Writing a Microservice in the Go Programming Language

by Glenn Engstrand

The Go programming language was designed mostly by some ex Bell Labs gurus who originally worked on Unix and worked for Google when Go was designed in 2007 and first released to the public in 2012. Since then, its popularity has steadily increased. It is known mostly as the programming language for infrastructure related systems such as Docker, Kubernetes, and Prometheus. What I wanted to learn was how effective Go was at writing business focused microservice based applications.

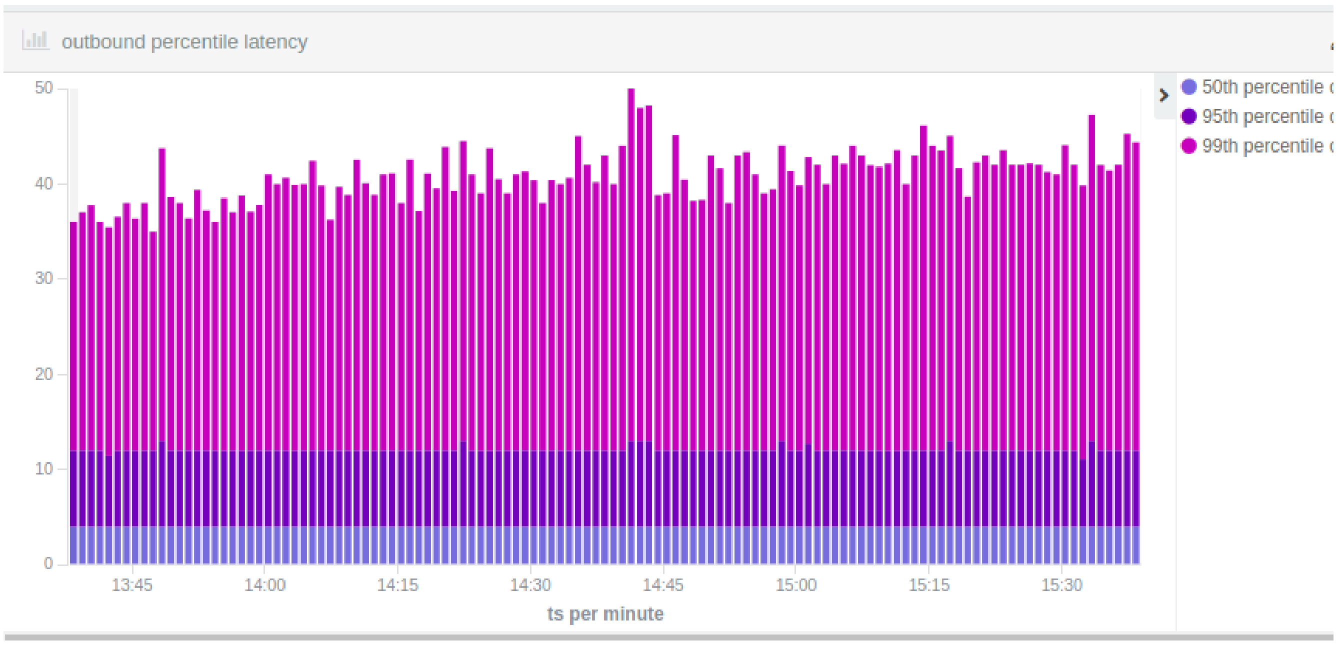
I have this github repo where I implement the same feature identical, polyglot persistent news feed microservice in different programming languages. I run each microservice on the same test lab then capture and analyze the performance results in order to form a basis for comparison between these various programming languages.

I wanted to see how Go compared to the other popular programming languages whose news feed microservces have been implemented previously including Dropwizard on Java, Spring Boot on Java, Node on Javascript, Flask on Python, Finatra on Scala, Scalatra on Scala, and Ring on Clojure.

Architecture and Design

The architecture for the Go version of the news feed microservice is the same as all the other versions. The mysql database is used to store participant and friend relationship information. The redis database is used as a cache that fronts read access to mysql. Cassandra is used to store both inbound and outbound news posts. Elasticsearch is used both for keyword based searching of news posts and to capture performance related data.

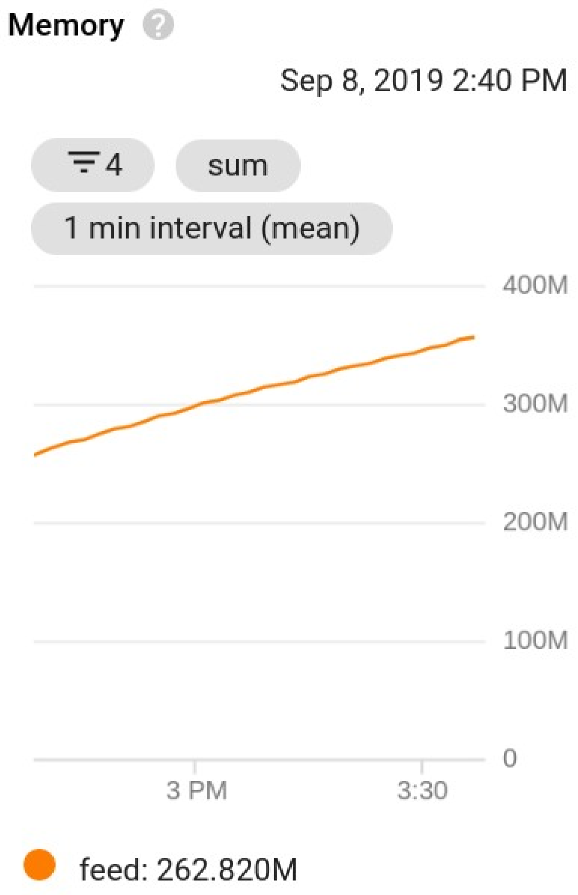
Like almost all of the implementations for the news feed microservice, I started from swagger codegen templates. The go-server templates use an open source project called mux as the request router. I used the officially sanctioned client libraries for mysql, redis, and cassandra. I had to use a more obscure client library for elasticsearch because the official one doesn't support the version of elasticsearch that I use. I ended up having to implement my own connection pool for that using Go’s built in sync package.

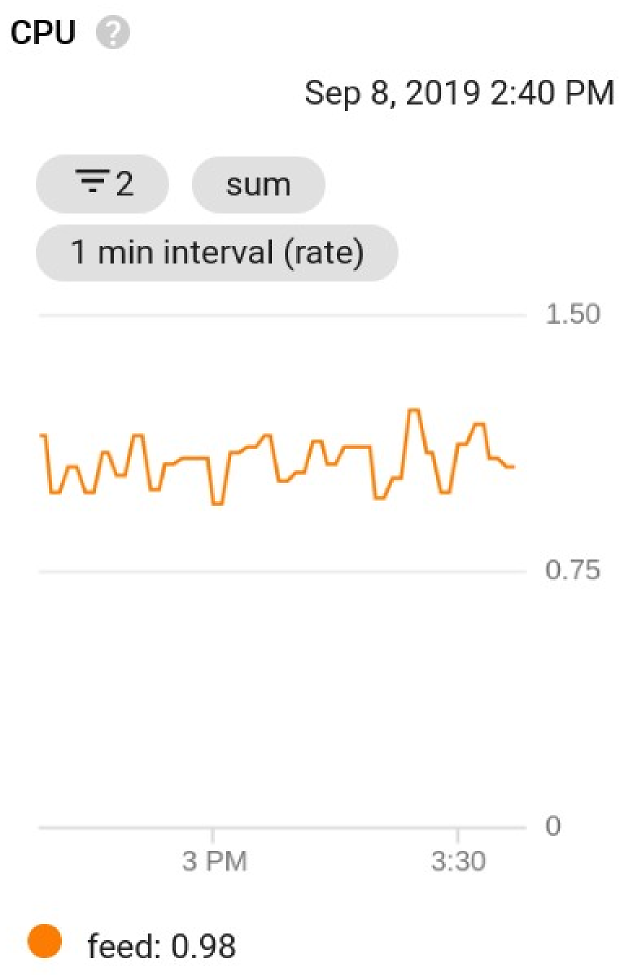


The design of the microservice is heavily influenced by the design of the programming language. Take one look at the bios of Go's inventors and you will quickly understand why the Go programming language is considered to be "a better C."

There are many improvements to Go over the C programming language. For those who like to approach concurrency via the CSP model, goroutines and channels are a very simple, yet effective, language feature. Go does have pointers but not pointer arithmetic. Slices let you do everything that you originally did with pointer arithmetic but without the need for unsafe operations. There is a rudimentary type inference system that allows you to have short variable declarations. The defer keyword before a statement means that statement won't get executed until control is returned from the current function. Functions can return multiple results. Both defer and multiple results become very important as you shall see in about four paragraphs from here.

There are also some quite significant features, that you find in other popular programming languages, missing in Go. I suspect that these omissions are deliberate so you should not expect them to be added later.





Go does not support Object Oriented Programming. Go has structs with fields. Functions can have structs as receivers. This can make structs kind of look like objects. What is missing is inheritance, encapsulation, and polymorphism. Go interfaces kind of look like polymorphism but it uses duck typing. Duck typing in a statically compiled language. Imagine that.

Go's approach to Functional Programming is similar in its approach to Object Oriented Programming. Provide some limited support but not enough for a proper solution.

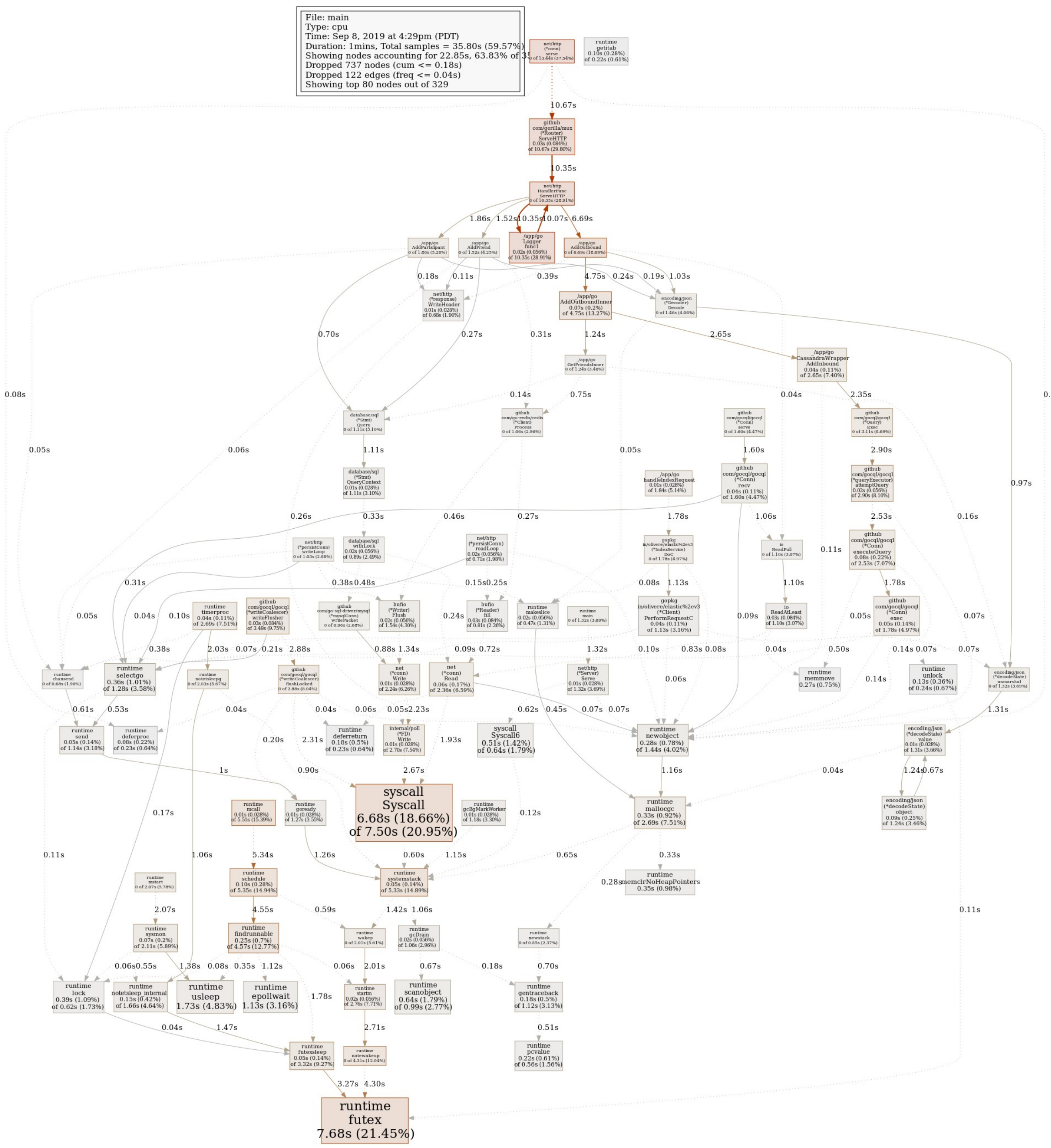
Function closures afford the lambda calculus but there is no support for monads in the Go programming language itself. You can kind of fake it by writing all of these map functions that apply a function closure inside a for loop over a range of slice. If you go that approach, then you are going to be writing a lot of code that you will wish was just being handled in Go.

Go does not have exceptions. No try and no catch. There is a panic and recover but that results in the process terminating. This is where defer and multiple results comes into play. With multiple results, each function can return both the success result and the error result. The code that makes a call to a function then tests if the error result is not nil. With defer, you can return prematurely in those error checks without writing a lot of complicated logic to determine which connections should be closed or struct values returned to their respective pools.

The features of the programming language exert a subtle yet powerful influence on the natural way that code is organized in files.

The last missing feature that I want to cover here is the lack of a method intercepting dynamic proxy. Why do you care? Because a dynamic proxy is very useful when writing unit tests. The key to writing unit tests is that each test covers only the unit of code that it is expected to vet. That means mocking out dependencies to external data stores or services. That mocking is easy to accomplish with a dynamic proxy capability. Since Go doesn't have a dynamic proxy, you have to compensate by wrapping all calls to the affected client libraries with a mixture of structs and interfaces. I had to add 8 structs (4 in the service and 4 in the unit test that mock the 4 in the service) and 4 interfaces to the Go version of the news feed implementation just to write a unit test for adding a news feed item.

Why write unit tests when Kubernetes makes it easy to write end-to-end tests where you are exercising all of the code? Even with the advent of CI / CD where you can automatically spin up pods in Kubernetes and run test automation against those pods, it is still important to write unit tests. Why is that? Because the faster you find and fix bugs, the quicker you can deliver quality software. You don't need to spin up any pods for unit tests so that is the fastest way to find bugs.


File: main
Type: cpu
Time: Sep 8, 2019 at 4:29pm (PDT)
Duration: 1mins, Total samples = 35.80s (59.57%)
Showing nodes accounting for 22.85s, 63.83% of 35
Dropped 737 nodes (cum <= 0.18s)
Dropped 122 edges (freq <= 0.04s)
Showing top 80 nodes out of 329
syscall Syscall 6.68s (18.66%) of 7.50s (20.95%)
runtime futex 7.68s (21.45%)

Here is a CPU profile of the service for one minute during the load test. As you can see, there is no apparent bottle neck. Most of the time was spent waiting on IO which, for a service like this, is appropriate. I used the net/http/pprof package to generate this profile which requires a small change to the service itself that is not in the github repo.

Code Complexity

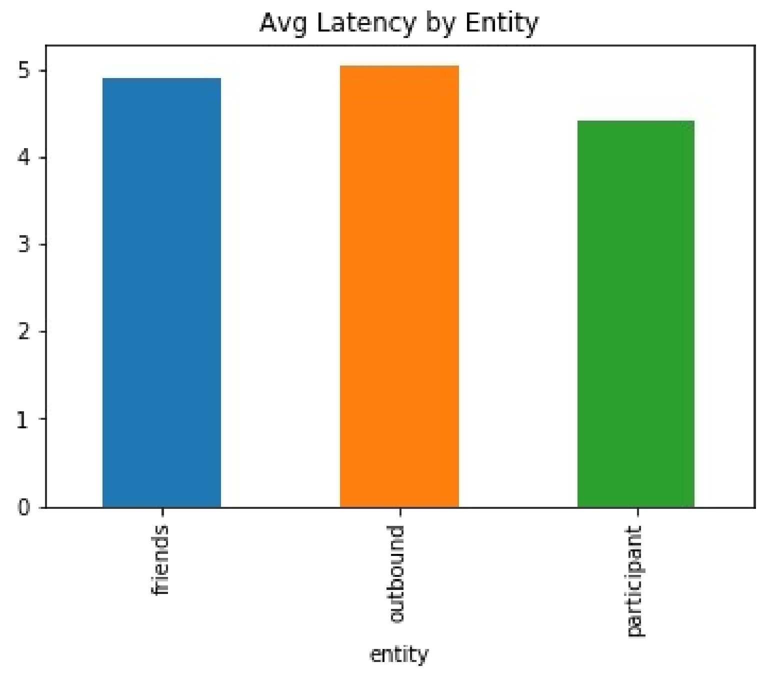
One basis for comparing different programming languages is to measure the complexity of feature identical programs written in those languages.

One simple metric that can be used across different languages is average per file Lines of Code (LoC). Many will argue that LoC really doesn't accurately reflect on complexity. I will agree that there is no mathematical proof here but I will say this in defense of using LoC. The best way for a developer to maintain quality is to understand the code before enhancing it. When a developer opens a file to change the code within, the developer is more likely to give up trying to understand the already existing code in the file if there is a lot of it.

You might be tempted to mandate a maximum LoC size for files but that doesn't work either. Files are a way of organizing code such that it can be easily found later. The organizing principle has to make sense to the developer in order for them to be able to effectively make use of it for that purpose. Enforcing a max file size does not help here because developers will end up with a lot of artificial abstractions in their code that doesn't make sense in the larger context of the problem domain. Less intelligible code will not help developers improve their understanding of it.

The features of the programming language exert a subtle yet powerful influence on the natural way that code is organized in files.

Why write unit tests when Kubernetes makes it easy to write end-to-end tests where you are exercising all of the code?

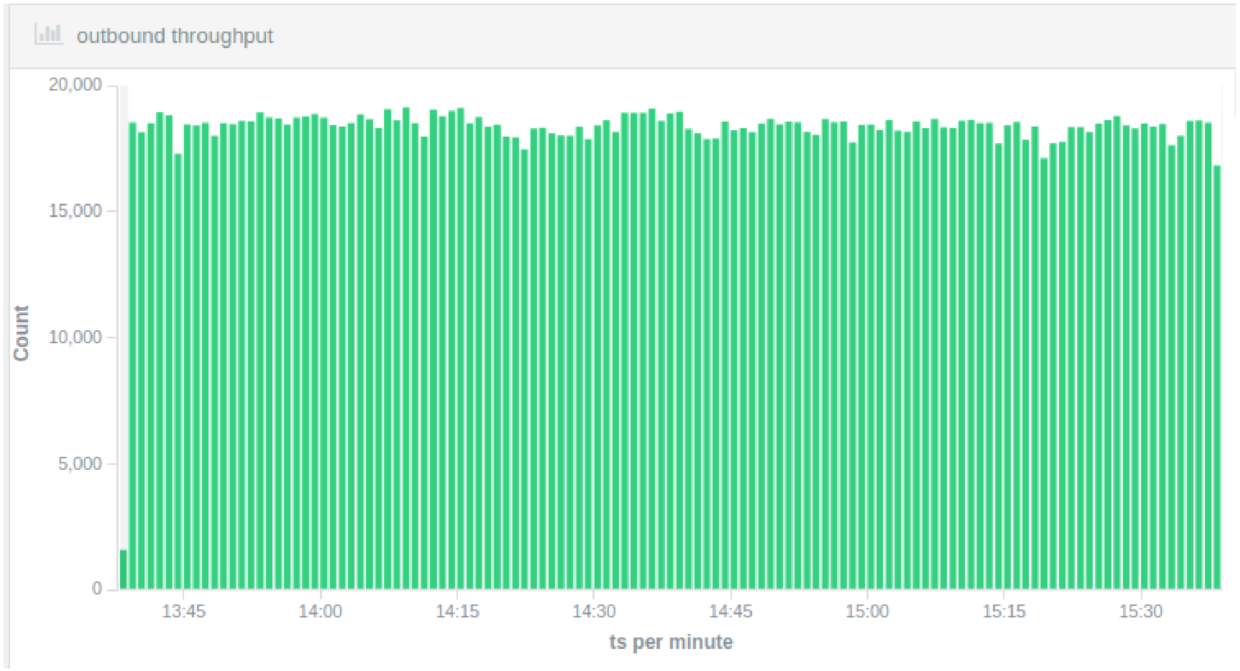


Where does Go stand when compared to the other news feed microservices in terms of average per file LoC? Closer to the largest files. Go files had fewer LoC (on average) than the Finatra based Scala microservice and the DropWizard based Java microservice. The Go files were larger than those in Clojure, Spring Boot based Java, Python, Node.js, and Scalatra based Scala.

Why is there so much Go code? Remember that Go doesn't support exceptions. That lack of exceptions means that you have to check for errors with every function call you make.

Another way to measure code complexity is to use the cyclomatic complexity formula as developed by Thomas McCabe in 1976. Loosely, the metric reflects on the number of different control flow paths that are possible within the program. There are two main issues with using cyclomatic complexity as a basis for comparison. Cyclomatic complexity does not strongly correlate with code readability. There is no single tool that can be used for all of these different programming languages and how that number gets computed is different for every tool. According to this metric, Go had the lowest complexity; however, the tool that I used to calculate that metric was very naive in its implementation of the formula. It calculated the complexity based on the assumption that there is only one entrance and one exit to each method. That assumption was incorrect for this code.

Performance Under Load



I tested this microservice the same way that I tested all of the other microservices, using the standard load test environment for two hours. I like to compare performance for the outbound post operation because it does a lot. Each call to that API fetches the poster's friends, creates inbound entries for each friend, creates the outbound entry for the poster, and finally creates a searchable document of the news feed item in elasticsearch. The average per minute throughput of outbound posts was 18,425 with an average duration of 5 ms, a median of 4 ms, and a 99th percentile of 29 ms.

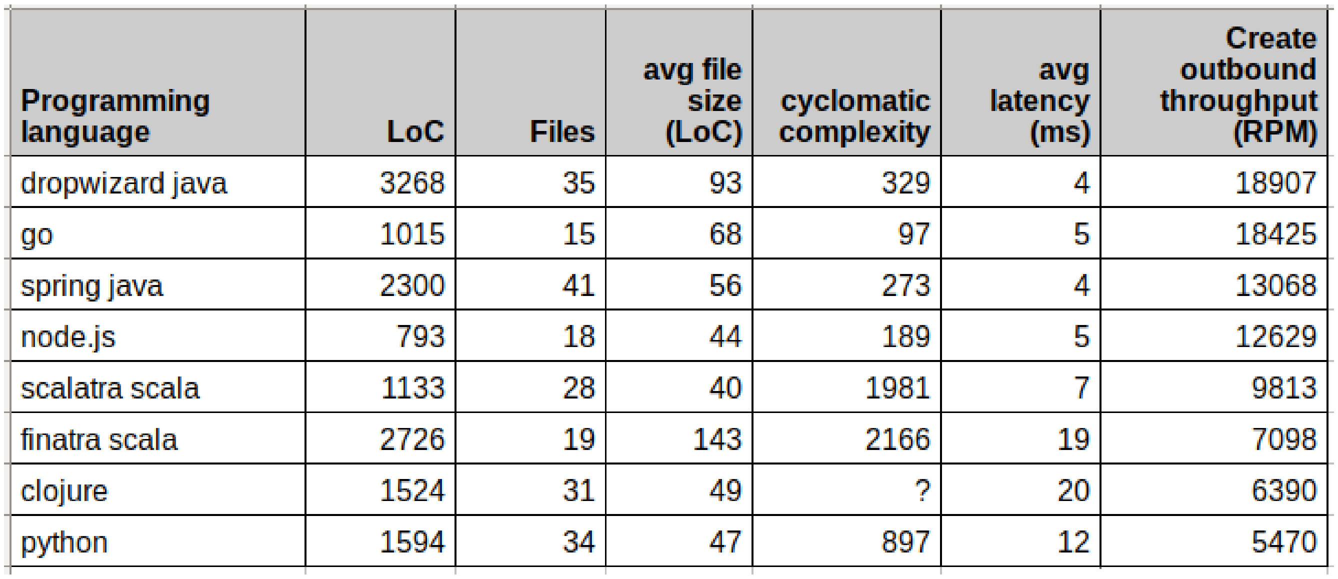
In terms of throughput, the Go version of the news feed microservice performed worse than Dropwizard Java and better than the Spring Boot Java, Node.js, Python, Scala, and Clojure versions.

In terms of latency, the Go version of the news feed microservice performed worse than both the Dropwizard and the Spring Boot Java versions but it performed better than the Node.js, Python, Scala, and Clojure versions.

Conclusion

The Go programming language is very popular with infrastructure based systems development and is starting to gain the attention of application developers. Its designers eschew almost all modern programming language features claiming that is how Go programs perform so well with a lot less complexity.

My own findings were very different. While the simplicity of the language itself yielded a much shorter learning time, that simplicity did not translate to less complex microservices. With regards to performance, Go did quite well but was a close second to Dropwizard Java which has all the modern language features missing in Go.



Programming language	LoC	Files	avg file size (LoC)	cyclomatic complexity	avg latency (ms)	Create outbound throughput (RPM)
dropwizard java	3268	35	93	329	4	18907
go	1015	15	68	97	5	18425
spring java	2300	41	56	273	4	13068
node.js	793	18	44	189	5	12629
scalatra scala	1133	28	40	1981	7	9813
finatra scala	2726	19	143	2166	19	7098
clojure	1524	31	49	?	20	6390
python	1594	34	47	897	12	5470