Seminar Paper

Go

Author: Patrik Schweigl
Supervisor: Dr. Dohan Kim
Date: July 12, 2023

Department of Computer Science
University of Innsbruck

Abstract

With the rise of modern computing technologies such as distributed systems and multicore processors and the dominance of networking, and the fact, that there was no major system language released in a while, the need for a programming language has become important. This seminar paper will introduce the programming language Go, a modern, simple and powerful programming language designed to tackle the challenges from the 21st century.

Contents

1 Introduction **1**

2 History **1**
- 2.1 Necessity of Go . . . 1
- 2.2 Development of Go . . . 1
- 2.3 Current state . . . 2

3 Language Details **3**
- 3.1 Variables, Types and Functions . . . 3
- 3.2 Control Flow . . . 4
- 3.3 Structs and Custom Types . . . 5
- 3.4 Concurrency . . . 6
  - 3.4.1 Goroutines . . . 6
  - 3.4.2 Channel . . . 7
- 3.5 Generics . . . 7
- 3.6 Idiomatic Error handling . . . 8

4 Go vs. Other Languages **8**
- 4.1 Package Dependencies: C vs. Go . . . 9
- 4.2 Memory Management: C vs. Java vs. Go . . . 10
- 4.3 Visibility: C vs. Java vs. Go . . . 10
- 4.4 Error handling: C vs. Java vs. Go . . . 11
- 4.5 Functional Programming in Go . . . 12
- 4.6 Object-Oriented Programming in Go . . . 12

5 Conclusion **12**

6 Literature Review **14**

1 Introduction

This paper shows an outline of the programming language Go. In Section 2, we will give a short summary of the History of Go. We will talk about the birth of the language, as well as its subsequent growth in popularity. We also give an overview of its current state of the language. In Section 3 we will explore the unique characteristics of the programming language Go that sets it apart. Although Go is a very simple programming language with only a few reserved keywords, this paper will not provide all features of the language. For those of you who are interested in learning more about the programming language Go, you can take a look at the homepage and learn everything about the standard library [7], the user manual [3] or a guideline for writing effective Go [4]. In Section 4 we will compare Go with other well-known programming languages. In the last Section 5 we will give a summarized of Go and I will provide a personal opinion on Go.

2 History

2.1 Necessity of Go

If we look at the past years, we can see how much the field of technology has changed. We have seen the rise of multicore processors, networking systems and massive computation clusters. However, the problems which were introduced were only worked around instead of solved, making the code less maintainable. This, and the fact, that the code bases at Google hold tens of millions of lines of code and are maintained by thousands of developers, led to several issues. To top that, compilation time was also a factor to not overlook [6].

> "Go is a programming language designed by Google to help solve Google's problems, and Google has big problems." [6]

With this quote in mind, Google started to think of a new programming language

2.2 Development of Go

The development of the programming language Go started in late 2007. Robert Griesemer, Rob Pike and Ken Thompson, three well-known software engineers at Google, started the design of Go. About half a year later in mid 2008, the design work was mostly finished and the first implementations, the compiler and the run-time, started to work. In November 2009, approximately two years after the design start, Google officially announced Go to the public and made Go an Open-Source project. Finally, in March 2012, the first stable go1 version was released [5]. Table 1 shows the brief history of Go as well as some of the most important releases.

Table 1: Brief history of Go

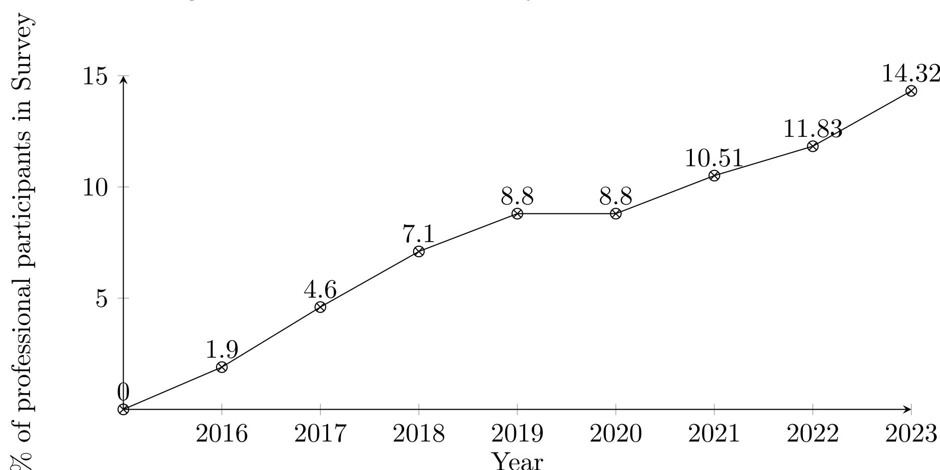
Date	Version
2007	design start by Robert Griesemer, Rob Pike and Ken Thompson
2009	public announcement, Open Source
2012	go1.0 - first stable major version
2022	go1.18 - added generics
2023	go1.20 - Current version
2023	go1.21 - next major release planned in Q3

2.3 Current state

Go1.20 is currently the latest version of Go, with go1.21 expected in Q3. According to the official webpage, many companies such as American Express, Netflix, Twitch, Cloudflare, Dropbox, Docker and many more are using Go. Even databases like CockroachDB and AresDB from Uber are written in Go [2].

> "Seven years along, with a lot of additional core functionality and a great deal of performance profiling under the bridge, we are still very happy with Go." [1]

Figure 1: Stack Overflow Survey from 2015 to 2023



If we look onto the annual Stack Overflow Developer Surveys from 2015 to 2022, an increase of popularity of the programming language Go can be noticed. In the first survey 2015, Go appeared as the 4th most loved programming language, ranking behind Swift, C++11 and Rust [8]. Additionally, Go started to appear as a serious programming language used by professional

developers in the survey from 2016. Although the percentage of 2% out of approximately 26,000 participants is pretty low, this number increased to over 14% out of approximately 67,000 participants over the course of eight years [8, 9, 10, 11, 12, 13, 14, 15, 16].

3 Language Details

In this chapter we will discuss the basic and the unique features of Go will be presented. Go is a statically typed, compiled high-level programming language. It uses a garbage collector to manage the memory automatically. To get started, we will examine one of the most iconic snippets, known as The Hello, World!.

```
1    package main
2    import(
3        "fmt"
4    )
5    func main() {
6        fmt.Println("Hello, World!")
7    }
```

Listing 1: Hello, World! Program in Go

The program begins with the package main declaration, which indicates that this code is part of the main package. Next, we import the `"fmt"` package, which we need to provide formatted print function for input and output. The entry point of the program is the main function. Unlike many other languages like C, C++ or Java, the main function does not have a parameter list and returns void. Inside this function, we call the method `Println` from the package `"fmt"`, which was imported earlier. The `Println` function takes the string "Hello, World!" as a parameter, which will printed out to the console.

3.1 Variables, Types and Functions

In this section we will talk about the fundamental concepts of programming languages: variables, types and functions. Go is a statically typed language, that means that the type of every variable must be known at compile time.

```
1    var myInt int32 = 32 // typed definition
2    var myFloat32 = 64.1234 // type deduction (float64)
3    myString := "Hello" // without var keyword
4    myPointer := &myString // pointer to adress of myString
5
6    const Pi = 3.14159 // constant of type "untyped float"
```

Listing 2: Definitions of variables

Listing 2 provides some variable definitions we will briefly look at. The first definition provides beside the name of the variable and the value also the type.

This definition is fine, but Go allows defining variables much shorter than shown in line three and four. Go uses type inference to receive the type from the assignment. Go also has the pointer type, which we all know from languages such C and C++, but Go omits the pointer arithmetic.

In line six we define a constant variable named Pi. Constants can be either characters, strings, boolean or numeric values such int or float. Numeric constants in Go has no type at the beginning, it will get a type, when you perform explicit conversion or if a function requires a typed variant of it. Constants are very useful, if you want to perform arithmetic, because they are with arbitrary precision.

```
1  func add(a, b int) int {
2      return a + b
3  }
4  func divide(a, b int) (res int, rem int) {
5      res = a / b
6      rem = a % b
7      return
8  }
```

Listing 3: Definitions of functions

Listing 3 shows two simple definitions of functions. The first function `add` takes two parameters, `a` and `b` of type `int`. Note, that `a` does not have an explicit type definition. In that case, both parameters are of the same type, so you are allowed to omit the first type name if the types of the consecutive parameters are the same. To call this function, simply write out the name and provide the parameters to it `add(10, 12);`

The second function `divide` shows a unique feature of Go, Go allows multiple return values. They either can be anonymous with an explicit return, or, in our case, be named. To get the result of the function `divide`, you have to destructure it: `a, b := divide(5, 2);`

Functions in Go often have more than one return value, because in Go errors are handled as return values, which is shown in 3.6.

3.2 Control Flow

This section provides information about the unique features about the two different constructs to handle control flow, branches and loops. In Go as well as many other programming languages, there are two ways to handle conditional branching: the `if` and `switch` statement. The syntax of those two constructs are the same as in C or Java. But they provide extra features, which are shown in Listing 4.

In Go, it is possible to add a preceding statement before the condition in an `if`, a `switch` or a `for` statement. This variable is accessible in the current branch and in all proceeding branches. In other languages you have to define a variable before the `if` statement, which means that the variable has a longer lifetime.

```go
if condition := true; condition {
    // condition is accessible here
}
// condition is not accessible here
switch j := 2; j {
    // j is accessible here
}
```

Listing 4: Structs and Custom Types

While other programming languages do have different keywords for loops like `for`, `foreach`, `do..while` or `while`, Go only has one keyword for it: `for`. This construct can be used in Go to get all the loop variants mentioned above.

3.3 Structs and Custom Types

In this section we will talk about structs and custom types in Go. Structs provide a way to define complex data structures, while custom types providing more clarity.

```go
type Person struct {
    firstname string
    lastname string
}
type Employee struct {
    Person
    Email string
    Department string
}
func (p Person) FullName() string {
    return p.firstname + " " + p.lastname
}
```

Listing 5: Structs and Custom Types

Listing 5 provides examples, how to define structs and provide methods for those. Line one to four defines a `struct` with two members: firstname and lastname. A struct is a simple data structure which can contain a collection of fields and embeds other structs, but not functions. Line 5 to 9 defines a `struct Employee`, which embeds the earlier defined `struct Person`. That means, that every `Employee` now holds all fields of `Person` a can be safely converted into it. As I said, structs do not allow function definitions inside a struct. In Go, you can define a method for a receiver type as shown in line ten to twelve. `(p Person)` is called the receiver type for the function `FullName`. This allows us, to simply call the function `person.Fullname()` instead of `FullName(person)`, which is much cleaner especially if you chain functions.

Listing 6 shows two examples, how to define custom types. Custom types are commonly used, to provide clarity.

```go
type Duration int
type IPv4Addr [4]byte

func (ip IPv4Addr) IPv4AddrToString() string {
    return fmt.Sprintf("%d.%d.%d.%d", ip[0], ip[1], ip[2], ip[3])
}
```

Listing 6: Structs and Custom Types

3.4 Concurrency

In this section we will dive into the concurrency feature of Go. Concurrency is a powerful mechanism, which allows running code simultaneously. If utilized correctly, it increases the performance of the program and can maintain interactivity, because the UI-thread does not block. Concurrency is one of its unique features which sets it apart from other programming languages. Go uses a variant of Communicating sequential process CSP [6]. CSP was introduced by Tony Hoare in the late 1970s and has been widely used in concurrent programming languages.

3.4.1 Goroutines

Goroutines are lightweight threads, which executes a given function concurrently.

```go
func print(s string) {
    for i := 1; i <= 5; i++ {
        time.Sleep(100 * time.Millisecond)
        fmt.Println(s, ": ", i)
    }
}
func main() {
    go print("Hello from 1st")
    go print("Hello from 2nd")

    time.Sleep(2 * time.Second)
}
```

Listing 7: Simple goroutine in Go

In the above Listing 7, we define a function `print`, which prints out a given string to the output 5 times with a delay of 100ms. If we call this function as a normal function, the main thread would block for the whole duration. To call this function as a goroutine, simply put the `go` keyword in front of the function name, as shown in line eight and nine. Because the goroutines are non-blocking by default, we have to wait for the goroutines to finish.

3.4.2 Channel

In addition to goroutines, Go provides a mechanism for communication and synchronization between goroutines: Channel. A channel is a typed pipe, which can send and receive values between goroutines. To write into and read from a channel, you have to use the `<-` operator.

```
1  func main() {
2      ch := make(chan int)
3      go func() {
4          ch <- 42
5      }()
6      value := <-ch
7      fmt.Println(value)
8  }
```

Listing 8: Simple channel in Go

In this Listing 8, we create a channel called `ch` with the built-in function `make(t Type, size ...int) Type`. We do not provide a size parameter, which means that the channel is an unbuffered channel. With the size parameter it is possible to make a buffered channel. Then we provide an anonymous goroutine, which writes the value `42` into the channel `ch`. Finally, we read the value from that channel and print it out.

3.5 Generics

Generics are a fairly new feature in Go, introduced 2022 with version go.1.18. Generics are one of the most requested features in Go, that has been missing from the language. They allow the developer to define a more general and reusable code that work with different types.

```
1      func Swap[T any](a, b *T) {
2          tmp := *a
3          *a = *b
4          *b = tmp
5      }
```

Listing 9: Generics

In Listing 9 we defined a generic `Swap` function, which swaps the content of two pointers. This function is defined with a type parameter `T`, which can by of `any` type. `Any` type is a type alias for the type `interface{}`, which serves the same purpose as the `object` type in Java. To call this function, we can either call the function explicitly typed `Swap[int](&x, &y)`, or let the compiler inference the type `Swap(&x, &y)`.

Generics are not limited to functions, they can be used with structs, interfaces and methods as well.

3.6 Idiomatic Error handling

Lastly, this section shows the idiomatic error handling, which is widely used in Go. There can be caused by various reasons, such as I/O errors, incorrect inputs, memory related errors, to name a few. Normally, you do not want your program to crash at such situations, you want to handle them. That is the reason why error handling is so crucial for every programming language. In Listing 3 we defined a function `divide(a, b int) (res int, rem int)`, which does not handle division by 0. Let's fix that.

```
1   func divide(a, b float64) (float64, error) {
2       if b == 0 {
3           return 0, errors.New("Cannot divide by 0!")
4       }
5       return a / b, nil
6   }
7   func main() {
8       if res, err := divide(12, 0); err != nil {
9           fmt.Println(err.Error())
10      } else {
11          fmt.Println(res)
12      }
13  }
```

Listing 10: Idiomatic error handling in Go

In Listing 10, we redefined the function `divide`, to take two `float64` parameters and return the result of the division and an error object. If the second parameter is different from zero, we return the result of the division and a `nil` object. `Nil` does have the same meaning as `null` have in other languages, it is the zero value for pointers, structs, channels and interfaces. If it is zero instead, the result is typically the zero value of its type and an error object is created with `Error.New(text string)`. Now that we have a stable divide function, we need to check, if an error is rose. In line eight we check, if the error object is nil, which indicates that the function works as expected. Otherwise, we may need to investigate what was the error cause and try to recover from that.

Sometimes, if an error is unrecoverable, Go has a construct named `panic`. If a `panic` occurs in `main`, the program exits with a non-zero exit code. In a goroutine a `panic` stops the goroutine from running and terminates the goroutine.

4 Go vs. Other Languages

In this section we will look onto other well-known programming languages as well as popular programming paradigms like Object-Oriented Programming and Functional Programming.

4.1 Package Dependencies: C vs. Go

In this section we will compare the dependency management in C and Go, which has a huge impact on compilation time.

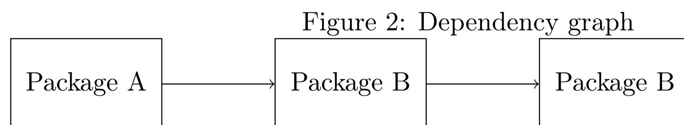
In C, dependencies are grouped in so-called header files. Every source file, which will references something outside its own package, has to include those header files, usually located on top of the file. If we compile our program, the first step is the preprocessing step. The Preprocessor replaces all the `\#include` directives with the corresponding source code. This may lead to multiple inserts of the same header file. To prevent redefinition errors, C has its own `\#ifndef` guard pattern as shown in Listing 11. These #define guards allow the developer to reorder the #include directives. On the other hand this could lead in big code bases to many redundant `\#include` directives, which will slow down the preprocessor step as well as all following steps afterwards. Let's take a look at the program ps, a program which will scan all running processes. The compilation of the source code included the `<sys/stat.h>` header file 37 times [6].

```
1   #ifndef HEADER_H
2   #define HEADER_H
3   // definitions, declarations goes here
4   #endif
```

Listing 11: Define Guards in C

Now that we know how C handles dependencies, let's have a look, how Go handles dependencies. One of the big improvements is, that every unused import in Go is a compile-time error. This is a nice upgrade, which prevents the program from increasing in size for no reason. Now let's have a look at Figure 2. We have three packages: package A includes package B, and B includes package C. Furthermore, package A has a transitive dependency to C over B. If we compile this program, the dependency graph recognizes, that package C has to be compiled first and generates an C.o file. Next, package B is compiled, and it read all the necessary information from the object file C.o, not C.go. Lastly, package A gets compiled, and it gets all the necessary information from B.o. The compiler does not need to access either B.go or C.o. This is the reason, why Go has a far superior compilation time than C or C++ [6].

Figure 2: Dependency graph



4.2 Memory Management: C vs. Java vs. Go

Memory management vary across many programming languages. In this section we will compare the different strategies of managing memory in C, Java and Go.

In C, the developer is fully responsible in acquiring and releasing memory. To acquire memory from the underlying operating system, the developer has to use the function `void *malloc(size_t size)`, which will, on success, reserve a block of `size_t` and return a pointer, which points on the first segment. To release this block, the function `void free(void *ptr)` has to be called on the pointer which points to the memory block. This flexible memory management comes with risks: memory leaks and so-called dangling pointers, if the developer does not manage the memory carefully.

In Java, there is an automatic memory management tool named garbage collector. The garbage collector runs in its own thread and will be periodically scan the memory for unreachable objects. This algorithm is fully automatic, which means the developer does not need to manage the memory manually at any point. Although the developer can call the Garbage Collector manually through `System.gc()`, this is not a reliable way to handle the GC manually.

Go uses the same approach as Java. Although a garbage collector might be a debatable feature in a system programming language, the design team around Go decided that it would be more beneficial to implement it [6].

Table 2: Advantages and Disadvantages of GC

Advantages	**Disadvantages**
• Developer does not need to worry about memory management, increased productivity • Reduces memory related bugs like dangling pointer and memory leaks • Automatic memory management can lead to better memory usage through dynamic memory allocation	• Performance overhead due to periodically running GC • GC calls my lead to latency issues through pause times • GC is usually non-deterministic, hard to predict when memory is free • Memory footprint is generally higher • GC is more difficult to implement

4.3 Visibility: C vs. Java vs. Go

In terms of visibility, C, Java and Go offers different approaches.

In C, header files are typically included into source files. Those header files provide declaration of functions, definitions of variables, macros and structs. These imports are generally replaced with its content when the program is compiled. Thus, every declaration in those header files is publicly available. C does not have the concept of packages or namespaces, which sometimes lead to potential naming conflicts.

Java uses access modifiers such as public, protected and private for visibility control. The modifier public is used, when the class or class member should be publicly accessed. Protected is used to limit the access to the same package and subclasses. Private, for the sake of completeness, is used, to restrict the access to the defining class. With these modifiers the developers can maintain a flexible level of encapsulation on package or class level.

Go, on the other hand, uses a simple rule to maintain package-level visibility. Variables, structs, types, constants, methods, functions and interfaces are publicly visible, if they start with an uppercase letter. Lowercase identifiers and identifiers which start with an underscore, on the other hand, are only accessible inside the same package. This is a clever approach to keep the language simpler. However, if you want to change the visibility of identifiers, you have to rename them and change all references within the package. Listing 12 provides some examples that show the visibility in Go.

```go
const Pi = 3.14159 // exported
const e = 2.71828 // not exported

type Point3D struct { // user defined struct gets exported
    X, y, _Z float64 // member X gets exported, y and _Z are not
}
```

Listing 12: Visibility in Go

4.4 Error handling: C vs. Java vs. Go

This section will compare the error handling of C and Java with Go.

First, C does not have a built-in feature for error handling. Fortunately, there are more or less standard ways, to deal with errors in C. The most common one is a concise return value, i.e. `NULL` or `-1`. This only works, if `NULL` is a valid return value or `-1` does not make sense context wise. A solution to this would be a return value, which indicates the state, and a pointer parameter, where the result of the function can be stored. We can take this divide function as an

example `int divide(float a, float b, float* result)`. Other ways are global states, such in `#include <errno.h>`

Java, on the other hand, has exception handling. Exceptions are objects, which can be thrown. If such an exception is thrown, the normal execution is interrupted and the program continues in the exception handler. In Java, the exception handler is called a `catch` block. If there is no surrounding handler, the program terminates. This mechanism provides a powerful tool, when it comes to handle errors, but comes with a major drawback. Exceptions can cause performance overhead due to so-called stack-unwinding.

The idiomatic error handling mechanism in Go is explained in detail in 3.6. It is hard to compare error handling in Go with C, because C does not have a standardized way. The way, Go handles errors, is completely different to Java.

4.5 Functional Programming in Go

While Go is not a functional programming language (FP), Go supports different functional programming aspects, which enables FP to some degree. Go supports first-class functions, where functions can be assigned to variables. This can be very handy, especially, when paired with anonymous functions, also called closures in Go. Go also supports higher-order functions, where functions can be function arguments. Famous examples of higher-order functions are `map`, `filter`, `reduce` and `fold`. It is also a good advice to keep your business functions pure, to maintain testability. However, Go was not designed to be a pure functional programming language.

4.6 Object-Oriented Programming in Go

Although Go is not a traditional object-oriented programming (OOP) language, Go offers features to achieve object-oriented programming to some degree. Classical OOP languages like Java and C# rely on class hierarchies and inheritance, Go on the other hand uses composition over inheritance. Go uses structs and methods with a receiver type to get the similar behavior to classes. Additionally, Go supports struct embedding, which allows code reuse. To enable Polymorphism, Go utilizes interfaces.

5 Conclusion

This seminar paper holds an overview of the programming language Go and unique features. We discussed the history of Go, which started way back in 2007 as a project at Google. Since then, Go has grown to a widely accepted and used programming language.

Overall, Go was a very easy programming language to learn. Due to its simple and C like syntax, which I am very familiar with, I had a great time learning Go. Another aspect, which I appreciated, is that the Go developer team offers such great documentation. This documentation offers the best way to

learn the language. Go offers some interesting and unique features to learn, like concurrency, composition over inheritance, the package system, how methods are implemented, just to name some. However, there are some design decisions, which I miss or which I do not like. Personally, I like the manual memory management over the garbage collector. Another point, which I do not like, is the idiomatic error handling in Go.

6 Literature Review

References

[1] Jessica Edwards. Why go was the right choice for cockroachdb, 2022. Last accessed 15 June 2023. URL: https://www.cockroachlabs.com/blog/why-go-was-the-right-choice-for-cockroachdb.

[2] The Go Team. Case studies. Last accessed 15 June 2023. URL: https://go.dev/solutions/case-studies.

[3] The Go Team. Documentation. Last accessed 15 June 2023. URL: https://go.dev/doc/.

[4] The Go Team. Effective go. Last accessed 15 June 2023. URL: https://go.dev/doc/effective_go.

[5] The Go Team. Release history. Last accessed 15 June 2023. URL: https://go.dev/doc/devel/release#go1.

[6] The Go Team. Go at google: Language design in the service of software engineering, 2012. Last accessed 15 June 2023. URL: https://go.dev/talks/2012/splash.article.

[7] The Go Team. Standard library, 2023. Last accessed 15 June 2023. URL: https://pkg.go.dev/std.

[8] The SO Team. 2015 developer survey. Last accessed 15 June 2023. URL: https://insights.stackoverflow.com/survey/2015.

[9] The SO Team. 2016 developer survey. Last accessed 15 June 2023. URL: https://insights.stackoverflow.com/survey/2016.

[10] The SO Team. 2017 developer survey. Last accessed 15 June 2023. URL: https://insights.stackoverflow.com/survey/2017.

[11] The SO Team. 2018 developer survey. Last accessed 15 June 2023. URL: https://insights.stackoverflow.com/survey/2018.

[12] The SO Team. 2019 developer survey. Last accessed 15 June 2023. URL: https://insights.stackoverflow.com/survey/2019.

[13] The SO Team. 2020 developer survey. Last accessed 15 June 2023. URL: https://insights.stackoverflow.com/survey/2020.

[14] The SO Team. 2021 developer survey. Last accessed 15 June 2023. URL: https://insights.stackoverflow.com/survey/2021.

[15] The SO Team. 2022 developer survey. Last accessed 15 June 2023. URL: https://survey.stackoverflow.co/2022.

[16] The SO Team. 2023 developer survey. Last accessed 15 June 2023. URL: `https://survey.stackoverflow.co/2023`.