RAPL 기반의 통제된 측정 환경에서 코드 300쌍을 평가하였다.

실험 결과, v2는 v0 대비 60.2%의 에너지를 절감하였고 300쌍 중 88%의 쌍에서 원본보다 낮은 에너지를 기록했다. 이는 인간 개선 코드인 v1의 절감률 35.3%에 비해 유의미하게 높은 수준이다. 전력과 시간을 나누어 분석했을 때, 72.4%의 쌍이 실질적인 전력 감소를 보여 단순 실행 시간 단축에 그치지 않음을 확인하였다.

향후 연구에서는 측정 순서 무작위화를 통해 열 편향을 제거하고 단일 Python 프로세스 내 in-process 마이크로 벤치마킹을 통한 인터프리터 오버헤드 분리를 수행할 계획이며, 다른 하드웨어 조건에서 교차 검증을 수행할 계획이다. 또한 RAG의 효과를 개선하기 위하여 단순 샘플 참조가 아닌 Context Engineering을 적용하고 AST 유사도에 기반한 검색 등의 고도화된 검색 전략의 탐구를 진행할 예정이다. 최종적으로는 알고리즘 문제 해결 도메인이 아닌 실제 소프트웨어에 적용하여 효과를 검증하는 것이 필요하다.

## 참 고 문 헌


[1] M. Mehditabar, S. Rajput, T. Sharma. “A Validated Taxonomy on Software Energy Smells.“ arXiv:2604.04809, 2026.

[2] A. Shypula et al. “Learning Performance-Improving Code Edits.“ In Proc. ICLR 2024, 2024.

[3] M. Gottschalk, M. Josefiok, J. Jelschen, A. Winter. “Removing energy code smells with reengineering services.“ In INFORMATIK 2012, Gesellschaft für Informatik (GI), 2012.

[4] A. Vetrò, L. Ardito, G. Procaccianti, M. Morisio. “Definition, implementation and validation of energy code smells: an exploratory study on an embedded system.“ In Proc. ENERGY 2013, 2013.

[5] M. Fowler. “Refactoring: Improving the Design of Existing Code.“ Addison-Wesley Professional, 1999.

[6] Luis Cruz, Rui Abreu, “Catalog of Energy Patterns for Mobile Applications.” Empirical Software Engineering, 24(4):2209–2235, 2019

[7] M. Hasan et al. “Energy profiles of Java collections classes.“ In Proc. ICSE ’16, ACM, 2016.

[8] T. Cappendijk, P. de Reus, A. Oprescu. “An Exploration of Prompting LLMs to Generate Energy-Efficient Code.“ arXiv:2411.10599, 2025.

[9] R. Apsan et al. “Generating Energy-Efficient Code via Large-Language Models – Where are we now?“ In Proc. ICSE ’26, ACM, 2026.

[10] M. A. Islam et al. “Evaluating the Energy-Efficiency of the Code Generated by LLMs.“ arXiv:2505.20324, 2025.

[11] L. Solovyeva, S. Weidmann, F. Castor. “AI-Powered, But Power-Hungry? Energy Efficiency of LLM-Generated Code.“ arXiv:2502.02412, 2025.

[12] T. Vartziotis et al. “Learn to Code Sustainably: An Empirical Study on Green Code Generation.“ In Proc. LLM4Code ’24, ACM, 2024.

[13] Y. Wu et al. “FasterPy: An LLM-based Code Execution Efficiency Optimization Framework.“ ACM Trans. Softw. Eng. Methodol., 2025.

[14] K. N. Khan, M. Hirki, T. Niemi, J. K. Nurminen, Z. Ou. “RAPL in Action: Experiences in Using RAPL for Power Measurements.“ ACM Trans. Model. Perform. Eval. Comput. Syst., 3(2):Article 9, 2018.

[15] R. Puri et al. “CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks.“ arXiv:2105.12655, 2021.