HOW THE TEMPERATURE SWEEP CAN HELP DESIGN COMPILER DIAGNOSTICS FOR AI CODING AGENTS

By

Akash Deo

Thesis Project
Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

June 2026

Simone Campanoni, First Reader

Christos Dimoulas, Second Reader

Technical Report Number: NU-CS-2026-29

ABSTRACT

Compiler diagnostics are crucial in coding agent workflows. However, there are no systematic approaches to evaluate compiler diagnostics for AI coding agents. This thesis proposes the CANARY method: a temperature sweep which leverages the power of language models to evaluate compiler diagnostics for AI coding agents automatically. CANARY evaluates compiler diagnostics on two axes: descriptiveness (explaining what is wrong) and prescriptiveness (how to fix it). I apply CANARY to compiler auto-vectorization diagnostics. First, I show that coding agents succeed 2.8x more often when they have compiler auto-vectorization diagnostics. Second, I use data from CANARY experiments to improve auto-vectorization compiler diagnostics in Clang. This thesis shows that it is possible to evaluate compiler diagnostics for AI coding agents automatically, paving the way for language-model-based approaches to improving them.

ACKNOWLEDGEMENTS

I have so many people to thank. I want to thank Tommy McMichen. He took me under his wing and taught me basically everything that I know about a compiler and how to research. Thank you for everything over this past year. I must thank Christos Dimoulas for introducing me to research. I will never forget the kindness he showed me by offering to teach me over the summer as a rising junior at Northwestern. Without Simone, I wouldn't have the opportunity to write this thesis.

I must thank several others on the third floor of Mudd: Federico Sossai, Yian Su, Haocheng Gao, Kaytlin Harrison, Yuchen Zhu, Leyla Latifova, Benjamin Ye, Nick Wanninger, Karl Hallsby, Alex Butler, Friedrich Doku, David Krasowska, Kirill Nagaitsev, Liam Strand, and more.

The number of times I talked to Akhil Deo about how to serve a small language model was incredible. Lastly, I am so deeply grateful for my parents. Thank you Rema and Sanjay Deo, for your love and support.

To my family.

TABLE OF CONTENTS

Acknowledgments 2

List of Figures 3

Chapter 1: Introduction and Background 5

Chapter 2: AI Coding Agents Need Better Compiler Diagnostics 9

2.1 Prior Work 9

2.2 Compiler Diagnostics Make a Significant Difference 9

Chapter 3: Descriptiveness and prescriptiveness by example 12

3.1 Descriptiveness and prescriptiveness 12

3.2 Example: UnsafeDep Diagnostic 12

3.3 Example: CantReorderFPOps Diagnostic 13

3.4 Example: NonReductionValueUsedOutsideLoop Diagnostic 14

3.5 Example: WritesInEarlyExit Diagnostic 15

Chapter 4: The CANARY Method 16

4.1 What does temperature have to do with descriptiveness and prescriptiveness? . . . 16

4.2 CANARY Temperature Sweeps 17

4.3 Applying CANARY to compiler auto-vectorization diagnostics 18

4.4 Applying CANARY to the TSVC benchmark to Analyze Compiler Auto-Vectorization Diagnostics . . . 19

Chapter 5: Improving Compiler Auto-Vectorization Diagnostics with CANARY 23

5.1 Improving Compiler Auto-Vectorization Diagnostics using the Canary Method . . . 23

5.2 Evaluating Improved Diagnostic . . . 24

Chapter 6: Discussion . . . 27

6.1 Discussing Limits of the CANARY Method . . . 27

6.1.1 Descriptiveness Flag . . . 27

6.1.2 CANARY is expensive . . . 27

6.2 Next Steps for the CANARY method . . . 28

Chapter 7: Related Work . . . 29

7.1 Evaluating Compiler Diagnostics . . . 29

7.2 Enhancing AI with Compilers . . . 29

7.3 Enhancing Compilers with AI . . . 30

Chapter 8: Conclusion . . . 31

References . . . 33

Appendix A: CANARY Prompts . . . 1

LIST OF FIGURES

1.1 The `vtvtv` TSVC kernel & the corresponding Clang vectorization remark. Clang automatically vectorizes the loop because no iterations touch overlapping memory locations. Iteration i only touches a[i], b[i], and c[i]. Iteration i+1 only touches a[i+1], b[i+1], and c[i+1], so on and so forth. 6

1.2 The `s212` TSVC kernel. There is a data dependence between the read from `a[i+1]` in the ith iteration and the write to `a[i]` in the $i + 1$th iteration. An AI coding agent could break this dependence by making a temporary copy of `a[i+1]` before Line 8. 7

1.3 The `s212` TSVC kernel with a temporary copy of `a[i + 1]` on Line 8. This breaks the data dependence in Figure 1.2. 7

2.1 The s291 C kernel in TSVC and the corresponding Clang analysis remark explaining why it was not vectorized. 11

3.1 The `s241` TSVC kernel & the corresponding Clang vectorization remark. 13

3.2 The repaired `s241` TSVC kernel. 14

3.3 The `vsumr` TSVC kernel & the corresponding Clang vectorization remark. 14

3.4 The `s481` TSVC kernel & the corresponding Clang vectorization remark. 15

3.5 The repaired `s481` TSVC kernel. 15

4.1 A temperature sweep. In the ablation study, an agent is given a C program (and sometimes a diagnostic), and the agent can either succeed or fail. This workflow is run 100 times. We count successes with the diagnostic and successes without the diagnostic. A success rate delta is computed by taking the difference between the number of successes with the diagnostic and without. Using this information, we compute a prescriptiveness score and a descriptiveness flag. 17

4.2 One trial in a temperature sweep. An agent is given some C program, and optionally some diagnostic information. It is supposed to generate a new draft of the program. If the new draft emits a diagnostic indicating the auto-vectorizer succeeded, then the agent succeeded. 18

4.3 CANARY results for compiler auto-vectorization diagnostics on TSVC. The graph shows the success-rate delta (with vs. without diagnostic) across temperatures; the table summarizes the prescriptiveness score and descriptiveness flag for each diagnostic. 19

4.4 The Clang CantIdentifyArrayBounds diagnostic. 19

4.5 The Clang CantReorderFPOps diagnostic. 20

4.6 The Clang NonReductionValueUsedOutsideLoop diagnostic. 20

4.7 The Clang UnsafeDep diagnostic. 21

4.8 The Clang UnsupportedUncountableLoop diagnostic. 21

4.9 The Clang WritesInEarlyExit diagnostic. 22

5.1 This describes how to improve a diagnostic. Diagnostic 0 has an actively misleading prescription. Removing the prescription transforms Diagnostic 0 into Diagnostic 1. Here, Diagnostic 1 is not prescriptive since it does not have a high success rate delta at low temperature and it is not descriptive either since it has a very flat slope. Diagnostic 2 is more descriptive because it has a steeper slope. Diagnostic 3 is highly prescriptive because it has a high success rate delta at a low temperature. A compiler engineer should have three goals: first, fix negative prescriptions (Diagnostic 0 to Diagnostic 1). Second, take diagnostics that lack description and prescription, and make them more descriptive (Diagnostic 1 to Diagnostic 2). Third, take diagnostics that are descriptive, and translate its descriptive messaging into prescriptive messaging (Diagnostic 2 to Diagnostic 3). 25

5.2 CANARY diagram after improving the `UnsafeDep` diagnostic. We observe a drastic improvement in the prescriptiveness score of the `UnsafeDep` diagnostic. . . . 26

CHAPTER 1
INTRODUCTION AND BACKGROUND

Modern AI coding agents operate in tight feedback loops with their tools. Prior work shows that tool calling is essential for AI agents' task performance [1, 2]. Based on this fact, coding agent harnesses like Claude Code, Codex, and Copilot expose tools for large language models to compile code, lint programs, run tests, or execute arbitrary shell commands. This feedback loop between AI coding agents and tools is indispensable for writing code [2].

Coding agents use compilers specifically to get immediate feedback on what code they generated. Prior work for code generation puts compiler feedback in iterative agent workflows to improve LLM-generated code [3, 4, 5]. Compiler feedback is crucial. It grounds inherently stochastic language models in deterministic static analysis, preventing them from indulging in semantic hallucinations or prematurely declaring success in tasks [6].

Even though coding agents will continue consuming more and more compiler diagnostics while generating code, we lack a method to evaluate the quality of compiler diagnostics in the context of AI coding agents. Prior works like Barik et al. [7] and Becker et al. [8] provide frameworks to evaluate compiler diagnostics for human software engineers which might not work for AI coding agents. Whether a diagnostic has a complete argument structure [7], or is concise and uses simple language [8] might not matter to a coding agent that behaves in a fundamentally different way from a human software engineer. First, the way a diagnostic is worded does not relate to whether the diagnostic is correct or incorrect. Second, whether a diagnostic is readable or concise for humans might not be relevant for an AI agent. After all, AI agents are different from human engineers. They are capable of quickly parsing vast amounts of diagnostic text in a fraction of the time it would take a human in some situations, but make surprisingly basic mistakes in others.

To fill this gap, this thesis introduces the CANARY method. CANARY is a 'temperature sweep': it uses the behavior of language models at different temperatures to inform diagnostic design for

AI coding agents. Models that are trained for instruction-following listen to suggestions in the prompt very closely at low temperatures, whereas models at higher temperatures are more 'creative', deriving necessary actions from the problem description provided by a diagnostic.

I apply CANARY to automatic vectorization diagnostics. The auto-vectorizer is an optimization pass in a compiler that takes independent iterations in a loop that use scalar instructions and replaces them with vector instructions that process multiple iterations simultaneously using specialized hardware. The challenge lies in transforming loops written in C to fit patterns that compilers can automatically vectorize. For instance, the compiler can automatically vectorize the code snippet in Figure 1.1 because no iterations in the loop touch overlapping memory locations. In this case, the compiler will emit an auto-vectorization diagnostic indicating that it successfully vectorized the loop. However, there are more difficult cases, like in Figure 1.2, where there is a data dependence between the read from `a[i+1]` in the ith iteration and the write to `a[i]` in the $i + 1$th iteration. An AI coding agent could break this dependence by making a temporary copy of `a[i+1]` before Line 8, as depicted in Figure 1.3. Because of these challenges, automatic vectorization is frequently used in academic literature to evaluate coding agents in the context of performance engineering with compiler assistance [3, 6, 5, 4, 9, 10].

```
1 float a[LEN_1D];
2 float b[LEN_1D];
3 float c[LEN_1D];
4
5 float vtvtv() {
6   for (int i = 0; i < LEN_1D; i++)
7     a[i] = a[i] * b[i] * c[i];
8   return 0.;
9 }
```

```
1 Pass:     loop-vectorize
2 Name:     Vectorized
3 DebugLoc: File: tsvc.c, Line: 3344,
4           Column: 9
5 Function: vtvtv
6 Args:     vectorized loop
7           (vectorization width: 8,
8            interleaved count: 4)
```

Figure 1.1: The `vtvtv` TSVC kernel & the corresponding Clang vectorization remark. Clang automatically vectorizes the loop because no iterations touch overlapping memory locations. Iteration i only touches a[i], b[i], and c[i]. Iteration i+1 only touches a[i+1], b[i+1], and c[i+1], so on and so forth.

This thesis argues that:

```
1  float a[LEN_1D];
2  float b[LEN_1D];
3  float c[LEN_1D];
4  float d[LEN_1D];
5
6  float s212() {
7    for (int i = 0; i < LEN_1D - 1; i++) {
8      a[i] *= c[i];
9      b[i] += a[i + 1] * d[i];
10   }
11   return 0.;
12 }
```

```
1  Pass:        loop-vectorize
2  Name:        UnsafeDep
3  DebugLoc:    File: tsvc.c, Line: 838,
4               Column: 21
5  Function:    s212
6  Args:        loop not vectorized: unsafe
7               dependent memory operations
8               in loop. Use #pragma clang
9               loop distribute(enable) to
10              allow loop distribution to
11              attempt to isolate the
12              offending operations into a
13              separate loop' Backward loop
14              carried data dependence."
15              Memory location is the same
16              as accessed at tsvc.c:837:13.
```

Figure 1.2: The `s212` TSVC kernel. There is a data dependence between the read from `a[i+1]` in the ith iteration and the write to `a[i]` in the $i+1$th iteration. An AI coding agent could break this dependence by making a temporary copy of `a[i+1]` before Line 8.

```
1  float a[LEN_1D];
2  float b[LEN_1D];
3  float c[LEN_1D];
4  float d[LEN_1D];
5
6  float s212() {
7    for (int i = 0; i < LEN_1D - 1; i++) {
8      float temp_a = a[i + 1];
9      a[i] *= c[i];
10     b[i] += temp_a * d[i];
11   }
12   return 0.;
13 }
```

```
1  Pass:        loop-vectorize
2  Name:        Vectorized
3  DebugLoc:    File: tsvc.c, Line: 836,
4               Column: 9
5  Function:    s212
6  Args:        vectorized loop (vectorization
7               width: 8, interleaved count: 2)
```

Figure 1.3: The `s212` TSVC kernel with a temporary copy of `a[i + 1]` on Line 8. This breaks the data dependence in Figure 1.2.

> The temperature sweep provides a signal for the quality of compiler auto-vectorization diagnostics, and diagnostics designed with this signal in mind can improve coding agent success rates on tasks to enable auto-vectorization.

This thesis requires answering two separate questions.

First, AI coding agents might not benefit from compiler auto-vectorization diagnostics. Perhaps the large language models are so powerful that they can enable auto-vectorization of a loop without a compiler's assistance. This raises the following research question:

Do coding agent success rates on enabling auto-vectorization significantly increase when coding agents have access to compiler auto-vectorization diagnostics?

I answer this question with a resounding yes in Chapter 2. I show that AI coding agents' task success rate shows a statistically significant improvement when they are given compiler diagnostics.

Second, I must determine whether it is possible to use the signal from a temperature sweep to improve compiler auto-vectorization diagnostics.

Does the temperature sweep provide a signal for evaluating the quality of compiler auto-vectorization diagnostics that can be used to design diagnostics to improve coding agent success rates on enabling auto-vectorization?

I apply the CANARY method to evaluate compiler auto-vectorization diagnostics. With this information, I improve an auto-vectorization diagnostic and show the improvement helps coding agents.

The remainder of this thesis is organized as follows. Chapter 2 explains the motivation for improving compiler diagnostics for AI coding agents. Chapter 3 describes what descriptiveness and prescriptiveness are by example. Chapter 4 then explains CANARY and evaluates existing compiler auto-vectorization diagnostics using the technique. Chapter 5 improves compiler auto-vectorization diagnostics using CANARY. Chapter 6 discusses the limits and the promises of CANARY. Chapter 7 discusses the method in relation to prior work. Chapter 8 concludes.

CHAPTER 2

AI CODING AGENTS NEED BETTER COMPILER DIAGNOSTICS

I begin by showing why improving compiler diagnostics helps AI coding agents succeed more often. First, I review some prior work that shows compiler diagnostics improve code generation. Second, I show that compiler diagnostics significantly increase how often AI coding agents succeed on automated performance engineering tasks.

2.1 Prior Work

Deo et al. [6] show that the bottleneck for AI coding agents' task performance is not model capability, but diagnostic quality. Their method evaluates an AI coding agent on transforming code to enable compiler auto-vectorization, and finds that compiler diagnostics improve agent performance on both Intel and Clang C compilers. The agent achieves a 3.3x success rate with Clang's diagnostics and 2.9x with Intel's compared to an agent without diagnostics.

However, the effects of individual diagnostics are not uniform. Each has a different effect and they vary across temperatures. For instance, Intel's 'Output Dependence' compiler diagnostic achieved a 26% improvement at $T = 0.8$ and Intel's 'Multiple Exits' compiler diagnostic achieved a 22% improvement at $T = 0.2$.

There are two limitations to prior work. First, it only uses an older model (Qwen2.5-Coder-7B). Second, it does not demonstrate that compiler diagnostics lead to a statistically significant performance difference. I build on prior work in the next section.

2.2 Compiler Diagnostics Make a Significant Difference

I extend these experiments to a newer model (Qwen3-Coder-30B-A3B) and show that compiler diagnostics do lead to a significant difference in the agent's performance.

Experiment setup: In one trial, I tell an AI coding agent to perform a source-to-source transformation of a C kernel in the TSVC benchmark [11]. There are 100 total trials. The agent is model Qwen3-Coder-30B-A3B set at temperature $T = 0.8$. The prompts used are in Appendix, Listing A.2. The compiler is Clang 21.1.8, with `-O3 -march=native -fsave-optimization-record`. [1] I run this experiment where the prompt has compiler auto-vectorization diagnostics, and where the prompt does not have diagnostics. The coding agent succeeds if the modified TSVC kernel triggers compiler automatic vectorization.

Quantitative results: Without diagnostics, the coding agent's overall success rate is **5.48%**. With diagnostics, the coding agent's success rate increases to **15.15%** (**2.8x success rate**). I apply a paired t-test in Table 2.1 to show a significant change in success rate. Each pair compares a function's success rate with and without a diagnostic. I find a p-value of $p = 0.0032$. I can claim that AI coding agents succeed significantly more often when compiler diagnostics are in the prompt.

Paired t-test	
Kernels (n)	62
Mean delta	$+0.0966$
t	3.07
p	0.0032

Sign test	
Diagnostic helps	10
Diagnostic hurts	4
No change	48
p (two-sided)	0.18

Table 2.1: A paired t-test and sign test. The paired t-test shows a statistically significant improvement from introducing compiler diagnostics, and the sign test shows the breakdown of gains from introducing compiler diagnostics. The reason the two tests disagree on significance is because the number of ties underpowers the sign test.

I also applied a sign test in Table 2.1. 10 functions succeed more with compiler diagnostics, 4 succeed less, and 48 are ties. The sign test is not statistically significant because it is underpowered. After all, there are 48 ties. This shows that the effect of compiler diagnostics is not uniform - 10 functions benefit from compiler diagnostics, and 4 regress.

Qualitative analysis: Next, I investigate exactly how different diagnostics can affect coding

[1] There is one change. There are several different optimization remark infrastructures in Clang, like Rpass and structured YAML output. In one case, Rpass recommends `#pragma clang loop vectorize(enable)`. I append these recommendations to the YAML output.

agents' success rates. Certain diagnostics are clearly helpful for the coding agent. On 3 out of 10 functions that had a positive difference with compiler diagnostics in TSVC, the compiler emitted the optimization remark CantReorderFPOps. This optimization remark indicates that different floating-point operations cannot be reordered, because floating-point operations are not associative. This optimization remark explicitly recommends inserting '#pragma clang loop vectorize(enable)' above a loop header in order to signal that these operations can be safely reordered. In the kernels where this diagnostic was emitted, the coding agent frequently inserted this pragma, which resolved the vectorization blocker.

However, some diagnostics are actively detrimental to coding agents. 2 functions that succeed less with diagnostics emit a diagnostic named 'NonReductionValueUsedOutsideLoop' because of an induction variable in the loop. Clang fails to compute a closed form expression for the induction variable, so it cannot automatically vectorize the loop. In these cases, the diagnostic does not describe the problem at all.

For example, in Figure 2.1, the induction variable `im1` has a closed form expression: `(LEN_1D + i - 1) % LEN_1D`. In this case, the diagnostic worsens performance because it misleads the coding agent towards solutions that construct a reduction pattern or motivates it to be more conservative by merely inserting the compiler hint `#pragma clang loop vectorize(enable)` above a loop instead of computing the closed form expression of the induction variable.

```
float s291(args_t *func_args) {
    int im1;
    im1 = LEN_1D - 1;
    for (int i = 0; i < LEN_1D; i++) {
        a[i] = (b[i] + b[im1]) * (float).5;
        im1 = i;
    }
    return 0.;
}
```

```
Pass:     loop-vectorize
Name:     NonReductionValueUsedOutsideLoop
DebugLoc: File: tsvc.c,
          Line: 1841, Column: 9
Function: s291
Args:     loop not vectorized: value that
          could not be identified as
          reduction is used outside the
          loop
```

Figure 2.1: The s291 C kernel in TSVC and the corresponding Clang analysis remark explaining why it was not vectorized.

Compiler diagnostics increase AI coding agents' task performance, but they must be designed properly, otherwise they are actively detrimental. The remainder of this thesis explores how we should evaluate and improve compiler diagnostics.

CHAPTER 3

DESCRIPTIVENESS AND PRESCRIPTIVENESS BY EXAMPLE

3.1 Descriptiveness and prescriptiveness

CANARY measures a *descriptiveness flag* and a *prescriptiveness score* for compiler diagnostics. The *prescriptiveness score* represents how much a diagnostic improves an AI coding agent's success rate at a low temperature. The *descriptiveness flag* indicates whether a diagnostic improves the agent's success rate more at a higher temperature than at the low temperature. This information can help compiler engineers refine their diagnostics to be more helpful for AI coding agents who will consume their tool's diagnostics more and more in the coming years.

I will now give an informal description of the notion of descriptiveness and prescriptiveness to help the reader understand what the CANARY method results mean for diagnostic design decisions. Informally, prescriptiveness is how much a diagnostic suggests a solution, and descriptiveness is how much actionable insight is exposed by the diagnostic that is not suggested as a solution. It requires some additional *creativity* to act on some descriptive content in a diagnostic, whereas prescriptive diagnostics can be acted on algorithmically. In this chapter, I give an understanding of descriptiveness and prescriptiveness by inspecting examples of compiler auto-vectorization diagnostics.

3.2 Example: UnsafeDep Diagnostic

The UnsafeDep auto-vectorization diagnostic says that a dependence between one memory instruction and another memory instruction in a loop makes it unsafe to execute multiple iterations of the loop in parallel using vector instructions. It also suggests inserting `#pragma clang loop distribute(enable)` before the loop.

UnsafeDep has bad prescriptive messaging but good descriptive messaging. At first glance,

```
1  float a[LEN_1D];
2  float b[LEN_1D];
3  float c[LEN_1D];
4  float d[LEN_1D];
5
6  float s241(args_t *func_args) {
7    for (int i = 0; i < LEN_1D - 1; i++) {
8      a[i] = b[i] * c[i] * d[i];
9      b[i] = a[i] * a[i + 1] * d[i];
10   }
11   return 0.;
12 }
```

```
1  Pass:        loop-vectorize
2  Name:        UnsafeDep
3  DebugLoc:    File: tsvc.c, Line: 1065, Column: 27
4  Function:    s241
5  Args:        loop not vectorized: unsafe dependent
6               memory operations in loop. Use #pragma
7               clang loop distribute(enable) to allow
8               loop distribution to attempt to isolate
9               the offending operations into a separate
10              loop Backward loop carried data
11              dependence. Memory location is the same
12              as accessed at tsvc.c:1064:13.
```

Figure 3.1: The `s241` TSVC kernel & the corresponding Clang vectorization remark.

you might think that UnsafeDep is highly prescriptive because it contains an explicit suggestion: Use `#pragma clang loop distribute(enable)` to allow loop distribution. However, `#pragma clang loop distribute(enable)` fails to automatically perform loop distribution on `s241` and every other loop in TSVC [1]. Put concretely, this prescription is misleading.

UnsafeDep is highly descriptive of the problem in loop `s241`. The issue blocking automatic loop vectorization in Figure 3.1 is a backward write-after-read dependence. On Line 9 in Figure 3.1, there is a read from `a[i + 1]`. On Line 8 in Figure 3.1, there is a write to `a[i]`. There is a data dependence that requires the read from `a[i + 1]` to execute before the write to `a[i]`, which means the ith iteration must execute before the $i + 1$th iteration of the loop. Equipped with this insight into the problem, a creative coding agent might propose a valid solution. The most effective repair is to make a temporary of `a[i + 1]` before Line 8, which would break the backward dependence. This repair is depicted in Figure 3.2. This is a well-known technique to break false data dependencies like write-after-write or write-after-read dependencies called privatization.

3.3 Example: CantReorderFPOps Diagnostic

The CantReorderFPOps diagnostic in Figure 3.3 states that the loop cannot be vectorized because floating-point operations cannot be safely reordered by default. The underlying reason is that floating-point arithmetic is not associative. The diagnostic also suggests inserting `#pragma`

[1] The compiler says loop distribution is unsafe when I insert `#pragma clang loop distribute(enable)` on any loop in TSVC compiled with Clang 21.1.8 with compiler options `-O3 -march=native`.

```
float a[LEN_1D];
float b[LEN_1D];
float c[LEN_1D];
float d[LEN_1D];
float s241(args_t *func_args) {
  for (int i = 0; i < LEN_1D - 1; i++) {
    float temp_a = a[i + 1];
    a[i] = b[i] * c[i] * d[i];
    b[i] = a[i] * temp_a * d[i];
  }
  return 0.;
}
```

Figure 3.2: The repaired `s241` TSVC kernel.

```
float a[LEN_1D];

float vsumr(args_t *func_args) {
  float sum;
  sum = 0.;
  for (int i = 0; i < LEN_1D; i++) {
    sum += a[i];
  }

  return sum;
}
```

```
Pass:        loop-vectorize
Name:        CantReorderFPOps
DebugLoc:    File: tsvc.c, Line: 3366,
             Column: 17
Function:    vsumr
Args:        loop not vectorized: cannot
             prove it is safe to reorder
             floating-point operations;
             allow reordering by specifying
             '#pragma clang loop
             vectorize(enable)' before the
             loop or by providing the
             compiler option '-ffast-math'
```

Figure 3.3: The `vsumr` TSVC kernel & the corresponding Clang vectorization remark.

`clang loop vectorize(enable)` before the loop, or by providing the compiler option `-ffast-math` which relaxes the restriction on floating point arithmetic.

The CantReorderFPOps diagnostic has good descriptive messaging and good prescriptive messaging. It is descriptive because it diagnoses the exact vectorization blocker in the `vsumr` kernel in Figure 3.3: floating-point operations in a reduction operation cannot be reordered. It is prescriptive since the diagnostic also proposes two correct solutions to the issue. Such actionable feedback can be directly applied by instruction-following models.

3.4 Example: NonReductionValueUsedOutsideLoop Diagnostic

The NonReductionValueUsedOutsideLoop diagnostic (see Figure 2.1) says that there is a variable in a loop that does not fit a standard reduction pattern, so the compiler cannot automatically vectorize it. While technically correct, the diagnostic provides very limited insight into the problem blocking vectorization. The NonReductionValueUsedOutsideLoop diagnostic is not descriptive or

prescriptive.

3.5 Example: WritesInEarlyExit Diagnostic

```
float a[LEN_1D];
float b[LEN_1D];
float c[LEN_1D];
float d[LEN_1D];

float s481(args_t *func_args) {
  for (int i = 0; i < LEN_1D; i++) {
    if (d[i] < (float)0.) {
      exit(0);
    }
    a[i] += b[i] * c[i];
  }
  return 0.;
}
```

```
Pass:            loop-vectorize
Name:            WritesInEarlyExitLoop
DebugLoc:        File: tsvc.c, Line: 2939,
                 Column: 9
Function:        s481
Args:            loop not vectorized: Cannot
                 vectorize early exit loop
                 with writes to memory
```

Figure 3.4: The `s481` TSVC kernel & the corresponding Clang vectorization remark.

The WritesInEarlyExitLoop diagnostic explains that the compiler cannot vectorize a loop that might exit if that loop also has writes to memory. This provides extremely clear insight into the problem blocking vectorization. Any coding agent could easily distribute the conditional exit and the arithmetic operations into two separate loops to enable compiler auto-vectorization like in Figure 3.5 . This diagnostic has good prescriptive messaging because of how actionable the diagnostic is. Note that a diagnostic can have prescriptive messaging even if there is not an explicit suggestion in the text of the diagnostic.

```
float a[LEN_1D];
float b[LEN_1D];
float c[LEN_1D];
float d[LEN_1D];

float s481(args_t *func_args) {
  for (int i = 0; i < LEN_1D; i++) {
    if (d[i] < (float)0.) {
      exit(0);
    }
   }
  for (int i = 0; i < LEN_1D; i++) {
    a[i] += b[i] * c[i];
  }
  return 0.;
}
```

Figure 3.5: The repaired `s481` TSVC kernel.

CHAPTER 4

THE CANARY METHOD

Now that we have an understanding of descriptiveness and prescriptiveness, I introduce a procedure to evaluate these properties in compiler diagnostics. This thesis proposes CANARY: a temperature sweep to investigate the descriptiveness and prescriptiveness of a compiler diagnostic. First, I define what a temperature sweep is in general, beyond this individual compiler auto-vectorization case study. Second, I discuss how to deploy the CANARY method to investigate compiler auto-vectorization diagnostics. Third, I evaluate existing compiler auto-vectorization diagnostics using this technique.

4.1 What does temperature have to do with descriptiveness and prescriptiveness?

A large language model's output is a series of logits. This series of logits is put through a softmax to convert it into a distribution over the next token. Temperature $T > 0$ is a parameter that divides a language model's logits before the softmax. Lower T makes the next-token distribution sharper (or more deterministic), and higher T makes it flatter (more random).

$$p_i = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)} \tag{4.1}$$

Temperature is a rough proxy for 'creativity'. Increasing the randomness of sampling from a token distribution also broadens the diversity of a language model's responses. Higher temperatures let the model explore beyond its single most likely next token, which can surface less run-of-the-mill solutions [12].

If a diagnostic has explicit prescriptions in it, then a model at a low temperature is very likely to select next tokens that follow that prescriptive messaging. It requires further exploration of the solution space for an AI agent to surface solutions from descriptive messaging. That is more likely

to happen at higher temperatures.

4.2 CANARY Temperature Sweeps

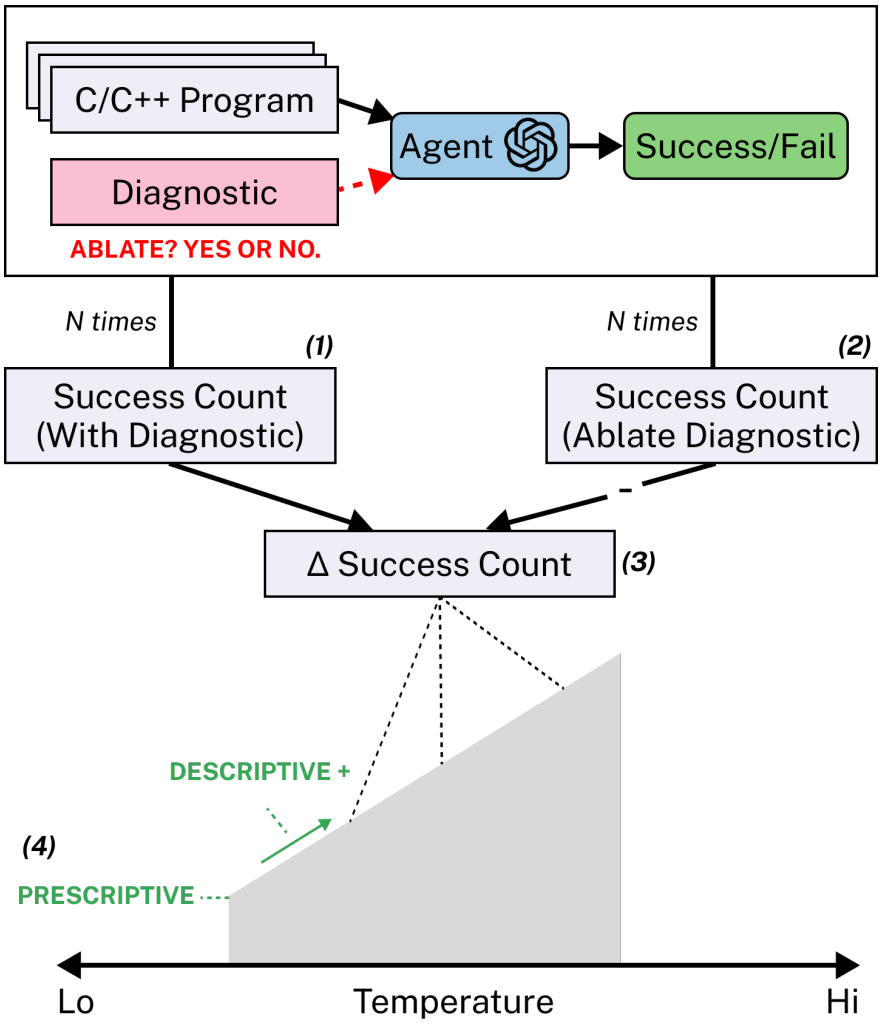


Figure 4.1: A temperature sweep. In the ablation study, an agent is given a C program (and sometimes a diagnostic), and the agent can either succeed or fail. This workflow is run 100 times. We count successes with the diagnostic and successes without the diagnostic. A success rate delta is computed by taking the difference between the number of successes with the diagnostic and without. Using this information, we compute a prescriptiveness score and a descriptiveness flag.

A temperature sweep is a method for isolating the effect of a diagnostic on a language model's task performance, and observing its effect across different language model temperatures. It has four steps depicted in Figure 4.1. (1) It measures the number of successes of an AI agent with a

compiler diagnostic across different temperatures. (2) It measures the number of successes of an AI agent without a compiler diagnostic across different temperatures. (3) It generates a success rate delta for a diagnostic, by subtracting the number of successes without the diagnostic from the number of successes with the diagnostic. (4) We do steps 1-3 over different temperatures, find the success rate at the low temperature, and the temperature T where we achieve the maximum success rate.

The success rate delta at the low temperature is what I refer to as a *prescriptiveness score*. If the temperature T where we achieve the maximum success rate is not the lowest temperature, then the descriptiveness flag is set to true.

4.3 Applying CANARY to compiler auto-vectorization diagnostics

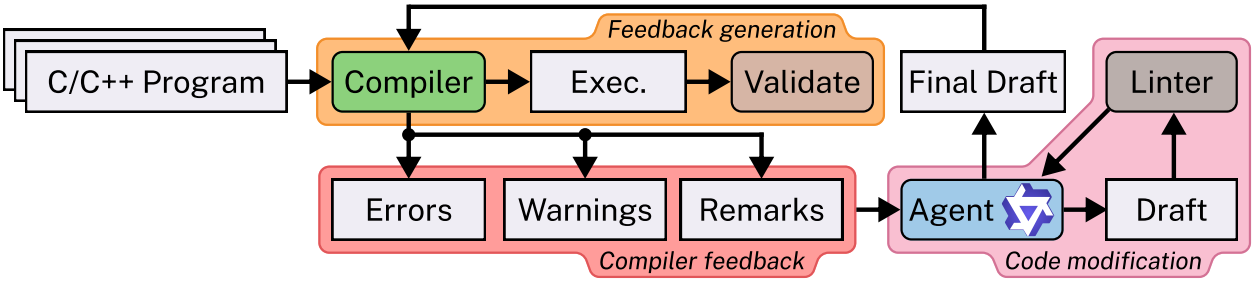


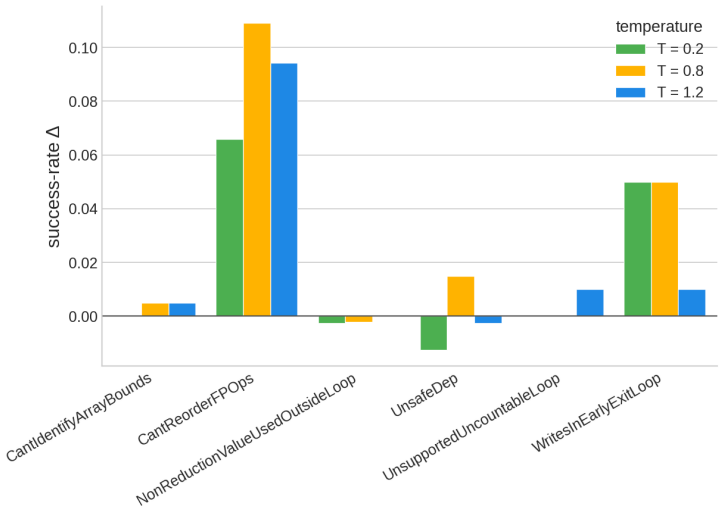
Figure 4.2: One trial in a temperature sweep. An agent is given some C program, and optionally some diagnostic information. It is supposed to generate a new draft of the program. If the new draft emits a diagnostic indicating the auto-vectorizer succeeded, then the agent succeeded.

CANARY applies the temperature sweep method to compiler auto-vectorization diagnostics. To apply CANARY, we need to specify what success means in this context:

> A C kernel has a loop that was not automatically vectorized. An AI coding agent applied a transformation to the C kernel to enable auto-vectorization. If the AI agent fails a syntax check, it gets to retry generating syntactically valid code 3 times. If the new draft of the C kernel enables automatic vectorization while passing tests, then that counts as a success.

4.4 Applying CANARY to the TSVC benchmark to Analyze Compiler Auto-Vectorization Diagnostics

I apply CANARY to compiler auto-vectorization diagnostics to evaluate them for coding agents. For each loop in TSVC, I run the CANARY workflow that I described in previous sections. I use Clang 21.1.8, with compiler options `-O3 -march=native`. The model is Qwen3-Coder-30B-A3B. I vary across 3 temperatures ($T = 0.2$, $T = 0.8$, and $T = 1.2$).



Diagnostic	Presc.	Desc.
ArrayBounds	0	Set
CantReorderFPOps	0.066	Set
NonReductionValue	-0.003	Not Set
UnsafeDep	-0.013	Set
UncountableLoop	0	Set
EarlyExit	0.05	Not Set

Figure 4.3: CANARY results for compiler auto-vectorization diagnostics on TSVC. The graph shows the success-rate delta (with vs. without diagnostic) across temperatures; the table summarizes the prescriptiveness score and descriptiveness flag for each diagnostic.

```
float s1161() {
  for (int i = 0; i < LEN_1D - 1; ++i) {
    if (c[i] < (float)0.) {
      goto L20;
    }
    a[i] = c[i] + d[i] * e[i];
    goto L10;
  L20:
    b[i] = a[i] + d[i] * d[i];
  L10:;
  }
  return 0.;
}
```

```
Pass:        loop-vectorize
Name:        CantIdentifyArrayBounds
DebugLoc:    File: tsvc.c, Line: 0,
             Column: 0
Function:    s1161
Args:        loop not vectorized: cannot
             identify array bounds

```

Figure 4.4: The Clang CantIdentifyArrayBounds diagnostic.

CantIdentifyArrayBounds (Figure 4.4): CANARY results for this diagnostic tell us that there is no prescriptive messaging present (prescriptiveness score of 0), but there is some descriptive

```
float s311(args_t *func_args) {
  float sum;
  sum = (float)0.;
  for (int i = 0; i < LEN_1D; i++) {
    sum += a[i];
  }
  return sum;
}
```

```
Pass:        loop-vectorize
Name:        CantReorderFPOps
DebugLoc:    File: tsvc.c, Line: 1970,
             Column: 17
Function:    s311
Args:        loop not vectorized: cannot
             prove it is safe to reorder
             floating-point operations;
             allow reordering by specifying
             '#pragma clang loop
             vectorize(enable)' before the
             loop or by providing the
             compiler option '-ffast-math'
```

Figure 4.5: The Clang CantReorderFPOps diagnostic.

```
float s291(args_t *func_args) {
    int im1;
    im1 = LEN_1D - 1;
    for (int i = 0; i < LEN_1D; i++) {
        a[i] = (b[i] + b[im1]) * (float).5;
        im1 = i;
    }
    return 0.;
}
```

```
Pass:     loop-vectorize
Name:     NonReductionValueUsedOutsideLoop
DebugLoc: File: tsvc.c,
          Line: 1841, Column: 9
Function: s291
Args:     loop not vectorized: value that
          could not be identified as
          reduction is used outside the
          loop
```

Figure 4.6: The Clang NonReductionValueUsedOutsideLoop diagnostic.

messaging present (the descriptiveness flag is set). Perhaps at the higher temperatures, the coding agent is able to discern that 'cannot identify array bounds' implies that the compiler does not know when the loop will stop traversing the arrays `a` or `b`.

CantReorderFPOps (Figure 4.5): CANARY results for this diagnostic tell us that there is good prescriptive messaging present (0.066), and there is good descriptive messaging present as well. That means some of the prescriptions are very high quality. In the workflow, the compiler options are fixed. This means that the `#pragma clang loop vectorize(enable)` suggestion is very helpful to the AI agent. I verified this by manually inspecting trials: I discovered that when the suggestion is present, 100 times out of 100 trials, the AI agent will insert a `#pragma clang loop vectorize(enable)` above the loop. According to CANARY, this diagnostic is highly descriptive as well. But in this case, since the diagnostic induces the AI agent to always attach a pragma, any regression in success rate without the diagnostic is attributable to increasing randomness due to high sampling temperature.

NonReductionValueUsedOutsideLoop (Figure 4.6): Results tell us that this diagnostic has no prescriptive or descriptive content. This diagnostic has a prescriptiveness score of -0.003, and

```
float a[LEN_1D];
float b[LEN_1D];
float c[LEN_1D];
float d[LEN_1D];

float s212() {
  for (int i = 0; i < LEN_1D - 1; i++) {
    a[i] *= c[i];
    b[i] += a[i + 1] * d[i];
  }
  return 0.;
}
```

```
Pass:        loop-vectorize
Name:        UnsafeDep
DebugLoc:    File: tsvc.c, Line: 838,
             Column: 21
Function:    s212
Args:        loop not vectorized: unsafe
             dependent memory operations
             in loop. Use #pragma clang
             loop distribute(enable) to
             allow loop distribution to
             attempt to isolate the
             offending operations into a
             separate loop' Backward loop
             carried data dependence."
             Memory location is the same
             as accessed at tsvc.c:837:13.
```

Figure 4.7: The Clang UnsafeDep diagnostic.

```
float s482() {
  for (int i = 0; i < LEN_1D; i++) {
    a[i] += b[i] * c[i];
    if (c[i] > b[i])
      break;
  }
  return 0.;
}
```

```
Pass:            loop-vectorize
Name:            UnsupportedUncountableLoop
DebugLoc:        File: tsvc.c, Line: 2961,
                 Column: 9
Function:        s482
Args:            loop not vectorized:
                 Cannot vectorize uncountable
                 loop
```

Figure 4.8: The Clang UnsupportedUncountableLoop diagnostic.

does not have its descriptiveness flag set.

UnsafeDep (Figure 4.7): CANARY results indicate that the UnsafeDep diagnostic has a bad prescription (prescriptiveness score of -0.013), and good descriptive messaging (descriptiveness flag is set, because $T = 0.8$ is the best performing temperature). This means that the prescription in the diagnostic is actively misleading [1]. But the fact that the descriptiveness flag is set implies that there is still good insight into the problem blocking vectorization. In the next section, I will explore fixing the negative prescription, and improving descriptive insights of this diagnostic.

UnsupportedUncountableLoop (Figure 4.8): Results show a prescriptiveness score of 0, and the descriptiveness flag is set. This tells us that this diagnostic does not have any prescriptive content but there is some insight into the problem present in the diagnostic.

WritesInEarlyExitLoop (Figure 4.9): Results tell us that this diagnostic is highly prescriptive (prescriptiveness score of 0.05), and that there is no more descriptive insight that is not exposed as prescriptive messaging. This is an interesting case where a diagnostic can be highly prescriptive

[1] This turns out to be true: attaching `#pragma clang loop distribute(enable)` does not automatically distribute any loop in TSVC.

```
float s481(args_t *func_args) {
  for (int i = 0; i < LEN_1D; i++) {
    if (d[i] < (float)0.) {
      exit(0);
    }
    a[i] += b[i] * c[i];
  }
  return 0.;
}
```

```
Pass:            loop-vectorize
Name:            WritesInEarlyExitLoop
DebugLoc:        File: tsvc.c, Line: 2939,
                 Column: 9
Function:        s481
Args:            loop not vectorized: Cannot
                 vectorize early exit loop
                 with writes to memory
```

Figure 4.9: The Clang WritesInEarlyExit diagnostic.

even without an explicit suggestion in the diagnostic. The diagnostic is so clear about the nature of the problem that agents at low temperatures know exactly what transformation would resolve the vectorization blocker. In practice, the agent frequently lands on the correct solution (loop distribution) to resolve the vectorization blocker.

In the next section, I leverage these insights to improve compiler auto-vectorization diagnostics for AI coding agents.

CHAPTER 5

IMPROVING COMPILER AUTO-VECTORIZATION DIAGNOSTICS WITH CANARY

In the prior chapter, I collected insights into the quality of compiler auto-vectorization diagnostics for AI coding agents. In this chapter, I leverage these insights to improve these diagnostics for AI coding agents. First, I discuss how to interpret a CANARY diagram to figure out the next step to improve a diagnostic. Second, I apply this procedure to the UnsafeDep diagnostic. Finally, I evaluate the improved diagnostic to show that this data-driven exercise can lead to significant improvements.

5.1 Improving Compiler Auto-Vectorization Diagnostics using the Canary Method

From the results of the CANARY method, we collect prescriptiveness scores and descriptiveness flags for diagnostics. This section describes how we use those metrics. Figure 5.1 describes the improvement process of a diagnostic. The goal is to move from Diagnostic 0 to Diagnostic 3. Diagnostic 0 has a detrimental prescription. We should remove the detrimental prescription from Diagnostic 0, which transforms it into Diagnostic 1. Diagnostic 1 lacks descriptive messaging and has little prescriptive messaging. Here, we aim to transform Diagnostic 1 into Diagnostic 2 by exposing more compiler analyses to make the diagnostic more descriptive. Diagnostic 2 is not very prescriptive. In this situation, we aim to convert the descriptive messaging in the diagnostic into explicit prescriptions. That produces Diagnostic 3.

From our results, we found that UnsafeDep has the highest ceiling for improvement. There are two reasons. First, it has a negative prescription. Removing negative prescriptive messaging is simple. Second, it has good descriptive messaging, and we might consider translating this descriptive messaging into something more prescriptive.

As we discussed earlier, the prescription in UnsafeDep Figure 4.7 is actively misleading. `#pragma clang loop distribute(enable)` does not automatically distribute a loop in

TSVC with our compiler options.

Based on this information, I remove the negative prescription, and second, I make the problem more precise to substitute for the bad pragma suggestion by supplementing data dependence type information. The new diagnostic text looks like Listing 5.1.

```
Pass:            loop-vectorize
Name:            UnsafeDep
DebugLoc:        File: tsvc.c, Line: 838,
                 Column: 21
Function:        s212
Args:            loop not vectorized: unsafe dependent memory operations in loop.
                 Attempt loop splitting to isolate the offending operations in another loop
                 Backward write-after-read data dependence. Memory location is the same as
                 accessed at tsvc.c:837:13
```

Listing 5.1: Improved UnsafeDep diagnostic with dependence-type information

5.2 Evaluating Improved Diagnostic

After the intervention, I re-ran CANARY on the improved UnsafeDep diagnostic. UnsafeDep's prescriptiveness score rose from -0.013 to 0.141 (Figure 5.2). This change matches our prediction: we removed a broken prescription and made the descriptive messaging more precise to augment prescriptiveness. That not only led to a jump in prescriptiveness, it also flattened the temperature curve.

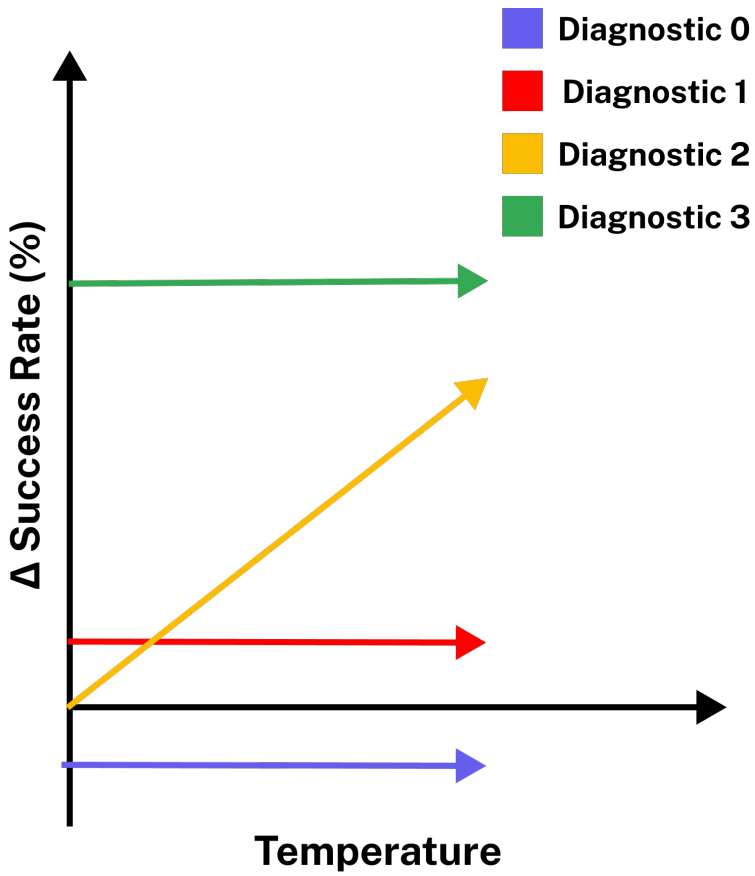


Figure 5.1: This describes how to improve a diagnostic. Diagnostic 0 has an actively misleading prescription. Removing the prescription transforms Diagnostic 0 into Diagnostic 1. Here, Diagnostic 1 is not prescriptive since it does not have a high success rate delta at low temperature and it is not descriptive either since it has a very flat slope. Diagnostic 2 is more descriptive because it has a steeper slope. Diagnostic 3 is highly prescriptive because it has a high success rate delta at a low temperature. A compiler engineer should have three goals: first, fix negative prescriptions (Diagnostic 0 to Diagnostic 1). Second, take diagnostics that lack description and prescription, and make them more descriptive (Diagnostic 1 to Diagnostic 2). Third, take diagnostics that are descriptive, and translate its descriptive messaging into prescriptive messaging (Diagnostic 2 to Diagnostic 3).

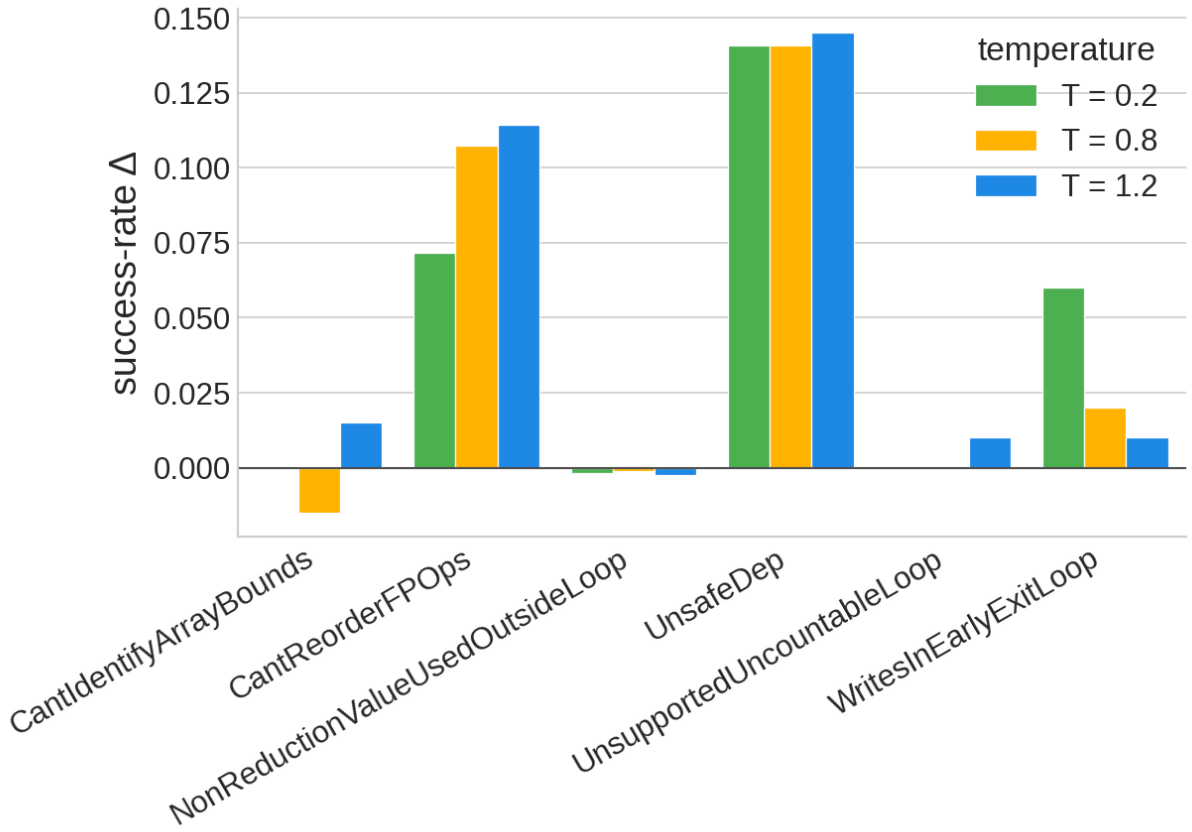


Figure 5.2: CANARY diagram after improving the `UnsafeDep` diagnostic. We observe a drastic improvement in the prescriptiveness score of the `UnsafeDep` diagnostic.

CHAPTER 6

DISCUSSION

6.1 Discussing Limits of the CANARY Method

There are limits to this approach. The descriptiveness flag might be set for the wrong reason, and the approach requires a lot of compute and time.

6.1.1 Descriptiveness Flag

First, the descriptiveness flag can be set for the wrong reason. There are two reasons that a diagnostic might have a higher success rate delta at higher model temperature. One, the diagnostic has descriptive messaging. It helps a model at higher temperatures. Two, language models regress at higher temperatures. More random sampling of tokens leads the model away from good solutions that might be explicit in the diagnostic.

One case stands out in my data to illustrate this: `CantReorderFPOps`. With the `CantReorderFPOps` diagnostic, the agent modifies the TSVC kernel by inserting `#pragma clang loop vectorize(enable)` in every single trial across every temperature. Without the diagnostic, the agent's performance regresses as temperature increases. In this case, a higher success rate delta at a higher temperature does not signal a strength of the diagnostic, it signals model degradation at higher temperatures.

6.1.2 CANARY is expensive

CANARY requires a large corpus of tasks to analyze diagnostics. If someone wanted to apply the method to their own tools, they would need a corpus to emit their diagnostics in a variety of situations. This was one constraint in these experiments. Clang fails to automatically vectorize only 62 loops in TSVC, so the corpus of loops that this thesis runs experiments with is small.

Moreover, CANARY is computationally expensive. I ran these experiments with 2 NVIDIA

A40s, serving Qwen3-Coder-30B-A3B with the vLLM inference engine. Collecting enough data to generate one of the CANARY diagrams above takes approximately 8 hours.

6.2 Next Steps for the CANARY method

A method to automatically evaluate compiler diagnostics for AI coding agents paves the way for automatically improving compiler diagnostics as well. If a model knows the prescriptiveness score and the descriptiveness flag of a diagnostic, then it can automatically refine compiler outputs to improve the diagnostic following a simple decision tree. If a diagnostic has an actively misleading prescription, an AI coding agent should remove or change the prescription. If a diagnostic is not descriptive, an AI coding agent should expose more analyses from the compiler. If a diagnostic is descriptive, an AI coding agent should turn descriptions into solutions or fixes.

CHAPTER 7

RELATED WORK

No prior work has investigated a mechanism to evaluate and improve compiler diagnostics for AI coding agents. But there are several other approaches that sit at the intersection of compilers, diagnostics, the loop vectorizer, and large language models.

7.1 Evaluating Compiler Diagnostics

Prior work has identified that compiler diagnostics are crucial in software engineering workflows and developed techniques to evaluate compiler diagnostics. Sun et al. compared GCC and Clang compiler implementations and used discrepancies in compiler warnings to report bugs in both compilers [13]. Barik et al. developed a framework for improving compiler error notifications emitted by IDEs [14]. Becker et al. proposes a qualitative method for assessing the readability of error messages focused on message length, message tone, and use of jargon [8]. Barik et al. propose a model to decompose compiler diagnostics into their constituent parts: claim, warrant, grounds, backing, qualifier, rebuttal, and resolution. [7]. Lastly, FeedbackEval evaluates how good large language models are at interpreting feedback from compilers, tools, tests, and humans. [15]. Deo et al. emphasized the importance of compiler diagnostics in this feedback loop between AI coding agents and the compiler [6].

7.2 Enhancing AI with Compilers

Prior works have utilized compilers to improve agent workflows for writing high-performance code. Several use compiler diagnostics in their agent workflows, but all of them treat the compiler diagnostic interface as fixed instead of evaluating and improving it. These prior works fall into three categories.

Inference-time techniques: CompilerGPT designs an agent workflow to transform C kernels in 3 parts: (1) interpret an optimization report, (2) analyze the code, and (3) improve the code's performance [16]. LOOPRAG uses static analysis on C loops to inform retrieval augmented generation of prompt examples from a dataset [9]. VecTrans departs from prior work and proposes that LLMs should perform source-to-source transformations of C kernels instead of emitting vector intrinsics [4].

Training: LLM-VeriOpt focuses on increasing model capability instead of modifying the agent workflow: it applies reinforcement learning techniques to large language models to improve their performance on optimization tasks instead of using out of the box LLMs [10].

Validation: LLM-Vectorizer engineered an agent workflow to transform C kernels into kernels containing vector intrinsics, and developed a 3-part framework to validate the agent's code transformations [3]. Compiler-Runtime Cooperative Chain of Verification employs runtime verification checks in addition to ahead of time techniques to validate AI code transformations [5].

7.3 Enhancing Compilers with AI

Some works suggested putting AI in the compiler. Many suggest using machine learning models to replace hand-crafted heuristics in the compiler. NeuroVectorizer decides on vectorization and interleaving factors based on machine learning models [17]. Other works take a much bolder approach; they suggest utilizing large language models to orchestrate entire transformation passes. IntOpt uses large language models to structure a plan ahead of time, and uses large language models to conduct individual transformations with the aid of compilers' static analysis and optimization pass lists [18]. ACCLAIM interleaves LLM calls with the traditional layers of a compiler - the front-end lowering, middle-end optimization passes, and the back-end lowering [19]. Some suggest using large language models as a wholesale replacement for the compiler [20].

CHAPTER 8

CONCLUSION

Compiler diagnostics are the bottleneck for coding agent performance, and compiler engineers lack a method to evaluate and improve their tool's diagnostics. This thesis attempts to remedy this issue by introducing CANARY: an experiment that observes large language model behavior across different temperatures to evaluate compiler diagnostics for coding agents. I applied the CANARY method to a compiler auto-vectorization diagnostic in Clang, improved it based on insight collected from CANARY, and showed that it improves coding agent performance. This work paves the way for evaluating and improving compiler diagnostics for AI, using AI.

REFERENCES

[1] S. Yao et al., "ReAct: Synergizing reasoning and acting in language models," in *International Conference on Learning Representations (ICLR)*, 2023.

[2] J. Yang et al., "Swe-agent: Agent-computer interfaces enable automated software engineering," in *Proceedings of the 38th International Conference on Neural Information Processing Systems*, ser. NIPS '24, Vancouver, BC, Canada: Curran Associates Inc., 2024, ISBN: 9798331314385.

[3] J. Taneja, A. Laird, C. Yan, M. Musuvathi, and S. K. Lahiri, "Llm-vectorizer: Llm-based verified loop vectorizer," in *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*, ser. CGO '25, Las Vegas, NV, USA: Association for Computing Machinery, 2025, 137–149, ISBN: 9798400712753.

[4] Z. Zheng et al., *Vectrans: Enhancing compiler auto-vectorization through llm-assisted code transformations*, 2025. arXiv: 2503.19449 [cs.SE].

[5] H. Kwon et al., "Compiler-runtime co-operative chain of verification for llm-based code optimization," in *2026 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2026, pp. 617–629.

[6] A. Deo, S. Campanoni, and T. McMichen, "Ai coding agents need better compiler remarks," Presented at Workshop on Co-Design for Agentic and Multimodal AI (CoDAIM), 2026.

[7] T. Barik, D. Ford, E. Murphy-Hill, and C. Parnin, "How should compilers explain problems to developers?" In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018, Lake Buena Vista, FL, USA: Association for Computing Machinery, 2018, 633–643, ISBN: 9781450355735.

[8] B. A. Becker, P. Denny, J. Prather, R. Pettit, R. Nix, and C. Mooney, "Towards assessing the readability of programming error messages," in *Proceedings of the 23rd Australasian Computing Education Conference*, ser. ACE '21, Virtual, SA, Australia: Association for Computing Machinery, 2021, 181–188, ISBN: 9781450389761.

[9] Y. Zhi et al., "Looprag: Enhancing loop transformation optimization with retrieval-augmented large language models," in *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS '26, USA: Association for Computing Machinery, 2026, 1113–1135, ISBN: 9798400723599.

[10] X. Fang, J. Kang, R. Rocha, S. Ainsworth, and L. Mukhanov, "Llm-veriopt: Verification-guided reinforcement learning for llm-based compiler optimization," in *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2026.

[11] UoB-HPC, *TSVC_2: Updated c version of the test suite for vectorising compilers*, `https://github.com/UoB-HPC/TSVC_2`, Accessed: 2026-02-05, 2025.

[12] H. Chen and N. Ding, "Probing the "creativity" of large language models: Can models produce divergent semantic association?" In *Findings of the Association for Computational Linguistics: EMNLP 2023*, H. Bouamor, J. Pino, and K. Bali, Eds., Singapore: Association for Computational Linguistics, Dec. 2023, pp. 12 881–12 888.

[13] C. Sun, V. Le, and Z. Su, "Finding and analyzing compiler warning defects," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16, Austin, Texas: Association for Computing Machinery, 2016, 203–213, ISBN: 9781450339001.

[14] T. Barik, J. Witschey, B. Johnson, and E. Murphy-Hill, "Compiler error notifications revisited: An interaction-first approach for helping developers more effectively comprehend and resolve error notifications," in *Companion Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE Companion 2014, Hyderabad, India: Association for Computing Machinery, 2014, 536–539, ISBN: 9781450327688.

[15] D. Dai et al., *Feedbackeval: A benchmark for evaluating large language models in feedback-driven code repair tasks*, 2026. arXiv: `2504.06939 [cs.SE]`.

[16] P. Pirkelbauer and C. Liao, *Compilergpt: Leveraging large language models for analyzing and acting on compiler optimization reports*, 2025. arXiv: `2506.06227 [cs.PL]`.

[17] A. Haj-Ali, N. K. Ahmed, T. Willke, S. Shao, K. Asanovic, and I. Stoica, *Neurovectorizer: End-to-end vectorization with deep reinforcement learning*, 2020. arXiv: `1909.13639 [cs.DC]`.

[18] L. Qiu, Z. Yang, F. Lyu, M. Zhong, H. Cui, and X. Feng, *Beyond pass-by-pass optimization: Intent-driven ir optimization with large language models*, 2026. arXiv: `2602.18511 [cs.PL]`.

[19] B. Mikek, D. Vashchilenko, B. Lu, and P. Xu, *Agentic code optimization via compiler-llm cooperation*, 2026. arXiv: `2604.04238 [cs.PL]`.

[20] C. Cummins et al., "LLM compiler: Foundation language models for compiler optimization," in *Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction*, 2025.

Appendices

HOW THE TEMPERATURE SWEEP CAN HELP DESIGN COMPILER DIAGNOSTICS FOR AI CODING AGENTS

Approved by:

Dr. Campanoni, Advisor
McCormick School of Engineering
Northwestern University

Dr. Dimoulas
McCormick School of Engineering
Northwestern University

Date Approved: June 2, 2026

APPENDIX A

CANARY PROMPTS

```
You are a C programmer. Fix the syntax error in the function below.

## Source Code
```c
{{ kernel }}
```

Compiler Error
{{ linter_output }}

Rules
- Fix only the syntax error. Do not change logic, structure, or comments!
- Do not add or remove includes!
- Do not change the function signature!

Output
Return only the change as a SEARCH/REPLACE block. SEARCH must match the
    existing source exactly, including whitespace and indentation:

<<<<<<< SEARCH
// exact lines to replace
=======
// corrected lines
>>>>>>> REPLACE
```

Listing A.1: Repair Syntax Prompt

```
You are a performance engineer. Your task is to modify a C/C++ function so
```

````
   that the compiler will successfully auto-vectorize it, while preserving
   exact semantic behavior.

## Source Code
```c
{{ kernel }}
```

## Compiler Remarks
{{ remarks }}

## Your Task
Identify why vectorization failed from the remarks above and apply the minimal
    source-level change that fixes it.

## Allowed Changes
- `#pragma` loop directives placed on the line immediately above the loop
   header
- Loop restructuring (distribution, peeling, scalar expansion)

## Prohibited Changes
- Do not change loop bounds or iteration count
- Do not add OpenMP or SIMD intrinsics
- Do not change the function signature

## Available Pragmas
- `#pragma clang loop vectorize(enable)` - force vectorization attempt
- `#pragma clang loop vectorize(assume_safety)` - disable dependency check (
   only if iterations are provably independent)
- `#pragma clang loop vectorize_width(N)`
- `#pragma clang loop interleave(enable)`
- `#pragma clang loop distribute(enable)`

````

```
## Output
Return only the change as a SEARCH/REPLACE block. SEARCH must match the
    existing source exactly, including whitespace and indentation:

<<<<<<< SEARCH
// exact lines from the source to replace
=======
// modified lines
>>>>>>> REPLACE
```

Listing A.2: Optimize Prompt