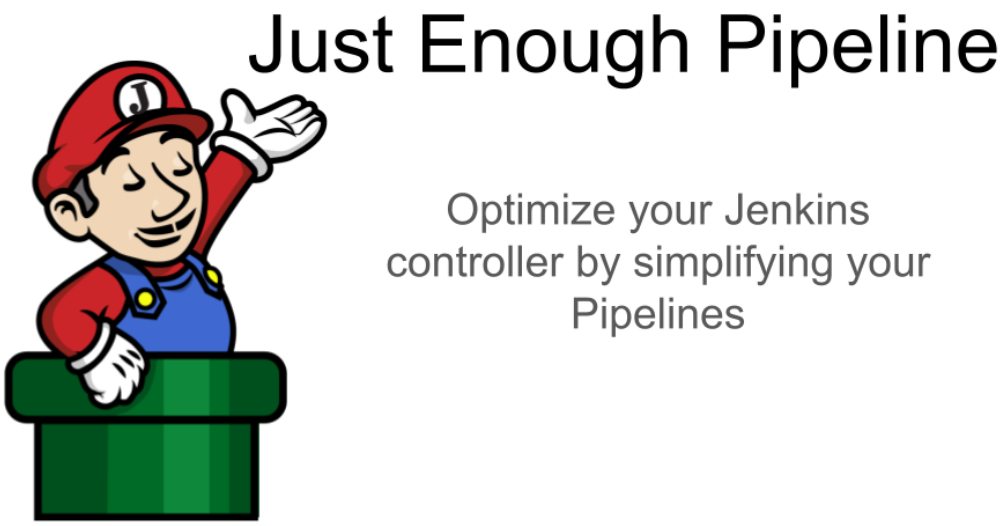
Use Just Enough Pipeline



- Jenkins Pipeline (or simply Pipeline with a capital P) is a suite of plugins that supports implementing and integrating continuous delivery pipelines into Jenkins.
- This allows you to automate the process of getting software from version control through to your users and customers.
- Pipeline code works beautifully for its intended role of automating build, test, deploy, and administration tasks.
- But, as it is pressed into more complex roles and unexpected uses, some users have run into snags.
- Using best practices – and avoiding common mistakes – can help you design a pipeline that is more robust, scalable, and high-performing.
- We see a lot of users making basic mistakes that can sabotage their pipeline. (Yes, you can sabotage yourself when you are creating a pipeline.)
- In fact, it is easy to spot someone who is going down this dangerous path and it is usually because they do not understand some key technical concepts about Pipeline.
- This invariably leads to scalability mistakes that you will pay dearly for down the line.

Do Not make this mistake!

- Perhaps the biggest misstep people make is deciding that they need to write their entire pipeline in a programming language.
- After all, Pipeline is a domain specific language (DSL). However, that does not mean that it is a general-purpose programming language.
- If you treat the DSL as a general-purpose programming language, you are making a serious architectural blunder by doing the wrong work in the wrong place.
- Remember that the core of Pipeline code runs on the controller.
- So, you should be mindful that everything you express in the Pipeline domain specific language (DSL) will compete with every other Jenkins job running on the controller.
- For example, it's easy to include a lot of conditionals, flow control logic, and requests using **scripted syntax in the pipeline job**.
- Experience tells us this is not a good idea and can result in serious damage to pipeline performance.
- We have seen organizations with poorly written Pipeline jobs bring a controller to its knees, while only running a few concurrent builds (Parallel Builds) .
- Jenkins allows for parallel execution of builds for a Job. Job configuration page has a check box, "Execute concurrent builds if necessary". Also, in the master node configuration set the "# of executors" field to more than 1. Once these two are done, parallel job execution is enabled.
- Wait a minute, you might ask, "Isn't handling code what the controller is there for?" Yes, the controller certainly is there to execute pipelines. But it is much better to assign individual steps of the pipeline to command line calls that execute on an agent.
- So, instead of running a lot of conditionals inside the pipeline DSL, it is better to put those conditionals inside a shell script or batch file and call that script from the pipeline.
- However, this raises another question: "What if I do not have any agents connected to my controller?" If this is the case, then you have just made another bad mistake in scaling Jenkins pipelines.
- Why? Because the first rule of building an effective pipeline is to make sure you use agents. If you are using a Jenkins controller and have not defined any agents, then your first step should be to define at least one agent and use that agent instead of executing on the controller.
- For the sake of maintaining scalability in your pipeline, the general rule is to avoid processing any workload on your controller.

- If you are running Jenkins jobs on the controller, you are sacrificing controller performance.
- So, try to avoid using Jenkins's controller capacity for things that should be passed off to an agent.
- Then, as you grow and develop, all your work should be running agents.
- Therefore, we always recommend setting the number of executors on the master to zero and you know why.

Use Just Enough Pipeline to Keep Your Pipeline Scalable

- All of this serves to highlight our overarching theme of "using just enough pipeline."
- Simply put, you want to use enough code to connect the pipeline steps and integrate tools but no more than that.
- Limit the amount of complex logic embedded in the Pipeline itself (similarly to a shell script) and avoid treating it as a general-purpose programming language.
- This makes the pipeline easier to maintain, protects against bugs, and reduces the load on controllers.
- Another best practice for keeping your pipeline lean, fast, and scalable is to use declarative syntax instead of scripted syntax for your Pipeline.
- Declarative naturally leads you away from the kinds of mistakes I just described. It is a simpler expression of code and an easier way to define your job.
- It is computed at the startup of the pipeline instead of executing continually during the pipeline.
- Therefore, when creating a pipeline, start with declarative, and keep it simple for as long as possible.
- Anytime a script block shows up inside of a declarative pipeline, you should extract that block and put it in a shared library step.
- That keeps the declarative pipeline clean. By combining declarative with a shared library, that will take care of most use cases you will encounter.
- That said, it is not accurate to say that declarative plus a shared library will solve every problem.
- There are cases where scripted is the right solution (great when executing the build process with maven and you want to download the all the downloads from the https://mvnrepository.com/).

- However, declarative is a great starting point until you discover that you absolutely must use scripted.
- Just remember, at the end of the day, you will do well to follow the adage: “Use just enough pipeline and no more.”