The Go Programming Language and Concurrency

Comparing Go's Concurrency Implementation to Other Popular Languages

James Palmer
Department of Computer Science and Information Systems
University of North Alabama
Florence, Alabama, United States
jpalmer8@una.edu

INTRODUCTION

The purpose of this work is to introduce the reader to the Go programming language, explain its history, give a brief overview of the language's features, review literature related to the topic, and then address the given research question. The research question that will be discussed is: how does Go support concurrency, and how does that implementation compare to other major languages in terms of performance, scalability, and ease of use?

BACKGROUND

1 Historical Background

The main contributors to the creation of Go were Robert Griesemer, Rob Pike, and Ken Thompson, all of whom worked at Google at the time. Work began on Go in 2007, and the language was released to the public in 2009. Since the language's public release, Go has been an open source project. Figure 1 contains a graph displaying older languages that influenced the initial design of Go.

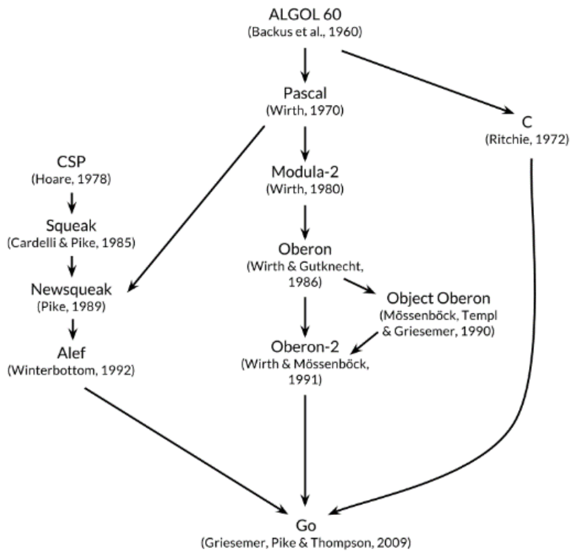


Figure 1: Diagram depicting the languages that influenced Go

The largest and most obvious influence was C/C++, with Go inheriting C's "expression syntax, control-flow statements, basic data types, call-by-value parameter passing, pointers, and above all, C's emphasis on programs that compile to efficient machine code and cooperate naturally with the abstractions of current operating systems" [4].

The creation of Go was spurred by a multitude of challenges that Google developers and engineers faced with the languages that they used previously, with C++ being the main culprit. C++ was seen as being wildly inefficient for large software products that included a large number of files. The proverbial 'straw that broke the camel's back' occurred in 2007, when 4.2 MB worth of C++ source code required the C++ preprocessor to access more than 8 GB worth of data. That means that for every byte of C++ source code, 2000 bytes of data were being accessed by the preprocessor and delivered to the compiler. Additionally, compilation of that C++ code took 45 minutes to complete. In 2012, after the code base had been converted to Go, compilation required much less data access and took only 27 minutes to complete, even though the source code was substantially longer.

Another major factor for the creation of Go was that, during its creation, multi-core systems were becoming increasingly popular. Unfortunately, programming languages and operating systems of the time had not quite caught up to the advances of multi-core systems, which made most concurrency implementations rather obtuse and heavyweight. Robert Griesemer, Rob Pike, and Ken Thompson, knowing that multi-core systems would only become more commonplace as time passed, decided that Go would need to have a focus on concurrency to take advantage of multi-core systems. This incentive becomes increasingly apparent when considering what kind of software Google produces: programs for large and distributed systems, including servers. Before converting to Go, some systems at Google ran multiple complete instances of the same program to handle requests since it was too challenging or inefficient to create a single, threaded program that could handle the number of requests that Google was processing.

Since its creation, Go has become an increasingly popular programming language that has been used on a wide variety of major projects in addition to the multitude of programs made by Google themselves. Companies like X (formerly

Twitter), Uber, Twitch, Netflix, and Meta all use software written in Go. According to the April 2024 TIOBE Index, Go is the seventh most popular programming language. That is a fairly significant improvement over Go's ranking only one year ago, when the TIOBE Index for April 2023 had Go ranked as the tenth most popular programming language.

2 Language Overview

Go, sometimes called Golang, is a multi-paradigm, general-purpose language, but it has a focus on systems programming and concurrency. The overall syntax of Go is a slight variation on Extended Backus-Naur Form (EBNF). Unsurprisingly, C++, Java, and Python all implement syntax that is a variation of EBNF. Thankfully, this shared attribute means that learning Go can be fairly straightforward for someone who has already learned a language with similar syntax.

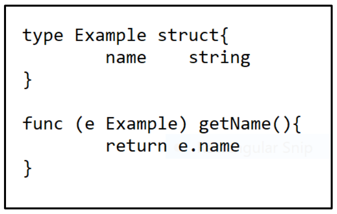
Go supports a wide range of data types. All variables in Go are statically typed, but the data type does not necessarily have to be explicitly written in a variable's declaration. The "bool" data type allows for Boolean values, represented as "true" and "false". The numeric types cover integers, floating point numbers, and complex numbers. The predefined numeric types are:

• uint8	unsigned 8-bit integer
• uint16	unsigned 16-bit integer
• uint32	unsigned 32-bit integer
• uint64	unsigned 64-bit integer
• int8	signed 8-bit integer
• int16	signed 16-bit integer
• int32	signed 32-bit integer
• int64	signed 64-bit integer
• float32	signed 32-bit floating point number
• float64	signed 64-bit floating point number
• complex64	real float32 part and imaginary part
• complex128	real float64 part and imaginary part
• byte	another name for uint8
• rune	another name for int32

Additionally, using either "uint" or "int" without any number to specify the size will default to the word size of the machine that compiles the program.

Go also implements a "string" data type that works similarly to other languages, with each character of the string occupying one byte of memory. Strings in Go do not use a null-terminating character. Go has an "array" data type very similar to the data structure of the same name in C and C++, with every element being the same data type and its size being explicitly declared and unable to change during program execution. Go's "slice" data type, however, does allow for its size to change dynamically. The length of a slice is defined as the number of elements it contains. The capacity of a slice is the maximum number of items the slice may hold before more memory must be allocated. Slices work by dynamically allocating an array, so exceeding the capacity, which is the size of the underlying array, requires that a new underlying array be allocated that is large enough to store all of the slice's elements. The first index of a string, array, or slice is always 0, and its last index is always one less than its length.

Go has a data structure called a "struct" that is not the same as a struct in C/C++. In fact, a struct in Go is much closer in design to classes in C++ or Java because structs allow for the inclusion of methods. Like classes, Go's structs consist of an aggregation of other data types and methods that can perform operations with or upon that data. The names of the data fields dictate privacy, with names starting with a capital letter being "exported", also known as public. On the other hand, field names starting with a lowercase character are private or "unexported" and cannot be directly accessed outside of the struct. Exported data fields can be accessed like so: structName.fieldName. A struct's methods are not declared inside of the struct declaration like one may expect if they have programmed in other object-oriented languages. Instead, methods are declared and defined after the struct using syntax very similar to creating a function in Go. Below is a short example to demonstrate:



```
type Example struct{
        name    string
}

func (e Example) getName(){
        return e.name
}
```

Figure 2: Simple method example

The added parameter field before the function name corresponds to the calling object. Unlike C++ and Java's "this" or Python's "self", Go allows the programmer to give the calling object a method-specific name. One way in which Go's structs are different from the classes of many object-oriented languages is that there is no mechanism for inheritance.

Go allows for the use of pointers. A pointer is written as an asterisk followed by the data type it is intended to point to. For example, "pointer *uint" would create a variable named "pointer" that is a pointer to an unsigned integer. Uninitialized pointers are set to "nil", which is Go's equivalent to "NULL" in C/C++.

A "map" in Go is a data structure that stores an unordered set of key-value pairs of user-defined types. The variable that refers to a map stores a pointer to the map in memory. Similar to a slice, maps are dynamically allocated and may change in size during program execution. Below is a short example to demonstrate a map:

```
exampleMap := map[string]int{
        "John": 38,
        "Mary": 21,
}
```

Figure 3: Simple map example

The above map has been initialized with two key-value pairs, with the key being of string type and the value being of integer type.

Like many other languages, Go supports a large number of operators. For the sake of brevity, only those that are different from what one would reasonably expect will be discussed. Go implements an operator specifically for declaring and initializing a variable, which is ":=". Interestingly, the use of := is not required; "example := 20" and "var example = 20" are equivalent expressions. In the area of bitwise operators, Go uses a curious "&^" operator, which is defined as AND NOT. For example, "x &^ y" can be written as "x and not y" or "x and ~y". Since Go allows for the use of pointers, the "&" and "*" operators as seen in C/C++ also work similarly in Go.

Concurrency, as previously stated, played a large factor in the development of Go. Because of this, Go has a rather unique approach to how concurrency is supported by the language. The most important concurrency feature is "goroutines", which are extremely lightweight threads. Goroutines are given only a few kilobytes of memory at creation and can be dynamically resized as needed, meaning that effective implementation of goroutines can allow for thousands of them to be run independently of each other. In fact, programs on servers at Google may have millions of goroutines written into them. Goroutines are capable of sending values to each other using "channels". These channels can be set to be bidirectional, receivers, or senders. Of course, if a channel is directional, that means that every goroutine other than the one that created the channel has the opposite permissions. For example, if a channel is declared in the main function and set to receive only, then every goroutine other than main can only send data to it, and only main can receive from it. Bidirectional channels allow for all goroutines to send data to and receive data from the channel. If a channel is given a capacity, that capacity is used as the size of the buffer allocated for the channel, meaning that data transfers can be buffered to improve performance. If no capacity is given, both the receiving and the sending goroutine must be ready to perform the data transfer, and it will be blocked otherwise. The size of a channel's buffer cannot change during program execution, and channels are statically typed.

In order to address the issues Google encountered with the large amount of data that was accessed by including libraries in C/C++ programs, Go has a package system that focuses on memory efficiency. In fact, every Go program is a collection of one or more packages. As can be seen in Figure 4 and Figure 6, all packages used in a program must be explicitly imported at the beginning of a source code file. In order to prevent reading a large number of files during compilation that is caused by a chain of dependencies, every part of an imported package used in the source code being compiled will be included in the compiled program. So, importing the "main" package defined by the program in Figure 6 would inherently import any part of the fmt, math/rand, sync, and time packages used without the need for the compiler to access those packages themselves to retrieve their data. This package system greatly decreases the amount of files that the compiler must access when compiling a new program that has a large number of dependencies when compared to C/C++, which means that compilation of a Go program is significantly faster than compilation of a C/C++ program.

It is also important to note that Go uses a garbage collector during program execution. The garbage collector runs concurrently to the main program when there is a CPU core available to be dedicated to that role. Without the need for the programmer to manually manage memory deallocation, Go implements a fool-proof improvement to security by removing the possibility of hanging pointers.

3 Literature Review

Despite Go being relatively new, there is a large amount of literature about the language. The first source one should turn to when attempting to learn about Go is The Go Programming Language Specification. This source contains a concise description of every feature offered by the language by default. Additionally, its opening sections describe the grammar upon which the language was built. For a more in-depth understanding of Go, there may be no source more valuable than *The Go Programming Language* written by Alan Donovan and Brian Kernighan. All 366 pages of content provided in that book are incredibly helpful for anyone looking to learn to write in Go.

The research question presented in this paper was greatly influenced by the paper "Concurrency Analysis of Go and Java" written by Ahbinav P Y, Avakash Bhat, Christina Terese Jospeh, and K Chandrasekaran. That paper analyzes the performance of Go and Java while using concurrent execution of matrix operations that are much more complex than the pseudorandom number generation and addition used in this paper's tests. In fact, "Concurrency Analysis of Go and Java" examines performance differences as the complexity of each individual thread increases rather than examining performance when the number of threads increases. Interestingly, the difference in methodology did not produce drastically different results: Java proved to perform better when the scale of concurrency passes a certain threshold.

METHODOLOGY

Clearly, concurrent execution was a major influence over the design of Go. Naturally, curiosities arise as to whether or not Go truly is an improvement over the languages that came before it and those it competes against today. The languages that will be compared to Go for the sake of this research will be C++, Java, and Python. While not all of these languages support concurrency directly, libraries are readily available that do. A functionally identical program will be written in all four languages. Additionally, all four programs will run on the same hardware, which uses an 11th Gen. Intel i7 CPU running at 2.8 GHz and 16 GB of RAM running at 3200 MHz. While testing each program, only the absolute minimum number of other programs will be running on the computer.

Performance can be measured objectively, but scalability and ease of use can be seen as subjective measures of a language. Because of this, certain objective measures will be used that are directly related to both scalability and ease of use. For the sake of brevity, the factors considered will be the number of characters required to write each of the four programs and how much extra writing is required to double the number of concurrent processes the programs will create. One major factor of ease of use that will be used as a metric is similarity to pseudocode. One block of pseudocode will be used to describe the behavior of all four programs, meaning that all four programs will be designed from identical pseudocode. The similarity between the source code written in each language and the pseudocode will be used to measure ease of use, since it can easily be argued that high similarity to the pseudocode proves the language's syntax to be more intuitive. Performance will be measured by running every program five times, recording the execution time of each run of the program. The execution times will then be used to calculate an average execution time for each program.

Below is the pseudocode that will be used to write all four programs:

```
•  Save the start time to startTime for measuring
   execution time
•  Initialize overallSum to 0 and numThreads to 100
•  For numThreads iterations:
   o  Spawn a thread
   o  In each thread, generate 1000000 random
      numbers in the range [1, 1000000]
   o  Sum together random numbers produced in the
      thread
   o  Send thread's sum back to main thread
•  Add every thread's result to overallSum
•  Display the execution time
```

The pseudocode is arbitrary in what it performs. Its only purpose is to demonstrate the time required to spawn threads, have all threads perform some computation, and then combine the results of every thread's computation. Below is the code for all four programs:

```go
package main

import (
	"fmt"
	"math/rand"
	"sync"
	"sync/atomic"
	"time"
)

func threadFunc(overallSum *int64, wg *sync.WaitGroup){
	rng := rand.New(rand.NewSource(time.Now().UnixNano()))
	threadSum := 0
	for i := 0; i < 100000; i++{
		threadSum += rng.Intn(1000000) + 1
	}
	atomic.AddInt64(overallSum, int64(threadSum))
	wg.Done()
}

func main(){
	startTime := time.Now()
	var overallSum int64 = 0
	numThreads := 100
	var wg sync.WaitGroup
	wg.Add(numThreads)
	for i := 0; i < numThreads; i++{
		go func(){
			threadFunc(&overallSum, &wg)
		}()
	}

	wg.Wait()

	endTime := time.Now()
	duration := endTime.Sub(startTime).Seconds()
	fmt.Printf("Execution time: %f seconds\n", duration)
}
```

Figure 4: Concurrent code for 100 threads in Go using atomic operations

```python
import threading
import random
import time

def thread_func(overall_sum, lock):
    rng = random.Random()
    thread_sum = sum(rng.randint(1, 1000000) for _ in range(100000))

    with lock:
        overall_sum[0] += thread_sum

start_time = time.time()
overall_sum = [0]
num_threads = 100
lock = threading.Lock()
threads = []

for _ in range(num_threads):
    thread = threading.Thread(target=thread_func, args=(overall_sum, lock))
    thread.start()
    threads.append(thread)

for thread in threads:
    thread.join()

end_time = time.time()
duration = end_time - start_time

print(f"Execution time: {duration:.6f} seconds")
```

Figure 5: Code for 100 "threads" in Python

```go
package main

import (
	"fmt"
	"math/rand"
	"sync"
	"time"
)

func threadFunc(sum chan<- int, wg *sync.WaitGroup){
	rng := rand.New(rand.NewSource(time.Now().UnixNano()))
	threadSum := 0
	for i := 0; i < 100000; i++{
		threadSum += rng.Intn(1000000) + 1
	}
	defer wg.Done()
	sum <- threadSum
}

func main(){
	startTime := time.Now()
	overallSum := 0
	numThreads := 100
	sums := make(chan int, numThreads)
	var wg sync.WaitGroup
	wg.Add(numThreads)
	for i := 0; i < numThreads; i++{
		go func(){threadFunc(sums, &wg)}()
	}

	wg.Wait()
	close(sums)

	for s := range sums{
		overallSum += s
	}

	endTime := time.Now()
	duration := endTime.Sub(startTime).Seconds()
	fmt.Printf("Execution time: %f seconds\n", duration)
}
```

Figure 6: Concurrent code for 100 threads in Go using a channel

```cpp
#include <iostream>
#include <thread>
#include <vector>
#include <random>
#include <chrono>
#include <future>

int threadFunc() {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<int> dist(1, 1000000);

    int threadSum = 0;
    for (int i = 0; i < 100000; i++) {
        threadSum += dist(gen);
    }

    return threadSum;
}

int main() {
    auto startTime = std::chrono::high_resolution_clock::now();

    int numThreads = 100;
    std::vector<std::future<int>> futures;

    for (int i = 0; i < numThreads; i++) {
        futures.push_back(std::async(std::launch::async, threadFunc));
    }

    int overallSum = 0;
    for (auto& future : futures) {
        overallSum += future.get();
    }

    auto endTime = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> duration = endTime - startTime;
    std::cout << "Execution time: " << duration.count() << " seconds\n";

    return 0;
}
```

Figure 7: Concurrent code for 100 threads in C++

```java
import java.util.Random;
import java.time.Duration;
import java.time.Instant;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.Callable;

class SumTask implements Callable<Long> {
    @Override
    public Long call() {
        Random rand = new Random();
        long threadSum = 0;
        for (int i = 0; i < 100000; i++) {
            threadSum += rand.nextInt(1000000) + 1;
        }
        return threadSum;
    }
}

public class Main {
    public static void main(String[] args) throws Exception {
        Instant startTime = Instant.now();

        int numThreads = 100;
        long overallSum = 0;

        ExecutorService executor = Executors.newFixedThreadPool(numThreads);
        Future<Long>[] futures = new Future[numThreads];

        for (int i = 0; i < numThreads; i++) {
            futures[i] = executor.submit(new SumTask());
        }

        for (int i = 0; i < numThreads; i++) {
            overallSum += futures[i].get();
        }

        executor.shutdown();
        Instant endTime = Instant.now();
        double duration = Duration.between(startTime, endTime).toNanos() / 1e9;
        System.out.printf("Execution time: %.6f seconds\n", duration);
    }
}
```

Figure 8: Concurrent code for 100 threads in Java

Once this code was generated for all four languages, new source code files were made that doubled the number of threads to examine how many edits needed to be made, the change in the size of the executable generated by compilation, and the effect the change would have on program performance.

It should be noted that two different programs were written in Go: one that uses mutual exclusion in the form of atomic operations and one that uses a buffered channel for sending data from the thread back to the main program. It is also important to keep in mind that since Python is an interpreted language, "threading" acts more as asynchronous execution rather than parallel execution. Even though the threading library was used, the execution of the Python program is not actually concurrent.

RESULTS/CONCLUSIONS

The sizes of the source code files are as follows:

- Python: 690 bytes
- C++: 1,021 bytes
- Java: 1,333 bytes
- Go (using a channel): 759 bytes
- Go (using mutex): 729 bytes

The language that required the least amount of characters to implement the aforementioned pseudocode was Python. Additionally, the code of the Python program appears to be the closest to the pseudocode, making it the best in terms of ease-of-use. Because of how all four programs were written, making changes to double the number of threads created was a simple process and was accomplished by changing the variable numThreads from 100 to 200.

It is also important to note the sizes of the executable files created by the compilation of the C++ program and the two Go programs. Their sizes are:

- C++: 3,507 KB
- Go (using a channel): 2,083 KB
- Go (using mutex): 2,082 KB

Therefore, the executable produced by the C++ program is approximately 164% the size of the executables produced by the Go programs. It is also clear that choosing between the use of a channel or atomic operations does not cause a major change in the size of the executable produced.

The execution times of the Go program that creates 100 threads and uses atomic operations are as follows:

- Run 1: 0.025479 seconds
- Run 2: 0.020905 seconds
- Run 3: 0.023680 seconds
- Run 4: 0.023649 seconds
- Run 5: 0.020634 seconds
- Average: 0.022869 seconds

The execution times of the Go program that creates 100 threads and uses a channel are as follows:

- Run 1: 0.020090 seconds
- Run 2: 0.020904 seconds
- Run 3: 0.020700 seconds
- Run 4: 0.020059 seconds
- Run 5: 0.020051 seconds
- Average: 0.020361 seconds

The execution times of the C++ program that creates 100 threads are as follows:

- Run 1: 0.089805 seconds
- Run 2: 0.079013 seconds
- Run 3: 0.081624 seconds
- Run 4: 0.076334 seconds
- Run 5: 0.080098 seconds
- Average: 0.081375 seconds

The execution times of the Java program that creates 100 threads are as follows:

- Run 1: 0.098659 seconds
- Run 2: 0.105611 seconds
- Run 3: 0.117862 seconds
- Run 4: 0.097875 seconds
- Run 5: 0.092549 seconds
- Average: 0.102511 seconds

The execution times of the Python program that creates 100 threads are as follows:

- Run 1: 6.667386 seconds
- Run 2: 6.128670 seconds
- Run 3: 5.989118 seconds
- Run 4: 5.869446 seconds
- Run 5: 5.738530 seconds
- Average: 6.078630 seconds

The fastest program tested with 100 threads was the Go program that used a buffered channel. Python produced the slowest program by far, with it taking about 299 times as long to complete as the fastest Go program.

The execution times of the Go program that creates 200 threads and uses atomic operations are as follows:

- Run 1: 0.042826 seconds
- Run 2: 0.043395 seconds
- Run 3: 0.043877 seconds
- Run 4: 0.044550 seconds
- Run 5: 0.044488 seconds
- Average: 0.043827 seconds

The execution times of the Go program that creates 200 threads and uses a channel are as follows:

- Run 1: 0.042529 seconds
- Run 2: 0.045351 seconds
- Run 3: 0.045639 seconds
- Run 4: 0.043419 seconds
- Run 5: 0.044323 seconds
- Average: 0.044252 seconds

The execution times of the C++ program that creates 200 threads are as follows:

- Run 1: 0.150066 seconds
- Run 2: 0.150900 seconds
- Run 3: 0.152880 seconds
- Run 4: 0.154529 seconds
- Run 5: 0.151967 seconds
- Average: 0.152068 seconds

The execution times of the Java program that creates 200 threads are as follows:

- Run 1: 0.151757 seconds
- Run 2: 0.121601 seconds
- Run 3: 0.126763 seconds
- Run 4: 0.157567 seconds
- Run 5: 0.116215 seconds
- Average: 0.134781 seconds

The execution times of the Python program that creates 200 threads are as follows:

- Run 1: 12.251883 seconds
- Run 2: 12.047211 seconds
- Run 3: 11.844175 seconds
- Run 4: 12.066260 seconds

- Run 5: 12.248680 seconds
- Average: 12.0916418 seconds

Interestingly, increasing the number of threads to 200 made the Go program that uses atomic operations to be the fastest rather than the program that uses a channel. Also, Java proved to be faster than C++ when the number of threads increased.

The increase in average execution time between the 100-thread version and 200-thread version of each program are as follows:

- Go (using mutex): 91.6% increase
- Go (using a channel): 117.3% increase
- C++: 89.9% increase
- Java: 31.5% increase
- Python: 98.9% increase

By those metrics, Java is the most scalable because it experienced the smallest decrease in performance when the number of threads increased.

After these tests, Go has proven to perform the best during concurrent execution, Python's incredibly powerful statements allow it to have the greatest ease-of-use, and Java performs the best in terms of scalability. Interestingly, As the number of threads created increases, there may be some very large number of threads where Java would begin to perform better than Go since it had a much smaller degradation in performance. From these results some broad conclusions can be inferred. Go clearly is an incredibly competent language in terms of ease-of-use and performance and is therefore a perfectly valid language choice for a developer or business looking to build a large, highly concurrent system. Go's implementation of lightweight goroutines allows it to perform well enough to make up for its relative lack of scalability when compared to Java, leading to its execution time being the fastest in every test.

FUTURE WORK

Ideally, the previous tests would be run on a larger scale on a more powerful system or on a distributed system to be able to more thoroughly quantify both performance and scalability. Considering the subjective nature of "ease of use," even more quantitative approaches could be formulated to try to argue objectively for what constitutes an easy-to-use language.

In terms of further research, it would be a worthwhile exercise to perform tests where the number of threads created remains the same, but the complexity of the work done by each thread increases. Additionally, testing with a wide variety of different computations may also produce greatly different results.

REFERENCES

[1] R. Cox, R. Griesemer, R. Pike, I. L. Taylor, and K. Thompson, "The Go programming language and environment," *Communications of the ACM*, vol. 65, no. 5, pp. 70–78, Apr. 2022, doi: https://doi.org/10.1145/3488716.

[2] "The Go Programming Language Specification," *go.dev*. https://go.dev/ref/spec

[3] "Documentation - The Go Programming Language," *go.dev*. https://go.dev/doc

[4] A. A. A. Donovan and B. W. Kernighan, *The Go Programming Language*. New York, N.Y. Addison-Wesley, 2016.

[5] "EBNF: A Notation to Describe Syntax." Available: https://ics.uci.edu/~pattis/misc/ebnf2.pdf

[6] TIOBE, "TIOBE Index | TIOBE - The Software Quality Company," *Tiobe.com*, 2023. https://www.tiobe.com/tiobe-index/

[7] P. Y. Abhinav, A. Bhat, Christina Terese Joseph, and K. Chandrasekaran, "Concurrency Analysis of Go and Java," *5th International Conference on Computing, Communication and Security (ICCCS)*, Oct. 2020, doi: https://doi.org/10.1109/icccs49678.2020.9277498.