Checker Framework Integration for Editors and Integrated Development Environments (IDEs) based on the Language Server Protocol (LSP)

1 Introduction

Programming languages can be divided into two categories: statically typed languages and dynamically typed languages. Examples of the former are C/C++ and Java, and ensuring that the type of a variable is correct can be more easily done in this kind of languages; since the type of a variable never changes once it is defined, which is enforced by the compiler, we never need to worry that a variable defined as an integer will ever hold an instance of string.

However, built-in type systems still cannot catch all errors that may cause a program to crash, and the most notorious error may be the null pointer exception (NPE) in Java. To resolve issues like this, a practical tool to use is pluggable type checkers[6] that are used as plugins to compilers. By inserting extra meta-information into the code, such as an annotation `@Nullable`, pluggable type checkers enable us to add more constraints to the source code without too much effort and changes to the original code.

Checker Framework[1] is probably the most commonly used tool to conduct such a job of pluggable type checking. It provides a simple way to define qualifiers and type checkers. In addition, it also ships with many built-in checkers that are ready for use out of the box. With the same set of parameters as a normal `javac` compiler, a developer does not need to learn much before utilizing this tool to enjoy the features provided by those many checkers.

In this project, we advance the easiness of using Checker Framework by one more step, by creating plugins for editors and IDEs based on Checker Framework. While running the Checker Framework by "compiling" source files from time to time is not too complex, it is still more attracting if we can get diagnostic information of our source code when editing the source file like getting auto-completes.

In conjunction with this, we also explore implementing such plugins using the Language Server Protocol (LSP)[2], a protocol that provides unified interfaces for plugins to communicate with editors and IDEs, which significantly eases the development and integration of plugins. In latter sections, we will show that our core module built with LSP, the language server, can be integrated into VS Code and IntelliJ IDEA with very few effort. We will also show that, with the help of another Undergraduate Research Assistant project, the language server can be easily integrated into Eclipse.

2 Related Work

2.1 Checker Framework

Proposed in [6], Checker Framework is a "pluggable" tool for Java compilers to conduct type-checking in a way that is stricter and richer in semantics. The most straightforward way to do type-checking is probably to use keywords that is built into the language and compiler to describe the metadata of variables, classes, etc. One example can be the keyword `final` in Java and `const` in C/C++. The problem of using keywords is that the whole toolchain will need to be modified to accommodate new keywords, and this can be difficult and time-consuming for an ordinary developer who only wants to add a seemingly simple constraint to variables.

Thanks to the evolution of Java, new functionalities were added, and they later formed the foundation of Checker Framework. In JSR 175[7], annotation was proposed and finally integrated into Java, and JSR 308[8] furthered it by including type declarations in the acceptable targets of annotations, which is used by Checker Framework to store metadata that defines constraints on variables. For instance, when using the Nullness Checker, a developer needs to add annotations, such as `@Nullable` and `@NonNull`, for Checker Framework to work:

```
@Nullable Object obj1;
@NonNull Object obj2;
```

With these annotations, we can express the constraints that `obj1` can hold a value of `null`, whereas `obj2` is not supposed to hold `null`, and by doing this, if there is a line that is dereferencing `obj1` without checking its value, the Nullness Checker will report an error:

```
[dereference.of.nullable] dereference of possibly-null reference obj1
```

In reality, there are more checkers that are available for use, and using them only requires running the customized script provided in the distribution of Checker Framework, whose arguments are compatible with those of `javac`.

2.2 Language Server Protocol

The Language Server Protocol (LSP) is a communication protocol, which is based on JSON-RPC, that is proposed by Microsoft to ease the development of plugins for source code editors. Previously, the common way of enabling syntax highlight, auto-complete, error detection and jumping through function definitions in source code editors is to develop a plugin for all editors. What this means is that for every editor, the developer will have to develop a plugin in a language that is supported by that editor, usually the language the editor is implemented in, and this also usually requires the developer to implement all those complicated language-aware features again and again.

Then LSP comes; it (greatly) abstracts the editor from the plugin. Instead of developing the plugin based on the interfaces provided directly from the editor, which are different between different editors, now we only need to develop against the unified interface provided by LSP. In other words, LSP is a bridge between editors and plugins.

In practice, developers of plugins will now develop a single backend service, the language server, for a language that implements all the features, in any language, and a simple wrapper plugin in the language required by the editor, in which the actual communication between the language server and the editor will be set up, and this communication can be done using the standard input/output.

In later sections of this report, we will show that with an existing language server based on Checker Framework, the integration of it in IntelliJ IDEA can be completed by filling only one string configuration.

3 Components of the Plugin

3.1 Overview

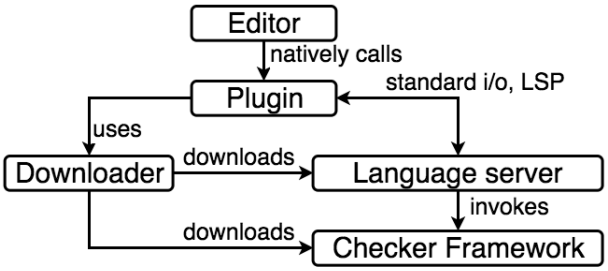

Figure 1: Relation of the three components.

In this section, we will introduce our design and implementation of the language server, a dependency downloader which acts as a helper for the plugin, and the actual plugin for VS Code. The first two are written in Java only because Checker Framework is written in Java, and the plugin is written in Typescript as advised by VS Code. They can be found at `https://github.com/eisopux/`.

These three components have some interactions when working, as shown in Figure 1. The plugin for VS Code communicates directly with VS Code, and it is the first to run among all components. During initialization, it checks if the language server and Checker Framework exists on the user's disk, and if they do not, it uses the downloader, which is bundled with the plugin, to download the latest versions of them from Github.

When the plugin finds that all dependencies are satisfied, it launches the language server, the core of the whole project, whose standard input and output follow the Language Server Protocol. After initialization, events such as the opening of a `.java` file is sent from the editor to the plugin natively, and then this event is sent to the language server by the plugin using LSP via standard input/output. When Checker Framework detects any error in a source file, the diagnostic information is sent in the reverse direction from the language server to the editor.

3.2 Language Server

This is the main module of the entire plugin that actually invokes the Checker Framework. It is a standalone program that can be launched, and as described previously, it uses the standard input/output

to communicate with other programs, following the Language Server Protocol. This module is primarily based on LSP4J[9], an implementation of LSP in Java.

As a concrete example, if an editor opens a source file `SomeClass.java`, then the input to the language server that is generated by the LSP library from the editor side is:

```
Content-Length: 671

{
    "jsonrpc": "2.0",
    "method": "textDocument/didOpen",
    "params": {
        "textDocument": {
            "uri": "file:///Users/joe/workspace/testproj/SomeClass.java",
            "languageId": "java",
            "version": 1,
            "text": "import org.checkerframework.checker.nullness.qual.*; ..."
        }
    }
}
```

The input is similar to an HTTP request. Note that the field `text` is truncated; in reality it is the content of the file. After the source file is checked by our language server, the output, following the Language Server Protocol as well, could be:

```
Content-Length: 759

{
    "jsonrpc": "2.0",
    "method": "textDocument/publishDiagnostics",
    "params": {
        "uri": "file:///Users/joe/workspace/testproj/SomeClass.java",
        "diagnostics": [
            {
                "range": {
                    "start": {
                        "line": 6,
                        "character": 4
                    },
                    "end": {
                        "line": 6,
                        "character": 8
                    }
                },
                "severity": 1,
                "code": "compiler.err.proc.messager",
                "source": "checker-framework",
                "message": "[dereference.of.nullable] dereference of possibly-null reference
                    obj1"
            }
        ]
    }
```

```
}
```

The response is not difficult to understand; it returns some diagnostic information to the editor, which specifies the path of the source file, the row and column range of related text, etc.

Several classes are created for this module, and they will be introduced below in top-down order. An overview of how information flows between the classes of this module is shown in Figure 2. Since the plugin only invokes the main entry point once, and the main entry point of the language server only starts one instance of the language server class, the lines between `Plugin`, `ServerMain` and `CFLanguageServer` are dotted to indicate this transience.

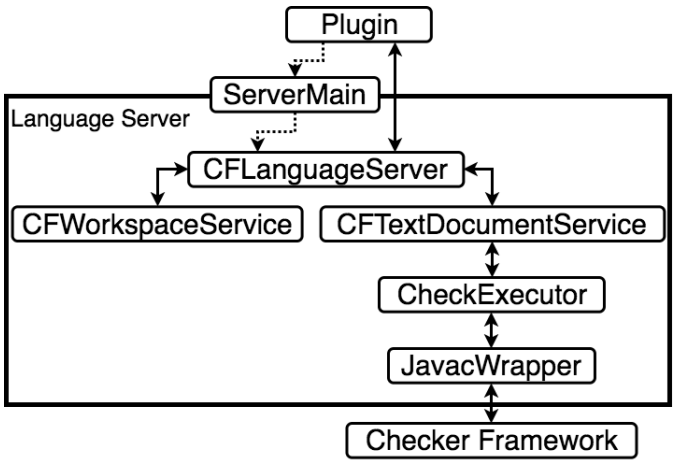

Figure 2: The flow of information within the language server.

3.2.1 ServerMain

`ServerMain` is the main entry point of this module. First it accepts command line options, including the path to the Checker Framework, list of checkers to use, and command line options that are going to be passed to Checker Framework. After this, it instantiates the language server class, `CFLanguageServer`.

3.2.2 CFLanguageServer

This class is the interchange for the source code editor client and other classes that do the actual dirty work. Several important components are in this class as member variables.

- `Settings settings`: this variable stores configuration information that can be set in the editor for this plugin, which are the same as the command line options. Although we can launch this language server with command line options, but LSP requires that the language server is able to accept new configuration information on the fly, and the reason command line options are needed is that some editors, such as Eclipse, do not send configuration information during initialization, so the language server will need a set of initial values.

- `CheckExecutor executor`: this is the (first-level) wrapper of Checker Framework; it accepts a list of files to check, and publishes diagnostics to the editor if there are any. We will describe its implementation in detail later.
- `CFTextDocumentService textDocumentService`: this variable receives events of a file from the editor, e.g. open, change, save, close. In our language server, a file is checked when it is opened or saved, and this is done by passing the file to `CheckExecutor` in respective methods. When a file is closed, it also clears diagnostics of this file, so the pane showing problems in editors only shows information about open files.
- `CFWorkspaceService workspaceService`: this variable receives events about the workspace (usually a folder opened in an editor). Currently it only propagates the event of changed configuration to other parts of the language server, so a new CheckExecutor reflecting the new configuration can be created.

3.2.3 CheckExecutor

This class is responsible for launching Checker Framework and using it to type-check files. Because in order to launch Checker Framework correctly, the class path and the bootstrap classpath need to be properly set to include several jar files of Checker Framework, and it is not very ideal to hard code all these arguments, the launcher class of Checker Framework, `CheckerMain`, is used by this class to retrieve arguments.

As you may notice, launching Checker Framework in this way is a little odd, as according to the documentation, we should be able to use a script that is already bundled with Checker Framework which accepts the same arguments as a normal javac. The reason we run Checker Framework in a programmatic way like this is that by doing this, we are able to get diagnostic information as Java objects (`javax.tools.Diagnostic`), otherwise we will have to parse the lines printed by Checker Framework, which is neither robust nor easy to implement.

Besides, there are two things we need to mention about starting Checker Framework. The first is that Checker Framework is started in a separate process. This is because the jar files needed by Checker Framework are very different from those needed by the language server, and we decided to give it a more dedicated environment to run to avoid any issue.

The second is that instead of starting Checker Framework directly, we launch a wrapper of it, `JavacWrapper`, which makes the communication between this class and Checker Framework easier. Recall that we want to run Checker Framework programmatically to obtain `javax.tools.Diagnostic` and at the same time we want to give it a dedicated environment. These two goals are actually conflicting; we can't fully separate a module while retaining the ability to call its methods. Thus, we choose to sacrifice the separation a bit by creating a very thin wrapper for Checker Framework, which can programmatically call the methods of Checker Framework and is also simple enough to not be affected by the environment of Checker Framework.

3.2.4 JavacWrapper

This class simulates the behavior of the actual Checker Framework as much as possible, and the only difference lies in its format of input and output, which is tailored for `CheckExecutor`.

This class accepts the same parameters that can be used to launch Checker Framework, except that it does not accept paths to files we wish to check. Upon launching, it continuously reads lines of paths to files from its standard input, which is piped by `CheckExecutor`. After the files are checked, diagnostics for them represented as `javax.tools.Diagnostic` objects are obtained.

In order to return diagnostics to `CheckExecutor`, they have to be serialized and printed to the standard output for `CheckExecutor` to read. However, `javax.tools.Diagnostic` cannot be serialized directly, so we first convert them into a custom class, and then convert this class to JSON. After this, these JSON objects representing the original diagnostics are printed to the standard output, which is piped by `CheckExecutor` as well and will finally be read and parsed, finishing the entire checking process.

3.3 Dependency Downloader

This is a utility program for plugins written for editors to get the latest version of language server and Checker Framework, so developers of different plugins do not need to bundle a fixed version of language server and Checker Framework with their plugins.

The implementation of this program is fairly straightforward; it uses the Github API to retrieve the latest releases of the language server and Checker Framework, and then downloads the corresponding assets. For example, the API used for Checker Framework is `https://api.github.com/repos/typetools/checker-framework/releases/latest`, and the response is a JSON object:

```
{
  ...
  "tag_name": "checker-framework-3.0.1",
  "name": "Checker Framework 3.0.1",
  "assets": [
    {
      ...
      "name": "checker-framework-3.0.1.zip",
      "browser_download_url": "https://github.com/typetools/checker-framework/releases/download
          /checker-framework-3.0.1/checker-framework-3.0.1.zip"
    }
  ]
}
```

With this, it is able to download the language server and Checker Framework, as long as they are published as a release and provides a distribution file.

The arguments of this program include a path to a folder in which the files will be downloaded, the organization and repository to download Checker Framework and language server. It prints the resulting jar file of language server and directory containing the unzipped Checker Framework to the standard output, starting with "Got: ". A sample run of this program is:

```
$ java -jar checker-framework-languageserver-downloader-all.jar \
-dest ~/Downloads -checkerframework_org typetools
Downloading from https://github.com/.../checker-framework-languageserver-0.1.0.jar \
to /Users/joe/Downloads/checker-framework-languageserver-0.1.0.jar
```

```
Got /Users/joe/Downloads/checker-framework-languageserver-0.1.0.jar
Downloading from https://github.com/.../checker-framework-3.0.1.zip \
to /Users/joe/Downloads/checker-framework-3.0.1.zip
Got /Users/joe/Downloads/checker-framework-3.0.1
```

3.4 VS Code Plugin

This program is the glue between VS Code and our language server, and is the only editor-specific module. It is developed with the implementation of LSP in VS Code, vscode-languageserver-node[11]. Only 1 method is required to be implemented by us, `activate`, which is called when this plugin is activated the first time a Java file is opened/changed/saved/closed.

In this module, we first check if Checker Framework and the language server exists on the user's computer, and run the downloader if they do not exist. If we need to download the language server and Checker Framework, their paths are set in the configuration file of the workspace opened in VS Code.

After all these are done, commands for launching our language server is returned, and then vscode-languageserver-node takes cares of the rest, including starting the language server and passing request-s/receiving responses.

With this design, users do not need to set anything except the JDK environment to enjoy the features of Checker Framework.

4 Usage Examples

4.1 VS Code

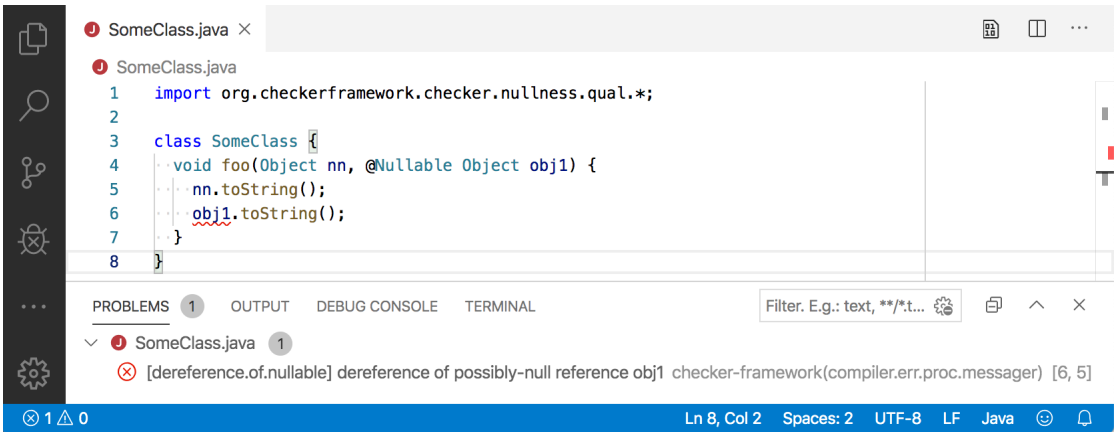

Figure 3: Detecting dereferencing null in VS Code.

For instructions on installing this extension, please refer to `https://github.com/eisopux/checker-framework-vscode`, as it may change. A basic usage of this plugin would be to use the Nullness Checker of Checker Framework to detect possible dereferencing of `null`s. A screenshot of this is Figure 3.

The argument of the method `foo`, `obj1`, is annotated with `@Nullable`, meaning that its value is possible to be `null`, so executing line 6 could result in a null pointer exception. This problem reported by our language server is listed in the PROBLEMS pane, and a squiggly line is drawn under the related text.

4.2 IntelliJ IDEA

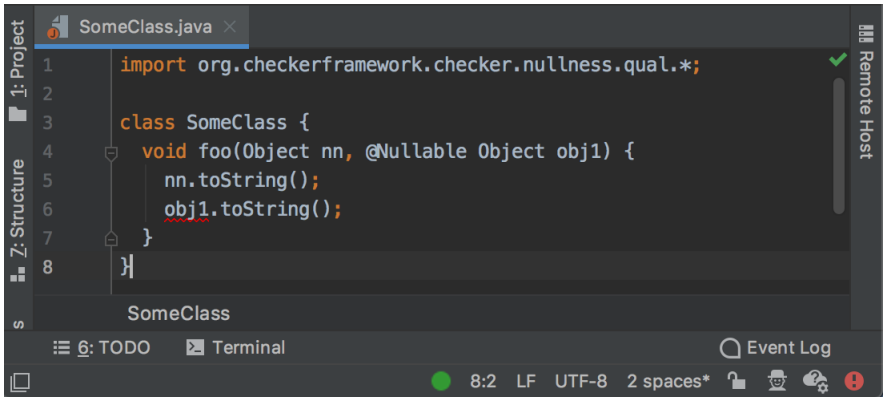

Figure 4: Detecting dereferencing null in IntelliJ IDEA.

IDEA has a plugin providing the LSP support[12], and we will use it to demonstrate the generality of our language server.

Although it is not as easy as using the extension for VS Code, but it is still very quick to set everything up. First, we need to download the language server and Checker Framework, and then unzip Checker Framework. Then, we need to provide the command for launching the language server to this plugin. A sample command could be:

```
java \
-cp /env/languageserver.jar:/env/checker-framework/checker/dist/checker.jar \
org.checkerframework.languageserver.ServerMain \
--frameworkPath /env/checker-framework \
--checkers org.checkerframework.checker.nullness.NullnessChecker
```

and the place to input this command is Preferences - Languages & Frameworks - Language Server Protocol - Server Definitions, then choose “Raw command”, and for the extension “java”. The errors detected are also shown as a squiggly line in the editor, as shown in Figure 4.

4.3 Eclipse

Unlike IntelliJ IDEA, there has not been a generic LSP-support plugin yet, so some code has to be written to use the language server. In Eclipse, LSP4E is the bridge between Eclipse and our language server, and it is used by the plugin implementation for Eclipse. This library defines several classes and interfaces that we can inherit or implement to establish the communication between Eclipse and our language server. While it is not as easy as building the VS Code plugin, its responsibility is pretty

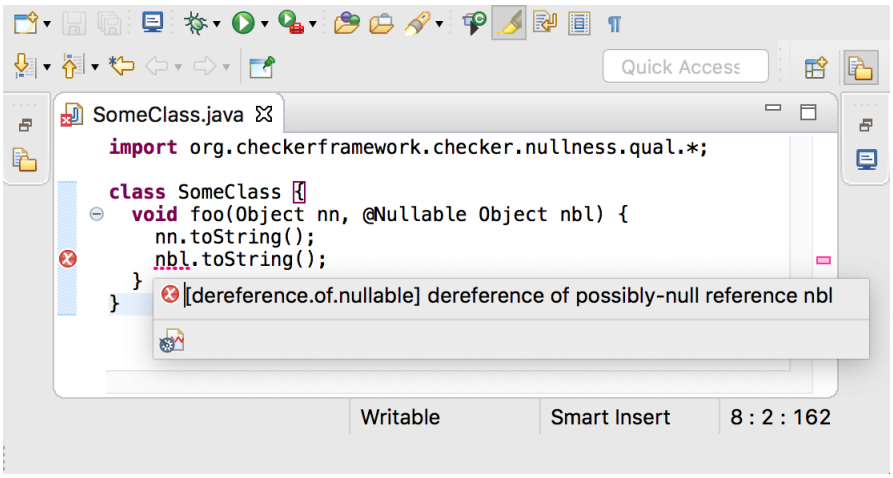

Figure 5: Detecting dereferencing null in Eclipse.

similar its VS Code counterpart: download and launch the language server upon activation, and pass parameters set by the user to the language server.

5 Future Work

In this project, we have implemented most of the functions required to have type-checking in our editors that have Language Server Protocol support, but in advanced IDEs like IntelliJ IDEA, the functionality of intelligently proposing fixes and refactors for problems is not rare, and make us ask: is it possible for Checker Framework to do this as well?

The answer is positive. Based on Checker Framework, Checker Framework Inference[13][14] is able to infer type annotations given a program with no annotations. The language server will be worth of much higher value for developers if it supports both type-checking and generating type annotations. Similarly, it is also very worthwhile to explore the possibility of refactoring code to resolve type errors.

In terms of the design and implementation of language server, the room for improvement clearly exists. Contrary to the practice of defining the type of member variables and arguments of methods as interfaces, concrete classes are mostly used in this project, which is not very scalable and violates the rule of high cohesion/low coupling. In the future, it would be beneficial to review the methods provided by each class and interface, and probably re-design the relation between classes.

Lastly, it would be ideal if more tests can be created and run, especially for the plugin module. The environments when running the plugin will certainly vary greatly from user to user, and there may be many cases that we forget to handle. What if the operating system is Windows? What if the user wants to download Checker Framework from another organization? What if ...?

6 Conclusion

In this report, we gave introductions on Checker Framework, a type-checking framework for Java language that is based on annotations, and Language Server Protocol, a communication protocol for

unifying and easing the development of plugins for source code editors. Based on these, we described the design and implementation of a plugin for VS Code which can provide the functionalities of Checker Framework in a very user-friendly way.

As shown, our core module, language server, can be easily integrated into any editor that has support for LSP, IntelliJ IDEA and Eclipse is used as an example. This demonstrates the potential of LSP, which significantly decreases the difficulty of developing plugins by separating the setting up and actual processing of source files.

References

[1] Checker Framework, `https://checkerframework.org/`.

[2] Language Server Protocol, `https://microsoft.github.io/language-server-protocol/`.

[3] Dietl W, Dietzel S, Ernst MD, Muşlu K, Schiller TW. Building and using pluggable type-checkers. In Proceedings of the 33rd International Conference on Software Engineering 2011 May 21 (pp. 681-690). ACM.

[4] Visual Studio Code, `https://code.visualstudio.com/`.

[5] IntelliJ IDEA, `https://www.jetbrains.com/idea/`.

[6] Papi MM, Ali M, Correa Jr TL, Perkins JH, Ernst MD. Practical pluggable types for Java. InProceedings of the 2008 international symposium on Software testing and analysis 2008 Jul 20 (pp. 201-212). ACM.

[7] JSR 175: A Metadata Facility for the Java Programming Language, `https://jcp.org/en/jsr/detail?id=175`.

[8] JSR 308: Annotations on Java Types, `https://jcp.org/en/jsr/detail?id=308`.

[9] LSP4J, `https://projects.eclipse.org/projects/technology.lsp4j`.

[10] LSP4e, `https://projects.eclipse.org/projects/technology.lsp4e`.

[11] vscode-languageserver-node, `https://github.com/microsoft/vscode-languageserver-node`.

[12] LSP Support for IDEA, `https://plugins.jetbrains.com/plugin/10209-lsp-support/`.

[13] Li, Jianchu. A General Pluggable Type Inference Framework and its use for Data-flow Analysis. MS thesis. University of Waterloo, 2017.

[14] Checker Framework Inference , `https://github.com/typetools/checker-framework-inference`.