# Findings

Findings of my research are the measurements recorded for each dependent variables of the experiment. I have used different metric to measure few dependent variables and for the variables not quantifiable, based on the level of support each language offer the output is measured. Also, AMQP and gRPC experiment could only be carried out on Java and Ballerina and not on Jolie due to unavailability of the libraries. Jolie is out of scope for the for AMQP and gRPC. In addition to the findings with respect to the dependent variables, I am considering the challenges that I faced during each phase of the experiment namely environment setup, development, build, deployment, and execution. This finding is purely based on my experience and the background I have for each language. Finally, outcome is derived based on the findings and the comparison is made of what language is better.

## Verbosity

When calculating verbosity, I have calculated the source line of code metric that includes different variables to be counted. These are - Physical lines that include the commented code (SLOC), the logical lines that include the executable statements, blank lines, etc [(Bhatt et al., 2012)](https://www.zotero.org/google-docs/?HwcorN). To get the count of each variable there are different automated tools. However, there was no single tool that provides support for the languages used in the experiment. Thus, I am manually counting the line of code based on the above metric.

Even though all the languages are structurally and syntactically different, while coding I have tried to be consistent in my style of coding to avoid as many differences as possible. I have achieved this by following certain guideline, this includes –

* Common naming conventions
* Common declaration style of class, functions, and variables.
* No comments.
* Use of blank lines in code is only when there is a declaration of class in java, service in ballerina and interface in Jolie or functions. Functions is the common term used across all the language.
* Every executable statement is in a single line in my program.
* There is only one function created for hello and world service. Additional function will be created only if it is unavoidable due to the way the language works. For example – The ReST service developed in Java requires a public constructor without that it fails to execute.

For import statements, Ballerina and Jolie has common style of importing a class. The import statement is for the entire module. However, in Java there are two different ways of importing the class, either import the entire package or give the fully qualified name of the class (https://docs.oracle.com/javase/tutorial/java/package/usepkgs.html). Each method is appropriate for different solutions. Since I am trying to have consistency in style across all the language. I am opting to import entire package instead of single class.

By following the above standards, I have ensured the consistency in coding.

The source line of code metric for every language looks as shown in the diagram.

Outcome

Number of physical lines is same in Ballerina and Jolie for the ReST service. Also, Ballerina and Jolie both has less number of physical lines than Java. For AMQP and gRPC, Ballerina has fewer physical lines than Java. However, compared to Java and Jolie. However, the executable lines have significant difference in each language. Like Ballerina, Jolie has only one executable line for hello service and Java has two executable lines. Hello service does not have large difference. However, for the World Service, Jolie has only 2 executable lines which is minimum of all the other languages used for the experiment. Ballerina on other hand has more logical lines than Jolie and Java.

For AMQP and gRPC services, the outcome is the same. The number of physical lines and logical lines is higher is Java than Ballerina.

## Ports

Unlike verbosity, port is not a quantifiable variable. A communication port concretely describes how some of the functionalities of a microservice are made available to the network. Each service may be equipped with many ports or ports of different kind such as input and output. Input ports describe the functionalities that the service provides to the rest of the MSA. Conversely, output ports describe the functionalities that the service requires from the rest of the MSA. According to Guidi et al. (2017), ports should be specified separately from the implementation of a service, so that one can see what a service provides and what it needs without having to check its actual implementation. Keeping this mind, the ports variable is measured on how each language allow the usage of the port.

### Java

#### ReST

For both hello and world service, the configuration of the port could not be defined in the code. This detail is mentioned in the configuration file of the glassfish server on which both the services are hosted. This detail goes into the network-listener section of the configuration file with the xpath as network-config/network-listeners/network-listener. Inside the network-listener tag, the attribute “port” defines the port number and the protocol information. The port information in the Glassfish server configuration file looks like as shown in Figure 0‑1.

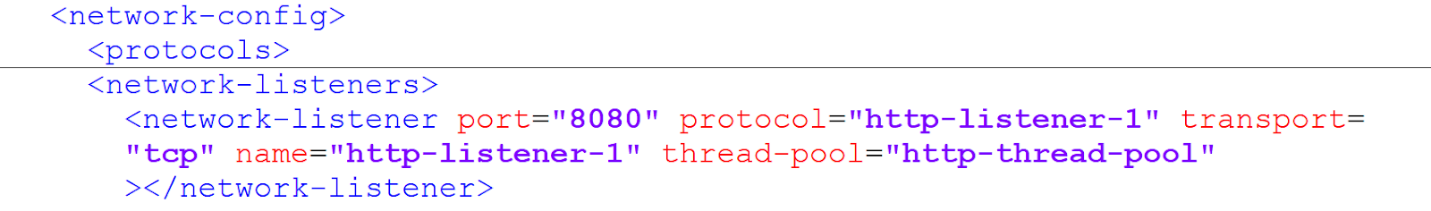


Figure ‑ Network Configuration for Glassfish Server

#### AMQP

In the AMQP experiment, hello service was a ReST service sending message to the AMQP broker. The port was again configured in the glassfish server configuration file where the hello service is hosted. For the world service, as it is a console application no port is required.

#### gRPC

In the gRPC experiment, hello service is the gRPC service listening on port 8080. The library that is used to develop the gRPC service allows to configure the port within the code unlike the ReST service. Figure 0‑2 is the screenshot of the code snippet, static method “forPort” of class ServerBuilder is used to set the port number.

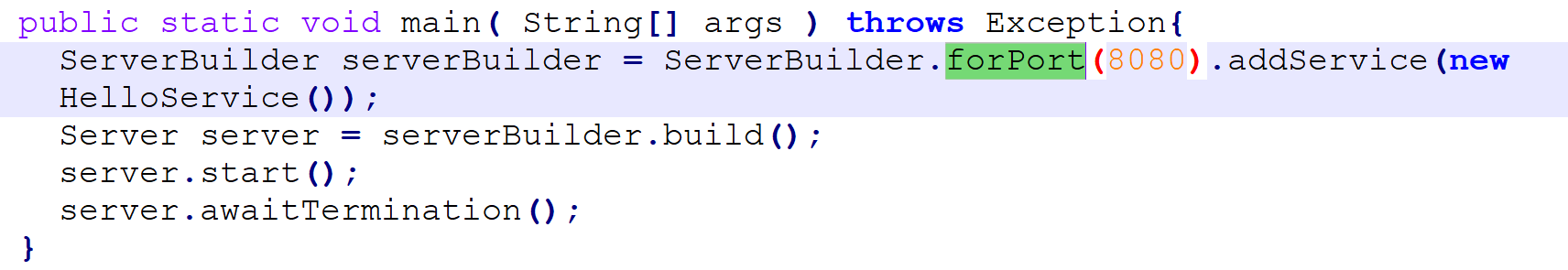


Figure ‑ gRPC code snippet for input port

World service is the consumer of the hello service calling it on port 8080. This output port information is also within the code that is available from the gRPC library. Figure 0‑3 is the screenshot of the code snippet, static method “forTarget” of ChannelBuilder is used to set the output port number.

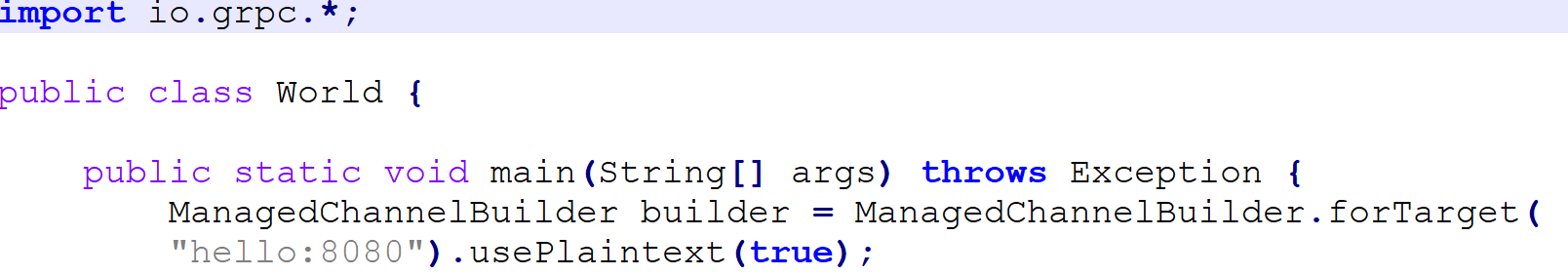


Figure ‑ gRPC code snippet for output port

### Ballerina

#### ReST

Ballerina allows to define the port for both the ReST services – hello and world in the code. This detail is given as the part of the service definition within the code in the http:listener as shown in the Figure 0‑4.



Figure ‑ Ballerina ReST code snippet

#### AMQP

In AMQP experiment, as the hello service is a ReST service, the port detail is provided the same way in the code. World service is the listener service that executes when a message is received in the RabbitMQ broker. The information of the RabbitMQ broker is configured in the definition of the RabbitMQ listener as shown in Figure 0‑5.

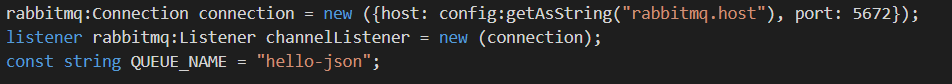


Figure ‑ Ballerina AMQP code snippet

#### gRPC

Port configuration of the Hello service is given when defining the service same as the ReST service. However, a listener used is grpc listener and not http listener.

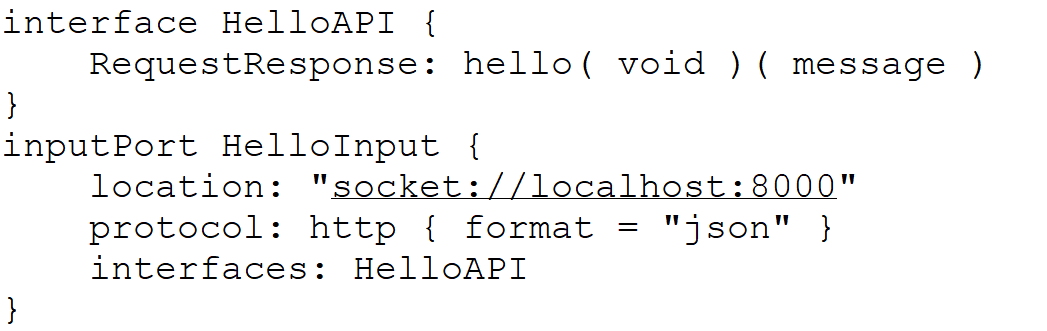


World service is the consumer of the hello service calling it on port 8080. This output port information is also within the code.



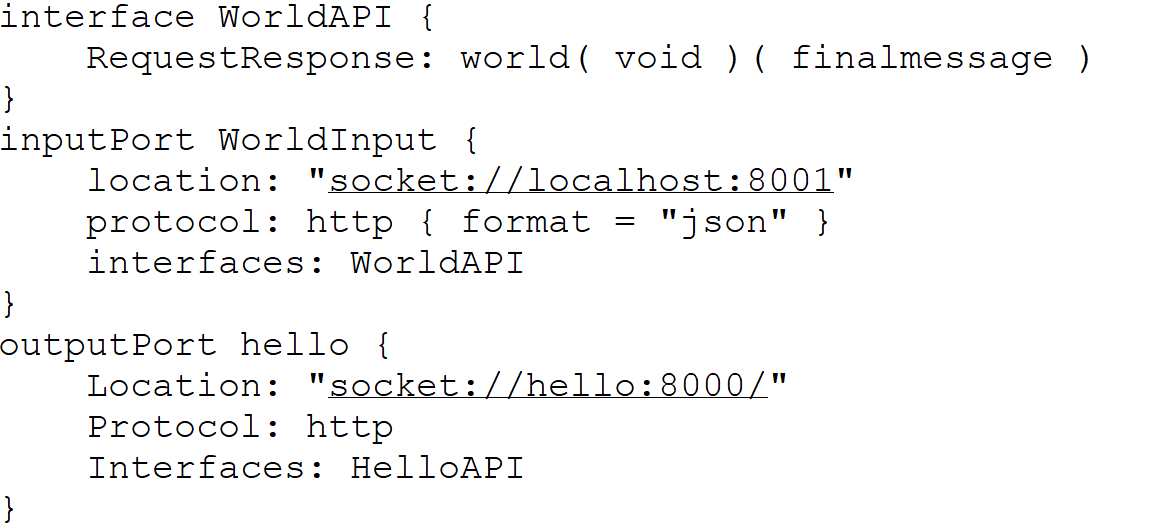
### Jolie

Jolie allows to define the port information as per the behaviour of the service. This includes if the port is used as input or output. Hello service is the ReST service exposed on port 8000. This information is given in the location attribute of the input port and the same is attached to the hello service by specifying the name of the service in the interfaces attribute of the port as shown in the image below. The name of the hello service is HelloAPI and the same value is given to the interfaces attribute.



World Service is the service exposed on port 8001. Also, world service calls hello service exposed on port 8000. Both this port information is configured as input port and output port with the respective interface information. Interface like mentioned above is the name of the defined in the code. As the hello service was called as HelloAPI, the output port of the world service has that as the interface value. Similarly, name of the world service is given as WorldAPI and the same name is given as the value for the interface of the input port.

Image illustrates the above –



### Outcome

For the ReST service it is evident that Jolie has flexibility as defined by (Guidi et al., 2017) where the language has support for port and is separate from the implementation . While the same is not achievable in Java and Ballerina.

For all the services written in Ballerina, language does offer the support for configuring the port. However, it is tightly coupled with the implementation logic of the service. If a new port detail must be added to the service, that would require change in the service implementation. Thus, slighlty impacting the software development life cycle.

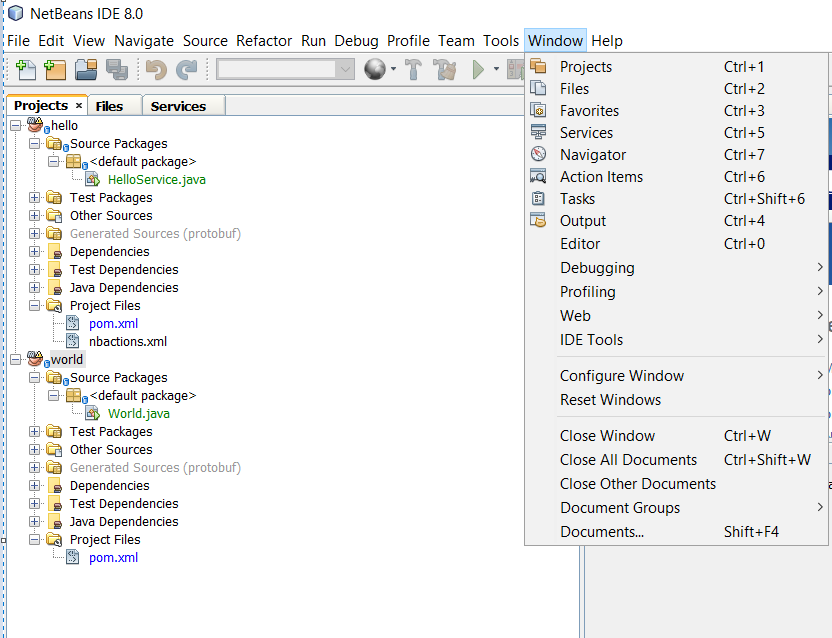
Rest and AMQP service in Java does not allow the developer to configure the port information in the language. However, the libraries designed for gRPC does have the support to define the port in the language. Like Ballerina, this library also have the limitation of port being tightly coupled with the implementation of the service.

## Graphical View Support

Graphical view support is the ability to display the service flow (incoming request/ outgoing response). This is the visual representation of the service. Graphical support by the language is not the part of the language libraries or the service definition. I have used the official website of each language to identify if there are additional plugins available that can be used with the Integrated Development Environment (IDE) used for the experiment.

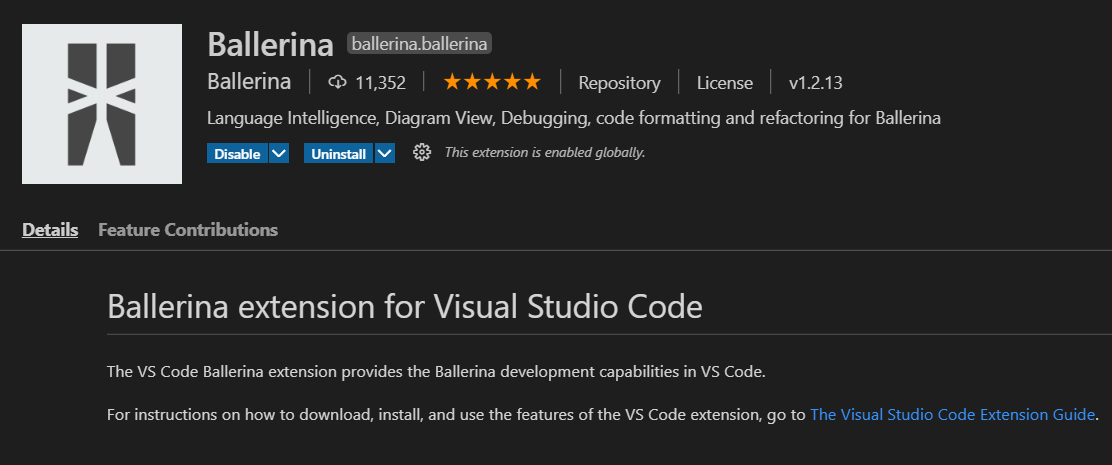
### Java

For Java, I was using NetBeans IDE. I checked the official website of Java and Netbeans for the visualisation tools/plugins but there was no result found. Also, I checked within the tools in the IDE and there was no plugin available that can be used to visualise the service. The Windows section of the NetBeans IDE displays what are the different windows available for the developer to use. If some window or IDE Tools are not available by default, the same can be exported using the plugins option. I searched inside the plugins option for the visualisation tool/ graphical editor but there was no result returned.

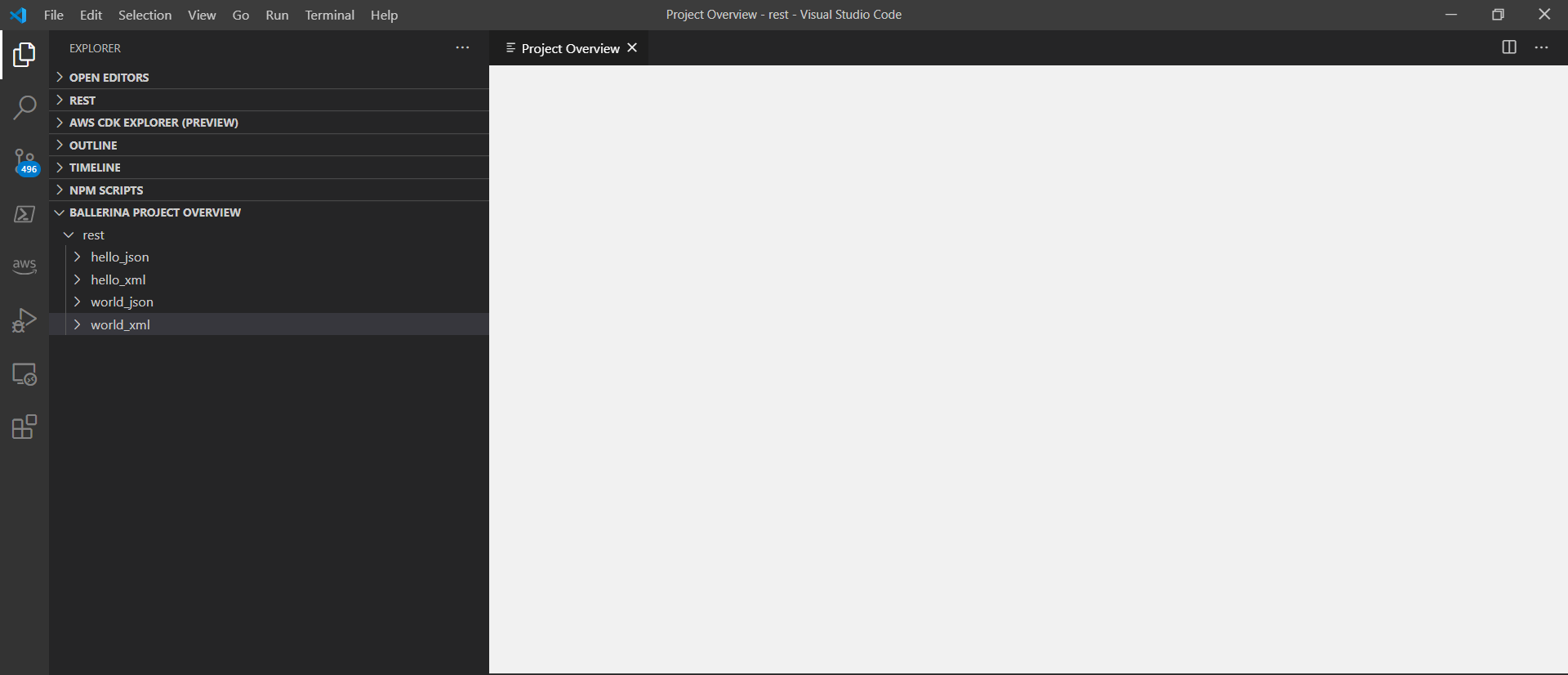


### Ballerina

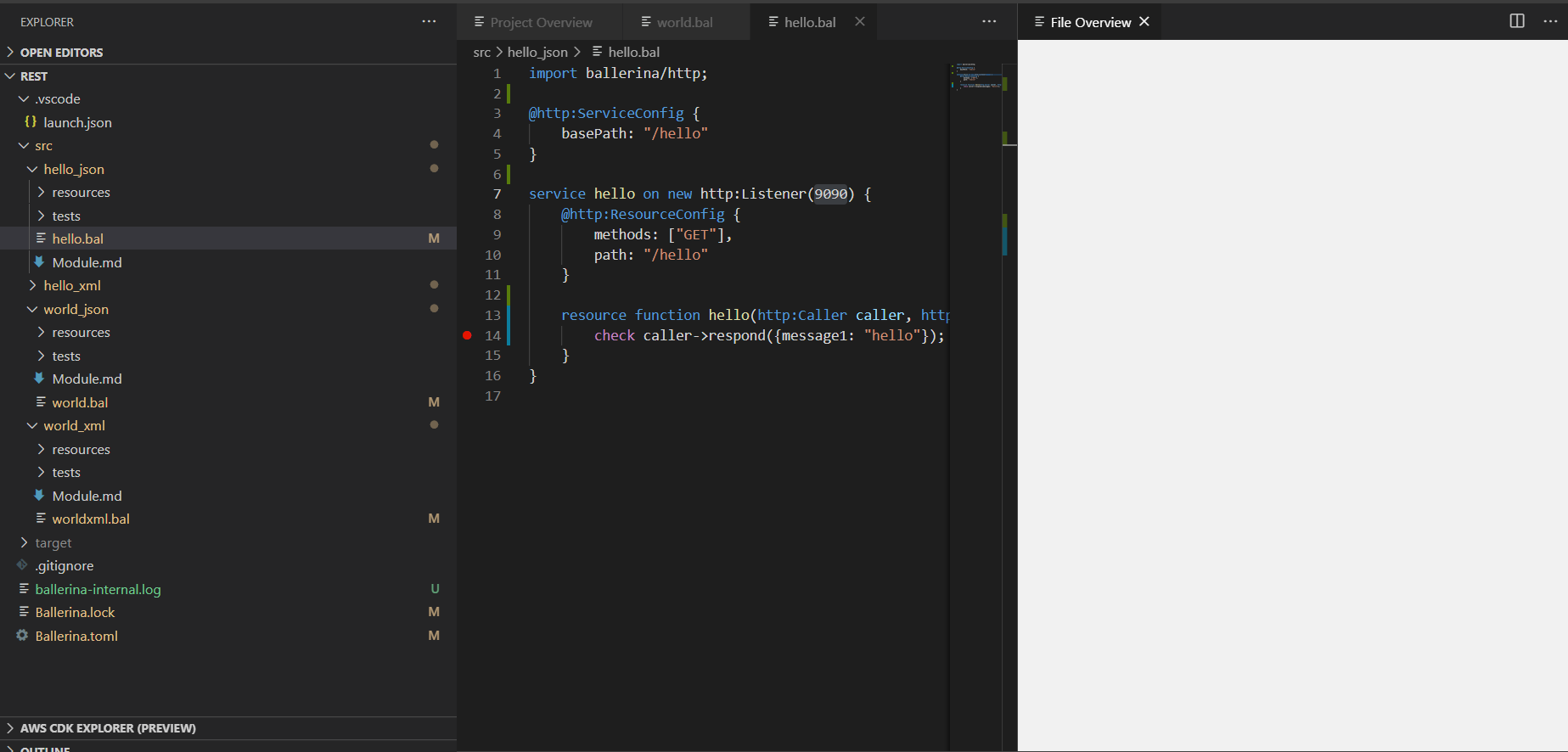
For developing services in Ballerina, I used Visual Studio Code for the development of the services, and I installed the extension for Ballerina. Extension is like plugins in IDE.



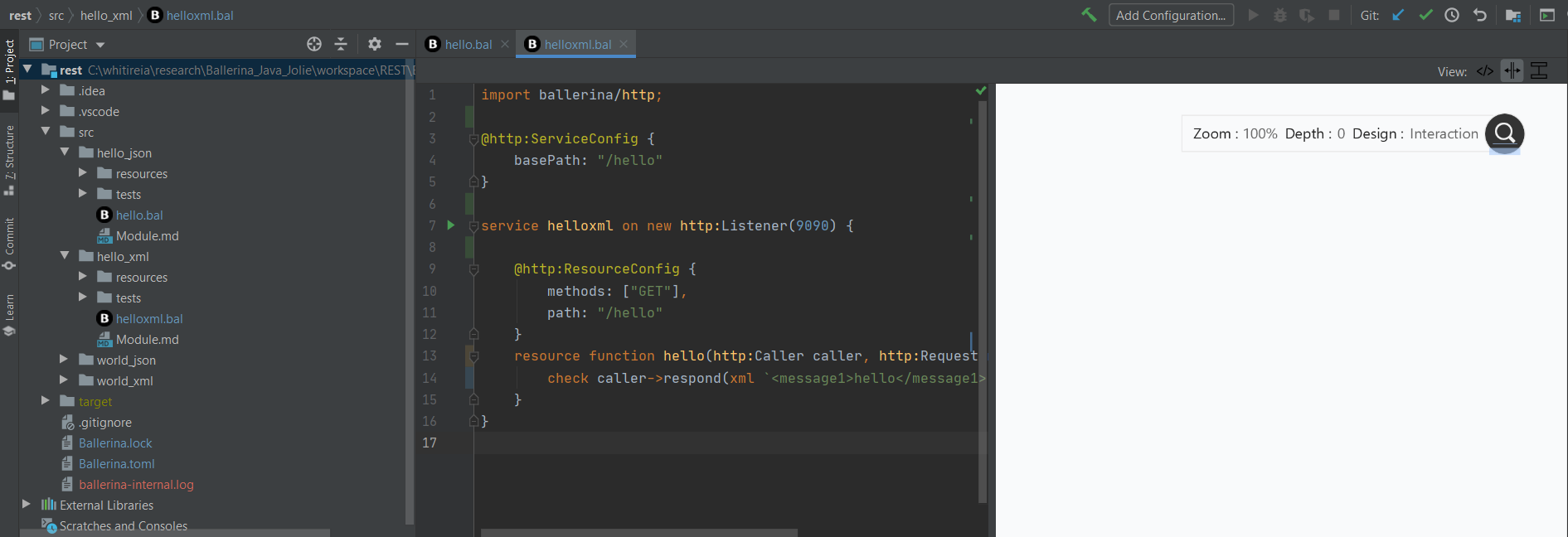
After I installed the extension for Ballerina, I could see a new window called Ballerina Project Explorer. As per the documentation, this editor should visually display the service. (<https://ballerina.io/learn/setting-up-visual-studio-code/graphical-editor/#launching-the-project-overview>). In my experiment, I was not able to successfully demonstrate this behaviour. Given image is the screenshot of my Visual Studio Code with the four ReST services written in Ballerina and the window open is of the Project Overview that visually displays the service which is blank in my case. Similar behaviour is observed for the services written in AMQP and gRPC.



I also tried to open individual service files in the “File Overview” window which gives the graphical representation of the Ballerina file. However, even that window opened blank.

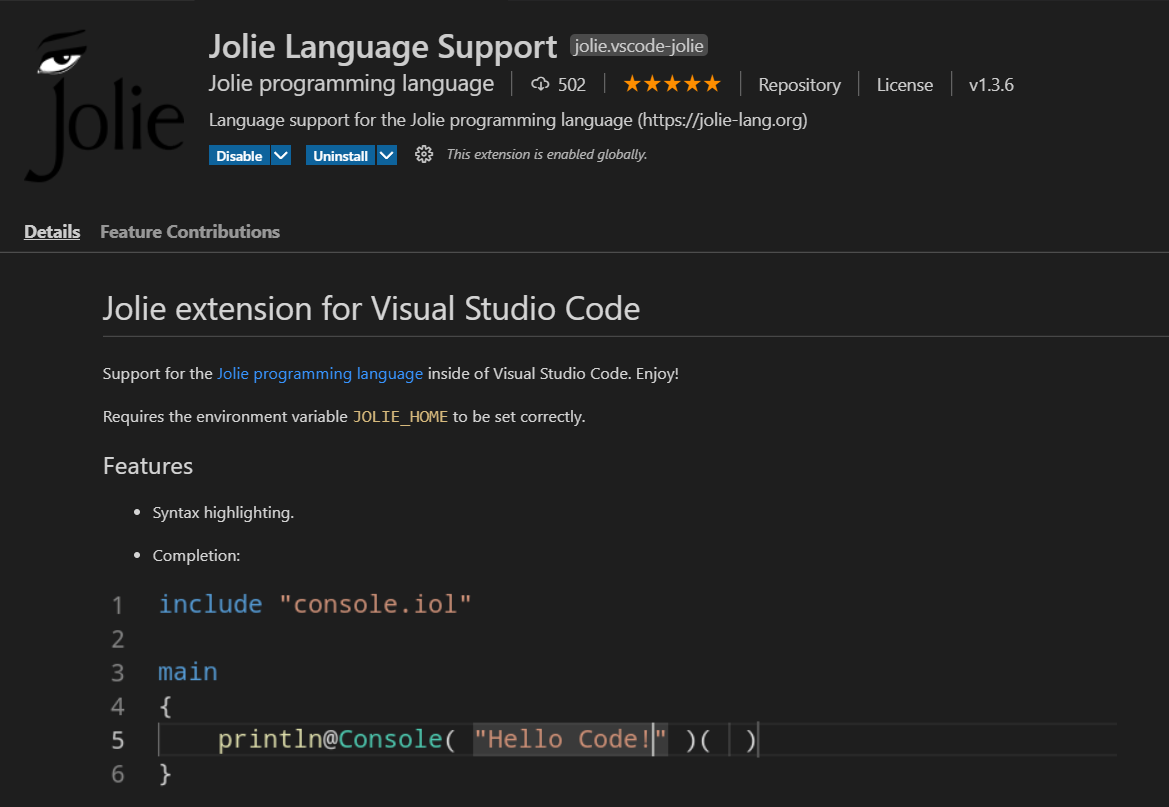


I tried to setup my project in other editor called IntelliJ – IDEA, which is also supported to develop services in Ballerina. I got same blank screen in IDEA as well. Given below is the screenshot.



### Jolie

For developing services in Ballerina, I use Visual Studio Code and I installed “Jolie language support” extension.



This extension does not have the feature to display the service developed visually. Also, the official documentation of Jolie did not mention the graphical support provided by the language.

### Outcome

From the official website of all the languages, Ballerina is the only language that has capability to visually view the services. However, I failed to view the services in different editor that support Ballerina.

## Debug

I tested the local debug feature of each language in the respective IDE used for the development of the microservices. There are different debug options available, I tested each language for local and remote debug. Local debug option is the debug feature that helps the developer to debug the code in the development stage. Local debugging options is important in the software development life cycle.

### Java

#### ReST

In this experiment, I was able to perform local debug in the NetBeans IDE for both the ReST services. I could also add the breakpoint. To debug the service, IDE has the debug feature and is only one click event. After selecting the Debug project option, the service is run in the debug mode. Upon making the request to the service on the exposed resource, I did this by entering the URI in the web browser for both the services. This caused the debugger to stop at the breakpoint and this way I tested my debugging for Java ReST service.

#### AMQP

For testing the debug, I did not do separate testing for both services. Instead, I created a single debug scenario. I added the breakpoints in both the services. First, I run the world service from the IDE in the debug mode and then I run the hello service from the IDE in the debug mode. I entered the URI for hello service from the web browser. This caused the debugger to stop at the breakpoint in the hello service. After resuming the debug, hello service sent the message to the RabbitMQ broker. As soon as the message was received by the broker, it was available for consumption to the world service. When world service consumed the message from the RabbitMQ broker, the debugger stopped at the breakpoint of the world service. In this way, I tested the debug for the AMQP inside NetBeans IDE.

#### gRPC

For testing the debug, I did not do separate testing for both services. Instead, I created a single debug scenario. I added the breakpoints in both the services. First, I run the hello service from the IDE in the debug mode and then I run the world service from the IDE in the debug mode. This results in the debugger to stop at the breakpoint of the world service. On resume, world service builds the message and makes request to the hello service. When the hello service receives the request, the debugger stops at the breakpoint of the service. On resume, the process completes, and world service prints out the final message.

### Ballerina

I had to setup debug in Visual Studio Code to debug the services written in Ballerina. I followed below steps to test the debug –

Click the Debug icon in the left menu or press the Control + Shift + D keys, to launch the Debugger view.

Click No Configurations and select Add Configuration….

Click Ballerina Debug. This opens the launch.json file. You can edit this file to change the debug configuration options as required.

Click on the name of the file to debug.

Click the Start Debugging icon.

#### ReST

Like Java I tested both the services for debug separately and was able to stop the debugger at the breakpoint and resume from there.

#### AMQP

I tried to use the same test scenario used in Java for AMQP. However, in Visual Studio Code I could not debug two processes simultaneously to test the debug for both the services separately. The sequence of running is the same but only service is run in debug mode at a time.

#### gRPC

Repeated the same steps of the testing scenario created for Java. Again, due to the limitation of having multiple debug processes running at the same. I ran only one service in debug mode at a time.

### Jolie

Visual Studio Code had no support for debugging the services written in Jolie. Also, the documentation and the official website did not mention how to debug the services in Jolie.

### Outcome

Debug is a powerful tool and the NetBeans Editor is stable to deliver debug feature without needing to do additional setup for debug. However, debug in Ballerina is not as mature as Java with the limitations in number of debug processes in parallel. Also, Debug as per official website is the experimental feature and is not fully stable.

## Size

My experiment compares the size of the docker image built for all the microservices developed for the experiment and disk space used by each container (https://docs.docker.com/storage/storagedriver/). My microservices are built into docker image and will be running in the container. To compare the size of every image it is important to understand how docker builds the image and how it stores the image in the disk and how the container space is utilised.

Docker image is built from the series of layers. Each layer is the instruction in the Docker file. Every instruction utilises space in the disk and thus the total disk space occupied by each instruction is the size of the docker image. The docker image size information can be extracted by running the command “docker image ls”. The size of the layers can be extracted by running the command “docker history $IMAGEID”. Both the commands will help to identify the size of the executable and the additional size required to setup for every language.

To extract the information of the container running the docker image below command is run – “docker container ls -s” This command returns displays disk size against the column value size. Size is the amount of data used for the writing operation by the container. (<https://docs.docker.com/storage/storagedriver/>)

Size metric for this experiment will hold values for docker image size, container size, executable size and others

Images below are the Charts that represent the size metric recorded for every microservice –

### Java

#### ReST

#### AMQP

#### gRPC

### Ballerina

#### ReST

#### AMQP

#### gRPC

### Jolie

#### ReST

### Outcome

## Execution Time

Execution time is the user time in seconds which is the total CPU time spent within the process. This is the CPU time spent in executing the actual process. As my research is focusing on the integration so I am calculating the total processing time including the integration. Thus, the test is always calculating the response time of the second service that calls the first service and sends back the response. The request to the services is made for 100 times. Execution time metric will capture the time for all the 100 runs, the average of 100 runs. Outcome will be derived based on the average of the runs.

To measure the execution time for the ReST services, I have used one of the ReST Clients called Postman. Postman has a tool called Collection Runner, that allows to make the request n number of times. In my case it will 100. Also, this tool allows me to export the result of all the 100 runs into the json file. From the json file, I capture the data for the 100 runs and feed to the Excel sheet and calculate the average. From the collection runner, I make request to the world service developed in all the three language.

For AMQP service, call to first service triggers the second service execution. Here the execution time is will be the total time when the first service starts and second service ends. I am using the Postman to make the request to the first service. To measure the execution time for the AMQP services, I am using the console logs that prints out the timestamp of every execution. Finally, I export this log data and use a java application that reads the log data and calculates the total execution time.

To calculate the execution time of the gRPC service, I am using the command “time” when starting the second service that gives the total time taken by the process to run the second service. This have achieved writing a powershell script that makes call to the second service for 100 times and records the time for every run. I export the data in the excel sheet and calculate the average of all runs.

Below images display the average of 100 runs for the second service in all the languages -

the ReST request for n number of times. for the ReST and gRPC request. For AMQP requests, the same is obtained using the powershell command.

### Java

#### ReST

#### AMQP

#### gRPC

### Ballerina

#### ReST

#### AMQP

#### gRPC

### Jolie

#### ReST

### Outcome

[Bhatt, K., Tarey, V., & Patel, P. (2012). Analysis Of Source Lines Of Code(SLOC) Metric. *IJETAE*, *2*.](https://www.zotero.org/google-docs/?CthOp9)

Guidi, C., Lanese, I., Mazzara, M., & Montesi, F. (2017). Microservices: A language-based approach. In *Present and Ulterior Software Engineering* (pp. 217–225). Springer.