The Need For Speed: Building Pipelines To Be Faster

Sam Van Oort, CloudBees Inc.
Where is our journey going?

• **Background**: What’s the problem and why do I care?

• **Tooling**: What can I use to look at pipeline performance?

• **Best practices**: How can I use pipeline more effectively?
Background
What Does “Fast” Really Mean: Throughput?

- **Throughput**: solved by scale-out or separation of concerns
  - Distributed builds: many build agents (slaves) per master
  - Multiple masters (one per team)
What Does “Fast” Really Mean: Resource Use?

• Resource Use:
  • AWS c4.xlarge: 4 vCPU, 7.5 GB RAM
    • ~$153/month On-demand ($95 reserved)
  • Each may support dozens of engineers
  • Software Engineer: (Just salary + vacation)
    • ~$3000-17000/month* (plus benefits!)
  • Conclusion: Commodity Hardware is CHEAP!

*Rough figures for US/EU engineers across geos, junior to principal level
What Does “Fast” Really Mean: LATENCY!

- **Latency**: the King!
  - Low turn around time = ship FASTER
  - Low turn around = less context switching for engineers
  - Context switching = lost time & mistakes
  - Staff time >> CPU time, so...
- YES, we have the answer!
What Are The Components Of Latency For Jenkins Pipelines?

1. **Triggering delay**: time from commit until a build is enqueued
2. **On-master overheads**: orchestration and tracking
3. **Queueing time**: waiting for an executor slot
4. **Executor time**: how long it takes to build, test, deploy
5. **Feedback delay**: time until someone that cares sees the key result (pass/fail)
Let’s Get The Basics Out Of The Way: Triggering and Master

• **Triggering delay:**
  • Use web hooks or commit hooks: faster than polling, easier on Jenkins and the SCM
    • Everyone loves “GitHub API rate limit exceeded”
  • Short polling cycles can give a fast response time, but dramatically increase resource use (see also: CloudBees support ticket history)

• **On-master overhead:**
  • Delete old build records
  • Don’t give masters any executors
  • Don’t dump GBs of data to logs (should go without saying, but I’ve seen it)
Let’s Get The Basics Out Of The Way: Executor Use

• Queueing Time
  • You must construct additional build agents (slaves)
  • Dynamic agents are an easy solution: cloud agents, Docker agents, etc

• Feedback Delay
  • Limit the spam! Only the culprits.
  • Make it meaningful, failure or prod
  • Use better systems: IM not email

We must create additional build agents!
Let’s Get The Basics Out Of The Way

• **Triggering delay:**
  - Use web hooks or commit hooks: faster than polling, easier on Jenkins and the SCM
  - Short polling cycles can give a fast response time, but dramatically increase resource use (see also: CloudBees support ticket history)

• **On-master overhead:**
  - Delete old build records
  - Don’t give masters any executors
  - Don’t dump GBs of data to logs (should go without saying, but I’ve seen it)

• **Queueing Time:**
  - You must construct additional build agents (slaves)
  - Dynamic slaves are an easy solution: cloud agents, Docker agents, etc

• **Notifications**
  - Limit the spam (people will ignore it), use IM not email
### Analysis, the Top Level: Pipeline Stage View

**Stage View**

<table>
<thead>
<tr>
<th>Stage Description</th>
<th>Time 1</th>
<th>Time 2</th>
<th>Time 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple on-master echo</td>
<td>4ms</td>
<td>818ms</td>
<td>285ms</td>
</tr>
<tr>
<td>Run local echo from shell</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Run local echo with a single shell script</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Average stage times:

- 4ms
- 818ms
- 285ms

```java
stage('simple on-master echo') {
    for(int i=0; i<3; i++) {
        echo 'printing simple message'
    }
}

stage('Run local echo from shell') {
    node {
        for(int i=0; i<3; i++) {
            sh 'echo "running shell"'
        }
    }
}

stage('Run local echo with a single shell script') {
    node {
        sh 'for i in {1..3}; do echo "printing $i"; done'
    }
}
```
# New: Pipeline Steps View As a Profiler

## Pipeline Steps

<table>
<thead>
<tr>
<th>Step</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start of Pipeline</td>
<td><img src="image" alt="Status" /></td>
</tr>
<tr>
<td>Stage: Start - (0.24 ms in block)</td>
<td><img src="image" alt="Status" /></td>
</tr>
<tr>
<td>simple-no-master-echo - (1 ms in block)</td>
<td><img src="image" alt="Status" /></td>
</tr>
<tr>
<td>Print Message</td>
<td><img src="image" alt="Status" /></td>
</tr>
<tr>
<td>Print Message - (1 ms in self)</td>
<td><img src="image" alt="Status" /></td>
</tr>
<tr>
<td>Print Message</td>
<td><img src="image" alt="Status" /></td>
</tr>
<tr>
<td>Stage: Start - (0.82 sec in block)</td>
<td><img src="image" alt="Status" /></td>
</tr>
<tr>
<td>Run local echo from shell - (0.81 sec in block)</td>
<td><img src="image" alt="Status" /></td>
</tr>
<tr>
<td>Allocate node - Start - (0.81 sec in block)</td>
<td><img src="image" alt="Status" /></td>
</tr>
<tr>
<td>Allocate node - Body - Start - (0.8 sec in block)</td>
<td><img src="image" alt="Status" /></td>
</tr>
<tr>
<td>Shell Script - (0.25 sec in self)</td>
<td><img src="image" alt="Status" /></td>
</tr>
<tr>
<td>Shell Script - (0.25 sec in self)</td>
<td><img src="image" alt="Status" /></td>
</tr>
<tr>
<td>Shell Script - (0.25 sec in self)</td>
<td><img src="image" alt="Status" /></td>
</tr>
<tr>
<td>Stage: Start - (0.28 sec in block)</td>
<td><img src="image" alt="Status" /></td>
</tr>
<tr>
<td>Run local echo with a single shell script - (0.28 sec in block)</td>
<td><img src="image" alt="Status" /></td>
</tr>
<tr>
<td>Allocate node - Start - (0.28 sec in block)</td>
<td><img src="image" alt="Status" /></td>
</tr>
<tr>
<td>Allocate node - Body - Start - (0.27 sec in block)</td>
<td><img src="image" alt="Status" /></td>
</tr>
<tr>
<td>Shell Script - (0.26 sec in self)</td>
<td><img src="image" alt="Status" /></td>
</tr>
</tbody>
</table>
Alternative Approach: Blue Ocean
How Long Does Pipeline Take? (Roughly)

- Hardware: modern AWS instance types with EBS storage (SSD)
- Standard modern AWS instances (with EBS SSD storage) - may improve over time!

<table>
<thead>
<tr>
<th>Step Description</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Echo Step (trivial)</td>
<td>1ms</td>
</tr>
<tr>
<td>node step: Obtain a workspace</td>
<td>20ms</td>
</tr>
<tr>
<td>Run a shell step</td>
<td>250ms</td>
</tr>
<tr>
<td>Quick build (1 min)</td>
<td>60,000ms</td>
</tr>
</tbody>
</table>

- Standard modern AWS instances (with EBS SSD storage) - may improve over time!
Best Practices
What should I do?
Hardcore Antipatterns That Can Breaking The World

- Input step that locks up an executor

```java
stage ('make pipeline developers cry') {
    node {
        sh 'mvn clean install'
        input 'can we do something already?'
    }
}
```

- Godot won’t show up: but angry coworkers will

```java
stage ('Why the pain?!?!') {
    waitUntil {
        node {
            try {
                sh './waitForGodot.sh'
            } catch (Exception fail) {
                return false;
            }
        }
    }

    // Deploy or publish
}
```
We get it, things fail - network requests, downtime, slow processes
Computers are dumb: they don’t know when to stop
Pipelines persist state: unbounded loops are like leaving your garden hose on
Best practice within a node{ } block

Better Practice: Bounded Loops

```
node('deployer') {
  try {
    retry (3) {
      deploy(serverName);
    }
  } catch (hudson.AbortException ex) {
    rollbackDeploy(serverName);
    throw new Exception("Deploy failed on $serverName", ex);
  }
}
```
Better Practice: Timeouts

- Subtler version of retry case
- Are you deploying? Are you doing network calls?
  - You need a timeout somewhere.
  - Yes, really.
- Lets you safely recover from hangups
- Yes, you can AND SHOULD mix with retries for critical bits
#1 Biggest Time Saver: Effective Use Of Parallel
#2 Biggest Time Saver: Effective Notifications

```groovy
// Notify with each failure in the parallel branches, but still get the full test results
Closure wrapTest(String testName, Closure test) {
    try {
        test.call()
    } catch (Exception ex) {
        // Basic example, you might include a link, or first line of stack trace, send email, etc
        String failure = "Build FAILED in test \$testName - \${env.JOB_NAME} \${env.BUILD_NUMBER}"
        hipchatSend notify:true, message: failure
    }
    throw ex;
}

def testBranches = ['failFast':false]
testBranches['linux'] = wrapTest('linux', {
    node('linux') {
        git repoName
        sh 'make build test'
    }
})
testBranches['windows'] = wrapTest('windows') {
    node('windows') {
        git repoName
        bat 'build.bat test'
    }
}
parallel testBranches
```
Optimization: Consolidate, Consolidate, Consolidate!

• Node blocks:
  • Giving up a workspace lease means someone else might snatch it!

• Shell/batch steps
  • Remember that ~0.25s overhead for each shell step? Consolidate!

• Complex processing logic (XML parsing etc):
  • CPS has some significant overheads for tracking all the things
  • Use @NonCPS functions for more complex processing w/ no steps
  • Next step: use a helper script for processing
Conclusions

• Focus on **latency** (turn around time)
  • Prioritizes what matters (results and technical labor) over what doesn’t (CPU time)
  • Easier to measure, easier to use

• **Tools:**
  • Stage View → Blue Ocean for top level
  • Pipeline steps for specifics
    • Pipeline step view: step and block level
    • Blue Ocean and Stage View* for per-step stats

• **Best Practices:**
  • Use parallel right, use notifications early and often
  • Don’t block things: input outside of workspace, retry, timeout
Thank you, and I hope everyone enjoyed their time at Jenkins World!

Sam Van Oort, CloudBees, Inc