

Sonatype Lift

Table of Contents

Table of Contents	1
About Sonatype Lift	6
Getting Started	6
GitHub App Installation	6
Analyze Your First Repository	7
Running Lift	10
Configuring Lift	11
Languages	11
Java: Specifying Build System	11
Java: Android Projects	12
Java: Specifying Tools	12
C and C++	13
JavaScript	13
Python	14
Ruby	14
Haskell	14
Environment Variables	15
Local Settings	15
Dependencies	16
Downloading Through Github	16
Using Apt Package Manager	16
Something Else	16
Infer Configuration	17
Pyre Configuration	19
Extending Lift	19

Execution Environment.....	19
APIv1	20
API Commands	20
BYO API Hello World.....	21
Suggested Development Process.....	22
Development Process Steps	22
Development Process by Example	23
Talking to Lift	24
Getting Liftbot's Attention	24
Ignore - Mark a Report as False Positives	24
Unignore - Undo an Ignore Reports.....	25
Help - Get Help from Liftbot	25
Troubleshooting.....	25
Lift thought there was a bug, but I'm confident it is wrong (impossible).....	25
Lift couldn't understand my build system / complained about not being able to produce a compilation database	26
I have a multi-project repository and I want Lift to only run on one particular project.....	27
My custom build script has the error "E: Unable to locate package 'packagename'"	27
My Android project is not building in Lift.....	27
I don't have my own repository, but I still want to try Lift.....	27
I am wondering what utilities are available to Lift analysis by default.....	29
I have a problem that is not addressed here	30
Included Analyzers.....	30
Infer	30
ErrorProne	30
FindSecBugs	31
Quandary	31
ESLint.....	31
Semgrep	32

Staticcheck.....	32
Golang-ci.....	32
Pyre	32
Bandit.....	33
Rubocop	33
Detekt.....	33
ShellCheck.....	33
Cobra	33
Clippy	34
HLint	34
Psalm.....	34
LuaCheck.....	34
Markdownlint - MDL (Disabled by Default).....	35
PMD (Disabled by Default)	35
Checkov (Disabled by Default)	35
Open Source Vulnerability Analyzer.....	35
Available ecosystems.....	36
Golang Analysis	37
Javascript Analysis.....	37
JVM Analysis.....	38
Python Analysis.....	39
Rust Analysis.....	40
Configuring Analyzers	41
Infer Quandary.....	41
Pyre/Pysa	41
Configuration Reference.....	42
In-Repo Options (The .lift.toml).....	42
Build System Support	44
Subprojects	45
APIs for Custom Tools	45

Example Extended Checks - Fixing the FIXME	46
Example Extended Checks - ShellCheck and PMD	46
Extended Tooling with Custom or Community-provided Tools	47
Security FAQs	47
Need More Help	48
Lift Pro Users (including free trials)	48
Lift Free Users	49
Deleting an Account	49
Scripts	49
Check Common Comments	49
Check Lift API	51
ESLint	53
goparamcount	54
Hello Lift	56
Line Length	57
mdl	59
pmd	60
Shell Check	61
Static Check	62
Extending Lift V3	64
Tool Invocation Format	64
Command: (none)	65
Command: applicable	65
Command: version	65
Command: run	65
Command: finalize	66
Command: talk	66
Data Types	67

Running Lift via GitHub Actions	68
Introduction	68
Installation Method	68
Docker Image Access	68
GitHub Actions YAML	68
Add Secrets to GitHub	71
Operation Walk Through.....	71
Debugging	71
Self Hosting Lift + GitLab	71
Overview	71
Performing Installation	72
Install Webhooks and Use the Application.....	73
GitHub Actions	73
Installation Method	73
Docker Image Access	73
GitHub Actions YAML	73
Add Secrets to GitHub	76
Operation Walk-Through.....	76
Debugging	76
Self Hosting Lift + GitHub.....	76
Overview	77
Create a GitHub App	77
Performing an Installation.....	78
Install The Lift GitHub App	79

Sonatype Lift is an innovative, code analysis platform built for developers to help them find and fix critical security, performance, reliability, and style issues in their code. Every time a developer submits a pull-request, Lift automatically analyzes their code, and then reports bugs as comments in code review, alongside other developers.

Lift overcomes the challenges of conventional code analysis tools by providing accurate and actionable feedback to developers at the right time and right place - during peer code review where developers are 70x more likely to fix bugs.

About Sonatype Lift

Lift integrates deep static analysis tools into your pull request workflow. Lift is packaged as a GitHub app and interacts primarily through GitHub. When enabled, LiftBot will analyze every new PR and then post PR comments highlighting any bugs it finds in the changed code. Because LiftBot focuses on the changed code and uses very precise tools, it tends to be very quiet (typically 2 or fewer comments per PR). Because Lift does a deeper analysis of your code than other platforms, it needs to understand how your code is built. Most of the time Lift can auto-detect your build system and automatically configure all the included analysis tools. Sometimes Lift needs you to specify some information about the build using a simple config file placed in the repository.

Read our [Getting Started Guide](#)(see page 6) to enable Lift on your repositories.

Getting Started

The Lift app is installed and managed through the GitHub marketplace at <https://github.com/apps/sonatype-lift>. Getting started with Lift is as easy as:

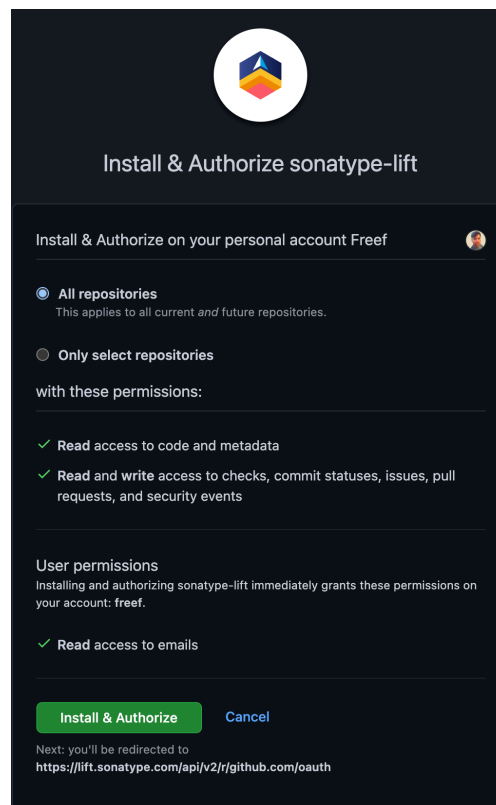
1. Installing the GitHub app
2. Selecting a few repositories to analyze
3. Viewing any bugs found in the Lift Console

NOTE: These steps let you quickly see existing issues within your code base. Lift normally runs automatically at each pull request and reports only on new errors, delivering results as inline code review comments. Learn about our developer-centric approach to code analysis at [Running Lift](#)(see page 10) for more information.

GitHub App Installation

1. Login to your GitHub account and go to the [application settings page](#)¹. Click `install`.
2. Lift can be configured to analyze all of your repositories, or a select subset of them. Chose one of these options from the dialog, as show below. Click `install`.

¹ <https://github.com/apps/sonatype-lift>





3. Review the application permissions and click **Install & Authorize sonatype-lift**.
4. Once the app is installed, you will be redirected to the Lift Console. Lift will now attempt to analyze pull requests for your repositories.

Analyze Your First Repository

When visiting [the Lift Console](https://lift.sonatype.com/)², you will see a list showing all repositories you have installed the Lift app on (for each organization separately).

² <https://lift.sonatype.com/>




Repositories
Community
Documentation





Repositories / Matt-Freeland

Repositories

ⓘ We're analyzing your repos. The status will update when they're ready. Click on an item in the Recent Activity tab to access analysis results directly, or click on a repo to see a list of recent checks for this repo.

 Matt-Freeland

This lists the status of all repositories Lift has analyzed

 Matt-Freeland/hyh-test	<div>main</div> <div>Analyze</div> <div>Most recent analysis succeeded</div>
 Matt-Freeland/vim-macdown	<div>master</div> <div>Analyze</div> <div>Most recent analysis succeeded</div>
 Matt-Freeland/Markdown-Files	<div>master</div> <div>Analyze</div>

From this screen, you can select any repository for Lift to get started analyzing. Simply click the “Analyze Now” button. You will then be forwarded to the analysis view where you can watch Lift inspect your code.

If you want to try out Lift, but you don’t have a suitable repository on-hand, see [this guide](https://help.sonatype.com/lift/troubleshooting-78578731.html#Troubleshooting-no-suitable-repo)³ to clone a test repo and see Lift working.

³ <https://help.sonatype.com/lift/troubleshooting-78578731.html#Troubleshooting-no-suitable-repo>

[Repositories](#) / [Matt-Freeland](#) / [hyh-test](#) / Analysis

Analysis Report

i This job is analyzing, this might take a while — about twice as long as a build. ×

ANALYZING

⌚ Loading...

Status updated: "a few seconds ago"

Analysis began: "a few seconds ago"

[Results](#)
[Tool Results](#)
[Build Logs](#)

...
Analysis in progress, please wait

The yellow color of the page indicates that Lift is building and analyzing your code. This can take some real time (roughly twice the normal build time of your project), so feel free to stretch, grab a cup of coffee, or whatever. Lift will take it from here. Once Lift has finished analysis, you will be able to view the results by clicking on the results tab. If there is a problem with analysis you may also click to see the failing build log. See [Troubleshooting](#)(see page 25) page for help if there is a build error.

[Repositories](#) / [Matt-Freeland](#) / [hyh-test](#) / Analysis

Analysis Report

ANALYSIS COMPLETE

Analysis began: "6 days ago"

[Results](#)
[Tool Results](#)
[Build Logs](#)

« 

 Filter

▼ Closure-Compiler

☐ All Types



1

☐ JSC_PARSE_ERROR

1

JSC_PARSE_ERROR

Parse error. ',' expected

 [pages/index.js](#) 17:2  Closure-Compiler

[Show Details](#)

If you made it here, you're all good to go! You can see the existing bugs that Lift found in your repositories, and any installed repositories will be analyzed every time you create a pull request in GitHub. For more information on that workflow see [Running Lift](#)(see page 10). Happy Lifting!

Running Lift

Lift's pull-request workflow is the primary way to interact with Lift. Once you have installed Lift on a repository which it can build and analyze (see [Getting Started](#)(see page 6)), you are ready to use the pull-request workflow. Lift will automatically analyze your repository each time a pull request is submitted to your repository, comparing the source and destination branches.

After analysis, Lift will report any introduced bugs that it finds as line comments on the GitHub pull-request diff. Any bugs not introduced by the pull-request or falling outside of the diff can be viewed on the [Lift Console](#)⁴.




The screenshot shows a GitHub pull request interface. At the top, a comment from the **lift-dev** bot is displayed, stating it was reviewed 4 minutes ago. Below the comment, a code diff for the file `src/acvp_enable_cap.c` is shown. The diff highlights four lines (1661-1664) with green markers, indicating additions. The code is as follows:


```
1661 + return ACVP_MALLOC_FAIL;
1662 + }
1663 +
1664 + return (acvp_append_cmac_caps_entry(ctx, cap, cipher, crypto_handler));
```

Below the code diff, the bot's comment is repeated: **lift-dev** (bot) 4 minutes ago. The comment text reads: **MEMORY_LEAK:** memory dynamically allocated by call to `calloc()` at line 1659, column 11 is not reachable after line 1664, column 5. At the bottom of the comment, there is a "Reply..." input field and a "Resolve conversation" button.



Lift will also take advantage of GitHub's status API to let you know if there were any new bugs introduced by the pull request:


⁴ <https://lift.sonatype.com/>




All checks have failed
[Hide all checks](#)

1 failing check



liftdev — Analysis completed (2 new bugs found)
 [Details](#)


This branch has no conflicts with the base branch

Merging can be performed automatically.

Merge pull request

▼

You can also [open this in GitHub Desktop](#) or view [command line instructions](#).

The pull-request workflow is the best way to use Lift. It will give you feedback on your code changes without any additional effort. Especially if you are already using a pull-request based workflow, Lift will simply be there like a trusted teammate, always ready to give you a thorough code review. Now that you have the basics down, check out the [Configuration Guide](#)(see page 11) to get the most out of Lift.

Having any problems? See our troubleshooting page for a [pull-request example](#)⁵ to make sure everything is configured properly.

Configuring Lift

Following are `.lift.toml` configuration guides which encapsulate common workflows. For a more detailed reference of all Lift configuration options, see the [Configuration Reference](#)(see page 42).

Languages

Java: Specifying Build System

Lift can guess which build system should be used in each directory by looking for common build files such as `pom.xml`, `build.gradle` and `gradlew`. Sometimes the detected build is not preferred, perhaps because there are several such files present. An explicitly specified build can be used to disable the detection and assume one system. Consider the below example `.lift.toml` file that specifies use of Maven and `jdk11`:

```
build = "mvn"
jdk11 = true
```

Build targets, flags, and environmental variables can be added to the build line too:

⁵ <https://help.sonatype.com/lift/troubleshooting-78578731.html#Troubleshooting-no-suitable-repo>

```
build = "gradle -DskipTests=true build"
```

NOTE: the build line is not executed verbatim and can not be a shell command. For each build system Lift has plugins to better understand the project files and dependencies. Normal behavior is if it builds then Lift will analyze the code but the difference surfaces rare cases. If the project's build system is layered, such as a gradle build that just calls out to Maven, then the build will not be understood. If your project's build falls into such a category then consider simplifying the build system and using a supported build system in a canonical manner.

Java: Android Projects

Android projects in Lift require special configuration. Android projects generally do not support JDKs later than 8, so the `jdk11` flag should be set to false. The android SDK version is required to be specified for all android projects.

Here is an example configuration for an Android project:

```
build = "gradlew"
arguments = [ "assembleAndroidTest" ]
jdk11 = false
androidVersion = 28
```

Lift supports android SDK versions 27 and 28.

Java: Specifying Tools

For Java, Lift uses multiple tools to analyze the code. Documentation for each of these tools can be found [here](#):

- [Errorprone](#)⁶ - for common programming mistakes
- [FindSecBugs](#)⁷ - security audits
- [Infer](#)⁸ - potential bug finder

Lift is quite by default. Large projects should consider further tuning Lift via explicit selection of which bugs to report and tools to run. For example, to run only Infer and FindSecBugs but ignore Infer's `RESOURCE_LEAK` message use a configuration of:

```
tools = ["infer", "findsecbugs"]
ignore = ["RESOURCE_LEAK"]
```

⁶ <https://errorprone.info/docs/flags>

⁷ <https://find-sec-bugs.github.io/bugs.htm>

⁸ <https://fbinfer.com/docs/all-issue-types>

C and C++

Lift supports C and C++ along with the common build systems of cmake, make, compilation databases, and GNU autotools including detecting then running `autogen.sh` and `configure`.

C and C++ analysis results are provided by Infer. Infer's bug reference documentation can be found on [their website](#)⁹. An example configuration that specifies the make build system and infer tool is:

```
build = "make"
tools = ["infer"]
```

As with Java, the build line can specify environmental variables and argument but it is not itself a place to write arbitrary shell script. If it is necessary to execute a script prior to build then read the [dependencies section](#)(see page 16).

If your project's build system is complex then consider writing a setup script to generate a compilation database. For example define a `.lift/setup.sh` file of:

```
#!/usr/bin/env bash
# execute any build dependency steps here
sudo apt update && sudo apt install -y $DEPS
# Generate a compilation database next
CC=gcc compdb -n make all
```

Then in the `.lift/config.toml` specify the build as `compdb` so Lift will ignore other build systems such as Make or CMake:

```
build = "compdb" setup = ".lift/setup.sh"
```

The end results is you have full control over the compilation steps in a manner that is easy to inspect and debug.

JavaScript

ESLint is the tool that Lift uses to analyze Javascript projects. Lift respects ESLint configuration files and more information about the config files can be found [on their website](#)¹⁰.

Lift runs ESLint by default. To run *only* ESLint make `.lift.toml` specify the `tools` list as:

⁹ <https://fbinfer.com/docs/all-issue-types>

¹⁰ <https://eslint.org/docs/user-guide/configuring>

```
tools = ["eslint"]
```

Lift includes five additional commonly used configurations and plugins:

- [Airbnb Config](#)¹¹
- [Prettier Config](#)¹²
- [Prettier Plugin](#)¹³
- [Babel Plugin](#)¹⁴
- [Jest Plugin](#)¹⁵

Each must be configured individually in order to be used. For example, in order to use the Airbnb configuration with hooks support, simply add "extends": ["airbnb", "airbnb/hooks"] to your .eslintrc. Other plugins and configurations can be installed if desired by [using a setup script](#)(see page 16).

Python

Lift supports Python projects with Bandit and Pyre. By default, Pyre support operates by first finding requirements.txt or setup.py files, installing dependencies, and analyzing the project. Bandit will run on any repository containing files with the .py extension.

Ruby

Ruby is supported by Rubocop and, in the case of ruby on rails, Brakeman. Both these tools run when the repository contains files with the .rb extension.

Haskell

[HLint](#)¹⁶ is a Haskell analysis tool that is integrated with Lift. HLint can be configured by a .hlint.yaml at the working directory of the project. For more information on configuration, [view HLint's documentation](#)¹⁷.

To run *only* HLint make .lift.toml specify the tools list as:

```
tools = ["hlint"]
```

¹¹ <https://www.npmjs.com/package/eslint-config-airbnb>

¹² <https://www.npmjs.com/package/eslint-config-prettier>

¹³ <https://www.npmjs.com/package/eslint-plugin-prettier>

¹⁴ <https://www.npmjs.com/package/eslint-plugin-babel>

¹⁵ <https://www.npmjs.com/package/eslint-plugin-jest>

¹⁶ <https://github.com/ndmitchell/hlint#readme>

¹⁷ <https://github.com/ndmitchell/hlint#readme>

Environment Variables

Environment variables are defined in the `build` field of `.lift.toml`. For example, we can specify which C compiler to use when executing `make`:

```
build = "CC=gcc GXX=g++ CXX=g++ make"
```

Proxy settings with Bash environment variables can also be defined here.

```
build = "https_proxy=https://server:port make"
```

Local Settings

Some tools such as Maven use a `settings.xml` to specify custom maven repository servers. If the configuration file is not in the repository then the `setup` section of the Lift config can create that file.

```
setup = ".lift/createConfig.sh"
```

And the repository would include a `.lift/createConfig.sh` file of:

```
$ cat .lift/createConfig.sh
#!/usr/bin/env bash
mkdir ~/.m2
cat <<EOF > ~/.m2/settings.xml
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
    https://maven.apache.org/xsd/settings-1.0.0.xsd">
  <localRepository>${user.home}/.m2/repository</localRepository>
  <interactiveMode>true</interactiveMode>
  <offline>false</offline>
  ...
</settings>
EOF
```

N.B. Lift does not currently have a secrets vault and can not inject secret information, such as Artifactory or Nexus credentials, into the analysis process.

Dependencies

Many projects require external dependencies to be able to be built. We will look at some of the common ways to orchestrate Lift to download and build these dependencies.

The Lift analysis image is based on [Ubuntu 20.04](#)¹⁸. A complete list of included utilities is available [here](#)¹⁹.

The following `.lift.toml` will be used for these examples and `.lift/getDependencies.sh` is defined in each example.

```
setup = ".lift/getDependencies.sh"
```

Downloading Through Github

Adding another repository that is a dependency from GitHub.

```
$ cat .lift/getDependencies.sh
#!/usr/bin/env bash
git clone github.com/<someuser>/<someproject>.git
cd someproject/ && make
```

Using Apt Package Manager

The server used to analyze is managed with the apt package manager providing access to a rich environment of packages. Be sure to use `sudo` when invoking commands requiring root permissions.

```
$ cat .lift/getDependencies.sh
#!/usr/bin/env bash
sudo apt update
sudo apt install -y libcrypto-dev
```

Something Else

Is there any additional functionality you are missing? Check out the Lift [“Bring Your Own Tool” API](#)(see [page 19](#)) to learn how to extend Lift functionality with custom tools.

Still experiencing a build error or any other problem? We have a [Troubleshooting page](#)(see [page 25](#)) or feel free to [contact us](#)(see [page 48](#)) for personal assistance.

¹⁸ https://hub.docker.com/_/ubuntu?tab=tags&page=1&name=20.04

¹⁹ <https://help.sonatype.com/lift/troubleshooting-78578731.html#Troubleshooting-default-utilities>

Infer Configuration

If you want to configure Infer, such as specifying the sources and sinks for the quandary information flow analysis, then you may commit a `.inferconfig` file to the repository. Here is an example of Infer configuration, which defines some common sources, sinks, and sanitizers for SQL injection attacks (downloadable file version below the block):

```
{
  "quandary-sources": [
    {
      "procedure": "javax.servlet.http.HttpServletRequest.getParameter",
      "kind": "UserControlledString"
    },
    {
      "procedure": "javax.servlet.http.HttpServletRequest.getHeader",
      "kind": "UserControlledString"
    },
    {
      "procedure": "java.io.BufferedReader.read",
      "kind": "UserControlledString"
    },
    {
      "procedure": "javax.servlet.http.HttpServletRequest.getAttribute",
      "kind": "UserControlledString"
    }
  ],
  "quandary-sanitizers": [
    {
      "procedure": "org.owasp.encoder.Encode.forHtml"
    },
    {
      "procedure": "org.owasp.esapi.Encoder.encodeForSQL"
    },
    {
      "procedure": "org.apache.commons.lang.StringEscapeUtils.escapeHtml"
    }
  ],
  "quandary-sinks": [
    {
      "procedure": "java.util.logging.Logger.info",
      "kind": "Logging"
    },
    {
      "procedure": "java.util.logging.Logger.log",
      "kind": "Logging"
    },
    {
      "procedure": "java.io.PrintWriter.write",
      "kind": "Other"
    },
    {
      "procedure": "org.springframework.jdbc.core.JdbcTemplate.queryForObject",
```

```

    "kind": "SQLRead"
  },
  {
    "procedure": "javax.jdo.PersistenceManager.newQuery",
    "kind": "SQLWrite"
  },
  {
    "procedure": "org.hibernate.Session.createQuery",
    "kind": "SQLWrite"
  },
  {
    "procedure": "org.apache.turbine.om.peer.BasePeer.executeQuery",
    "kind": "SQLWrite"
  },
  {
    "procedure": "javax.persistence.EntityManager.createQuery",
    "kind": "SQLWrite"
  },
  {
    "procedure": "java.sql.Statement.executeQuery",
    "kind": "SQLWrite"
  },
  {
    "procedure": "java.sql.PreparedStatement.executeUpdate",
    "kind": "SQLWrite"
  },
  {
    "procedure": "java.sql.PreparedStatement.executeQuery",
    "kind": "SQLWrite"
  },
  {
    "procedure": "com.google.codeu.data.Datastore.storeMessage",
    "kind": "HTML"
  },
  {
    "procedure": "org.springframework.web.servlet.ModelAndView",
    "kind": "HTML"
  },
  {
    "procedure": "javax.servlet.http.HttpSession.setAttribute",
    "kind": "HTML"
  }
]
}

```

[Link to file version](#)²⁰.

²⁰ <https://help.sonatype.com/lift/files/78578725/78578726/1/1623180759408/inferconfig>

Pyre Configuration

```
{
  "binary": "/usr/local/bin/pyre.bin",
  "search_path": [
    "/opt/lift/pyre-venv/lib/python3.8/site-packages/"
  ],
  "taint_models_path": "/usr/local/lib",
  "typeshed": "/usr/local/lib/pyre-check/typeshed"
}
```



pyre_configuration

Link to file: https://help.sonatype.com/lift/files/78578727/78578728/1/1623180762879/pyre_configuration

Extending Lift

Custom Tool APIs are interface specifications about program arguments, standard input and standard output formats a tool must support in order to interoperate with Lift. The [configuration reference](#)(see page 42) presents how to configure your repository so that each analysis will invoke such a tool.

Execution Environment

Each tool runs in an Ubuntu 20.04 environment specifically for analysis of either the source or destination branches of a pull request. Shell scripts are used in examples below because they are a common platform supported, but there is no requirement to use shell scripts. More complex tools might benefit from a shell wrapper that verifies the environment before invoking the analysis tool.

For each issue discovered by a tool it should emit a tool note specifying the type of the issue, a detailed message, and location including file and line.

The Android SDK will be available on the file system accessible under \$ANDROID_HOME.

When Lift runs your tool the issues it discovers are categorized into those that are introduced (appearing only in the source branch), fixed (appearing only in the destination branch) and preexisting (appearing in both branches). Only introduced issues are propagated to the user by way of a pull request comment. All tool results are available to the user via the console.

APIv1

Bring Your Own (BYO) tools are called with arguments of a repository directory, commit hash, and a command. In addition, they must output a JSON value specific to the command. These tools are called once on the whole repository and are expected to analyze every relevant file in the repository.

API Commands

The command and their associated standard outputs are:

- `version`: Always print the number `1` on standard out and return with exit success (zero).
- `applicable`: Perform any needed analysis of the repository to determine if this tool is applicable (ex: a tool might only be applicable to particular languages). Print the string `true` or `false` to standard out dependently and return with exit success.
- `run`: Evaluate the repository and emit JSON results, detailed below.

The `run` command must emit JSON of:

```
[ { "type"      : <string>,
  "message"    : <string>,
  "file"       : <string>,
  "line"       : <int>,
  "details_url" : <optional string>,
  }
]
```

The tool is always passed arguments of repository directory, branch of interest, and command. An example of running the three commands is:

```
$ ./tool.sh /lift/code/path ce44de9a version
1
$ ./tool.sh /lift/code/path ce44de9a applicable
true
$ ./tool.sh /lift/code/path ce44de9a run
[{"type": "Invalid configuration keyword",
  "message": "The keyword 'customTols' is not a valid configuration value."},
```

```

    "file": "configs/config.toml",
    "line": 12,
    "details_url": null
  },
  { "type": "Invalid configuration keyword",
    "message": "The keyword 'riika' is not a valid configuration value.",
    "file": "configs/config.toml",
    "line": 11
  }
]

```

The tool must always exit successfully, returning exitcode 0.

BYO API Hello World

Let's make a hello-world example of a Lift custom tool at work. We'll use a shell script to case over the commands and take appropriate action. As a framework let's start with:

```

#!/usr/bin/env bash
# dir=$1 but it is not needed
commit=$2
cmd=$3

# ... to be filled in ...

if [[ "$cmd" = "run" ]] ; then
  run
fi
if [[ "$cmd" = "applicable" ]] ; then
  applicable
fi
if [[ "$cmd" = "version" ]] ; then
  version
fi

```

Now we'll fill in functions for each command. The `version` command is the simplest as there is no decision needing made. Just echo 1 out:

```

function version() {
  echo 1
}

```

For applicability let's always run - this tool isn't specific to configuration files, languages, times of day or any other variable.

```
function applicable() {
  echo "true"
}
```

Before tackling `run`, recall that Lift runs on both the source and destination branches before classifying the bugs as introduced/fixed/preexisting. Only introduced bugs are sent as comments. Therefore we need to do something branch specific and not just emit a tool note saying “hello world”. Our hello world tool will emit issues including the commit hash:

```
function run() {
  echo "[{ \"type\": \"Hello World\", \
    \"message\": \"We are analyzing commit $commit\", \
    \"file\": \"N/A\", \
    \"line\": 0, \
    \"details_url\": \"https://help.sonatype.com/lift/extending-lift\" \
  }]"
}
```

Suggested Development Process

We suggest you keep development local, and not push scripts to upstream repositories or test via Lift servers until after ensuring functionality locally. You should start with docker, access to the Ubuntu 20.04 image, and a shell. You can develop entirely locally and with a fast feedback loop by working directly inside of the docker container.

First we will look at the raw commands you can use to test your tool. After that we’ll write our own tool and verify it operates using the same method.

Development Process Steps

Create a test repository (or many) that is fast to analyze and sufficiently exercises your tool. Commit your tool, or a wrapper, in the repository such as at `.lift/tool.sh`. Also commit a `.lift/config.toml` with the entry `customTools = [".lift/tool.sh"]`.

Once setup, you can test the tool by executing the commands manually inside the container:

```
docker run --rm -it -v /path/to/the/repository:/code ubuntu:20.04 bash
cd /code
# execute any setup script
[[ -x ".lift/setup.sh" ]] && ./lift/setup.sh
./lift/tool.sh $(pwd) $(git rev-parse HEAD) version
./lift/tool.sh $(pwd) $(git rev-parse HEAD) applicable
./lift/tool.sh $(pwd) $(git rev-parse HEAD) run
```

Before concluding development you can verify the tool conforms with Lift's API with the check-lift-api tool:

```
docker run --rm -it -v /path/to/the/repository:/code ubuntu:20.04 bash
cd /code
curl -LO https://help.sonatype.com/lift/files/78578749/78578750/1/1623180836097/check-lift-api.sh
chmod +x check-lift-api.sh
./check-lift-api.sh
```

N.B. Make sure you test your tool on a repository that doesn't have any findings. Make sure the standard output is still a JSON list such as [].

Development Process by Example

As an example, let's develop a Bash based tool that complains about committed files which are not JSON. This is only for example; your tool can be written in any language and do any desired deep inspection of the code.

First we will create our test workspace:

```
mkdir lift-test-project ; cd lift-test-project
echo "{ }" > empty_dictionary.json
echo "[ ]" > empty_list.json
echo "" > empty_string.json
echo '{ "number": 42, "string" : "nuh-uh", "meaning of life" : null }' > complex.json
git init
git add *.json
git commit -m 'json files'
```

Then we'll create the tool:

```
mkdir .lift
cat <<EOF >.lift/config.toml
customTools = [ ".lift/tool.sh" ]
EOF
cat <<EOF >.lift/tool.sh
#!/usr/bin/env bash
function run() {
    for i in $(git ls-files | grep -v lift) ; do
        if ! jq \${i} >/dev/null 2>/dev/null ; then
            echo "[ { \"type\" : \"bad json\", \"file\" : \"\${i}\", \"line\": 1,\
                \"message\" : \"JSON is many things, but it is not this.\",\
                \"details_url\": null } ]"
            exit 0
        fi
    done
    echo "[ ]" ; exit 0
}

[[ \"\$3\" = \"version\" ]] && echo "1"
```

```
[[ "\$3" = "applicable" ]] && echo "true"
[[ "\$3" = "run" ]] && run
[[ -z "\$3" ]] && echo '{ "version" : 1, "name" : "json-verifier" }'
EOF
chmod +x .lift/tool.sh
git add .lift/*
git commit -m "lift tool"
```

Now we can test the tool manually and iterate, fixing the tool as appropriate:

```
./lift/tool.sh $(pwd) $(git rev-parse HEAD) version
./lift/tool.sh $(pwd) $(git rev-parse HEAD) applicable
./lift/tool.sh $(pwd) $(git rev-parse HEAD) run
```

And finally test the tool with the script:

```
docker run --rm -it -v $(pwd):/code ubuntu:20.04 bash
cd /code
apt update && apt install -y curl jq git
curl -LO https://help.sonatype.com/lift/files/78578749/78578750/1/1623180836097/check-lift-api.sh
chmod +x check-lift-api.sh
./check-lift-api.sh .lift/tool.sh $(pwd)
```

This tool is incomplete, such as outputting at most one message, but it does conform to the Lift API and passes the verification.

Talking to Lift

Liftbot doesn't just comment, it also listens for comments as commands. While its vocabulary is limited right now, it will grow over time.

Getting Liftbot's Attention

To issue Liftbot a command, simply @sonatype-lift followed by the command. This ensures Liftbot doesn't chime in when you aren't talking to it.

Ignore - Mark a Report as False Positives

Liftbot is preconfigured to minimize noise from false positives, but isn't perfect and wants to improve. Should Liftbot report an issue that you determine isn't a bug, you can tell Liftbot to ignore that report going forward. When you comment @sonatype-lift ignore, Liftbot will remove the bug report from results and from the status bar in Github.

Unignore - Undo an Ignore Reports

Should you or a colleague comment `@sonatype-lift ignore`, but change your mind and wish the report to remain, you can comment `@sonatype-lift unignore` and Liftbot will undo the ignore notation and treat the report as a legitimate bug report.

Help - Get Help from Liftbot

If you need a reminder of Liftbot's commands or how to get further information, or to add Lift to another project, comment `@sonatype-lift help` and Liftbot will respond with a list of commands and quick links to our documentation and our console where you can view results or add Lift to other projects.

Troubleshooting

- [Lift thought there was a bug, but I'm confident it is wrong \(impossible\).](#)(see page 0)
- [Lift couldn't understand my build system / complained about not being able to produce a compilation database](#)(see page 26)
- [I have a multi-project repository and I want Lift to only run on one particular project](#)(see page 27)
- [My custom build script has the error "E: Unable to locate package 'packagename'"](#)(see page 27)
- [My Android project is not building in Lift](#)(see page 27)
- [I don't have my own repository, but I still want to try Lift](#)(see page 27)
- [I am wondering what utilities are available to Lift analysis by default](#)(see page 29)
- [I have a problem that is not addressed here](#)(see page 30)

Lift thought there was a bug, but I'm confident it is wrong (impossible).

By design, Lift only surfaces the new bugs as pull request comments. False positive can be ignored by clicking "resolve" or responding with "ignore bug". If you're still saying "How do I understand how Lift came to its conclusions?" or want to improve the situation then consider:

1. Think through your project - is the type of bug reasonable? Not all bugs are reasonable to surface for all projects and it might be worth adding this bug to the ignore list in your [Lift configuration](#)(see page 42).
2. Lift orchestrates a collection of tools. Look at which tool detected the bug. In <https://lift.sonatype.com/> you can see your analysis results and which tool was responsible.
3. Tool in hand, consider the tool's configuration parameters. Most Lift tools are open source and support in-repository configurations.

4. [Contact us](#)(see page 48)! We're working hard to reduce false positives and make Lift results actionable out of the box without investing time on your part. We want to understand when any tool or rule causes too much pain. By monitoring repositories and github chatter we get an idea but your needs are ground truth.

Lift couldn't understand my build system / complained about not being able to produce a compilation database

Because Lift includes advanced program analysis tools, it has to be able to build the application. There are three common cases:

1. Building the code requires Java 11 (Lift defaults to JDK 1.8). In this case, add a `.lift.toml` file in the root directory of your repository containing

```
jdk11 = true
```

2. Analyzing the code requires a non-standard build target. When possible, it is best to use a build target that doesn't invoke tests, perform benchmarks, or pull in non-code resources such as images or binaries. This will make the analysis faster and more robust. To specify the build target, add a `.lift.toml` file containing the following:

```
build = "<build command>"
```

For example,

```
build = "./gradlew assembleAndroid"
```

3. Some dependencies are needed before build and analysis can proceed. Add a `.lift.toml` file that contains the following:

```
setup = ".lift/setup.sh"
```

Now add a `.lift/setup.sh` file that contains commands to download any needed dependencies. You can `apt install` packages or `git clone` public GitHub repositories. This script will be run from the root directory of the repository prior to the build and analyze steps.

I have a multi-project repository and I want Lift to only run on one particular project

Normally, Lift will do its best to autodetect your projects and analyze them. In the case that you had something else in mind, simply place a `.lift.toml` configuration file in the directory containing the project you want analyzed.

For more information, see [subprojects](#)(see page 45).

My custom build script has the error “E: Unable to locate package ‘packagename’”

First ensure your script runs `apt update` to get the latest package definitions before attempting adding another package.

Lift uses Ubuntu 20.04 “focal” as the base image for all analysis builds. Check if it is a valid “focal” package using the [ubuntu package search](#)(see page 42).

My Android project is not building in Lift

Android projects in Lift will require some special configuration. See [this guide](#)²¹ for instructions and an example.

I don’t have my own repository, but I still want to try Lift

In this case here is an example repository you can fork to see the power of Lift for yourself.

For this tutorial, we are going to analyze a C project that uses the ‘make’ build system. Fork our example repo at https://github.com/Lift-Dev/hello_lift²².

NOTE: if you didn’t give the Lift app permissions to all repositories then you’ll need to give it permission to this new repository. It’s easy, just go to <https://github.com/apps/sonatype-lift>, click `configure` and include this new repository by selecting it under ‘repository access’.

The `develop` branch has had multiple bugs introduced. Lift can analyze the changed code and call out any bugs that have appeared. You can start the analysis process by making a pull request. Navigate to the URL below and click `Create pull request`. Again, replace `$USER` with your GitHub username.

`https://github.com/$USER/hello_muse/compare/master...develop`

²¹ <https://help.sonatype.com/lift/configuring-lift-78578724.html#ConfiguringLift-java-android>

²² https://github.com/Muse-Dev/hello_muse

After a few minutes Lift will respond with its discoveries through pull request comments such as the one below.



lift-dev bot reviewed 23 minutes ago

View changes

```

main.c
...  ...  @@ -4,6 +4,11 @@ static char text[] = "Hello Muse!";
4      4
5      5      int main() {
6      6      char * printme = text;

```



lift-dev bot 23 minutes ago

...

DEAD_STORE: The value written to &printme (type char*) is never used.



lift-dev bot reviewed 23 minutes ago

View changes

```

main.c
...  ...  @@ -4,6 +4,11 @@ static char text[] = "Hello Muse!";
4      4
5      5      int main() {
6      6      char * printme = text;
7      +      printme = NULL;
8      +      printme[0] = 'X';

```





lift-dev bot 23 minutes ago

...

NULL_DEREFERENCE: pointer `printme` last assigned on line 7 could be null and is dereferenced at line 8, column 5.

Lift will also take advantage of GitHub's status API to let you know if there were any new bugs introduced by the pull request:







All checks have failed


Hide all checks

1 failing check

liftdev — Analysis completed (2 new bugs found)

Details



This branch has no conflicts with the base branch

Merging can be performed automatically.

Merge pull request

You can also [open this in GitHub Desktop](#) or view [command line instructions](#).

The full set of comments can be seen at the following URL:

```
https://github.com/$USER/hello_lift/pull/1
```

I am wondering what utilities are available to Lift analysis by default

Great! You are in the right place. The Lift analysis image is based on [Ubuntu 20.04](https://hub.docker.com/_/ubuntu?tab=tags&page=1&name=20.04)²³. Here is a list of packages installed by default:

```
apt-get install --yes --no-install-recommends \  
  autoconf \  
  automake \  
  awscli \  
  curl \  
  git \  
  software-properties-common \  
  gcc-7 \  
  gcc-9 \  
  g++-9 \  
  g++-multilib \  
  libc6-dev \  
  libgmp-dev \  
  libsqlite3-dev \  
  opam \  
  libtool \  
  libtool-bin \  
  autotools-dev \  
  openjdk-8-jdk-headless \  
  openjdk-11-jdk-headless \  
  pkg-config \  
  ruby \  
  libsqlite3-dev \  
  gawk \  
  tzdata \  
  maven \  
  libpcre3-dev \  
  netbase \  
  npm \  
  locales \  
  libcurl4-gnutls-dev \  
  libssl-dev \  
  ant \  
  jq \  
  vim \  
  zlib1g-dev \  
  python3 \  
  python3-dev \  
  python3-pip \  
  python-setuptools
```

²³ https://hub.docker.com/_/ubuntu?tab=tags&page=1&name=20.04

I have a problem that is not addressed here

Whether it's a cryptic build error or just confusing results, don't hesitate to [contact our support](#)(see page 48) for personalized assistance.

Included Analyzers

Lift leverages a broad range of analyzers that can be tailored to your code-base. Below is a list of the analyzers we've incorporated and made available within the Lift platform.

Want us to add your favorite open-source analysis tool to Lift? Visit our [community](#)²⁴ and let us know what other tools you would like to use via Lift.

If you need to add a new tool right away, [use our API](#)²⁵ to add your tool and Lift will automatically include it in every analysis run.

Infer

Languages: Java, C, C++

Website: <https://fbinfer.com>²⁶

Error Patterns: <https://fbinfer.com/docs/all-issue-types>

Infer was developed at Facebook and uses advanced compositional analysis techniques to provide deep insight into code behavior while keeping analysis times low. Infer checks Java for null pointer exceptions, resource leaks, performance issues, command injection and other information flow vulnerabilities, annotation consistency, and concurrency errors such as race conditions and deadlocks. Infer checks C/C++/ObjectiveC code for null pointer dereferences, memory leaks, coding convention violations, and API misuse errors.

ErrorProne

Languages: Java

Website: <https://errorprone.info>²⁷

Error Patterns: <https://errorprone.info/bugpatterns>

²⁴ <https://community.sonatype.com/c/sonatype-lift/50>

²⁵ <https://help.sonatype.com/lift/extending-lift>

²⁶ <https://fbinfer.com/>

²⁷ <https://errorprone.info/>

Developed by Google, this Java bug detection tool looks for language-specific error patterns and API mis-use errors. It is implemented as a compiler extension, and so has access to type information, class hierarchies, and dependency data. This gives it deeper insight into the code than most linters and allows it to detect more bugs while maintaining a low false positive rate. ErrorProne can also be extended with custom rules.

FindSecBugs

Languages: Java

Website: <https://find-sec-bugs.github.io>²⁸

Error Patterns: <https://find-sec-bugs.github.io/bugs.htm>

This tool provides static analysis for security audits of Java web applications and Android applications targeting many of the OWASP Top Ten.

Quandary

Languages: Java, C, C++

Website: <https://fbinfer.com/docs/checker-quandary>

Error Patterns: <https://fbinfer.com/docs/checker-quandary#list-of-issue-types>

Quandary is an extension of Infer that detects flows of values between sources and sinks that don't pass through a "sanitizer". It has a small list of built-in sources and sinks, but also provides a means for defining custom sources and sinks via an `.inferconfig` file included in the same directory as the build files (see an example [here](#)²⁹). Quandary can be used to detect cross-site scripting, shell/sql injection, untrusted data use, and logging of private data.

ESLint

Languages: JavaScript

Website: <https://eslint.org>³⁰

Error Patterns: <https://eslint.org/docs/rules>

ESLint is an open source JavaScript linting utility that helps find problematic patterns or code that doesn't adhere to certain style guidelines.

²⁸ <https://find-sec-bugs.github.io/>

²⁹ <https://github.com/facebook/infer/blob/master/infer/tests/codetoanalyze/java/quandary/.inferconfig>

³⁰ <https://eslint.org/>

Semgrep

Languages: Multiple languages (Python, JavaScript, Java, Golang, and C and more coming)

Website: <https://github.com/returntocorp/semgrep>

Lightweight static analysis for many languages. Find and block bug variants with rules that look like source code.

Staticcheck

Languages: Golang

Website: <https://staticcheck.io>³¹

Staticcheck is a state of the art linter for the Go programming language that finds bugs and performance issues, offers simplifications, and enforces style rules.

Golang-ci

Languages: Golang

Website: <https://github.com/golangci/golangci>

Our golangci-lint configuration aggregates and runs 9 Golang linters to catch a broad range of common Go errors.

Pyre

Languages: Python

Website: <https://pyre-check.org>³²

Error Patterns: <https://pyre-check.org/docs/error-types.html>

Pyre is a fast, scalable type checker for large Python 3 codebases, designed to help improve code quality and development speed by flagging type errors, following the typing standards introduced in PEP484 and PEP526. Pyre also includes the Pysa information flow analysis tool which can be configured to detect and warn on flows between functions labeled as sources and sinks.

³¹ <https://staticcheck.io/>

³² <https://pyre-check.org/>

Bandit

Languages: Python

Website: <https://pypi.org/project/bandit>

Bandit is a security linter from PyCQA designed to find common security issues in Python code. To do this Bandit processes each file, builds an AST from it, and runs appropriate plugins against the AST nodes.

Rubocop

Languages: Ruby

Website: <https://rubocop.org>³³

RuboCop is a Ruby code style checker (linter) and formatter based on the community-driven [Ruby Style Guide](#)³⁴.

Detekt

Languages: Kotlin

Website: <https://github.com/detekt/detekt>

Detekt a static code analysis tool for the Kotlin programming language. It operates on the abstract syntax tree provided by the Kotlin compiler.

ShellCheck

Languages: Bash, Shell

Website: <https://www.shellcheck.net>³⁵

Error Patterns: <https://github.com/koalaman/shellcheck/wiki/Checks>

ShellCheck is a code analyzer for your shell scripts.

Cobra

Languages: C/C++

Website: <http://spinroot.com/cobra/>

³³ <https://rubocop.org/>

³⁴ <https://rubystyle.guide/>

³⁵ <https://www.shellcheck.net/>

Cobra is a structural source code analyzer. Fast and easy to configure, Cobra is an ideal choice for enforcing API rules without needing compilation.

Clippy

Languages: Rust

Website: <https://github.com/rust-lang/rust-clippy>

Clippy is a Rust linting tool with hundreds of lints to surface a broad range of issues in Rust code.

HLint

Languages: Haskell

Website: <https://github.com/ndmitchell/hlint#readme>

HLint is a tool for analysing Haskell projects and outputting discovered short-comings in a helpful way with possible solutions for detected problems. HLint is searching for not only performance or error-prone code pieces, but it also can help with establishing and applying best-practices from the whole Haskell ecosystem.

Psalm

Languages: PHP

Website: <https://psalm.dev/>

Error Patterns: https://psalm.dev/docs/running_psalms/issues/

Psalm is a static analysis tool for finding errors in PHP applications, including not just stylistic issues, but can also perform taint analysis to catch command injection vulnerabilities. Psalm also suggests fixes to many common issues it surfaces.

LuaCheck

Languages: Lua

Website: <https://github.com/mpeterv/luacheck>

Error Patterns: <https://luacheck.readthedocs.io/en/stable/warnings.html>

Luacheck is a static analyzer and a linter that detects various issues such as usage of undefined global variables, unused variables and values, accessing uninitialized variables and unreachable code.

Markdownlint - MDL (Disabled by Default)

Languages: Markdown

Website: <https://github.com/markdownlint/markdownlint>

MDL is a tool to check markdown files and flag style issues.

PMD (Disabled by Default)

Languages: Java, JavaScript, and others

Website: <https://pmd.github.io>³⁶

Error Patterns: https://pmd.github.io/pmd-6.23.0/pmd_rules_java.html

The tool finds common programming flaws like unused variables, empty catch blocks, unnecessary object creation, and so forth.

PMD is not enabled by default. To add it, copy this line into your Lift configuration file:

```
customTools = [ "https://help.sonatype.com/lift/files/78578763/78578764/1/1623180860953/pmd.sh rulesets/java/quickstart.xml" ]
```

[Click here](#)(see page 42) to learn how to create a Lift configuration file.

Checkov (Disabled by Default)

Languages: Infrastructure Code (Terraform, Cloudformation, K8S)

Website: <https://checkov.io>³⁷

Error Patterns: <https://github.com/bridgecrewio/checkov/blob/master/docs/5.Policy%20Index/all.md>

Checkov is a static code analysis tool for infrastructure-as-code. It scans cloud infrastructure managed in Terraform, Cloudformation, Kubernetes, Arm templates or Serverless Framework and detects misconfigurations.

Open Source Vulnerability Analyzer

The Lift Open Source Vulnerability Analyzer provides insights on vulnerabilities found in dependencies used throughout your application. Vulnerable dependencies are highlighted within each pull request, helping you focus on changes at the time of implementation.

³⁶ <https://pmd.github.io/>

³⁷ <https://checkov.io/>

The Open Source Vulnerability Analyzer is not available on-prem (see [Nexus Lifecycle](#)³⁸ for on-prem SCA analysis).

Available ecosystems

Language	Build System	Scan file required	Manifest scanning	Transitive dependencies ⁽¹⁾	Transitive Dependency Highlighting ⁽²⁾	Notes
Java	Maven	pom.xml	✓	✓	✓	
	Gradle	build.gradle, build.gradle.kts	✓	✓	✓	Modules of a multi-module project are scanned individually; therefore vulnerabilities are only highlighted at the module level
JavaScript	npm	package-lock.json	✓	✓	✓	
	Yarn	yarn.lock	✓	✓	✓	
Go	Go modules	go.mod	✓	✓		All transitive dependencies are scanned but currently highlighting the responsible direct dependency is not enabled
Python	Python	requirements.txt	✓			All dependencies listed will be scanned (this may include development and platform dependencies)
Rust	Cargo	Cargo.lock	✓	✓	✓	

(1) - The ecosystem has the capability to scan all direct and transitive dependencies for the project.

(2) - The ecosystem has the capability to establish a graph of dependencies which allows highlighting the direct dependency that brought in a vulnerable transitive dependency. Otherwise, the vulnerability will be highlighted at the top of the relevant file.

³⁸ <https://help.sonatype.com/display/NXIQ>

Unless stated otherwise, development dependencies (such as dependencies listed under 'devDependencies' for npm) are *not* included as part of the scan.

Golang Analysis

What is supported

Go modules by scanning a runtime generated list of dependencies.

To scan as a Go package a **go.mod** file must be present.

Javascript Analysis

What is supported

NPM/Yarn packages by scanning the following files (file name must be preserved):

- **yarn.lock**: auto-generated file for projects using Yarn as package manager. To learn how, please refer to [documentation](#)
- **package-lock.json**: auto-generated file for projects using npm as package manager. To learn how, please refer to [documentation](#)

 Were possible vulnerabilities will be highlighted in **both** the package.json and associated lock file

What do we parse from files?

In yarn.lock

Name and version fields will be evaluated and additionally any declared dependencies. For example:

- name: @dangl/angular-material-shared
- version: 2.0.0

```
@dangl/angular-material-shared@2.0.0:
  version "2.0.0"

@progress/kendo-theme-material@0.3.2:
  version "0.3.2"
```

```
@angular@0.0.0.1:
  version "0.0.1"
```

In package-lock.json

Name and version fields from **dependencies** objects will be evaluated. For example:

ansi-regex	3.0.0
wordwrap	0.0.3

```
{
  "requires": true,
  "lockfileVersion": 1,
  "dependencies": {
    "ansi-regex": {
      "version": "3.0.0",
      "resolved": "https://registry.npmjs.org/ansi-regex/-/ansi-regex-3.0.0.tgz",
      "integrity": "sha1-7QMXwyIGT3lGbAKWa922Bas32Zg="
    },
    "wordwrap": {
      "version": "0.0.3",
      "resolved": "https://registry.npmjs.org/wordwrap/-/wordwrap-0.0.3.tgz",
      "integrity": "sha1-o9XabNXAvAAI03I0u68b7WMFkQc="
    }
  }
}
```

JVM Analysis

What is supported

Maven and Gradle projects by scanning their respective pom.xml and build.gradle/ build.gradle.kts manifest files.

What do we parse from a file?

pom.xml

Dependencies with group, artifact, and exact version are required and evaluated, extension and classifier are optional. For example:

```
<dependency>
  <groupId>org.example</groupId>
  <artifactId>ACME-business</artifactId>
  <version>1.0-SNAPSHOT</version>
</dependency>
```

i During analysis all dependencies and their subsequent transitive dependencies (excluding test dependencies) are analyzed.

build.gradle (groovy) / build.gradle.kts (kotlin)

Dependencies with group, artifact, and exact version are required and evaluated, extension and classifier are optional. For example:

```
dependencies {
  compile 'org.example:ACME-business:1.0-SNAPSHOT'
}
```

i The following dependency scopes are supported: "runtimeClasspath", "compileClasspath"

i All internal modules are included in a scan, however, if *Module A* depends on *Module B* and *Module B* has vulnerabilities, then the vulnerabilities will only be reported at the *Module B* level despite *Module A* being dependant

Python Analysis

What is supported

Files named requirements.txt (generated using a pip command) will be analyzed.

What do we parse from the file?

Requirements using the "==" operator and version without wildcards will be considered. The scanner does not distinguish against the use of "sys_platform" markers and will therefore be considered regardless. For example:

```
altgraph==0.10.2
pywin32 ==1.0 ; sys_platform == 'win32'
```

Creating requirements

Run pip freeze

`pip freeze > requirements.txt`

i The requirements.txt encoding is UTF-8. Special note for Microsoft Windows users, the cmd.exe encoding may need to be changed to UTF-8. Please refer to Microsoft documentation on how to do this.

Ensure the requirements.txt is committed to your repository to be scanned.

Environment Markers

Any [environment markers](#)³⁹ are currently ignored as part of the scan. For example:

```
Django==1.6; sys_platform == 'win32'
```

Rust Analysis

What is supported

Files named Cargo.lock will be analyzed.

i When possible, vulnerabilities will be highlighted in **both** the Cargo.lock and Cargo.toml files

What do we parse from the file?

The fields name and version of the dependency under each "package" section are evaluated. For example:

³⁹ <https://www.python.org/dev/peps/pep-0508/#environment-markers>


```
[[package]]
name = "core-nightly"
version = "1.26.2"
```

Configuring Analyzers

Most analyzers are distributed with default configurations that allow them to run out of the box. For tools that don't have suitable defaults, Lift provides its own default configurations that are consistent with its philosophy of producing high confidence / high impact findings. Users can override these defaults by providing their own in-repository configuration files specific to the analysis tool being utilized.

See below for more details on the selection of these default configurations.

Infer Quandary

Infer [Quandary](#)⁴⁰ is an information flow [taint analysis](#)⁴¹ tool that detects unwanted flows of data.

Infer looks for a `.inferconfig` next to the `.lift.toml` file or `.lift` directory. If no file is found we insert our [default configuration](#)⁴².

Our default configuration can be overridden by committing a `.inferconfig` file to the root of your repo or [at a subproject](#)⁴³.

Pyre/Pysa

[Pysa](#)⁴⁴ is the information flow analysis portion of the Pyre Python static analysis project.

It uses a `.pyre_configuration` file to define search paths for `taint.config` and `.pysa` files. These define stub models and the source/sink definitions.

Our default `.pyre_configuration`⁴⁵ defines the search path to the [taint.config](#)⁴⁶, [stubs](#)⁴⁷, and [typeshed](#)⁴⁸ as included in the [pyre-check@0.0.59](#)⁴⁹ project.

To modify the `.pyre_configuration` you can download it next to the `.lift.toml` file or `.lift` directory, and update it according to the Pysa documentation.

⁴⁰ <https://fbinfer.com/docs/checker-quandary>

⁴¹ https://en.wikipedia.org/wiki/Taint_checking

⁴² <https://help.sonatype.com/lift/files/78578725/78578726/1/1623180759408/inferconfig>

⁴³ <https://help.sonatype.com/lift/configuration-reference-78578742.html#ConfigurationReference-subprojects>

⁴⁴ <https://pyre-check.org/docs/pysa-running/>

⁴⁵ https://help.sonatype.com/lift/files/78578727/78578728/1/1623180762879/pyre_configuration

⁴⁶ https://github.com/facebook/pyre-check/blob/v0.0.59/stubs/taint/core_privacy_security/taint.config

⁴⁷ https://github.com/facebook/pyre-check/tree/v0.0.59/stubs/taint/core_privacy_security

⁴⁸ <https://github.com/facebook/pyre-check/tree/v0.0.59/stubs/typeshed>

⁴⁹ <https://pypi.org/project/pyre-check/0.0.59/>

Note that the search path references a virtual environment, this is activated and requirements.txt is pip installed before your analysis is run in order to exclude pyre-check files from the analysis. This saves considerable time.

Configuration Reference

Most deep analysis tools build the project as part of operating. Lift can detect the build system and a build target to identify the code and decide which analyzer should be applied. Alternatively, direct configuration of Lift is possible by writing and committing a `.lift.toml` file to your repository. Explicit configuration is not usually required unless custom compilation steps must be taken, such as using non-standard build targets or installing external library dependencies. This page describes the supported build systems and configuration options available to influence build, analysis, and general Lift operation on a per-repository basis.

In-Repo Options (The `.lift.toml`)

By default Lift will attempt to run all its analysts on a repository, generating reports from all analyses that succeed. The configuration file can be used to enable or disabled analysis passes, specify the build system, and filter types of bugs reported. The file `.lift.toml` controls Lift's behavior using the fields:

```

setup                = <path to setup script>
build                = ENV=<env> <build option> [target]
importantRules       = <exclusive list of issues to report back to user>
ignoreRules          = <list of issues never to report to the user>
ignoreFiles          = <gitignore format string of files to ignore>
tools                = <infer | errorprone | eslint | hlint | findsecbugs >
disableTools         = <list of tools to not run on the project>
customTools          = <list of user-provided tools conforming to a tool API>
allow                = <list of users whose pull requests can trigger analysis>
jdk11                = <true or false (defaults to false)>
androidVersion       = <the android SDK version number>
errorproneBugPatterns = <list of bug patterns>
summaryComments      = <true or false (defaults to false)>

```

- **setup**: A script that can perform pre-build configuration, such as installing tools (ex: `apt install dependency`) or libraries. The path should be either absolute or relative to the *root* of the repository.
- **build**: The name of a build tool (overrides auto detection). Valid build tools include "make", "gradlew", "maven", "autogen", "configure", "cmake", "compdb" and "gradle".
 - Autogen implies `./autogen.sh ; ./configure ; make` a la typical autotools workflow.
 - "configure" similarly starts one step later.
 - "gradlew" without an explicit target will try the `assemble` target.
 - "compdb" requires a `compile_commands.json` file is present in the repository and its paths are all relative to the project's root.

- You may set environment variables from the build line: `build = "CC=gcc GXX=g++ CXX=g++ make"`
- `importantRules`: An allow list of issues considered important enough to report. If defined, only bugs of types that are in this list will ever be reported to the user. The values here match the bold faced header, aka issue “type”, seen on the reports.
- `ignoreRules`: The opposite of important, a deny list of issues that should never be reported. Any issue in the ignore list is never reported to the user, even if it is marked important. For example, you can ignore ESLint’s no-empty message by specifying `ignore = ["no-empty"]`.
- `ignoreFiles`: A gitignore formatted specification of files from which results should be ignored. You may wish to use Toml’s [multiline string](#)⁵⁰ to specify multiple files or directories to be ignored. As an alternative, you may place a gitignore formatted file in the path `.lift/ignoreFiles` relative to the root of your repository.
- `tools`: List of analysis tools to apply to the repository. By default each tool supported by Lift will be applied.
- `disableTools`: A deny list of analysis tools to not be applied to the repository. For example, specifying `disableTools = ["errorprone"]` will run all the default tools except for errorprone.
- `customTools`: List of custom tools to run. This can run any manner of tool. For a basic example that can serve as a starting point for your own tool see the [Custom Tools Section](#)(see page 45).
- `allow`: List of user names whose pull requests will cause analysis and result in comments being posted. The default behavior is to run Lift on all pull requests regardless of author.
- `jdk11`: A boolean field indicating use of OpenJDK11 instead of OpenJDK8.
- `androidVersion`: The android SDK version number. This option is required for all android projects. Valid values are 27 and 28.
- `errorproneBugPatterns`: Use the provided, and only the provided, [errorprone bug patterns](#)⁵¹. For example: `errorproneBugPatterns = ["ArrayEquals", "ArrayToString"]`.
- `summaryComments`: A boolean field indicating whether or not to post a comment with a summary. When true, Lift will post a comment that looks similar to "Complete (6 min, 6/7 checks) no new bugs found".

An actual configuration might look like:

```

setup      = ".lift/script_that_downloads_deps.sh"
build      = "gradlew assemble"

# We only care about NULL_DEREFERENCE (from Infer)
# and no-extra-boolean-cast (from ESLint)
importantRules = ["NULL_DEREFERENCE", "no-extra-boolean-cast"]

# Ignore results from test and build directories
ignoreFiles = ""

```

⁵⁰ <https://toml.io/en/>

⁵¹ <https://errorprone.info/bugpatterns>

```
build/
src/test/
"""

# Only run infer and eslint (do not run errorprone, hlint, findseccbugs)
tools = [ "infer", "eslint" ]

# Only analyze and post responses to PRs from developers with these usernames
allow = [ "jill", "dave", "shawn" ]

jdk11 = false
```

The file and all fields are optional. By default, Lift will attempt to infer appropriate values for any setting that isn't specified in the file.

Build System Support

Lift analyses look for build system files on the repository root and through all the subdirectories. A build system is necessary for Lift to learn which files are appropriate to analyze, which dependencies (libraries and class paths) to leverage, and which compilation flags are appropriate. The supported build systems include:

- gradlew:
 - Description: Lift will use the gradle wrapper
 - Condition: When a `gradlew` file is found
 - Command line equivalent: `./gradlew assemble`
- gradle:
 - Description: Lift will use the Gradle build system
 - Condition: When a `build.gradle` file is found.
 - Command line equivalent: `gradle assemble`
- CompDB:
 - Description: Lift will read compile commands from a JSON file
 - Condition: When a `compile_commands.json` file exists
 - Command line equivalent: Sequentially invoking the commands with arguments and files specified in the JSON file
- Maven:
 - Description: Lift will use Maven
 - Condition: When a `pom.xml` file is found
 - Command line equivalent: `mvn compile -B`
- Make:
 - Description: Lift will use make
 - Condition: When a `{M,m}akefile` file is found
- Autotools:

- Description: Lift will invoke the common GNU AutoTools of `autogen.sh` and `configure` to try and produce a Makefile
- Condition: Used when `autogen.sh` or `configure` files are found
- Command line equivalent: `./autogen.sh || ./configure && make`
- Compilation database:
 - Description: Lift will use the individual compilation commands and arguments specified by a compilation database
 - Condition: Used when `compile_commands.json` is found

Subprojects

Lift supports nested subprojects and projects not at the root level. This lets you easily build Lift into your monolithic repos and lets you specify entirely different build systems for different parts of your code. To set this up, each subproject needs its own Lift configuration file in the root directory of that subproject. This configuration file should use the same format as before, and in it you can configure specific tools and build systems to use for your subproject.

The top-level `.lift.toml` file can still optionally exist and can be used to build a top-level project. Additionally, this top-level configuration file is the only place you should put an `allow` list. Because the allowlist is global the `allow` field is only read from the top-level configuration file.

The directory structure might look something like this:

```
.lift.toml

project_a/
project_a/.lift.toml
project_a/Makefile
project_a/main.c

project_b/
project_b/.lift.toml
project_b/Makefile
project_b/main.c
```

Lift will then automatically run on both Project A and Project B.

APIs for Custom Tools

Lift supports arbitrary tools for users to extend and customize the types of comments made on pull requests. The customizations can either be a file path or HTTPS URL to a script followed by arguments. Here are some example configurations and links to the supported APIs so you can add your own checks.

Example Extended Checks - Fixing the FIXME

Let's say you have a repository that you wish to keep clean of cruffy code comments such as TODO, XXX, and FIXME. Add to the `.lift.toml` file:

```
customTools = ["https://docs.lift.dev/scripts/check-common-comments.sh"]
```

There is no magic in the URL - you can download that script, change it, re-host it on your own domain, and have Lift run checks of your own design. The `check-common-comments` script will `grep` through your source code and report any instance of the offending strings.

In many cases the script might be project-specific and can be hosted in the repository. In this case you might place the file `.lift/my_script.sh` in your repository and have a tools line of:

```
customTools = [ ".lift/my_script.sh" ]
```

Example Extended Checks - ShellCheck and PMD

Let's look at two more examples. The first is a Java analysis tool [PMD](https://pmd.github.io/)⁵² and the second is a shell script checker [ShellCheck](https://shellcheck.net/)⁵³.

The PMD script is parameterized by the rule set so we'll pass in an argument. Our ShellCheck support script hard codes some simple assumptions and does not consider arguments. We further disable the built-in tools, yielding a `.lift.toml` file of:

```
tools = [ ]
customTools = [ "https://docs.lift.dev/scripts/pmd.sh rulesets/java/quickstart.xml"
, "https://help.sonatype.com/lift/files/78578765/78578766/1/1623180865504/shellcheck.sh"
]
```

The hosted example scripts use binaries of `pmd` and `shellcheck` that are already on the image - the scripts just run the tool then translate the resulting JSON. If your preferred tool isn't installed consider specifying a installation script in the `setup` configuration field to adjust the environment first.

⁵² <https://pmd.github.io/>

⁵³ <https://shellcheck.net/>

Extended Tooling with Custom or Community-provided Tools

The above configuration shows how to invoke an existing tool. These tools all must conform to one of Lift's APIs. Please refer to our [tool API](#)([see page 19](#)) for information necessary to build a tool.

Security FAQs

How is Lift deployed?

Lift software is delivered as service. It integrates with your source repository to automatically run at each pull request.

How does Lift work?

Upon each pull request (which Lift monitors via the repo host), Lift clones your repo and runs its analyzers over the code, delivering results as code comments within the repo's code review tool. Upon completion of the analysis of private repositories, Lift will delete its copy of your repository.

What's the high-level architecture?

Lift is a container-based platform on Linux running on Amazon Web Services (AWS). The cloud platform integrates directly with repository hosts like Github and requires no installation of code into your environment.

How does Lift handle our source code and other confidential information?

Lift recognizes the value of its customers' source code and the importance of maintaining confidentiality. Lift retains its customers' data only to the extent required to deliver its service and for only as long as required to do so. Lift treats its customers' source code and related information as highly confidential, and cares for it with the same degree of care we use to preserve our own confidentiality. Lift encrypts its data at rest using industry standard encryption and for data in transit, Lift relies upon TLS and shared secrets with Github/Gitlab/Bitbucket to encrypt source code and other data transmitted to/from Lift. Lift further separates its customers' data by providing a dedicated single-tenant AWS node for the duration of each analysis. For Lift on-premise deployments, neither your source code nor our analysis results leave the Lift server.

Does Lift process "Personal Data" as defined by GDPR and similar privacy laws?

Lift captures Personal Data solely of its own end-users, i.e. those individual developers with Lift accounts. Specifically, we capture name, email address and other information received from Github through our SSO integration. We retain and use such data only as long as necessary and in compliance with law.

How do you handle authentication and otherwise manage user accounts?

Lift relies on integrates with 3rd party single-sign on (SSO) providers like Github so customers can use their existing accounts on those platforms to log into Lift. User accounts in the Lift platform are associated with the credentials of the SSO providers.

Why does Lift ask for these permissions on GitHub?

Lift performs a variety of operations depending on what tools it uses to analyze your code. The permissions requested are:

- GitHub's text: "User permissions" and "Installing and authorizing sonatype-lift immediately grants these permissions on your account: **<name>**. - **Read** access to emails"
 - Reason: to know who you are when you log into the console at <https://lift.sonatype.com> and to provide email communications regarding the Lift service.
- GitHub's text: "**Read** access to members, metadata, organization administration, organization plan, and organization projects"
 - Members: to help grant access to users who visit the console
 - Metadata: For learning default branches and other basic information
 - Organization administration: For allowing administrators to control seat provisioning
 - Organization plan: Unused
 - Organization projects: Unused
- GitHub's text: "**Read** and **write** access to checks, code, commit statuses, issues, pull requests, and security events"
 - Checks: This is a status API allowing CI jobs to report their current status.
 - Code: Read access is needed to scan your code. Write access is used by some tools to (optionally) create new branches and open associated pull requests in order to automatically suggest fixes for code quality and security issues.
 - Commit statuses: This is an API that allows apps like Lift to report the status of an analysis job associated with a pull request.
 - Issues: Allows the app to (optionally) open new GitHub issues as a means of recording and tracking code quality and security issues.
 - Pull requests: Allows the app to (optionally) open new pull requests to automatically suggest fixes for code quality and security issues.
 - Security events: Lift integrates with GitHub's code scanning dashboard and can upload issues directly to the GitHub code scanning system.

Need More Help

Lift Pro Users (including free trials)

You can contact our support team right away by opening a case on our [Support Portal](https://support.sonatype.com)⁵⁴ and we'll respond in less than one business day - often in less than a few hours. You can also visit our community to get your questions answered.

⁵⁴ <https://support.sonatype.com>

Lift Free Users

If you are unable to find the answer to your question in the documentation, [visit our community](#)⁵⁵ to get help.

Deleting an Account

Your Lift account can be canceled via the [GitHub marketplace subscription page](#)⁵⁶. On this page you can end a subscription, change the subscription to alternative plan, or cancel with one account and install on another.

If you'd like to additionally prohibit Lift access to your GitHub account simply go to [our App Settings](#)⁵⁷ where you can uninstall the application.

We're sorry to see you go.

Scripts

- [Check Common Components](#)(see page 49)
- [Check Lift API](#)(see page 51)
- [ESLint](#)(see page 53)
- [goparamcount](#)(see page 54)
- [Hello Lift](#)(see page 56)
- [Line Length](#)(see page 57)
- [mdl](#)(see page 59)
- [pmd](#)(see page 60)
- [Shell Check](#)(see page 61)
- [Static Check](#)(see page 62)

Check Common Comments

```
#!/bin/bash

# This script conforms to the Lift Script API v1, providing
# a 'version', 'applicable <path>', and 'run <path>' operation.

declare -a patterns
patterns=(XXX TODO FIXME)

directory=$1
commit=$2
```

⁵⁵ <https://community.sonatype.com/c/sonatype-lift>

⁵⁶ <https://github.com/marketplace/sonatype-lift>

⁵⁷ <https://github.com/apps/sonatype-lift>

```

command=$3

SEP=""

check_for_bad_pattern() {
    local pattern=$1
    lines=$(grep -I -H -i -n "$pattern" * -R 2>/dev/null | \
        grep -v 'check-common-comments')
    while read -r i ; do
        file=$(echo $i | cut -d ':' -f 1)
        line=$(echo $i | cut -d ':' -f 2)
        if [[ ! -z "$file" && ! -z "$line" ]] ; then
            echo ${SEP}
            SEP=", "
            echo "{"
            echo "\"type\" : \"$pattern\","
            echo "\"message\" : \"Marker found at line $line\","
            echo "\"file\" : \"$file\","
            echo "\"line\" : $line"
            echo "}"
        fi
    done <<< "$lines"
}

if [[ "$command" = "version" ]] ; then
    echo "1"
    exit 0
elif [[ "$command" = "applicable" ]] ; then
    echo "true"
    exit 0
elif [[ "$command" = "run" ]] ; then
    pushd $directory 1>/dev/null 2>&1
    echo "["
    for i in ${patterns[@]} ; do
        check_for_bad_pattern "$i"
    done
    echo "]"
    popd 1>/dev/null 2>&1
else
    exit 1
fi

```



check-common-comments.sh

Link to file: <https://help.sonatype.com/download/attachments/78578003/check-common-comments.sh>

Check Lift API

```
#!/usr/bin/env bash

function checkVersion() {
    local result
    result=$( $script "$test_repo" "$test_commit" version | jq . )
    if [[ "$result" = "1" ]] ; then
        echo "$script version    GOOD"
    else
        echo "$script version    BAD (returned $result)"
        echo "    should emit '1' to indicate API version 1."
    fi
}

function checkApplicable() {
    local aresult
    aresult=$( $script "$test_repo" "$test_commit" applicable | jq . )
    if [[ ( "$aresult" = "true" ) || ( "$aresult" = "false" ) ]] ; then
        echo "$script applicable GOOD"
    else
        echo "$script applicable BAD (returned '$aresult')"
        echo "    should emit a boolean (true or false)"
    fi
}

function checkRun() {
    if [[ -d "$test_repo" ]] ; then
        local result
        result=$( $script "$test_repo" "$test_commit" run 2>/dev/null )
        toptype=$( echo "$result" | jq 'type' )
        if [[ ! ( "$toptype" = "array" ) && ! ( "$toptype" = "\"array\"" ) ]] ; then
```

```

    echo "$script run    BAD - results should be an array (found $toptype)"
    exit 1
fi
correctLength=$(echo "$result" | jq 'length')
withFile=$(echo "$result" | jq 'map(.file) | map(strings) | length')
withType=$(echo "$result" | jq 'map(.type) | map(strings) | length')
withMessage=$(echo "$result" | jq 'map(.message) | map(strings) | length')
withLine=$(echo "$result" | jq 'map(.line) | map(numbers) | length')
if [[ ! ( "$correctLength" = "$withFile" ) ]] ; then
    echo "$script run    BAD - bad or missing 'file' field(s)."
    exit 1
fi
if [[ ! ( "$correctLength" = "$withType" ) ]] ; then
    echo "$script run    BAD - bad or missing 'type' field(s)."
    exit 1
fi
if [[ ! ( "$correctLength" = "$withMessage" ) ]] ; then
    echo "$script run    BAD- bad or missing 'message' field(s)."
    exit 1
fi
if [[ ! ( "$correctLength" = "$withLine" ) ]] ; then
    echo "$script run    BAD - bad or missing 'line' field(s)."
    exit 1
fi
echo "$script run      GOOD"
else
    echo "No repository specified on the command line."
    echo "Not checking the 'run' functionality due to missing test repository."
fi
}

function checkDeps() {
    if [[ ! ( -f "$(command -v jq)" ) ]] ; then
        echo "Missing jq"
        exit 1
    fi
    if [[ ! ( -f "$(command -v git)" ) ]] ; then
        echo "Missing git"
        exit 1
    fi
}

function main() {
    checkDeps
    script="$(pwd)/$1"
    test_repo=$2
    test_commit=""
    echo "Checking script '$script'"

    if [[ ! ( -z "$test_repo" ) ]] ; then
        pushd "$test_repo" || exit 1
        test_commit=$(git rev-parse HEAD 2>/dev/null)
    else
        test_repo="some_repo_directory"
        test_commit="some_commit"
    fi
}

```

```
fi
checkVersion
checkApplicable
checkRun
}

main "$@"
```



check-lift-api.sh

Link to file: <https://help.sonatype.com/lift/files/78578749/78578750/1/1623180836097/check-lift-api.sh>

ESLint

```
#!/bin/bash

# Use: In the .lift/config specify:
# ```
# customTools = "https://help.sonatype.com/download/attachments/78578030/eslint.sh"
# ```
dir=$1
# commit=$2
cmd=$3
shift
shift
shift
args=$*

if [[ "$cmd" = "run" ]] ; then
    ESLINT_OUT_TMP=$(mktemp)
    # Break down of this sinful awk sed pipeline
    # sed -r 's/\\+([nrt])/\\1/g' Any escaped newlines should use one backslash.
    # sed -r 's/\\+([^nrt])/<invalid character>/g' Any other escaped sequence shouldn't exist
    # tr Actual newlines (vs escaped) shouldn't exist
    # awk '{ gsub("\\n", "\\n") ; print $0 }' RS= Escaped newlines actually do need two backslashes
    # LC_ALL=C sed -r -e 's/[^ -~]/g' Finally remove all control and non asc characters
```

```

eslint --no-eslintrc $args --format=json "." | \
    sed -r 's/\\+([nrt])/\\1/g' | \
    sed -r 's/\\+([^\nrt])/<invalid character>/g' | \
    tr '\n' '\\n' | \
    awk '{ gsub("\\n", "\\n") ; print $0 }' RS= | \
    LC_ALL=C sed -r -e 's/[^ -~]//g' \
    > "$ESLINT_OUT_TMP"

# For each record in the array
# rename key 'filePath' to 'file'
# For each .messages record
# Produce a new object with the top level .filePath, this .message, and a hard-coded 'type' field
# And rewrite the path to make it relative to the $dir
jq "[ .[] | .filePath as $file | .messages[] | { file:$file, line, message, type: \"ESLint\"} | .file
|= sub(\"$dir/\"; \"\") ]" 2>/dev/null < "$ESLINT_OUT_TMP"
fi

if [[ "$cmd" = "applicable" ]] ; then
    echo "true"
fi

if [[ "$cmd" = "version" ]] ; then
    echo 1
fi

```



eslint.sh

Link to file: <https://help.sonatype.com/download/attachments/78578030/eslint.sh>

goparamcount

```

#!/usr/bin/env bash
function tellApplicable() {
    files=$(git ls-files | egrep '.go$' | head)
    res="broken"
    if [[ -z "$files" ]] ; then

```

```

        res="false"
    else
        res="true"
    fi
    printf "%s\n" "$res"
}

function tellVersion() {
    echo "1"
}

function getTool() {
    apt update >/dev/null && apt install -y golang >/dev/null
    go get github.com/ericcornelissen/goparamcount
}

function emit_result() {
    echo "{ \"file\": \"$1\", \"
        \"line\": $2, \"
        \"type\": \"Too many parameters\", \"
        \"message\": \"There are too many parameters in procedure $3\" \"
    }"
}

function emit_results() {
    local FIRST=1
    echo "["
    echo "$1" | while read line ; do
        file=$(echo $line | cut -d ':' -f 2)
        lineNumber=$(echo $line | cut -d ':' -f 3 | cut -d ' ' -f 1)
        procedure=$(echo $line | cut -d ':' -f 3 | cut -d '"' -f 2)
        emit_result "$file" "$lineNumber" "$procedure"
        [[ $FIRST = 1 ]] || echo ", "
        FIRST=0
    done
    echo "]"
}

function run() {
    raw_results=$(HOME/go/bin/goparamcount ./...)
    code=$?
    getTool
    if [[ $code = 0 ]] ; then
        emit_results "$raw_results"
    else
        echo "Oh no, non-zero exit code!" >&2
        exit $code
    fi
}

case "$3" in
    run)
        run
    ;;

```

```

applicable)
    tellApplicable
    ;;
name)
    echo "goparamcount"
    ;;
*)
    tellVersion
    ;;
esac

```



goparamcount.sh

Link to file: <https://help.sonatype.com/download/attachments/78578043/goparamcount.sh>

Hello Lift

```

#!/usr/bin/env bash
function tellApplicable() {
    printf "true"
}

function tellVersion() {
    printf "1"
}

function run() {
    local SEP=""
    echo "["
    for file in $(git ls-files) ; do
        if [[ ( -f "$file" ) && ( $(wc -l < "$file" ) -gt 1337 ) ]] ; then
            printf "%s" "$SEP"
            author=$(git blame "$file" --porcelain 2>/dev/null | grep "^author " | head -n1)
            msg="$author did a bad job and they should feel bad"
            printf "{ \"message\": \"%s\", \"file\": \"%s\", \"

```



```

        \"line\": 123, \
        \"type\": \"Over-length file\" \
    }\n" "$msg" "$file"
    SEP=", "
fi
done
echo "]"
}

case "$3" in
    run)
        run
        ;;
    applicable)
        tellApplicable
        ;;
    version)
        tellVersion
        ;;
    default)
        echo "What? Check my version, I'm 1 (bulk)"
esac
esac

```



hello-lift.sh

Link to file: <https://help.sonatype.com/lift/files/78578756/78578757/1/1623180848696/hello-lift.sh>

Line Length

```

#!/bin/bash

# Use: In the .lift/config specify:
# ```

```

```
# customTools = "https://help.sonatype.com/lift/files/78578759/78578760/1/1623180852664/line-length.sh
LENGTH SUBDIR"
# ```
dir=$1
commit=$2
cmd=$3
length=${4:-80}
subdir=$5

if [ -z ${subdir} ] ; then
    subdir=$dir
else
    subdir="$dir/$subdir"
fi

if [[ "$cmd" = "run" ]] ; then
    lines=$(cd ${subdir} ; grep -IHnR ".\${length}\}" * 2>/dev/null)
    echo "["
    SEP=""
    if [ -z $lines ] ; then
        echo "]"
        exit 0
    fi
    while read -r i ; do
        file=$(echo $i | cut -d ':' -f 1)
        line=$(echo $i | cut -d ':' -f 2)
        echo ${SEP}
        SEP=", "
        echo "{"
        echo "\"type\" : \"Line Length\","
        echo "\"message\" : \"Line ${line} is too long (>${length})\","
        echo "\"file\" : \"${file}\","
        echo "\"line\" : ${line}"
        echo "}"
        done <<< "${lines}"
        echo "]"
    fi

if [[ "$cmd" = "applicable" ]] ; then
    echo "true"
fi

if [[ "$cmd" = "version" ]] ; then
    echo 1
fi
```



line-length.sh

Link to file: <https://help.sonatype.com/lift/files/78578759/78578760/1/1623180852664/line-length.sh>

mdl

```
#!/bin/bash

# Use: In the .lift/config specify:
# ```
# customTools = "https://help.sonatype.com/lift/files/78578761/78578762/1/1623180857208/mdl.sh"
# ```
dir=$1
# commit=$2
cmd=$3
shift
shift
shift
args=$*

if [[ "$cmd" = "run" ]] ; then
    gem install mdl 1>&2
    MDLOUT=$(mdl --json $args ${dir})
    echo "$MDLOUT" | jq '[ .[] | .file = .filename | .type = "MDL (" + .rule + ", " + .aliases[0] + ")"
| .message = .description | del(.aliases) | del(.description) | del(.filename) | del(.rule) ]'
fi

if [[ "$cmd" = "name" ]] ; then
    echo "MDL"
fi

if [[ "$cmd" = "applicable" ]] ; then
    echo "true"
fi

if [[ "$cmd" = "version" ]] ; then
```

```
echo 1
fi
```



mdl.sh

Link to file: <https://help.sonatype.com/lift/files/78578761/78578762/1/1623180857208/mdl.sh>

pmd

```
#!/bin/bash

# Use: In the .lift/config specify:
# ```
# customTools = "https://help.sonatype.com/lift/files/78578763/78578764/1/1623180860953/pmd.sh rulesets/
# java/quickstart.xml"
# ```
dir=$1
# commit=$2
cmd=$3
ruleset=$4
shift
shift
shift
shift
args=$*

if [[ "$cmd" = "run" ]] ; then
    if [[ -z "$args" ]] ; then
        jsonout=$(/opt/pmd/bin/run.sh pmd -d "$dir" -R "$ruleset" \
            -f codeclimate | \
            sed 's|\\w|\\\\w|g' | \
            jq '. | if type == "object" then . else empty end')
    else
        jsonout=$(/opt/pmd/bin/run.sh pmd -d "$dir" -R "$ruleset" \
            -f codeclimate "$@" | \
            sed 's|\\w|\\\\w|g' | \
```

```
jq '. | if type == "object" then . else empty end')

fi

echo "$jsonout" | jq "del(.type) | .[\\"type\\"] = .check_name | del(.\\"check_name\\")" \
| jq "del(.find) | .[\\"file\\"] = .location.path | del(.\\"location.path\\")" \
| jq "del(.line) | .[\\"line\\"] = .location.lines.begin | del(.
\\"location.lines.begin\\")" \
| jq "del(.message) | .[\\"message\\"] = .description | del(.\\"description\\")" \
| jq ".file = ( .file | sub(\"^$(pwd)\\\"; \".\\\"))" \
| jq -s .

fi

if [[ "$cmd" = "applicable" ]] ; then
    echo "true"
fi

if [[ "$cmd" = "version" ]] ; then
    echo "1"
fi
```



pmd.sh

Link to file: <https://help.sonatype.com/lift/files/78578763/78578764/1/1623180860953/pmd.sh>

Shell Check

```
#!/bin/bash

# Use: In the .lift.toml specify:
# ```
# customTools = "https://help.sonatype.com/lift/files/78578765/78578766/1/1623180865504/shellcheck.sh"
# ```
#dir=$1
#commit=$2
```

```
cmd=$3
shift
shift
shift

if [[ "$cmd" = "run" ]] ; then
    jsonout=$(find . -executable -name '*.sh' -print0 | xargs --null shellcheck -S error -f json "$@")
    if [[ ! ( "[" = "$jsonout" ) ]] ; then
        echo "$jsonout" | jq --arg type ShellCheck 'map (. + {type: $type})'
    else
        echo "$jsonout"
    fi
fi

if [[ "$cmd" = "applicable" ]] ; then
    echo "true"
fi

if [[ "$cmd" = "version" ]] ; then
    echo "1"
fi
```



shellcheck.sh

Link to file: <https://help.sonatype.com/lift/files/78578765/78578766/1/1623180865504/shellcheck.sh>

Static Check

```
#!/usr/bin/env bash
function tellApplicable() {
    files=$(git ls-files | egrep '.go$' | head)
    res="broken"
    if [[ -z "$files" ]] ; then
        res="false"
    fi
}
```

```

    else
        res="true"
    fi
    printf "%s\n" "$res"
}

function tellVersion() {
    echo "1"
}

function getTool() {
    pushd /tmp >/dev/null
    apt update >/dev/null && apt install -y golang >/dev/null
    curl -LO https://github.com/dominikh/go-tools/releases/download/2020.1.5/staticcheck_linux_amd64.tar.gz >
/dev/null
    tar xzf staticcheck_linux_amd64.tar.gz >/dev/null
    popd >/dev/null
}

function emit_results() {
    echo "$1" | \
        jq --slurp | \
            jq '.[] | .file = .location.file | .line = .location.line | .type = .code | del(.location) |
del(.severity) | del(.code) | del(.end)' | \
            jq --slurp
}

function run() {
    getTool
    raw_results=$(/tmp/staticcheck/staticcheck -f json -fail "" ./...)
    emit_results "$raw_results"
}

case "$3" in
    run)
        run
        ;;
    applicable)
        tellApplicable
        ;;
    *)
        tellVersion
        ;;
esac

```



Link to file: <https://help.sonatype.com/lift/files/78578767/78578768/1/1623180870140/staticcheck.sh>

Extending Lift V3

Lift's V3 Tool API abstracts over build systems allowing tool authors to focus on the analysis. Each project's build system is analyzed and broken down into a compilation database. The compilation commands are provided to V3 tools one at a time via the `run` command. Once the tool has been provided with all compilation commands the `finalize` command is called which gives tools a chance to produce the completed analysis.

Custom Tool APIs, such as V3 (presented here) and V1, are interface specifications about program arguments, standard input and standard output formats a tool must support in order to interoperate with Lift. The configuration reference page shows how to configure your repository so that each analysis will invoke tools of your choice.

Tool Invocation Format

As with the V1 API, the tool is always invoked with a file path and commit. Unlike v1, v3 commands sometimes indicate additional data is available in standard input.

- Typical invocation: `./tool.sh <filepath> <commit> <command>`
- stdin: command-specific JSON or nothing
- stdout: command-specific JSON or nothing
- stderr: ignored

The commands, their inputs, and expected output (if any) are listed below. For any one working directory inside a repository being analyzed the order of operations is:

1. Lift asks for the human readable name and version (no arguments)
2. applicable command to determine if the tool is appropriate for this repository.

3. run is called zero or more times, usually once for each entry in the compilation database.
4. finalize is called exactly once.

Command: (none)

When no arguments are provided the output should be a JSON dictionary:

```
{
  name : Text,
  version: 3
}
```

This is a special case - no arguments means the commit and repository directory also are not provided. These arguments are available when Lift calls every other command.

Command: applicable

- * Input Arguments: none
- * Output: boolean (true or false value as text output on standard out)

Command: version

- * Input Arguments: none
- * Output: Integer (3)

Command: run

The tool is called with the run command once for every source code file in the compilation.

Input Arguments:

```
{
  cwd : Text,
  cmd : Text,
  args : [Text],
  classpath : [Text]?,
  outputDir : Text?,
  files : [Text],
  residue : <JSON value>?
}
```

Cwd is the compiler working directory, cmd is the compiler command to build the source file (ex 'gcc', 'clang', or 'javac'), classpath is the classpaths of the compilation, files are the files to compile.

V3 tools must produce JSON output on standard out with the schema:

```
{
  toolNotes : [ToolNote]?,
  residue : <JSON value>?,
}
```

- toolNotes: A list of notes to be posted to the repository host.
- residue: If provided, text that will be provided to subsequent calls to command or finish

If there are no tool notes or residue then an empty JSON dictionary is expected (i.e { }).

Command: finalize

The finalize command is always called after all input files have been provided via the `run` command. The standard input include the immediately preceding residue, if any:

```
{
  residue : <JSON Value>?
}
```

As with run, the finalize output is a JSON dictionary but all the fields are optional:

Output arguments:

```
{
  toolNotes: [ToolNote]?,
  summary: Text?,
  residue: <JSON Value>?,
  pullRequest: PullRequest?
}
```

Command: talk

The talk command is used any time a user replies to a tool note via GitHub (GitLab, etc.) comment system. If the message is not recognized as a command by the Lift system then the text is passed through to the tool and it may respond with markdown text of its own.

Standard Input:

```
{
  residue : <JSON Value>?,
  messageText : Text,
  user : Text,
  noteID : Text?
}
```

Standard Output:

```
{
  message : Text?,
  toolNotes : [ToolNote]?
  residue : <JSON Value>?
}
```

Data Types

A ToolNote is a note (e.g., a bug) reported by the tool and is what is displayed on the pull request page. Rendering is in the form of ``<type>: <message>`` and allows github style markdown. The schema is:

```
{
  type : Text,
  message : Text,
  file : Text,
  line : Int?,
  column : Int?,
  function : Text?,
  detailsUrl: Text?,
  noteId : Text?
}
```

Residue is arbitrary JSON data that can aid future steps. Because the tool's `run` and `finalize` calls share a container the main use of residue is communicating from `finalize` to `talk`.

PullRequest is a dictionary describing what to place in the title and body of a pull request in addition to which branch to target and which source commit is the code to merge. The source commit must exist, committed into the working directory when this field is included.

```
{
  title: Text,
  body: Text,
  target_branch: Text?, // Defaults to the default branch of the repository if omitted
  source_commit: Text   // Commit must exist in the repository when this is emitted
}
```

Running Lift via GitHub Actions

Introduction

Lift can run on Actions to gain the benefits of retaining source code within your infrastructure, thus speeding up deployment by reducing IT burden and avoiding security review requirements.

[GitHub Actions](#)⁵⁸ is a powerful tool to run CI jobs in a developer friendly manner. Actions provides integrated CI coordination while allowing your organization to safely maintain its secrets and control the compute infrastructure.

Installation Method

There are three steps to installation:

1. Obtain the images
2. Commit the GitHub Actions workflow yaml file to your repositories
3. Add the secrets to your Github repositories or organization

Docker Image Access

Provide your <https://hub.docker.com>⁵⁹ username to Sonatype so we can grant read permissions to the necessary docker images. If there are corporate limitations regarding Docker Hub then request an S3 link of the images from your sales representative and push the images to your corporate registry.

GitHub Actions YAML

Copy the below GitHub actions YAML description and paste it as `.github/workflows/lift.yaml` in your repository.

Be sure to edit the YAML to reflect your project's specifics. For example, some corporate projects require access to a private Nexus server. This is your opportunity to provide secrets safely and construct `.m2/settings.xml` or install library dependencies. Notice there are two separate steps which each would need your build setup and system configuration.

```
name: Lift

on: [pull_request, workflow_dispatch]
```

⁵⁸ <https://github.com/features/actions>

⁵⁹ <https://hub.docker.com/>

```

jobs:
  analyze_src:
    name: Source Branch Analysis
    runs-on: ubuntu-latest
    container:
      image: musedev/analyst
      credentials:
        username: ${ secrets.DOCKER_HUB_USERNAME }}
        password: ${ secrets.DOCKER_HUB_PASSWORD }}
    steps:
      - uses: actions/checkout@v2
        with:
          fetch-depth: 0
      # - name: Setup build environment
      #   env:
      #     nexus-token: ${ secrets.MY_NEXUS_TOKEN }}
      #   run: |
      #     ... put shell script to create .m2/settings.xml
      #     .... put shell script to install build dependencies if needed
      - name: Produce src artifact
        run: |
          if [[ -z "$GITHUB_BASE_REF" ]] ; then
            echo "This is not a pull request."
            echo "Look for any result in the 'Destination Branch Analysis' job"
          else
            export SRC_SHA=$(cat $GITHUB_EVENT_PATH | jq -r -j .pull_request.head.sha)
            export DST_SHA=$(cat $GITHUB_EVENT_PATH | jq -r -j .pull_request.base.sha)
            analyst -t "$GITHUB_WORKSPACE" -C $SRC_SHA > lift-src-results.json
            cat lift-src-results.json | jq . | sed 's/\\n/\\n/g'
            echo -n "$SRC_SHA" > lift-commit
            echo -n "Source commit: " ; cat lift-commit ; echo ""
            echo "SRC_SHA -> DST_SHA: $SRC_SHA -> $DST_SHA"
            git -C $GITHUB_WORKSPACE diff ${DST_SHA}..${SRC_SHA} > lift.git.diff
          fi
      - name: Upload src artifact
        uses: actions/upload-artifact@v2
        with:
          name: lift_src_results
          path: lift*

  analyze_dst:
    name: Destination Branch Analysis
    runs-on: ubuntu-latest
    container:
      image: musedev/analyst
      credentials:
        username: ${ secrets.DOCKER_HUB_USERNAME }}
        password: ${ secrets.DOCKER_HUB_PASSWORD }}
    steps:
      - uses: actions/checkout@v2
        with:
          fetch-depth: 0
      # - name: Setup build environment
      #   env:
      #     nexus-token: ${ secrets.MY_NEXUS_TOKEN }}
      #   run: |

```

```
# ... put shell script to create .m2/settings.xml
# .... put shell script to install build dependencies if needed
- name: Produce dst artifact
  run: |
    if [ -z "$GITHUB_BASE_REF" ] ; then
      export DST_SHA="$GITHUB_SHA"
    else
      export DST_SHA=$(cat $GITHUB_EVENT_PATH | jq -r -j .pull_request.base.sha)
    fi
    analyst -t "$GITHUB_WORKSPACE" -C $DST_SHA > lift-dst-results.json
    cat lift-dst-results.json | jq . | sed 's/\\n/\\n/g'
- name: Upload dst artifact
  uses: actions/upload-artifact@v2
  with:
    name: lift_dst_results
    path: lift-dst-results.json

unify_results:
  name: Distill Result
  needs: [analyze_src, analyze_dst]
  runs-on: ubuntu-latest
  container:
    image: musedev/github-comment-composer
    credentials:
      username: ${ secrets.DOCKER_HUB_USERNAME }
      password: ${ secrets.DOCKER_HUB_PASSWORD }
  steps:
    - name: Get Dst
      uses: actions/download-artifact@v2
      with:
        name: lift_dst_results
    - name: Get Src
      uses: actions/download-artifact@v2
      with:
        name: lift_src_results
    - name: Combine
      run: |
        github-comment-composer lift.git.diff lift-src-results.json lift-dst-results.json > github-
comments.json
    - name: Post comments
      uses: actions/github-script@v4
      with:
        github-token: ${secrets.GITHUB_TOKEN}
        script: |
          const { promises: fs } = require('fs')
          const commit = await fs.readFile('lift-commit', 'utf8')
          console.log('Commit for comment: ' + commit)
          const comments = await fs.readFile('github-comments.json', 'utf8')
          console.log('Comments string (raw): ' + comments)
          for(const comment of JSON.parse(comments)) {
            console.log('Comment: ' + comment)
            github.pulls.createReviewComment({
              ...comment,
              owner: context.repo.owner,
              repo: context.repo.repo,
              pull_number: context.issue.number,
```

```
        commit_id: commit,  
    });  
}
```

Add Secrets to GitHub

Provide the GitHub Actions with any needed secrets such as the docker hub username and password.

GitHub Secrets are [documented here](#)⁶⁰ and can be found for a repository at https://github.com/<organization>/<repository_name>/settings/secrets/actions (see page 68). Organization-wide secrets can be added by browsing to <https://github.com/organizations/<organization>/settings/secrets/actions> (see page 68).

Operation Walk Through

Once the Action is configured, it is initiated whenever a developer makes a pull request and reports any discovered issues as comments in code review.

When a developer will make a pull request, GitHub will start both a source and destination Branch Analysis job. Once those jobs complete successfully, the Unify Results job reports any relevant issues as comments on the pull request.

Debugging

As with any CI environment, debugging a failing build requires an abnormal workflow. GitHub actions provides a console to which Lift emits the build information. If the actions terminal does not provide sufficient insight then try running the actions manually inside of a docker container. Start with `docker run --rm -it musedev/build-test bash` and step through each of your actions to isolate any issues.

Self Hosting Lift + GitLab

These instructions will guide you through installing the Lift to work with a GitLab Server.

Overview

Lift self-hosted platform is installed and managed using docker. The first step is to acquire a get-lift script from your Sonatype contact. From there, this document walks through the download of docker images and sets the initial configuration of Lift.

⁶⁰ <https://docs.github.com/en/actions/reference/encrypted-secrets>

Following Lift's installation you will be set up to configure the integration with your repository host. The high-level step summary is:

1. Provision a server (Suggestion: 2+ cores, 16+ GB RAM, 60+ GB HDD)
2. Create a new GitLab user for use by Lift and generate a token for this user.
3. Install Lift on the new server.
4. Configure Lift with the GitLab URL and token.
5. Configure one or more GitLab projects with a Lift webhook.
6. Test the connectivity by making a pull request.

Performing Installation

Start with Linux (recommended Ubuntu 20.04 or Amazon Linux with curl and docker installed). Then run the script provided by your sales representative, such as with `./get-lift.sh`. The script will first display a message and download the container images from S3. Upon completion of `get-lift` the protected links have been used to download the Lift software. The current directory now has docker images and an installation script name `lift.sh`.

As we proceed with the installation of Lift we will ask for, and store, configuration information. This will result in a new directory, `musedev-configuration`, complete with a configuration file of `muse.dhall`. We'll use the `NO_AUTO_CONFIGURE` option. This means we later edit the configuration file but this avoids auto-detection, ping tests, and other operations that can be foiled by corporate firewalls or unexpected operating systems.

```
NO_AUTO_CONFIGURE=true ./lift.sh
```

The script will verify basic tooling of docker, jq, and git are installed. The Lift images will be verified and loaded into the system's docker instance. Finally, a skeleton Lift configuration will be generated and placed in `musedev-configuration/muse.dhall`.

Edit the Lift configuration file. We should adjust fields for our repository host. Find, uncomment, and fill in the matching lines such as the line with a field of 'gitlab'.

For authentication information, make sure to use a token and not a password. In your GitLab instance, create a user for the bot, login and create a token by browsing to `https://<domain>/profile/personal_access_tokens`. Make sure and grant access to the API, repository read, and write access.

With configuration complete, we can run the system with `lift.sh start`. The system is ready to respond to any GitLab messages. Now configure a GitLab project to send webhook data to Lift.

```
./lift.sh start
```


Install Webhooks and Use the Application

Users can now install webhooks and have Lift respond to pull requests with insightful comments. For any given repository, browse to `https://<gitlab-domain>/<user>/<project>/hooks` (or click settings->webhooks from the GitLab project page). For the hook, use a URL of `http://<muse-domain>:2000/gitlab-event`. Also check the Merge Request Events box so Lift learns of the important happenings.

GitHub Actions

Lift can run on Actions to gain the benefits of retaining source code within your infrastructure, thus speeding up deployment by reducing IT burden and avoiding security review requirements.

[GitHub Actions](#)⁶¹ is a powerful tool to run CI jobs in a developer-friendly manner. Actions provides integrated CI coordination while allowing your organization to safely maintain its secrets and control the compute infrastructure.

Installation Method

There are three steps to installation:

1. Obtain the images
2. Commit the GitHub Actions workflow YAML file to your repositories
3. Add the secrets to your GitHub repositories or organization

Docker Image Access

Provide your [DockerHub](#)⁶² username to Sonatype, so we can grant read permissions to the necessary docker images. If there are corporate limitations regarding Docker Hub then request an S3 link of the images from your sales representative and push the images to your corporate registry.

GitHub Actions YAML

Copy the below GitHub actions YAML description and paste it as `.github/workflows/lift.yaml` in your repository.

Be sure to edit the YAML to reflect your project's specifics. For example, some corporate projects require access to a private Nexus server. This is your opportunity to provide secrets safely and construct `.m2/`

⁶¹ <https://github.com/features/actions>

⁶² <https://hub.docker.com>

settings.xml or install library dependencies. Notice there are two separate steps which each would need your build setup and system configuration.

```
name: Lift
on: [pull_request, workflow_dispatch]

jobs:
  analyze_src:
    name: Source Branch Analysis
    runs-on: ubuntu-latest
    container:
      image: musedev/analyst
      credentials:
        username: ${ secrets.DOCKER_HUB_USERNAME }
        password: ${ secrets.DOCKER_HUB_PASSWORD }
    steps:
      - uses: actions/checkout@v2
        with:
          fetch-depth: 0
      # - name: Setup build environment
      #   env:
      #     nexus_token: ${ secrets.MY_NEXUS_TOKEN }
      #   run: |
      #     ... put shell script to create .m2/settings.xml
      #     .... put shell script to install build dependencies if needed
      - name: Produce src artifact
        run: |
          if [[ -z "$GITHUB_BASE_REF" ]] ; then
            echo "This is not a pull request."
            echo "Look for any result in the 'Destination Branch Analysis' job"
          else
            export SRC_SHA=$(cat $GITHUB_EVENT_PATH | jq -r -j .pull_request.head.sha)
            export DST_SHA=$(cat $GITHUB_EVENT_PATH | jq -r -j .pull_request.base.sha)
            analyst -t "$GITHUB_WORKSPACE" -C $SRC_SHA > lift-src-results.json
            cat lift-src-results.json | jq . | sed 's/\\n/\\n/g'
            echo -n "$SRC_SHA" > lift-commit
            echo -n "Source commit: " ; cat lift-commit ; echo ""
            echo "SRC_SHA -> DST_SHA: $SRC_SHA -> $DST_SHA"
            git -C $GITHUB_WORKSPACE diff ${DST_SHA}..${SRC_SHA} > lift.git.diff
          fi
      - name: Upload src artifact
        uses: actions/upload-artifact@v2
        with:
          name: lift_src_results
          path: lift*

  analyze_dst:
    name: Destination Branch Analysis
    runs-on: ubuntu-latest
    container:
      image: musedev/analyst
      credentials:
```

```

        username: ${ secrets.DOCKER_HUB_USERNAME }}
        password: ${ secrets.DOCKER_HUB_PASSWORD }}
steps:
  - uses: actions/checkout@v2
    with:
      fetch-depth: 0
  # - name: Setup build environment
  #   env:
  #     nexus_token: ${ secrets.MY_NEXUS_TOKEN }}
  #   run: |
  #     ... put shell script to create .m2/settings.xml
  #     ... put shell script to install build dependencies if needed
  - name: Produce dst artifact
    run: |
      if [ -z "$GITHUB_BASE_REF" ] ; then
        export DST_SHA="$GITHUB_SHA"
      else
        export DST_SHA=$(cat $GITHUB_EVENT_PATH | jq -r -j .pull_request.base.sha)
      fi
      analyst -t "$GITHUB_WORKSPACE" -C $DST_SHA > lift-dst-results.json
      cat lift-dst-results.json | jq . | sed 's/\n/\n/g'
  - name: Upload dst artifact
    uses: actions/upload-artifact@v2
    with:
      name: lift_dst_results
      path: lift-dst-results.json

unify_results:
  name: Distill Result
  needs: [analyze_src, analyze_dst]
  runs-on: ubuntu-latest
  container:
    image: musedev/github-comment-composer
    credentials:
      username: ${ secrets.DOCKER_HUB_USERNAME }}
      password: ${ secrets.DOCKER_HUB_PASSWORD }}
  steps:
    - name: Get Dst
      uses: actions/download-artifact@v2
      with:
        name: lift_dst_results
    - name: Get Src
      uses: actions/download-artifact@v2
      with:
        name: lift_src_results
    - name: Combine
      run: |
        github-comment-composer lift.git.diff lift-src-results.json lift-dst-results.json > github-
comments.json
    - name: Post comments
      uses: actions/github-script@v4
      with:
        github-token: ${ secrets.GITHUB_TOKEN }}
        script: |
          const { promises: fs } = require('fs')
          const commit = await fs.readFile('lift-commit', 'utf8')

```

```
console.log('Commit for comment: ' + commit)
const comments = await fs.readFile('github-comments.json', 'utf8')
console.log('Comments string (raw): ' + comments)
for(const comment of JSON.parse(comments)) {
  console.log('Comment: ' + comment)
  github.pulls.createReviewComment({
    ...comment,
    owner: context.repo.owner,
    repo: context.repo.repo,
    pull_number: context.issue.number,
    commit_id: commit,
  });
}
```

Add Secrets to GitHub

Provide the GitHub Actions with any needed secrets such as the docker hub username and password.

GitHub Secrets are [documented here](#)⁶³ and can be found for a repository at <https://github.com/<org>/<repository name>/settings/secrets/actions>. Organization-wide secrets can be added by browsing to <https://github.com/organizations/<org>/settings/secrets/actions>.

Operation Walk-Through

Once the Action is configured, it is initiated whenever a developer makes a pull request and reports any discovered issues as comments in code review.

When a developer will make a pull request, GitHub will start both a source and destination Branch Analysis job. Once those jobs complete successfully, the Unify Results job reports any relevant issues as comments on the pull request.

Debugging

As with any CI environment, debugging a failing build requires an abnormal workflow. GitHub actions provides a console to which Lift emits the build information. If the actions terminal does not provide sufficient insight then try running the actions manually inside a docker container. Start with ``docker run --rm -it musedev/build-test bash`` and step through each of your actions to isolate any issues.

Self Hosting Lift + GitHub

These instructions will guide you through installing the Lift to work with GitHub Enterprise.

⁶³ <https://docs.github.com/en/actions/reference/encrypted-secrets>

Overview

Lift self-hosted platform is installed and managed using docker. The first step is to acquire a get-lift script from your Sonatype contact. From there, this document walks through the download of docker images and sets the initial configuration of Lift. If you wish to deploy on Kubernetes based infrastructure, such as EKS or OpenShift, then contact a sales representative to receive Helm charts.

Following Lift's installation you will be set up to configure the integration with your repository host. The high-level step summary is:

1. Create a new GitHub App for use by Lift, record the shared secrets and application Id.
2. Provision a server (Suggestion: 2+ cores, 16+ GB RAM, 60+ GB HDD)
3. Install Lift on the new server.
4. Configure Lift with the GitHub App information.
5. Install the GitHub App one or more GitHub organizations.
6. Test the system by making a pull request.

Create a GitHub App

Here we will add the application to your private GitHub Enterprise server. The end result is that organizations and users are able to install the app on their individual accounts as desired.

Start logged in as the user who will own and manage the application itself. This can be, but does not have to be, a system administrator. This process does not result in the app receiving any permissions to the account.

1. In a web browser, open [https://<github.example.com>/settings/apps/new?name=Lift&description=Better%20Code%2c%20No%20Fuss&url=https://sonatype.com&public=true&contents=read&pull_requests=write&events\[\]=pull_request](https://<github.example.com>/settings/apps/new?name=Lift&description=Better%20Code%2c%20No%20Fuss&url=https://sonatype.com&public=true&contents=read&pull_requests=write&events[]=pull_request) fixing the URL to match your GitHub Enterprise deployment.
2. In the Webhook URL textbox, enter <https://<lift.example.com>/github> (taking care to update the URL to match your deployment)

Now click Create GitHub App. The app is now created - users can install it by visiting <https://<github.example.com>/github-apps/Lift>. Before users can try the app we need to setup the server at that URL - lift.example.com - and that server will need some authentication information for this newly created app. On the GitHub app administration page (<https://<github.example.com>/settings/apps/Lift>) there are four important pieces of information:

1. Under About section note the ID: (ex: if this is the first app it will be "1").
2. Under About section note the Client ID.
3. Under Private key click Generate private key and save the PEM file.
4. Under Basic Information generate and record the webhook secret.

Make note of this information as it will be used during the installation of the server.

Performing an Installation

Start with Linux (recommended Ubuntu 20.04 or Amazon Linux with curl and docker installed). This server should be reachable with the webroute you provided to github in the above section (where the lift.example.com placeholder appears).

Run the script provided by your sales representative, such as with `./get-lift.sh`. The script will first display a message and download the container images from S3. Upon completion of get-lift the protected links have been used to download the Lift software. The current directory now has docker images and an installation script named lift.sh.

The installation will result in a new directory, musedev-configuration, complete with a configuration file of muse.dhall. We'll use the NO_AUTO_CONFIGURE option. This means we later edit the configuration file but this avoids auto-detection, ping tests, and other operations that can be foiled by corporate firewalls or unexpected operating systems.

```
NO_AUTO_CONFIGURE=true ./lift.sh
```

The script will verify basic tooling of docker, jq, and git are installed. The Lift images will be verified and loaded into the system's docker instance. Finally, a skeleton Lift configuration will be generated and placed in musedev-configuration/muse.dhall.

Edit the Lift configuration file. We should adjust fields for our repository host. Find, uncomment, and fill in the matching lines such as the line with a field of 'github'. Fill in the GitHub application's Client ID, application Id, and application secret. Copy the PEM file provided by github to the directory musedev-configuration/github.pem. The github field of the configuration should appear something like the below snippet. Notice the appRSAKey should not be modified.

```
, github = Some { clientId = "Iv1.abcdef"
  , appId      = "12345"
  , appSecret  = "abcd1234ef"
  , appRSAKey  = /etc/muse/muse_github_privatekey.pem as Text
  , oauth = Some
    { clientId = "Iv1.abcdef"
      , authType = "autolink"
    }
  , clientSecret = "abcd1234ef"
}
```

With configuration complete, we can run the system with `lift.sh start`. The system is ready to respond to any GitHub messages.

```
./lift.sh start
```

Install The Lift GitHub App

Users can now install the github App and have Lift respond to pull requests with insightful comments. As a GitHub Enterprise user - be it the owner of the app or any other - visit <https://<github.example.com>/github-apps/Lift> and click "Install". You can give the application permission to all or select repositories.

From here the behavior will match what is found on MuseDev's GitHub cloud offering. We strongly recommend you create a project with a simple shell script and then make a pull request introducing a bug in the script. For example, commit and push to the main branch an executable file of:

```
#!/usr/bin/env bash
key1=value1
```

And make a new branch add invalid code caught by one of the analysis tools such as:

```
#!/usr/bin/env bash
key1=value1
key2 = value2
# Notice the spaces around '=' in the above
```

This trivial bug serves as a fast smoke test of the system including network connectivity, docker/kubernetes volume allocation, container creation, and github credentials. A successful result would include a comment being posted by the Lift application back to the pull request noting the bug on line 3.