Early notice: GenAI-based Datarace Fix for Real-World Golang Programs

Feiyang Jin [*]
Georgia Institute of Technology
Atlanta, GA
`fjin35@gatech.edu`

Zhizhou Zhang
Uber Technologies Inc.
Sunnyvale, CA
`zzzhang@uber.com`

Rajkishore Barik[†]
Gitar Inc.
`raj@gitar.co`

Gautam Korlam[†]
Gitar Inc.
`gautam@gitar.co`

Milind Chabbi
Uber Technologies Inc.
Sunnyvale, CA
`milind@uber.com`

Abstract

Data race detection has been a subject of extensive research for decades; the practical deployment of race detectors has also become increasingly commonplace in industrial settings. However, the focus has mainly been on the detection aspect, with relatively little attention directed toward the challenging task of autonomously repairing programs with data races. This discrepancy is understandable given the inherent complexities of fixing the data race and the substantial engineering efforts required to integrate fixes into existing workflows.

In this paper, we introduce a novel closed-loop application that harnesses the power of Generative AI to fix data races automatically. Our early experiments involving this application within Uber's internal codebase have yielded promising results. The evaluation results suggest a bright future for integrating this application into Uber's infrastructure, potentially revolutionizing how data races are handled in large-scale software development environments.

1 Introduction

Golang (Go for short) is a popular programming language, and one of its main features is the support of concurrency as a first-class citizen. By simply prefixing the keyword "go", the immediately following function call will run asynchronously with the following continuation as a lightweight **goroutines** [3] managed by Go's runtime. However, as with other concurrency features, programmers must take care of *data races* [6, 9], a notorious bug caused by conflicting concurrent access to the same memory location. Researchers have been developing various data race detection techniques [4, 10, 13, 12, 6, 9] for a long time. Repair of data races has also been studied for structured parallel programs [11].

Uber has deployed a race detector on its industrial codebase since 2021. The race detector has found more than 6000 races over 81 million lines of Go code in Uber's internal Go monorepo [8]. At this point, there are about 2600 fixes submitted by engineers of Uber.

Nevertheless, as new code comes in, it is extremely challenging to depend solely on programmers' efforts to fix all of them. For Uber, around 4.5K engineers generate about 100K diffs per month split across monorepos. On average, 5-10 new data races are detected per day. Indeed, even addressing

---

[*] Work done during the internship at Uber.

[†] Work done at Uber.

Machine Learning for Systems Workshops at 37th NeurIPS 2023.

a single race involves a multistep process: reproducing the race locally, devising a logically sound fix, submitting the diff, communicating with other teams, etc. In addition, average developers are not concurrency experts, and fixing data race can be quite complicated and requires domain-specific knowledge and expertise.

An autonomous way of fixing data races could be immensely beneficial to Uber's engineers. In this project, we use Generative AI (GenAI) to create patches for code with races. The main challenges lie on two fronts: first, we must ensure that the patch successfully fixes the race without affecting other functionality. Second, the AI-generated fix must be similar to what a human engineer would propose, such as avoiding using locks as much as possible to prevent code serialization and following Go coding idioms. To handle the challenges, we build a closed-loop workflow by utilizing Uber's internal infrastructure. The contributions of this work are as follows:

1. A closed-loop application using GenAI that fixes data races in the Uber internal codebase. As the first step, it supports races where the racy accesses happened within the same function.
2. Empirical evaluation of the application in the industrial environment.

2 Background

A data race occurs when multiple goroutines concurrently access the same memory location, and at least one performs a write. For example, in the original code on the left side of Fig 1, there exist concurrent writes to the `err` variable. The err object is initially declared in line 1. However, the goroutine created by `go func()` in line 9 will currently write to the same err on line 10. The proposed solution of GenAI involves declaring a local err variable for the goroutine in line 10 on the right figure to avoid a data race on the original err. Such technique is known as **privatization** in the compiler and programming language community [5]. This fix not only aligns with logical correctness but also closely mirrors the approach a human engineer would take.

```
1  data, err := Preprocess()
2  if err != nil {
3    return err
4  }
5  var ad Advertisement
6  var item Item
7  var wg sync.WaitGroup
8  wg.Add(1)
9  go func() {
10     ad, err = FetchAd()
11     if err != nil {
12         log.Log("failed to fetch Ad")
13     }
14     wg.Done()
15 }()
16 item, err = getItem()
17 ...
18 wg.Wait()
```

```
1  data, err := Preprocess()
2  if err != nil {
3    return err
4  }
5  var ad Advertisement
6  var item Item
7  var wg sync.WaitGroup
8  wg.Add(1)
9  go func() {
10   // Create a local err to avoid race.
11   var err error
12     ad, err = FetchAd()
13     if err != nil {
14         log.Log("failed to fetch Ad")
15     }
16     wg.Done()
17 }()
18
19 item, err = getItem()
20 ...
21 wg.Wait()
```

Figure 1: Original bug(left) with a correct and idiomatic fix (right)

The Google distribution of Go comes with a built-in race detector based on ThreadSanitizer [10]. It can be enabled by passing the "-race" flag when compiling the program. The Uber internal race detection runs the built-in detector every night and creates an issue ticket (in a dedicated database) for each new race found.

3 Design

In this early experiment, our primary objective was to repair races within a single function. We specifically concentrate on common pitfalls related to transparent capture-by-reference. As discussed in previous research [2], Go automatically captures free variables by reference in nested functions, which contrasts with other programming languages like C++, where programmers typically have explicit control over whether a variable should be captured by value or by reference.

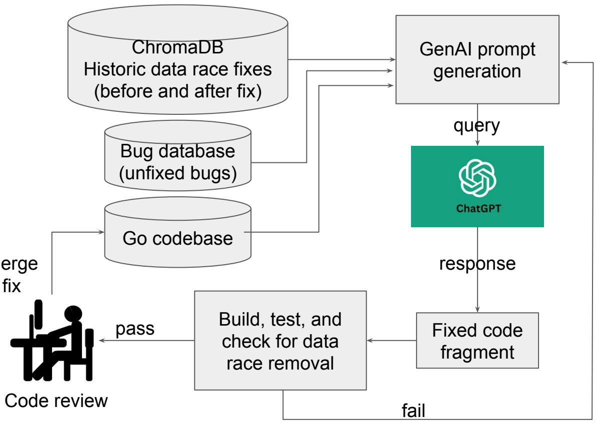

Figure 2: Datarace fixing workflow using chatGPT generative AI.

Fig. 2 shows the overall design for our closed-loop application. As a prerequisite, we create a database of all previously fixed data races maintaining the original racy function and the corresponding fixed function. We use Chromadb [1] to store it. We have thousands of such data race patches from previously fixed data races. Later we query this database to identify most similar examples when we want to automatically fix a new data race. We can provide some exemplar fixes to the generative AI to aid in solving new data races while adhering to common coding styles.

When a new data race needs to be fixed, we have the source locations (functions and lines) involved in the data race from the database of tickets. We extract the racy function from the file and query the most similar function pairs (before the fix and after the fix) from the Chromadb.

We compose the query message to OpenAI, consisting of three primary components:

1. three few-shot examples chosen by Chromadb based on code similarity. Each example contains two parts: function before with races and function after without races.
2. the function where the racy accesses lie in. We provide the entire function definition.
3. the text of the two lines that race with each other.

We query OpenAI chatGPT 4.0 [7] model to return an updated version of the function in which the race is removed. To verify the patch, the old function with race(s) is replaced by the one returned by OpenAI. We then run the race detection test again. If the number of races is reduced, we claim it as a success. Otherwise, the test may fail to build, or the number of races does not reduce, in which case we count it as a failure. If the patch removes the race, we need to verify if the fix is logically correct and analogous to a human's solution, hence it is sent for code review for the code owners.

Go's built-in race detector is non-deterministic [4, 12, 13], which detects races based on the execution schedule. To mitigate the impact of this issue, we adopt a strategy where each test is executed multiple times. From the output, we tally the number of unique races by generating hash values for each race trace. We ensure the data race is reproducible prior to the fix and eliminated post-fix.

4 Evaluation

In this experiment, we chose a subset of 157 tickets from a set of around 2000 tickets. GenAI provides a successful fix for 70 (44.6%) tickets (the patch can be built and all tests passed). We conducted a second experiment on 29 buildable open tickets where the races are on error variables. We observed a higher success rate as GenAI fixed 20 (69%) tickets. Now, we will discuss some representative examples we chose.

Fig. 1 is a good fix by privatizing the variable err. This technique is not an uncommon approach, as it is evident across all proposed solutions, underscoring GenAI's adeptness in understanding **privatization**.

In Fig. 3, GenAI opts to privatize the variables err in line 10 by creating a local variable $localErr$. In Golang, $:=$ is a shortcut operator to declare a variable and assign a value in one step. Although this approach is technically sound, it diverges from idiomatic practices. The reason is user can still accidentally write to the global err within the goroutine and cause data races.

Fig. 4 is considered an inferior fix by us because it uses a Mutex lock unnecessarily. The race arises due to concurrent appends to the $workflowErrs$ array. An ideal fix would be assigning a unique array index to each goroutine so that each one writes to its respective index of the array. However, simply using the Mutex lock will serialize the code and defeat the purpose of concurrent goroutine execution.

```
1  data, err := Preprocess()
2  if err != nil {
3    return err
4  }
5  var ad Advertisement
6  var item Item
7  var wg sync.WaitGroup
8  wg.Add(1)
9  go func() {
10     ad, err = FetchAd()
11     if err != nil {
12        log.Log("failed to fetch Ad")
13     }
14     wg.Done()
15 }()
16 item, err = getItem()
17 ...
18 wg.Wait()
```

```
1  data, err := Preprocess()
2  if err != nil {
3    return err
4  }
5  var ad Advertisement
6  var item Item
7  var wg sync.WaitGroup
8  wg.Add(1)
9  go func() {
10     ad, localErr := FetchAd()
11     if localErr != nil {
12        log.Log("failed to fetch Ad")
13     }
14     wg.Done()
15 }()
16
17 item, err = getItem()
18 ...
19 wg.Wait()
```

Figure 3: Original bug(left) with a correct but non-idiomatic fix (right)

```
var workflowErrs []error
var wg sync.WaitGroup
wg.Add(4)

go func() {
  defer wg.Done()
  err := ...
  if err != nil {
    workflowErrs = append(workflowErrs, ...)
  }
}()
```

```
var workflowErrs []error
var workflowErrsLock sync.Mutex
var wg sync.WaitGroup
wg.Add(4)

go func() {
  defer wg.Done()
  err := ...
  if err != nil {
    workflowErrsLock.Lock()
    defer workflowErrsLock.Unlock()
    workflowErrs = append(workflowErrs, ...)
  }
}()
```

Figure 4: Original bug(left) with an inferior fix(right)

5 Conclusion and future work

Data race detection has been extensively explored across various programming models. Nevertheless, there has been a noticeable gap in the literature regarding repairing data races in programs. There are practical hurdles in applying automated fixes to data races due to the the intricate nature of the task, from identifying diverse race patterns to localization of these races, and the subsequent correct fix.

Generative AI presents a unique opportunity to ease the tasks that previously required meticulous engineering effort. The work presented here represents an early experiment stemming from this potential. Leveraging Uber's internal infrastructure and GenAI, we have constructed an application designed to fix data races identified within Uber's Go monorepo. Preliminary evaluation results underscore the promising prospects of integrating this workflow into Uber's expansive codebase. There exist several aspects we would like to improve on: a) extend the application to support data races spanning multiple functions and files, b) enhance the code matching based on concurrency constructs, and c) cover more types of race patterns beyond variable capture [2].

References

[1] chroma-core/chroma: the ai-native open-source embedding database. `https://github.com/chroma-core/chroma`. (Accessed on 09/11/2023).

[2] Milind Chabbi and Murali Krishna Ramanathan. A study of real-world data races in golang. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 474–489, 2022.

[3] Golang developers. Effective go: Goroutine. `https://go.dev/doc/effective_go#goroutines`. Accessed: 2023-09-03.

[4] Feiyang Jin, Lechen Yu, Tiago Cogumbreiro, Jun Shirako, and Vivek Sarkar. Dynamic Determinacy Race Detection for Task-Parallel Programs with Promises. In Karim Ali and Guido Salvaneschi, editors, *37th European Conference on Object-Oriented Programming (ECOOP 2023)*, volume 263 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 13:1–13:30, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[5] Ken Kennedy and John R Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., 2001.

[6] John Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, pages 24–33, 1991.

[7] OpenAI. Gpt-4 technical report, 2023.

[8] Rachel Potvin and Josh Levenberg. Why google stores billions of lines of code in a single repository. *Communications of the ACM*, 59(7):78–87, 2016.

[9] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. Scalable and precise dynamic datarace detection for structured parallelism. *Acm Sigplan Notices*, 47(6):531–542, 2012.

[10] Konstantin Serebryany and Timur Iskhodzhanov. Threadsanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, page 62–71, New York, NY, USA, 2009. Association for Computing Machinery.

[11] Rishi Surendran, Raghavan Raman, Swarat Chaudhuri, John Mellor-Crummey, and Vivek Sarkar. Test-driven repair of data races in structured parallel programs. PLDI '14, page 15–25, New York, NY, USA, 2014. Association for Computing Machinery.

[12] Adarsh Yoga, Santosh Nagarakatte, and Aarti Gupta. Parallel data race detection for task parallel programs with locks. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 833–845, New York, NY, USA, 2016. Association for Computing Machinery.

[13] Lechen Yu, Feiyang Jin, Joachim Protze, and Vivek Sarkar. Leveraging the dynamic program structure tree to detect data races in openmp programs. In *2022 IEEE/ACM Sixth International Workshop on Software Correctness for HPC Applications (Correctness)*, pages 54–62, 2022.

Appendix

In this appendix, we provide the prompt used to interact with OpenAI and example responses.

A Context-setting message to OpenAI (role: "system")

> you are an expert in parallel computing and helping user fix data race in the golang programs. The user will provide you a function with data race; you will try to fix the race. Your response should only contain the fixed function. Do not say any other thing. Do not skip any code by saying ‘the rest of the code stays the same’. Do not use locks unless absolutely necessary. If possibile, create a local variable with the same name as the captured variable to eliminate the data race. You will be given 3 examples that fix data race in golang function.

B Few-shot examples conversation

```
User:  Example code snippet 1 before
func (m *Mitigator) IsMigrated(
  ctx context.Context,
  userCtx UserContext,
  operation string,
) (bool, error) {
  userCtxConfig := getConfig(userCtx)
  membershipUserCtx, err := Parse(
    ctx,
    userCtxConfig,
  )
  if err != nil {
    return false, err
  }
  var isConfigured bool
  g, ctx := errgroup.WithContext(ctx)
  g.Go(func() error {
    isConfigured, err = m.isConfigured(ctx, membershipUserCtx)
    return err
  })
  if err = g.Wait(); err != nil {
    return false, err
  }
  return isConfigured, nil
}

User:  Example code snippet 1 after
func (m *Migrator) IsMigrated(
  ctx context.Context,
  userCtx UserContext,
  operation string,
) (bool, error) {
  userCtxConfig := getConfig(userCtx)
  membershipUserCtx, err := Parse(
    ctx,
    userCtxConfig,
  )
  if err != nil {
    return false, err
  }
  var isConfigured bool
  g, ctx := errgroup.WithContext(ctx)
  g.Go(func() error {
    var err error
    isConfigured, err = m.isConfigured(ctx, membershipUserCtx)
    return err
  })
  if err = g.Wait(); err != nil {
    return false, err
  }
  return isConfigured, nil
}
```

Figure 5: Few-shot examples 1

User: Example code snippet 2 before

```
func (c *controller) GetMemberships(ctx context.Context, request Request) (Response, error) {
  logger := FromContext(ctx)
  logger.Info("GetMemberships controller called")
  membershipInfos, err := c.GetInfo(ctx, request.UserUUID)
  if err != nil {
    logger.With(zap.Error(err))
    return Response{}, err
  }
  var wg sync.WaitGroup
  wg.Add(len(membershipInfos))
  memberships := make([]Membership, len(membershipInfos))
  for i, membershipInfo := range membershipInfos {
    go func(ictx context.Context, index int, memInfo MembershipInfo) {
      defer wg.Done()
      memberships[index], err = c.process(ictx, memInfo)
      if err != nil {
        logger.With(zap.Error(err))
      }
    }(ctx, i, membershipInfo)
  }
  wg.Wait()
  return Response{
    Memberships: memberships,
  }, nil
}
```

User: Example code snippet 2 after

```
func (c *controller) GetMemberships(ctx context.Context, request Request) (Response, error) {
  logger := FromContext(ctx)
  logger.Info("GetMemberships controller called")
  membershipInfos, err := c.GetInfo(ctx, request.UserUUID)
  if err != nil {
    logger.With(zap.Error(err))
    return Response{}, err
  }
  var wg sync.WaitGroup
  wg.Add(len(membershipInfos))
  memberships := make([]Membership, len(membershipInfos))
  for i, membershipInfo := range membershipInfos {
    go func(ictx context.Context, index int, memInfo MembershipInfo) {
      defer wg.Done()
      err := err
      memberships[index], err = c.process(ictx, memInfo)
      if err != nil {
        logger.With(zap.Error(err))
      }
    }(ctx, i, membershipInfo)
  }
  wg.Wait()
  return Response{
    Memberships: memberships,
  }, nil
}
```

Figure 6: Few-shot example 2

User: Example code snippet 3 before

```
func TestConcurrentDo(t *testing.T) {
  client, err := NewClient()
  require.NoError(t, err)
  defer func() { require.NoError(t, client.Close()) }()
  concurrency := 10
  var eg util.Group
  // Set redis key values concurrently
  for i := 0; i < concurrency; i++ {
    var (
      i = i
    )
    go func() error {
      key := "foo" + strconv.Itoa(i)
      _, err = client.Do(CommandSET, key, "bar")
      return err
    }()
  }
  err = eg.Wait()
  require.NoError(t, err)
  // Fetch redis key values and assert on expected values
  for i := 0; i < concurrency; i++ {
    key := "foo" + strconv.Itoa(i)
    val, err := redis.String(client.Do(CommandGET, key))
    require.NoError(t, err, key)
    assert.Equal(t, "bar", val, key)
  }
}
```

User: Example code snippet 3 after

```
func TestConcurrentDo(t *testing.T) {
  client, err := NewClient()
  require.NoError(t, err)
  defer func() { require.NoError(t, client.Close()) }()
  concurrency := 10
  var eg util.Group
  // Set redis key values concurrently
  for i := 0; i < concurrency; i++ {
    var (
      i = i
    )
    go func() error {
      key := "foo" + strconv.Itoa(i)
      _, err := client.Do(CommandSET, key, "bar")
      return err
    }()
  }
  err = eg.Wait()
  require.NoError(t, err)
  // Fetch redis key values and assert on expected values
  for i := 0; i < concurrency; i++ {
    key := "foo" + strconv.Itoa(i)
    val, err := redis.String(client.Do(CommandGET, key))
    require.NoError(t, err, key)
    assert.Equal(t, "bar", val, key)
  }
}
```

Figure 7: Few-shot example 3

C Query to fix data race

User: using the above examples to guide you, refactor the below code to fix data race in golang function. two lines involved in the data race:

```
    _, err = decoder.Decode(buf, nil)
    _, err = decoder.Decode(buf, nil)
```

Do not output any other information

```
func TestDecodeBadInput(t *testing.T) {
    _, err := NewDecoder("test_topic", 0, &Config{
        AppID:  "a_test_app",
        Schema: &schema.Config{Type: "type"},
    })
    assert.Equal(t, schema.ErrInvalidType, err)

    _, err = NewDecoder("test_topic", 0, testDataEmitterConfig)
    assert.Error(t, err, "Should report that topic is not found
         ")

    _, err = NewDecoder("hp.event.user", -1,
         testDataEmitterConfig)
    assert.Error(t, err, "Should report that schema is not
         found")

    decoder, err := NewDecoder("hp_demand_job-canceled", 9,
         testDataEmitterConfig)
    require.NoError(t, err)

    var wg sync.WaitGroup
    testFunc := func(decoder Decoder) {
        buf := &bytes.Buffer{}
        buf.Write(headerV2)
        binary.Write(buf, binary.BigEndian, uint16(2))
        binary.Write(buf, binary.BigEndian, uint16(1000))
          . err = decoder.Decode(buf. nil)
         assert.Error(t, err, "Should fail to fetch writer schema")
        wg.Done()
    }
    for i := 0; i < 1000; i++ {
        wg.Add(1)
        go testFunc(decoder)
    }
    wg.Wait()

    _, err = decoder.Decode(&bytes.Buffer{}, nil)
    assert.Error(t, err, "Should fail to parse header")

    buf := &bytes.Buffer{}
    buf.Write(headerV2)
    binary.Write(buf, binary.BigEndian, uint16(10))
    binary.Write(buf, binary.BigEndian, uint16(0))
    _, err = decoder.Decode(buf, nil)
    assert.Error(t, err, "Should fail to fetch meta schema")

    _, err = decoder.Decode(bytes.NewBuffer(b), 123)
    assert.Error(t, err, "Should fail to construct Record")
}
```

OpenAI:

```
func TestDecodeBadInput(t *testing.T) {
  _, err := NewDecoder("test_topic", 0, &Config{
    AppID:  "a_test_app",
    Schema: &schema.Config{Type: "type"},
  })
  assert.Equal(t, schema.ErrInvalidType, err)

  _, err = NewDecoder("test_topic", 0,
       testDataEmitterConfig)
  assert.Error(t, err, "Should report that topic is not
       found")

  _, err = NewDecoder("hp.event.user", -1,
       testDataEmitterConfig)
  assert.Error(t, err, "Should report that schema is not
       found")

  decoder, err := NewDecoder("hp_demand_job-canceled", 9,
       testDataEmitterConfig)
  require.NoError(t, err)

  var wg sync.WaitGroup
  testFunc := func(decoder Decoder) {
    buf := &bytes.Buffer{}
    buf.Write(headerV2)
    binary.Write(buf, binary.BigEndian, uint16(2))
    binary.Write(buf, binary.BigEndian, uint16(1000))
     . localErr := decoder.Decode(buf. nil)
     assert.Error(t, localErr, "Should fail to fetch writer schema")
    wg.Done()
  }
  for i := 0; i < 1000; i++ {
    wg.Add(1)
    go testFunc(decoder)
  }
  wg.Wait()

  _, err = decoder.Decode(&bytes.Buffer{}, nil)
  assert.Error(t, err, "Should fail to parse header")

  buf := &bytes.Buffer{}
  buf.Write(headerV2)
  binary.Write(buf, binary.BigEndian, uint16(10))
  binary.Write(buf, binary.BigEndian, uint16(0))
  _, err = decoder.Decode(buf, nil)
  assert.Error(t, err, "Should fail to fetch meta schema")

  _, err = decoder.Decode(bytes.NewBuffer(b), 123)
  assert.Error(t, err, "Should fail to construct Record")
}
```

Figure 8: Query to fix data race and OpenAI's response