Of course! It can be confusing when you're starting out. Let's break it down into simple, clear steps to get your JMeter test running with Maven inside VS Code.

The main idea is to use Maven to **automatically download JMeter, run your test script (.jmx file), and generate a report**.

### Prerequisites

Before we start, make sure you have these installed:

* **Java Development Kit (JDK):** JMeter and Maven both need Java to run.
* **Apache Maven:** Make sure it's installed and configured in your system's PATH.
* **Visual Studio Code:** Your code editor.
* **Apache JMeter:** You still need the JMeter application to *create* your test plans. You won't use it to run the test in this process, but you need it to build the script.

### Step 1: Create the Project Folder Structure 📁

First, you need to create the exact folders that Maven and the JMeter plugin expect.

1. Create a main project folder. Let's call it my-jmeter-project.
2. Inside my-jmeter-project, open the VS Code terminal (Ctrl+ ` or by going to Terminal > New Terminal).
3. Type these commands one by one to create the required nested folders:
4. Bash
5. 

mkdir -p src/test/jmeter

1.   
   This single command creates the src folder, then test inside it, and finally jmeter inside that.

Your folder structure inside VS Code should now look like this:



my-jmeter-project/

└── src/

└── test/

└── jmeter/



### Step 2: Create a Simple JMeter Test Plan (Test.jmx) 📝

This is the most important part. The .jmx file is your actual test script. You create it using the JMeter GUI. The "copied from jmeter/extras" suggestion in the original text is a bit misleading; it's better to create your own.

1. **Launch the JMeter application** on your computer.
2. Right-click on **Test Plan** > **Add** > **Threads (Users)** > **Thread Group**.
3. Right-click on **Thread Group** > **Add** > **Sampler** > **HTTP Request**.
4. In the HTTP Request panel, enter google.com in the **Server Name or IP** field. This tells JMeter to send a request to Google.
5. **Save the test plan.** Go to **File > Save**. Navigate to the folder you created in Step 1 (my-jmeter-project/src/test/jmeter/) and save the file as Test.jmx.

You have now created a test script and placed it where the Maven plugin will look for it.

### Step 3: Create and Configure the pom.xml File ⚙️

Now, let's create the "brain" of the project—the pom.xml file. This file tells Maven what to do.

1. In the root of your my-jmeter-project folder (the same level as the src folder), create a new file named pom.xml.
2. Copy and paste the complete, corrected XML code below into your pom.xml file. The snippet in your question was incomplete.

XML



<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.mycompany.jmeter</groupId>

<artifactId>my-jmeter-project</artifactId>

<version>1.0-SNAPSHOT</version>

<build>

<plugins>

<plugin>

<groupId>com.lazerycode.jmeter</groupId>

<artifactId>jmeter-maven-plugin</artifactId>

<version>3.7.0</version> <executions>

<execution>

<id>jmeter-tests</id>

<phase>verify</phase>

<goals>

<goal>jmeter</goal>

</goals>

</execution>

</executions>

</plugin>

</plugins>

</build>

</project>

**What this file does:**

* **Section 1:** Basic information about your project.
* **Section 2:** Declares that you want to use the **jmeter-maven-plugin**. This is the tool that connects Maven to JMeter.
* **Section 3:** This is the critical "event." It tells Maven: "When you get to the **verify** step of your build process, execute the **jmeter** goal from the plugin."

### Step 4: Run the Test from VS Code 🚀

Everything is now in place! Your final folder structure is:



my-jmeter-project/

├── src/

│ └── test/

│ └── jmeter/

│ └── Test.jmx

└── pom.xml

1.   
   Make sure you are in the VS Code terminal and that you are in the root directory (my-jmeter-project).
2. Type the following command and press Enter:
3. Bash
4. 

mvn verify

1. 
2. **Watch the magic happen!** Maven will start running. The first time, it will download the JMeter plugin and JMeter itself. Then, it will find your Test.jmx file, run the test in the background (you won't see the JMeter GUI), and show the results in the terminal.

If everything worked, you will see a lot of text, ending with a message like:



[INFO] ------------------------------------------------------------------------

[INFO] BUILD SUCCESS

[INFO] ------------------------------------------------------------------------



### Step 5: Check the HTML Report ✅

The best part of using this plugin is the automatic report generation.

1. After the mvn verify command succeeds, a new folder called target will appear in your project directory.
2. Navigate into target/jmeter/reports/.
3. You will find a folder named after your .jmx file (e.g., Test-20250917).
4. Inside that folder, open the index.html file in a web browser.

You will see a beautiful, detailed dashboard with graphs and statistics about your test run.

That's it! You have successfully configured and run a JMeter test using Maven from within VS Code. You can now create more complex .jmx files and run them the same way.

Excellent news! Your log shows that you have **successfully fixed the network certificate issue**. 🎉 Maven is now downloading all the required files perfectly.

Now we have a new, much simpler error related to the JMeter plugin's configuration.

### The New Problem Explained 💡

The key part of the new error message is:

C:\Users\USER\Downloads\my-jmeter-project\target\config.json (The system cannot find the file specified)

The error message itself even gives you the solution! The JMeter Maven plugin works in two main steps:

1. **configure**: It sets up the test environment. This includes downloading the correct version of Apache JMeter and creating necessary configuration files (like config.json).
2. **jmeter**: It actually runs the test using the environment that was just set up.

Your pom.xml file correctly tells Maven to run the jmeter step, but it's missing the instruction to run the configure step first.

**Analogy:** You told a chef to cook a meal (jmeter goal) but never told them to preheat the oven or get the ingredients out of the fridge (configure goal). 👨‍🍳

### The Solution: Update Your pom.xml ✅

You need to add a second <execution> block to your pom.xml to explicitly tell the plugin to run the configure step.

1. Open your pom.xml file.
2. **Replace the entire content** with the corrected code below. I've added the new required section and commented it for you.

XML



<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.mycompany.jmeter</groupId>

<artifactId>my-jmeter-project</artifactId>

<version>1.0-SNAPSHOT</version>

<build>

<plugins>

<plugin>

<groupId>com.lazerycode.jmeter</groupId>

<artifactId>jmeter-maven-plugin</artifactId>

<version>3.7.0</version>

<executions>

<execution>

<id>configuration</id>

<goals>

<goal>configure</goal>

</goals>

</execution>

<execution>

<id>jmeter-tests</id>

<phase>verify</phase>

<goals>

<goal>jmeter</goal>

</goals>

</execution>

</executions>

</plugin>

</plugins>

</build>

</project>



### What to Do Now

1. Save the updated pom.xml file.
2. Go back to your terminal in VS Code.
3. Run the following command:
4. Bash
5. 

mvn clean verify

1.   
   (The clean part is good practice; it deletes the old target directory to ensure a fresh start.)

This time, you should see it successfully download JMeter, configure it, execute your Test.jmx script, and finish with a **BUILD SUCCESS** message. Afterward, you will find your HTML report in the target/jmeter/reports directory as expected.

Excellent, you've successfully run the test! The screenshot shows a BUILD SUCCESS and the .csv result file, which is a huge step forward. 🎉

You're right, the HTML report is missing. This is happening because the JMeter Maven plugin separates **running the test** from **generating the report**. You've successfully run the test, but you need to explicitly tell Maven to generate the HTML dashboard.

### The Solution: Add the results Goal

The fix is a simple one-line addition to your pom.xml file. You need to add the <goal>results</goal> to the execution that runs your tests.

1. Open your pom.xml file.
2. Find the <execution> block with the id jmeter-tests.
3. Add the line <goal>results</goal> right after <goal>jmeter</goal>.

Here is the corrected <executions> section. You can just replace your existing one with this.

XML



<executions>

<execution>

<id>configuration</id>

<goals>

<goal>configure</goal>

</goals>

</execution>

<execution>

<id>jmeter-tests</id>

<phase>verify</phase>

<goals>

<goal>jmeter</goal>

<goal>results</goal>

</goals>

</execution>

</executions>



### What to Do Now

1. **Save** your updated pom.xml file.
2. Go to your VS Code terminal and run the test again using the clean command to ensure everything is fresh:
3. Bash
4. 

mvn clean verify

1. 
2. After you see **BUILD SUCCESS**, check your target folder again. You will now see a new **reports** directory.  
   Inside that folder, you will find your interactive HTML dashboard. Just open the index.html file in a browser. 📊

Your file structure will look like this:



target/

└── jmeter/

├── logs/

├── results/

│ └── ... .csv

└── reports/ <-- THIS WILL NOW BE CREATED

└── ... /

└── index.html



This is actually a good sign! 👍 Your pom.xml is now configured perfectly. The build is failing for a new, and very important, reason: **your performance test itself had errors, and the plugin is correctly flagging it as a failure.**

This is a feature, not a bug. It's a quality gate to prevent bad performance from passing your build.

### The Problem Explained: A Built-in Quality Check 🚧

Look at this key line from your error log:

Failing build because error percentage ... is above accepted threshold 5.88. ... '0.0'

Here's what it means:

* Your test ran and had a **5.88% error rate** (1 failed request out of 17).
* By default, the JMeter Maven plugin has a **0.0% accepted error threshold**.
* Since 5.88% is greater than 0.0%, the plugin intentionally fails the build to alert you that the application you're testing is producing errors under load.

You have two ways to solve this, depending on your goal.

### Solution 1: Fix the Test (Recommended)

The best practice is to investigate *why* the test is failing. An error rate of almost 6% is usually a sign of a real problem.

1. **Check the Logs:** Navigate to target/jmeter/logs/ and open the .jmx.log file to see if JMeter reported any technical issues.
2. **Analyze the Results:** The most valuable file is the .csv result file located at target/jmeter/results/. You can open this file in the JMeter GUI to see exactly which request failed and why.
   * Open the **JMeter GUI**.
   * Add a **View Results Tree** listener.
   * In the "Filename" field of the listener, click **"Browse..."** and select your .csv file.
   * JMeter will load the results. Click on the failed request (it will have a red icon) to see the full request, response code, and response data to understand why it failed.

### Solution 2: Increase the Error Threshold (If Errors are Acceptable)

If you expect some small percentage of errors and want the build to pass anyway, you can configure the plugin to be more lenient.

1. Open your pom.xml file.
2. Add a <configuration> section to your jmeter-maven-plugin declaration and set an errorRateThresholdInPercent.

Here is the complete <plugin> block with the new configuration added. Let's set the threshold to 10%, which is higher than your current error rate.

XML



<plugin>

<groupId>com.lazerycode.jmeter</groupId>

<artifactId>jmeter-maven-plugin</artifactId>

<version>3.7.0</version>

<configuration>

<errorRateThresholdInPercent>10</errorRateThresholdInPercent>

</configuration>

<executions>

<execution>

<id>configuration</id>

<goals>

<goal>configure</goal>

</goals>

</execution>

<execution>

<id>jmeter-tests</id>

<phase>verify</phase>

<goals>

<goal>jmeter</goal>

<goal>results</goal>

</goals>

</execution>

</executions>

</plugin>



### What to Do Now ✅

1. **We strongly recommend trying Solution 1 first** to be sure you don't have a real issue with your application.
2. If you decide the errors are acceptable, implement **Solution 2** by updating your pom.xml.
3. Save the file and run the build again:
4. Bash
5. 

mvn clean verify

1. 

With the error threshold increased, your build should now finish with **BUILD SUCCESS**, and you will finally find your HTML report in target/jmeter/reports/.

Of course. The CSV file you provided clearly shows why the build is failing.

The build fails because your JMeter test script has a failed request, and the Maven plugin is correctly configured to stop the build if any errors are found. Reports are not generated because the process halts when it detects the failure.

### The Exact Error in Your Test 🎯

The problem is in the very last line of your CSV results file.

**success,false,,1122,0,1,1,http:/,0,0,0**

Let's break down this specific failure:

* **Success Column:** It's marked **false**.
* **Response Message:** URI does not specify a valid host name: http:/
* **URL Column:** The URL is just http:/, which is an invalid web address.

This means you have an "HTTP Request" sampler in your JMeter test plan that has a typo in the URL.

### Why This Fails the Build

When you added the <goal>results</goal>, you activated the plugin's quality gate. By default, this gate is set to a **0% error tolerance**. Since your test now has one failed request, the error rate is greater than 0%, so the plugin intentionally fails the build to alert you to the problem.

This is a good thing—it's preventing you from accepting a broken test as a success.

### How to Fix It 🔧

You need to correct the error in your JMeter test plan.

1. Open your **Toolsvilla Loadtest plan.jmx** file in the JMeter GUI.
2. Look through your Thread Group for an **HTTP Request** sampler that is either unlabeled or has an error.
3. Find the one that has the invalid URL (http:/) in the **Server Name or IP** field.
4. Correct this typo to be a valid URL (e.g., http://example.com).
5. **Save** your .jmx file.

After you've fixed the typo in your test plan, run the command again in VS Code:

Bash



mvn clean verify

Your test should now pass with a 0% error rate, the build will succeed, and the HTML reports will be generated correctly. ✅

You can definitely edit your Toolsvilla Loadtest plan.jmx file directly in VS Code. A .jmx file is just a structured XML text file, and VS Code is a powerful text editor.

This method is perfect for making small, quick changes, like fixing the URL typo you have.

### How to Edit the .jmx File in VS Code ✍️

Here are the steps to find and fix the error directly in your editor.

#### 1. Open the File

In the VS Code explorer on the left, navigate to src/test/jmeter and click on **Toolsvilla Loadtest plan.jmx** to open it. You will see a large block of XML code.

#### 2. Find the Error

You know from the error log that the bad URL is http:/.

* Press **Ctrl+F** (or Cmd+F on Mac) to open the search bar in VS Code.
* Type **http:/** into the search bar and press Enter. VS Code will immediately jump to the line with the error.

#### 3. Understand the XML Structure

You will find the error inside an <HTTPSamplerProxy> element, which represents an HTTP Request sampler. It will look something like this:

XML



<HTTPSamplerProxy guiclass="HttpTestSampleGui" testclass="HTTPSamplerProxy" testname="HTTP Request" enabled="true">

<elementProp name="HTTPsampler.Arguments" elementType="Arguments" guiclass="HTTPArgumentsPanel" testclass="Arguments" testname="User Defined Variables" enabled="true">

<collectionProp name="Arguments.arguments"/>

</elementProp>

<stringProp name="HTTPSampler.domain"></stringProp>

<stringProp name="HTTPSampler.port"></stringProp>

<stringProp name="HTTPSampler.protocol">http</stringProp>

<stringProp name="HTTPSampler.path">/</stringProp>

<stringProp name="HTTPSampler.method">GET</stringProp>

</HTTPSamplerProxy>

Your error is likely a typo in one of the stringProp elements like HTTPSampler.domain, HTTPSampler.protocol, or HTTPSampler.path.

#### 4. Correct the Error

Let's say your broken URL http:/ was supposed to be https://api.toolsvilla.com/some/path. You would correct the XML properties like this:

XML



<HTTPSamplerProxy guiclass="HttpTestSampleGui" testclass="HTTPSamplerProxy" testname="HTTP Request" enabled="true">

<elementProp name="HTTPsampler.Arguments" elementType="Arguments" guiclass="HTTPArgumentsPanel" testclass="Arguments" testname="User Defined Variables" enabled="true">

<collectionProp name="Arguments.arguments"/>

</elementProp>

<stringProp name="HTTPSampler.domain">api.toolsvilla.com</stringProp>

<stringProp name="HTTPSampler.port"></stringProp>

<stringProp name="HTTPSampler.protocol">https</stringProp>

<stringProp name="HTTPSampler.path">/some/path</stringProp>

<stringProp name="HTTPSampler.method">GET</stringProp>

</HTTPSamplerProxy>

#### 5. Save and Re-run

* Press **Ctrl+S** (or Cmd+S) to save the changes to your .jmx file.
* Go to your terminal and run the test again:
* Bash
* 

mvn clean verify

* 

## The test should now pass, and your reports will be generated. ✅

### ⚠️ Important Warning

Editing .jmx files by hand is great for quick fixes, but it can be risky for complex changes.

* **DO** use VS Code to: Change a server name, update a URL path, or tweak a request parameter.
* DO NOT use VS Code to: Add new samplers, add logic controllers (like If/Loop), or delete elements. The XML structure is complex, and it's very easy to break the file.  
  For any major changes, always use the JMeter GUI to ensure the file structure remains valid.

You've found the exact source of the problem! The XML snippet you've shared is for the broken "HTTP Request" sampler.

### The Problem

The issue is that your <HTTPSamplerProxy> block is **missing the essential lines that define the URL**. It doesn't have the properties for the protocol (http/https), the domain (the website name), or the path.

This is why JMeter reports the error URI does not specify a valid host name—the sampler is completely empty and doesn't know where to send the request.

### The Solution 🔧

You need to **replace** your broken XML block with a complete one.

1. Find the broken <HTTPSamplerProxy> in your .jmx file that you posted above.
2. Delete that entire block, from <HTTPSamplerProxy ...> to </HTTPSamplerProxy>.
3. Replace it with the corrected code below.

XML



<HTTPSamplerProxy guiclass="HttpTestSampleGui" testclass="HTTPSamplerProxy" testname="HTTP Request" enabled="true">

<elementProp name="HTTPsampler.Arguments" elementType="Arguments" guiclass="HTTPArgumentsPanel" testclass="Arguments" testname="User Defined Variables" enabled="true">

<collectionProp name="Arguments.arguments"/>

</elementProp>

<stringProp name="HTTPSampler.domain">api.toolsvilla.com</stringProp>

<stringProp name="HTTPSampler.port"></stringProp>

<stringProp name="HTTPSampler.protocol">https</stringProp>

<stringProp name="HTTPSampler.path">/some/path</stringProp>

<stringProp name="HTTPSampler.method">GET</stringProp>

<boolProp name="HTTPSampler.follow\_redirects">true</boolProp>

<boolProp name="HTTPSampler.auto\_redirects">false</boolProp>

<boolProp name="HTTPSampler.use\_keepalive">true</boolProp>

<boolProp name="HTTPSampler.postBodyRaw">false</boolProp>

<stringProp name="HTTPSampler.embedded\_url\_re"></stringProp>

<stringProp name="HTTPSampler.connect\_timeout"></stringProp>

<stringProp name="HTTPSampler.response\_timeout"></stringProp>

</HTTPSamplerProxy>

1.   
   **Customize the new code.** Change the values for HTTPSampler.domain and HTTPSampler.path to match the actual URL you want to test.
2. Save the Toolsvilla Loadtest plan.jmx file.
3. Run the test again from your terminal:
4. Bash
5. 

mvn clean verify

1. 

This will fix the test failure, allow your build to pass, and generate the HTML report. ✅

You've asked the perfect question. No, **the changes to your pom.xml were correct**. The results goal and the error threshold are standard configurations.

This is a new and different error, and your screenshot gives us a huge clue as to what's wrong. I know this can be frustrating, but we're very close.

### The New Problem: A Corrupted .jmx File 💡

The key error message has changed. It's no longer about the error percentage. It's now:

Test failed with exit code: 1

This is a general error that means the JMeter program itself crashed. The lines just before the failure tell us why:

[INFO] path : /jmeterTestPlan/hashTree/hashTree

[INFO] line number : 440

This means JMeter failed while trying to **read and parse your Toolsvilla Loadtest plan.jmx file**. The manual edits we tried previously have accidentally broken the XML structure of the file around **line 440**. This is very easy to do and is a common risk when editing .jmx files by hand.

### The Solution: Rebuild the File with the JMeter GUI 🔧

Manually finding and fixing broken XML is very difficult. The fastest and safest way to fix this is to let the JMeter GUI rebuild the file for you.

1. **Open JMeter:** Launch the Apache JMeter application on your computer.
2. **Open Your Test Plan:** Go to **File > Open** and select your src/test/jmeter/Toolsvilla Loadtest plan.jmx file.
   * JMeter will likely show you an error message when it tries to open the file. This confirms the file is corrupted.
3. **Delete the Broken Sampler:** In the test tree on the left, find the broken "HTTP Request" sampler that we were trying to fix before. **Right-click on it and choose "Remove"**.
4. **Add a New Sampler:** Right-click on your **Thread Group** and go to **Add > Sampler > HTTP Request**.
5. **Configure the New Sampler:** Fill in the correct details for the new sampler (Protocol, Server Name, Path, etc.).
6. **Save the Plan:** Go to **File > Save**. This will overwrite your broken .jmx file with a fresh, structurally correct version.

### Final Step ✅

Now that you have a clean, valid .jmx file saved from the JMeter GUI:

* Go back to your VS Code terminal.
* Run the command one more time:
* Bash
* 

mvn clean verify

* 

This time, JMeter will be able to read the file correctly, your test will execute, and the build should finally succeed.

That's great news! If it opened without an error, it means the file is easy to fix.

The problem is still there, but now we can see it visually. Look at the last sampler in your test plan.

### Find and Remove the Broken Sampler 🎯

In your screenshot, look at the bottom of the list under "Virtual user group". You will see a sampler named simply **"HTTP Request"**.

This is the default name for a blank, unconfigured sampler. This is the one causing your build to fail because it has no server name or path to test.

Here's what to do:

1. In the JMeter GUI, **right-click** on that last sampler named **"HTTP Request"**.
2. From the context menu, choose **Remove**.
3. Click **"Yes"** to confirm. 🗑️
4. The most important step: Go to **File > Save** to save the changes to your .jmx file. This will permanently remove the broken element.

### Final Step ✅

Now that you've removed the bad sampler and saved the file:

* Go back to your terminal in VS Code.
* Run the command again:
* Bash
* 

mvn clean verify

* 

Your test plan is now clean, and the Maven build should finally run successfully.

Congratulations on the successful build! 🎉 You're right, the HTML report should be there.

This happens for a simple reason: sometimes the plugin needs to be explicitly told to generate the final HTML dashboard, even when the results goal runs. We can fix this with one more small addition to the pom.xml.

### The Solution: Add the generateReports Flag ⚙️

You need to add a <configuration> section to your plugin and set the <generateReports> flag to true. This leaves no doubt that you want the report to be created.

1. Open your **pom.xml** file.
2. Find the <plugin> section for the jmeter-maven-plugin.
3. **Replace your entire <plugin> block** with the complete version below. This includes our previous fixes and adds the new flag.

XML



<plugin>

<groupId>com.lazerycode.jmeter</groupId>

<artifactId>jmeter-maven-plugin</artifactId>

<version>3.7.0</version>

<configuration>

<generateReports>true</generateReports>

<errorRateThresholdInPercent>10</errorRateThresholdInPercent>

</configuration>

<executions>

<execution>

<id>configuration</id>

<goals>

<goal>configure</goal>

</goals>

</execution>

<execution>

<id>jmeter-tests</id>

<phase>verify</phase>

<goals>

<goal>jmeter</goal>

<goal>results</goal>

</goals>

</execution>

</executions>

</plugin>



### Final Step ✅

Now that you've told Maven to generate the report:

* Save the pom.xml file.
* Run the command in your terminal one last time:
* Bash
* 

mvn clean verify

* 

After the build succeeds, check your file explorer. A **reports** folder will now appear inside target/jmeter, and your index.html file will be inside it.

Yes, you can ignore that message. Congratulations, you have successfully configured the entire process! 🎉

That message is **not a build error**. It is a helpful **warning** from a VS Code extension that analyzes your pom.xml file.

### What the Warning Means

The message "Plugin execution not covered by lifecycle configuration" is simply telling you that one of your <execution> blocks is missing a <phase> tag.

* Your jmeter-tests execution is correctly tied to the verify phase.
* Your configuration execution doesn't have a phase specified.  
  It still works because the plugin has a default phase it runs in, but the VS Code extension is flagging it as a best-practice suggestion.

### How to Fix It (Optional but Recommended)

To make the warning disappear and have a perfectly configured file, you can add the <phase> tag to the configuration block as well.

1. Open your pom.xml.
2. Find the <execution> block with the id configuration.
3. Add the line <phase>verify</phase> to it.

Here is the final, complete <executions> block. This is considered the best practice.

XML



<executions>

<execution>

<id>configuration</id>

<phase>verify</phase>

<goals>

<goal>configure</goal>

</goals>

</execution>

<execution>

<id>jmeter-tests</id>

<phase>verify</phase>

<goals>

<goal>jmeter</goal>

<goal>results</goal>

</goals>

</execution>

</executions>

After adding that one line and saving, the warning in VS Code will disappear.

But to be clear: Your setup is working perfectly. You can consider this final tweak as optional code cleanup. Great job getting everything to run! ✨

Of course. Moving to automation and proper report storage is the perfect next step. Here's a complete roadmap to get you there.

## The Roadmap: From Local to Automated

Our plan has three main steps:

1. **Archive Reports:** Set up a way to store the HTML report from each test run.
2. **Version Control:** Put your project into GitHub. This is essential for both Jenkins and GitHub Actions.
3. **Automate:** Create a CI/CD pipeline in either Jenkins or GitHub Actions to run your tests automatically.

## Step 1: Storing the Dashboard Records

## Right now, every time you run mvn clean verify, the target directory (which contains your report) is deleted. The standard way to store these reports is to have your automation server archive them. How it works: After the Maven command finishes, the automation server (Jenkins or GitHub Actions) will copy the generated HTML report from the target/jmeter/reports/ directory and attach it to the specific build that created it. This gives you a complete history of every test run and its corresponding report. We will configure this in Step 3.

## Step 2: Put Your Project on GitHub 🐙

This is the foundation for automating your tests.

1. **Create a GitHub Repository:** Go to GitHub and create a new, empty repository. Let's say you call it jmeter-performance-tests.
2. Prepare Your Local Project: In your project's root folder (my-jmeter-project), create a new file named .gitignore. This file tells Git which files and folders to ignore. This is crucial to avoid committing large, unnecessary files.  
   Put the following content into your .gitignore file:
3. 

# Maven build output

target/

# Log files

\*.log

# IDE files

.vscode/

.idea/

1. 
2. **Push to GitHub:** In your VS Code terminal, run these commands one by one to upload your project.
3. Bash
4. 

# Initialize a git repository

git init -b main

# Add all files (except those in .gitignore)

git add .

# Make your first commit

git commit -m "Initial JMeter project setup"

# Link your local repo to the one on GitHub (get the URL from GitHub)

git remote add origin https://github.com/your-username/jmeter-performance-tests.git

# Push your code to GitHub

git push -u origin main

1. 

## Your project now lives on GitHub, ready for automation!

## Step 3: Automate with Jenkins or GitHub Actions

Here are two paths to achieve the same goal. You only need to choose one.

### Path A: Using Jenkins 🤖

Jenkins uses a file called a Jenkinsfile to define the automation pipeline.

1. **Create the Jenkinsfile:** In the root of your project, create a new file named Jenkinsfile (no extension).
2. **Add the Pipeline Script:** Copy the following code into your Jenkinsfile.
3. Groovy
4. 

pipeline {

agent any

tools {

// These must be configured in your Jenkins Global Tool Configuration

maven 'Maven3'

jdk 'JDK17'

}

stages {

stage('Checkout Code') {

steps {

// Clones your repository from GitHub

git url: 'https://github.com/your-username/jmeter-performance-tests.git', branch: 'main'

}

}

stage('Run Performance Tests') {

steps {

// Runs the Maven command

sh 'mvn clean verify'

}

}

}

post {

always {

// This block runs after the stages are complete

echo 'Archiving reports...'

archiveArtifacts artifacts: 'target/jmeter/reports/', fingerprint: true

}

}

}

1. 
2. **Commit and Push:** Add the Jenkinsfile to GitHub.
3. Bash
4. 

git add Jenkinsfile

git commit -m "Add Jenkins pipeline"

git push

1. 
2. **Configure in Jenkins:**
   * Create a new "Pipeline" job in Jenkins.
   * Under the "Pipeline" section, choose **"Pipeline script from SCM"**.
   * Select **"Git"** as the SCM.
   * Enter your repository URL.
   * Click Save.  
     Now, when you run this job in Jenkins, it will check out your code, run the tests, and you will see a link to the "Archived Artifacts" (your HTML report) on the build's page.

### Path B: Using GitHub Actions 🚀

GitHub Actions is built right into your repository. The configuration is done with a YAML file.

1. **Create the Workflow File:** In your project, create a folder path .github/workflows/. Inside the workflows folder, create a new file named run-performance-test.yml.
2. **Add the Workflow Script:** Copy the following YAML into your run-performance-test.yml file.
3. YAML
4. 

name: JMeter Performance Test

on:

push:

branches: [ main ] # Runs every time you push to the main branch

jobs:

run-jmeter-test:

runs-on: ubuntu-latest

steps:

- name: Checkout Code

uses: actions/checkout@v4

- name: Set up JDK 17

uses: actions/setup-java@v4

with:

java-version: '17'

distribution: 'temurin'

- name: Run Maven JMeter Test

run: mvn clean verify

- name: Upload JMeter Report

uses: actions/upload-artifact@v4

with:

name: jmeter-html-report

path: target/jmeter/reports/

1. 
2. **Commit and Push:** Add this new file to GitHub.
3. Bash
4. 

git add .github/

git commit -m "Add GitHub Actions workflow"

git push

1. 

## That's it! GitHub will automatically detect this file and run the workflow. Go to the "Actions" tab in your GitHub repository. You will see the job running. When it's done, you can click on the completed run and download the "jmeter-html-report" artifact.

## Bonus: How to Run a Specific JMX File

By default, the plugin runs all .jmx files found in src/test/jmeter. To run only one specific file, you can override this from the command line using the -D flag.

Let's say you only want to run Test-Scenario-A.jmx. You would use this command:

Bash



mvn verify -Djmeter.testfiles.include="\*\*/Test-Scenario-A.jmx"

This tells the plugin to include only the file that matches the pattern. This is very useful when you have many different test plans in the same project.

Of course. Setting up Jenkins for the first time can seem daunting, but it's a straightforward process. Let's walk through it step-by-step.

We'll use **Docker** to run Jenkins, as it's the easiest and cleanest way to get started without installing Java or other dependencies directly on your machine.

## Phase 1: Install and Run Jenkins with Docker 🐳

This phase gets a fresh, working Jenkins server running on your computer.

#### 1. Install Docker Desktop

If you don't already have it, download and install Docker Desktop for your operating system (Windows/Mac/Linux). This is a one-time setup.

#### 2. Run the Jenkins Docker Command

Open your terminal (PowerShell, Command Prompt, or Terminal on Mac/Linux) and run the following command. This command will download the official Jenkins image (with Java 17 included) and start it.

Bash



docker run -p 8080:8080 -p 50000:50000 -v jenkins\_home:/var/jenkins\_home --name jenkins-lts jenkins/jenkins:lts-jdk17

*   
  **-p 8080:8080:** Connects your local port 8080 to the container's port 8080, so you can access the Jenkins web interface.
* **-v jenkins\_home:/var/jenkins\_home:** Creates a persistent storage volume. This is **very important**—it saves all your Jenkins data even if you stop or restart the container.
* **--name jenkins-lts:** Gives your container a simple name.
* **jenkins/jenkins:lts-jdk17:** Tells Docker to use the official Long-Term Support version of Jenkins that comes with JDK 17.

## Wait for the terminal to output a message similar to "Jenkins is fully up and running".

## Phase 2: First-Time Jenkins Setup

Now that Jenkins is running, you need to configure it through your web browser.

#### 1. Get the Admin Password

* Open your web browser and go to http://localhost:8080.
* Jenkins will ask for an administrator password to unlock it.
* Go back to your terminal and run this command to get the password from inside the Docker container:
* Bash
* 

docker exec jenkins-lts cat /var/jenkins\_home/secrets/initialAdminPassword

* 
* Copy the long string of text and paste it into the "Administrator password" box in your browser, then click **Continue**.

#### 2. Install Plugins

* On the next screen, click the **"Install suggested plugins"** button. This will install all the essential tools we need, like Git and Pipeline.

#### 3. Create Your Admin User

* Once the plugins are installed, you'll be prompted to create your first admin user. Fill in your details (username, password, etc.) and click **"Save and Continue"**.
* On the "Instance Configuration" screen, you can just click "Save and Finish".  
  Your Jenkins server is now set up and ready to use!

## Phase 3: Configure Jenkins Tools 🔧

You need to tell Jenkins where to find Maven so your Jenkinsfile can use it.

1. On the Jenkins dashboard, go to **Manage Jenkins > Tools**.
2. **Configure JDK:** The JDK is already included in the Docker image, but we'll define it for the pipeline.
   * Scroll to **"JDK"** and click **"Add JDK"**.
   * Give it the **Name**: JDK17 (this name must **exactly** match what's in the Jenkinsfile).
   * For the installer, you can leave the defaults.
3. **Configure Maven:**
   * Scroll down to **"Maven"** and click **"Add Maven"**.
   * Give it the **Name**: Maven3 (this must also **exactly** match the name in the Jenkinsfile).
   * Enable **"Install automatically"**.
   * From the "Version" dropdown, choose the latest 3.x.x version (e.g., 3.9.8).
4. Click the **"Save"** button at the bottom of the page.

## Phase 4: Create Your Jenkins Pipeline Job 🚀

This is the final step where you connect your GitHub repository to Jenkins.

1. Go back to the Jenkins dashboard by clicking the logo in the top-left.
2. Click **"New Item"**.
3. Enter an item name, for example, jmeter-test-pipeline.
4. Select **"Pipeline"** as the job type and click **"OK"**.
5. On the configuration page, scroll down to the **"Pipeline"** section.
6. Change the **"Definition"** dropdown from "Pipeline script" to **"Pipeline script from SCM"**.
7. In the SCM section that appears, select **"Git"**.
8. In the **"Repository URL"** field, paste the URL of your GitHub repository (e.g., https://github.com/your-username/jmeter-performance-tests.git).
9. Leave the "Script Path" as Jenkinsfile. Jenkins will automatically look for that file in your repository.
10. Click **"Save"**.

### Run Your Job!

You will be taken to your new pipeline's page. Click "Build Now" on the left menu.

Jenkins will start running the steps defined in your Jenkinsfile: it will check out your code, run mvn clean verify, and archive the HTML report. You can watch the progress in real-time by clicking on the build number and then "Console Output".

When it's done, you'll see a link on the build's page called "Archived Artifacts" where you can access your JMeter dashboard.

Yes, you can absolutely do it without Docker. Automating the job is also a core feature of Jenkins.

## Running Jenkins Without Docker (Local Installation)

Installing Jenkins directly on your operating system is the traditional method. The main difference is you'll be responsible for installing and managing Java and Maven yourself.

### Installation Steps

1. **Install Java:** Jenkins requires Java to run. Download and install a compatible version like JDK 17 from a trusted source like **Adoptium (Temurin)**. After installation, make sure your JAVA\_HOME environment variable is set correctly.
2. **Install Jenkins:** Go to the [official Jenkins download page](https://www.jenkins.io/download/) and choose the package for your system (e.g., the Windows Installer, macOS installer, or Linux packages). Run the installer and follow its instructions.
3. **Install Maven:** Download Apache Maven from its [official website](https://maven.apache.org/download.cgi). Unzip the folder to a permanent location (like C:\Program Files\Apache\Maven). Then, add the bin directory from that folder to your system's PATH environment variable so you can run mvn from any terminal.
4. **Initial Setup:** Once Jenkins is running, the setup process is the same as before. You'll access it at http://localhost:8080, get the initial password, install suggested plugins, and create an admin user.

### Tool Configuration (The Main Difference)

In Manage Jenkins > Tools, your configuration will be slightly different. Instead of "Install automatically," you will tell Jenkins where you installed your tools.

* **JDK:** Click "Add JDK," give it the name JDK17, uncheck "Install automatically," and provide the path to your JAVA\_HOME directory.
* **Maven:** Click "Add Maven," give it the name Maven3, uncheck "Install automatically," and provide the path where you unzipped Maven.

After that, creating and running the pipeline job is exactly the same.

## Automating the Build at 7:00 AM ⏰

You can schedule your Jenkins job to run at a specific time using a feature called "Build Triggers." This is done using Cron syntax, which is a standard way to define time-based schedules.

The format uses five fields separated by spaces:

MINUTE HOUR DAY-OF-MONTH MONTH DAY-OF-WEEK

To run a job every day at 7:00 AM, the correct Cron expression is:

0 7 \* \* \*

* 0: At the 0th minute of the hour.
* 7: At the 7th hour of the day (7 AM).
* \*: On every day of the month.
* \*: In every month.
* \*: On every day of the week.  
  Note: The schedule runs based on your Jenkins server's time zone, which for you will be India Standard Time (IST).

## Where to Put the Schedule

You can configure this schedule in two places, but the easiest way is through the Jenkins user interface.

### In the Jenkins Job Configuration (Recommended for Beginners)

1. Go to your Jenkins dashboard and click on your pipeline job (jmeter-test-pipeline).
2. Click on **"Configure"** in the left-hand menu.
3. Go to the **"Build Triggers"** section.
4. Check the box that says **"Build periodically"**.
5. An input box labeled **"Schedule"** will appear. Paste your Cron string into this box: 0 7 \* \* \*
6. Click **"Save"**.

That's it! Jenkins will now automatically start a new build of your job every single day at 7:00 AM.

### In the Jenkinsfile (Alternative "as-code" method)

You can also define the schedule directly within your Jenkinsfile. This is considered a best practice for advanced users as it keeps all configuration in your code. To do this, you would add a triggers block to your pipeline script:

Groovy



pipeline {

agent any

triggers {

// This is where you define the schedule

cron('0 7 \* \* \*')

}

tools {

maven 'Maven3'

jdk 'JDK17'

}

// ... rest of your stages ...

}



Of course. The "path not acceptable" error is very common and it almost always means that Jenkins can't find what it's looking for in the directory you've provided. Jenkins is looking for a very specific folder structure.

Let's break down exactly how to find the correct paths and what Jenkins expects for each.

## 📁 Configuring the JDK Path

Jenkins needs the main home directory of your Java Development Kit (JDK), not the bin folder inside it.

#### Step 1: Find Your JAVA\_HOME Path

This is the top-level folder of your JDK installation. It's the directory that contains the bin, lib, and other subfolders.

* **On Windows**, it will typically be a path like:
  + C:\Program Files\Java\jdk-17
  + C:\Program Files\Eclipse Adoptium\jdk-17.0.8.7-hotspot
* **On macOS**, it's often:
  + /Library/Java/JavaVirtualMachines/temurin-17.jdk/Contents/Home

**✅ How to Verify:** Open this path in your file explorer. You should see folders named **bin**, **lib**, **conf**, etc., directly inside it. If you see those, you have the correct path.

#### Step 2: Enter the Path in Jenkins

1. Navigate to **Manage Jenkins > Tools**.
2. Find the **JDK** section and click **Add JDK**.
3. Set the **Name** to JDK17 (to match your Jenkinsfile).
4. **Uncheck** the box for "Install automatically".
5. In the **JAVA\_HOME** field, paste the full, verified path you found in Step 1.

If the path is correct, the red error message next to the input box will disappear.

## 📁 Configuring the Maven Path

This is the exact same principle. Jenkins needs the main home directory of your Maven installation.

#### Step 1: Find Your MAVEN\_HOME Path

This is the top-level folder you created when you unzipped Maven.

* **On Windows**, this will be a path like:
  + C:\Program Files\apache-maven-3.9.8
  + C:\dev\tools\apache-maven-3.9.8
* **On Linux/macOS**, it might be:
  + /opt/apache-maven-3.9.8

**✅ How to Verify:** Open this path in your file explorer. You must see folders named **bin**, **lib**, and **conf** directly inside it.

#### Step 2: Enter the Path in Jenkins

1. In the same **Manage Jenkins > Tools** screen, find the **Maven** section and click **Add Maven**.
2. Set the **Name** to Maven3 (to match your Jenkinsfile).
3. **Uncheck** the box for "Install automatically".
4. In the **MAVEN\_HOME** field, paste the full, verified path you found in Step 1.

Again, the error message should disappear if the path is correct.

## 🕵️ Troubleshooting: Why Your Paths Are "Not Acceptable"

If you still see an error, it's almost certainly one of these reasons:

1. **❌ You're using the bin folder:** This is the most common mistake. Do not include \bin at the end of your path.
   * **Incorrect:** C:\Program Files\Java\jdk-17\bin
   * **Correct:** C:\Program Files\Java\jdk-17
2. **Permissions Issue:** The user that runs the Jenkins service may not have permission to access the C:\Program Files directory.
   * **Quick Solution:** Try moving your JDK and Maven installations to a more accessible path, like C:\tools\jdk-17 and C:\tools\maven, and use those paths in the Jenkins configuration instead.
3. **A Typo in the Path:** Double-check for any typos or extra spaces. It's best to copy the path directly from your file explorer's address bar.

You've found the exact problem! The error is happening because you have included the **\bin** folder in your path.

Jenkins requires the main JAVA\_HOME directory, which is the folder that *contains* the bin folder, not the bin folder itself.

### The Fix 🔧

It's a very simple correction.

1. Go to the **JAVA\_HOME** field in your Jenkins configuration.
2. Delete \bin from the end of the path.

* **Incorrect Path:** ❌ C:\Program Files\Java\jdk-21\bin
* **Correct Path:** ✅ C:\Program Files\Java\jdk-21

Once you remove \bin, the orange warning message will disappear, confirming that Jenkins accepts the path. After that, just click **Save**.

This new error is happening because your Jenkins is running inside a Docker container, but you're giving it a file path from your Windows computer. The Docker container has its own isolated file system and cannot see your C: drive.

Think of Docker as a separate, sealed-off room (the container) inside your house (your Windows PC). Jenkins is inside that room and has no idea what's outside of it. The error message is correct: the path C:\Program Files\Java\jdk-21 does not exist *inside the container*.

## The Solution: Let Jenkins Install Tools Automatically

The easiest and best way to solve this when using Docker is to let Jenkins download and manage the tools for you *inside* the container.

1. Go back to the **Manage Jenkins > Tools** page.
2. Delete the JDK entry you've been trying to configure by clicking the red 'X' icon.
3. Click **Add JDK** again.
4. Set the **Name** to JDK17. (It's best to use JDK17 to match the Jenkinsfile and the Docker image we are using).
5. This time, **check the box for "Install automatically"**.
6. A new section will appear. From the "Add Installer" dropdown, select **"Install from Adoptium.net"** and choose a version of JDK 17.
7. Do the same thing for Maven. Make sure the name is Maven3, check **"Install automatically"**, and choose a version.
8. Click **Save**.

By doing this, you are telling Jenkins to download and manage its own copies of the JDK and Maven inside its own private storage within the Docker container. This completely avoids the problem of trying to access your local Windows files.

After saving this new configuration, your Jenkins job will run correctly. ✅

Excellent questions. Let's clear this up.

Yes, using **JDK 21 is perfectly fine**. Both JDK 17 and JDK 21 are Long-Term Support (LTS) versions and work great. The most important thing is to be consistent.

The installer options you see are the advanced ones. We need to install a simple plugin to get a much easier "download and install" option.

## JDK 21 vs. JDK 17

Since you have Java 21, let's stick with it.

1. **In Jenkins Tools:** Name your JDK installation **JDK21** as you've already done.
2. **In your Jenkinsfile:** You must update the tools section to match this name.
3. Groovy
4. 

// Jenkinsfile

pipeline {

agent any

tools {

maven 'Maven3'

jdk 'JDK21' // Make sure this name matches exactly

}

//... rest of file

}

1. 

## Choosing the Right Installer (By Adding a Plugin)

The easy installer options aren't there by default. We'll add a plugin to get them.

### 1. Install the Temurin Installer Plugin

1. On your Jenkins dashboard, go to **Manage Jenkins > Plugins**.
2. Click on the **"Available plugins"** tab.
3. In the search box on the right, type Temurin.
4. You will see a plugin named **"Temurin Tool Installer"**. Check the box next to it.
5. Click the **"Install"** button. Jenkins will download and install it.

### 2. Configure the JDK with the New Installer

1. Go back to **Manage Jenkins > Tools**.
2. Click on **Add JDK** (or configure your existing JDK21 entry).
3. Set the **Name** to JDK21.
4. Check the box for **"Install automatically"**.
5. Click the **"Add Installer"** dropdown. You will now see a new, much better option.
6. Select "Install from Adoptium.net".  
   7. A Version dropdown will appear. Select a recent version of OpenJDK 21.
7. Finally, click **Save**.

### What to Do Now

1. Make the change to your **Jenkinsfile** (updating JDK17 to JDK21) and push it to your GitHub repository.
2. Run the "Build Now" button in Jenkins.  
   Your pipeline will now automatically download JDK 21 and run your tests successfully. ✅

Perfect, you've found the correct plugin. Now you just need to install it. ✅

## Next Steps

1. **Check the box** next to "Eclipse Temurin installer".
2. Click the blue **Install** button at the top right.
3. Jenkins will show you an installation progress page. Wait for it to complete.

## After Installation

Once the plugin is installed, go back to **Manage Jenkins > Tools**.

Now, when you configure your JDK and click **"Add Installer"**, you will see the new **"Install from Adoptium.net"** option available to select. 👉

Your setup is perfect! The schedule you've set in your screenshot will work exactly as expected.

The schedule itself doesn't run the .jmx file directly. Instead, at 3:30 PM every day, it simply **triggers your Jenkins job to start**. The job then follows the instructions you've already given it in your Jenkinsfile.

### How It All Connects

Here's the step-by-step process that will happen automatically at your scheduled time:

1. The Trigger Fires ⏰  
   At 3:30 PM, the "Build periodically" scheduler tells your jmeter-test-pipeline job, "It's time to run!"
2. Jenkins Reads the Pipeline 📜  
   The job looks at its configuration, which tells it to get the Jenkinsfile from your GitHub repository.
3. The Jenkinsfile Gives the Command ⚙️  
   Jenkins reads the Jenkinsfile, which contains the command sh 'mvn clean verify'.
4. Maven Executes the Test 🚀  
   Maven starts, reads your pom.xml, and activates the jmeter-maven-plugin. This plugin is configured to automatically find and run all the .jmx test plans in your src/test/jmeter/ folder.

So, the schedule is just the starting pistol. The real work of running your test plan is handled by the **mvn clean verify** command inside your Jenkinsfile.

As your screenshot correctly shows, Jenkins understands your schedule and confirms the next run time. If you don't want to wait, you can always click the **"Build Now"** button on the job's page to run the entire process manually.