# Findings

This chapter provides the findings of this research and measurements recorded for each dependent variables of the experiment. Different metric is used in the research to measure different dependent variables of the experiment. These metrics are namely “Source Line Of Code (SLOC)” to measure the verbosity, “docker storage” to measure executable size and “user time” metric to measure the execution time. For the variables not quantifiable, the output is measured based on the level of support each language offer. 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, challenges faced with respect to the dependent variables during each phase of the experiment are described in this section, namely environment setup, development, build, deployment, and execution. Finally, outcome is derived based on the findings and the comparison is made of what language is better.

## Verbosity

When calculating verbosity, the source line of code metric is calculated from different variables. These are variables are physical lines, commented lines, logical lines, and the blank lines [(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, lines of code were manually counted based on the SLOC metric.

With different languages having different structure and syntax, consistent style of coding was used in this study to avoid differences. This was achieved by setting following guideline, including:

* 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. To achieve consistency in style across all the languages, entire package is imported instead of the single class. 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 have less number of physical lines than Java. For AMQP and gRPC, Ballerina has fewer physical lines than Java. 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 in mind, the ports variable is measured on how each language allow the usage of the port. The following section provides details of how different languages used in this study 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.

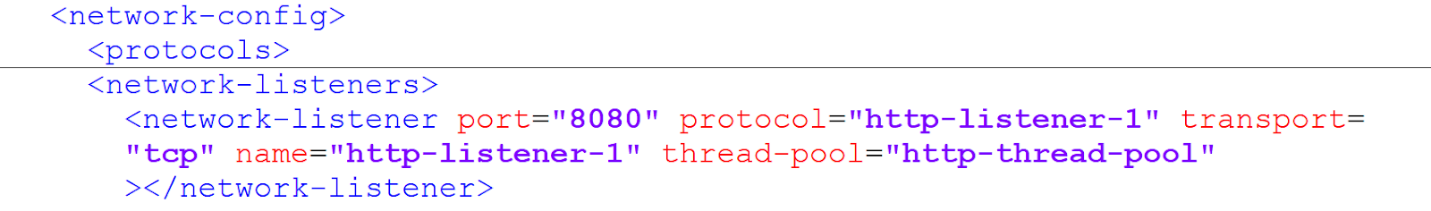


Figure 0‑1 Network Configuration for Glassfish Server

#### 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.

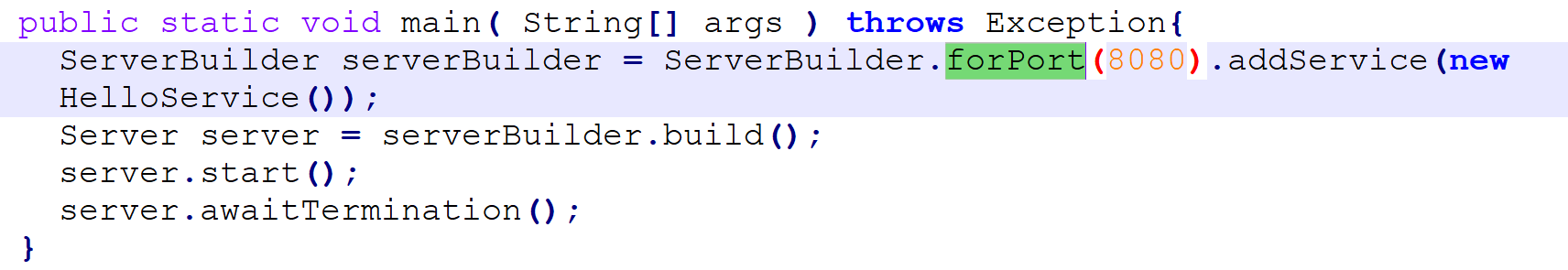


Figure 0‑2 gRPC code snippet for input port

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.

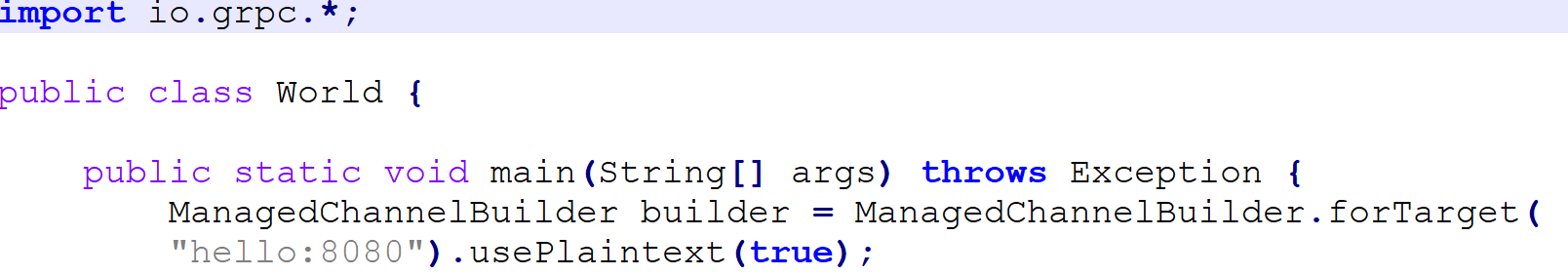


Figure 0‑3 gRPC code snippet for output port

### 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 0‑4 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.

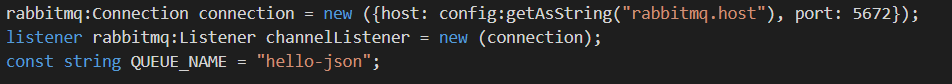


Figure 0‑5 Ballerina AMQP code snippet

#### 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.

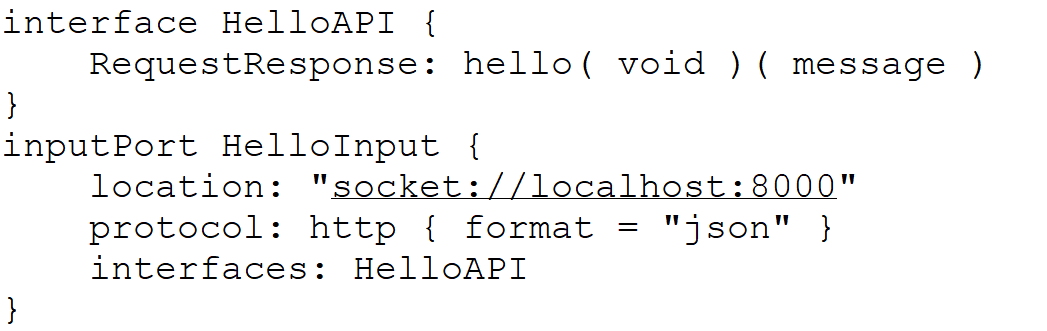


World service is the consumer of the hello service calling it on port 8080. This output port information is also within the code.



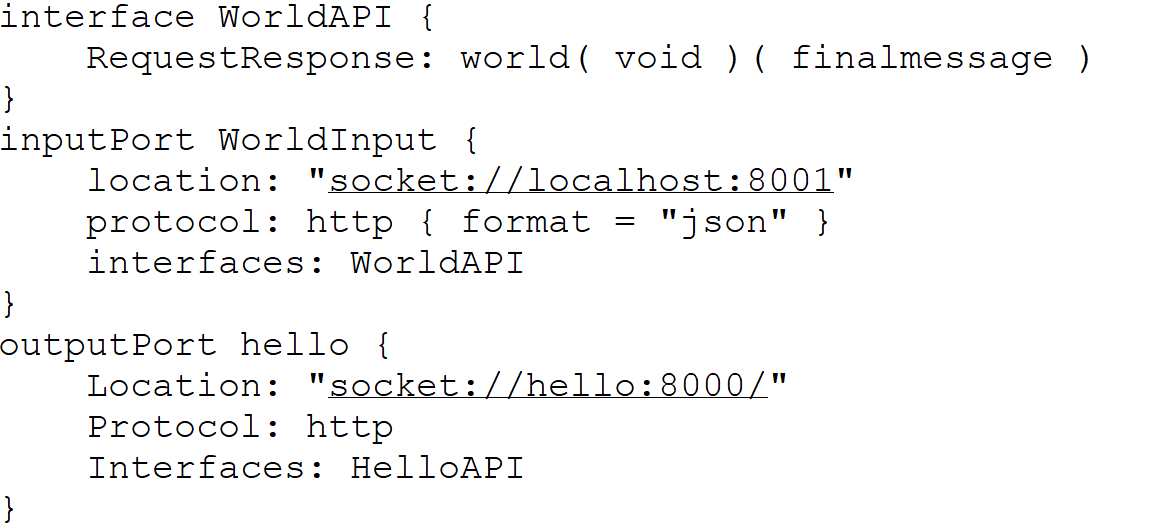
### Jolie

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.

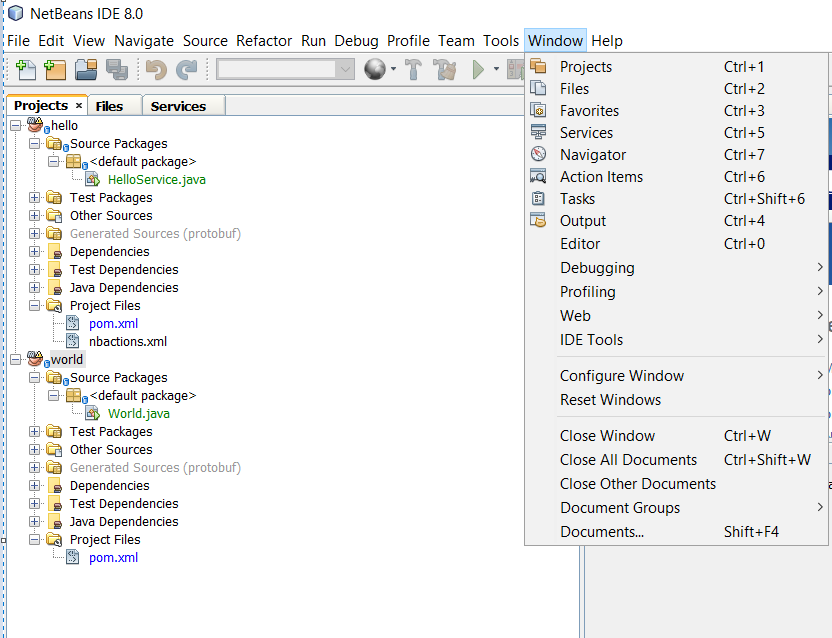
Rest and AMQP service in Java does not allow the developer to configure the port information in the language. However, the libraries designed for gRPC does have the support to define the port in the language. Like Ballerina, this library also have the limitation of port being tightly coupled with the 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. The official website of each language is used to identify if there are additional plugins available that can be used with the Integrated Development Environment (IDE) used for the experiment.

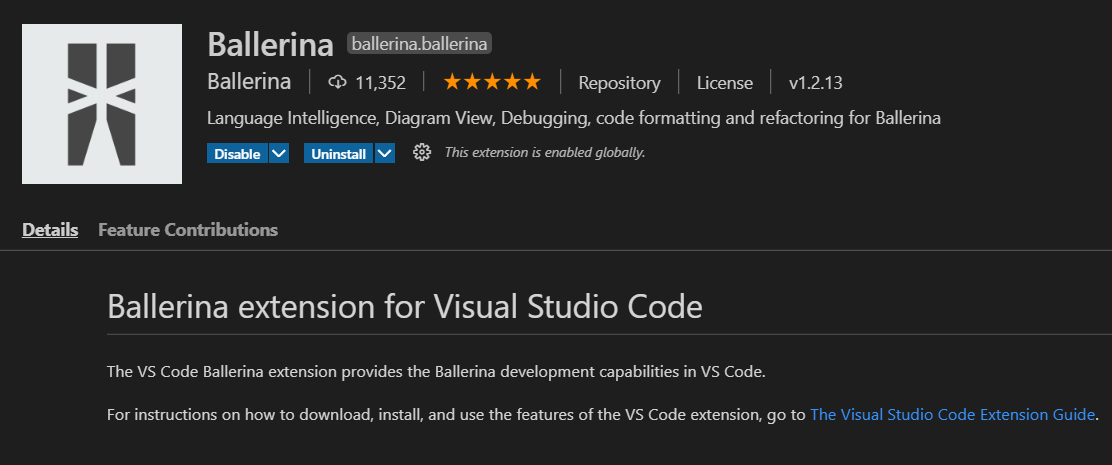
### Java

For developing services in Java, NetBeans IDE was used. The official website of Java and Netbeans was checked for the visualisation tools/plugins, but there were no results found. Also, within the tools in the IDE, 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. Upon searching inside the plugins option for the visualisation tool/ graphical editor, again there were no results returned.

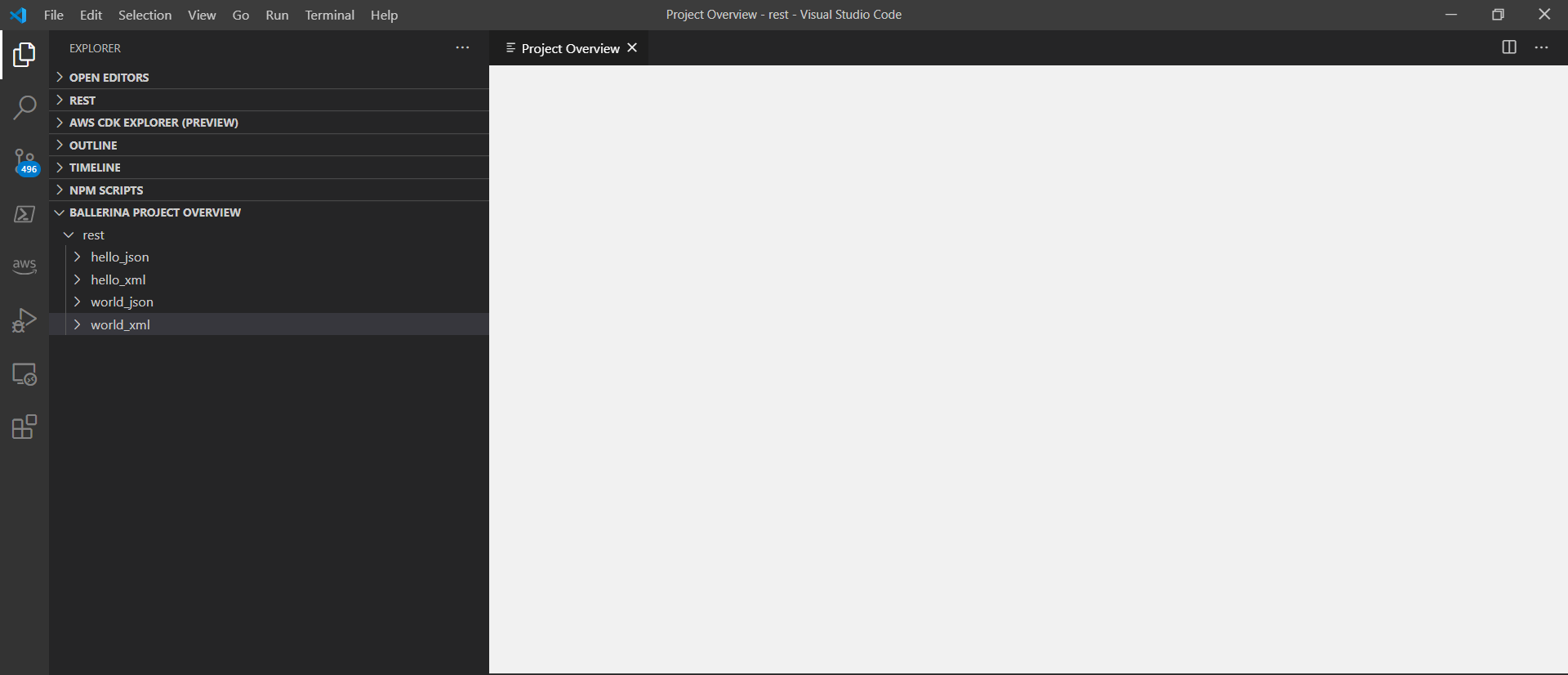


### Ballerina

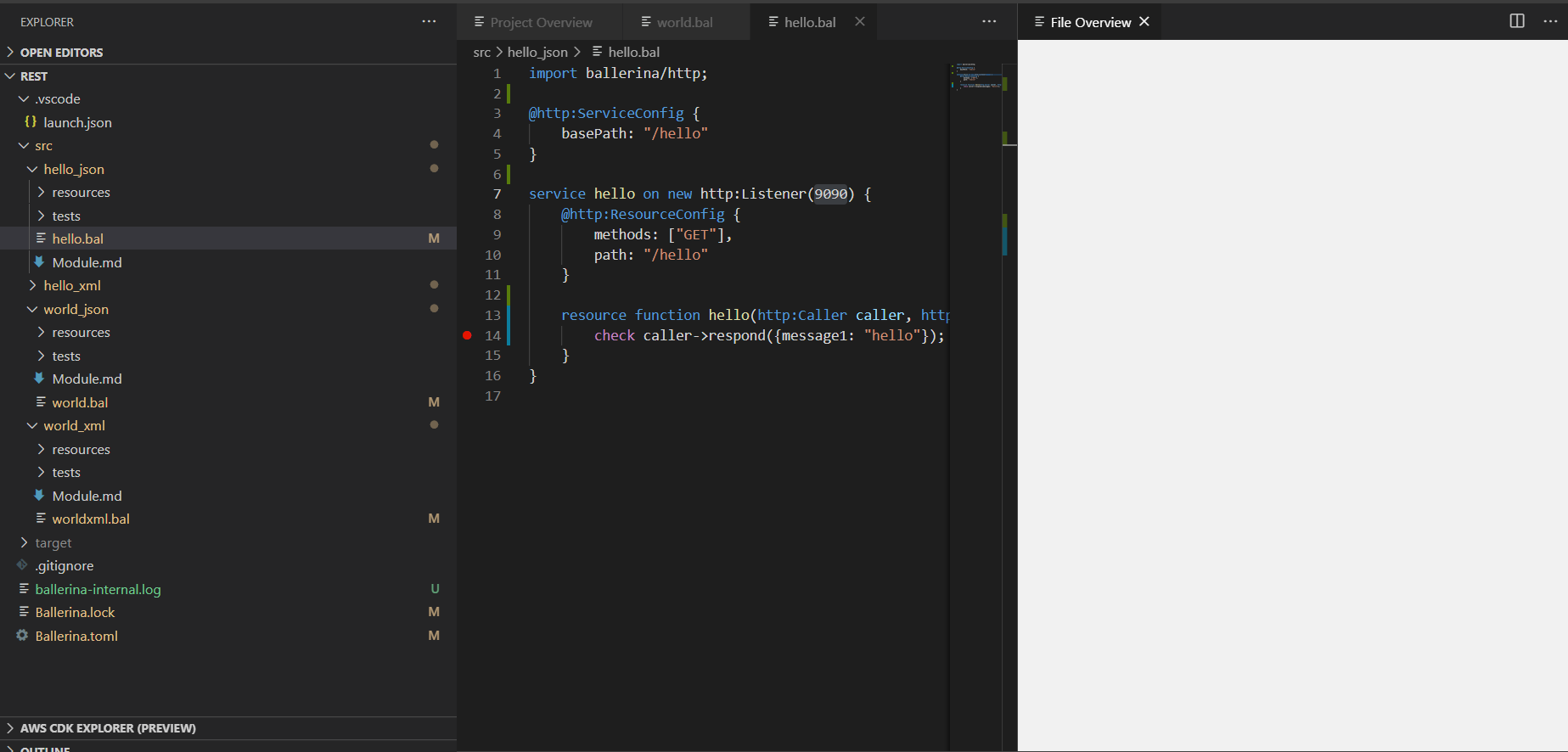
For developing services in Ballerina, Visual Studio Code was used. Visual Studio Code required installation of additional plugins for the development of the services. The additional plugins are called as extensions in Visual Studio Code, the extension for Ballerina was installed. The image shows how the Ballerina extension was installed from inside Visual Studio Code. There is button to install/uninstall the extension.



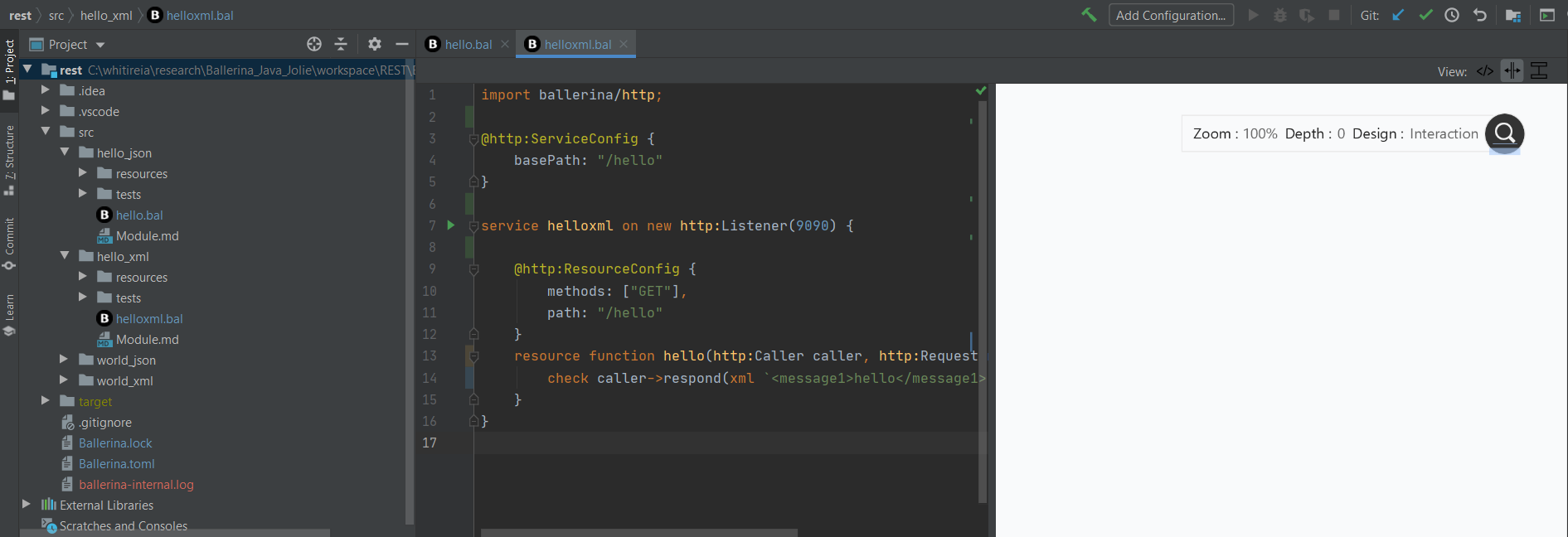
After the installation of the Ballerina extension, new window called Ballerina Project Explorer appears in the left panel as shown in the image. As per the documentation, this editor should visually display the service. (<https://ballerina.io/learn/setting-up-visual-studio-code/graphical-editor/#launching-the-project-overview>). In the research, the Project overview screen did not display any diagram that represents the flow of the service. Given image is the screenshot of 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.



There is another window that comes as a part of the Ballerina extension is the “File Overview”. In this window, individual services can be viewed as the diagram instead of the entire project. However, even that window opened blank failed to load the diagram. The diagram below shows the empty window when clicked to view the File Overview for the hello.bal.

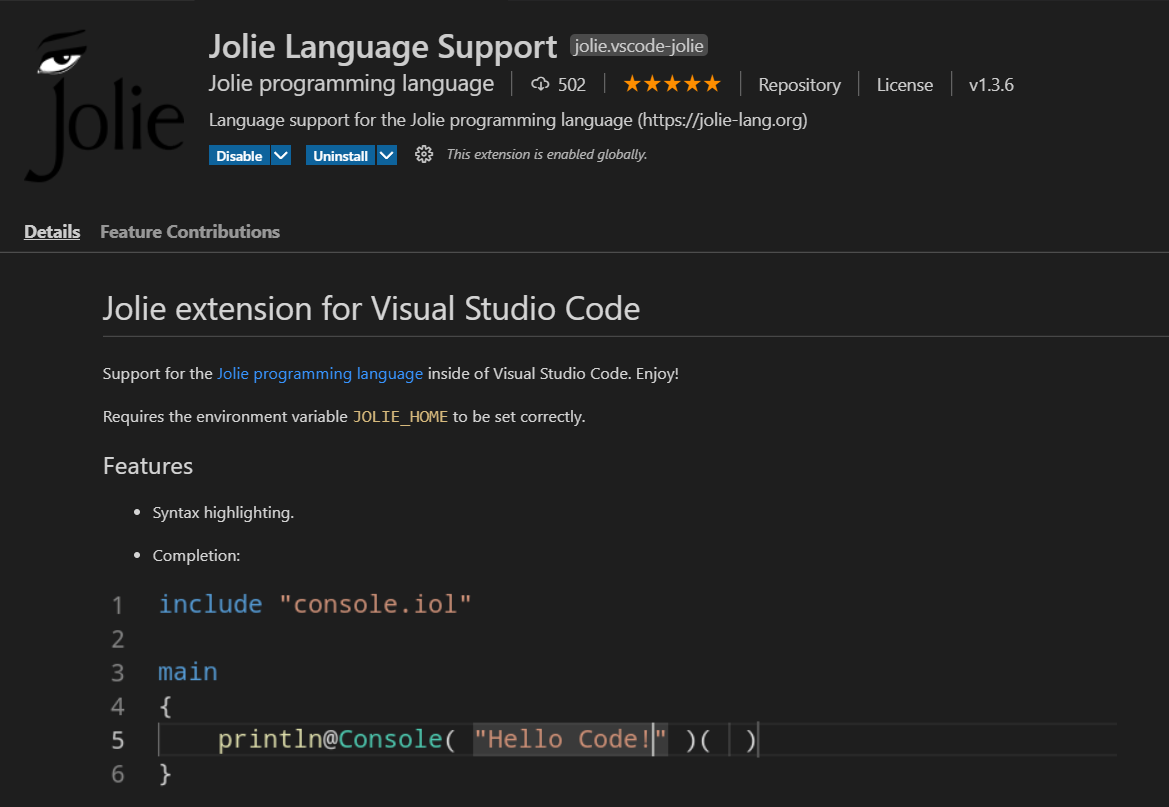


As this feature was supported in Ballerina but the research could not demonstrate the same, same project was tried in a different editor supported to develop the services in Ballerina. The new IDE used was IntelliJ – IDEA. However, nothing was displayed in the IDEA editor. The diagram below is the IDEA editor that displays the image of the window that fails to load the flow.



### Jolie

For developing services in Jolie, Visual Studio Code was used and the additional plugin called “Jolie language support” was installed. The image shows how the Jolie extension was installed from inside Visual Studio Code. There is button to install/uninstall the extension.



The extension for Jolie does not have the feature to display the service developed visually. Also, the official documentation of Jolie does 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, in the research the same could not be observed.

## Debug

The local debug feature of each language was tested in the respective IDE that was used for the development of the services. 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. Following was the testing scenario for debug –

* IDE should allow to add breakpoints.
* Service should run in the debug mode.
* Make the request to the services.
* Breakpoint is hit.
* IDE should allow the navigation such go to next step.
* Resume the services.

The debug scenario will remain the same for all language. However, the making the request part will vary for different integration technology scenario. This is explained in the integration technology section of every language.

### Java

#### ReST

In this experiment, local debug was tested in the NetBeans IDE for both the ReST services against the breakpoint added in the code. Debug feature in the IDE is only one click event. After selecting the debug project option, both the services are run in the debug mode and the test scenario was achieved for both the services. For the ReST service, the request is made to the second service only as second service calls first service before sending the response back. This will cause the first service to pause and both the services are tested for the debug scenario. Debug scenario was achieved for both the services without any challenges.

#### AMQP

Both the services are started in the debug mode with the breakpoints. For the AMQP, the request must be made to both the services separately as both the services do not communicate directly to each other. Thus, the first request is made to the hello service, that produces the message and sends to the RabbitMQ broker. Once the message reaches the RabbitMQ broker, world service is executed. The service execution should pause the service at the respective breakpoint. Debug scenario was achieved for both the services without any challenges.

gRPC

Both the services are started in the debug mode with the breakpoints. For the gRPC, the request is made like the ReST service as the world service communicates directly with the hello service. Thus, the debug scenario here involves making only one single request to the world service. Debug scenario was achieved for both the services without any challenges.

### Ballerina

Local debug was tested in the Visual Studio Code for Ballerina services. Debug required additional setup which involved performing below steps–

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

Ballerina ReST services were tested for the same debug scenario as the Java ReST services. There was a limitation observed when starting two services in debug mode. Both the services could not be started at the same time in the debug mode. Thus, the single request debug scenario could not be tested. To overcome this, both the services were tested separately with only one service started in debug mode and other service in regular mode (without debugging). All the other scenarios were tested successfully without any challenges.

#### AMQP

Ballerina AMQP services were tested for the same debug scenario as the Java AMQP services with the same limitation observed while running the ReST service. Only one service could be started in the debug mode. All the other scenarios were tested successfully without any challenges.

#### gRPC

Ballerina gRPC services were tested for the same debug scenario as the Java gRPC services with the same limitation observed while running the ReST service. Only one service could be started in the debug mode. All the other scenarios were tested successfully without any challenges.

### 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 the research is focuses on the integration so the total processing time is calculated for the entire integration process. Thus, the output of the execution time is calculated from the response time of the second service. 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, one of the ReST Clients called Postman is used. Postman has a tool called Collection Runner, that allows to make the request n number of times. Postman Collection Runner is used to call the service 100 times. Also, this tool allows to export the result of all the 100 runs into the json file. From the json file, the data is captured for the 100 runs and fed to the Excel sheet to calculate the average. From the collection runner, the request to the second service developed in all the three language is made.

For AMQP service, as both the services do not communicate directly calculating the time for the second service will not give the integration time. Thus, the total time first service starts, and second service ends is calculated, as this will give the entire integration time. Postman ReST client is used to make the request to the first service and to measure the execution time for the AMQP services. Postman Collection Runner is used to call the service 100 times. The console log of 100 runs is used for the calculation of the execution time. This log file has the timestamp of every execution including when the request started and completed. Finally, this log data is given as a input to the java application that reads the log and calculates the total execution time. This java application was developed for the research to read the log and calculate the difference of the timestamp of the when the first service started, and second service finished.

The execution time for gRPC is also derived from the total request response time of the second service as ReST service. To calculate the execution time of the gRPC service, the command “time” is used. To run the service 100 times, a powershell script is written where counter of 100 is maintained. The data from the console is fed to the excel sheet and the average of all runs is recorded.

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.