



HPC USER SERVICES • TRAINING SERIES

Basic Shell Scripting

From typing commands to writing your own automation.

Siva Prasad Kasetti | HPC User Services | 23 September 2026

Louisiana State University • Baton Rouge • sys-help@loni.org

```
$ whoami
kasetti
$ echo $SHELL
/bin/bash
$ |
```



HPC User Environment 1

- Intro to HPC
- Getting started
- Into the cluster
- Software environment (modules)



HPC User Environment 2

- Basic concepts
- Preparing your job
- Submitting your job
- Managing your jobs

Prior knowledge you need before running jobs → How to actually run your jobs



1. Introduction

01

What is Shell, and what is it (and isn't) good for



2. Basic Knowledge

02

Interactive vs Non-interactive (Shell Script)
Basic Commands & Syntax, Variables
Arrays, Arithmetic Operations



3. Beyond Basics

03

Subshells, flow control, and text processing



4. Bonus

04

Where to get help — including generative AI



<https://www.hpc.lsu.edu/static/training/weekly-materials/Downloads/ShellScripting.zip>

Download, then unzip on the cluster — every example we walk through today lives in this archive.

Tip: `unzip ShellScripting.zip && cd ShellScripting`

Previously in HPC User Environment 2, you met two ways to run work on the cluster:



Interactive Job

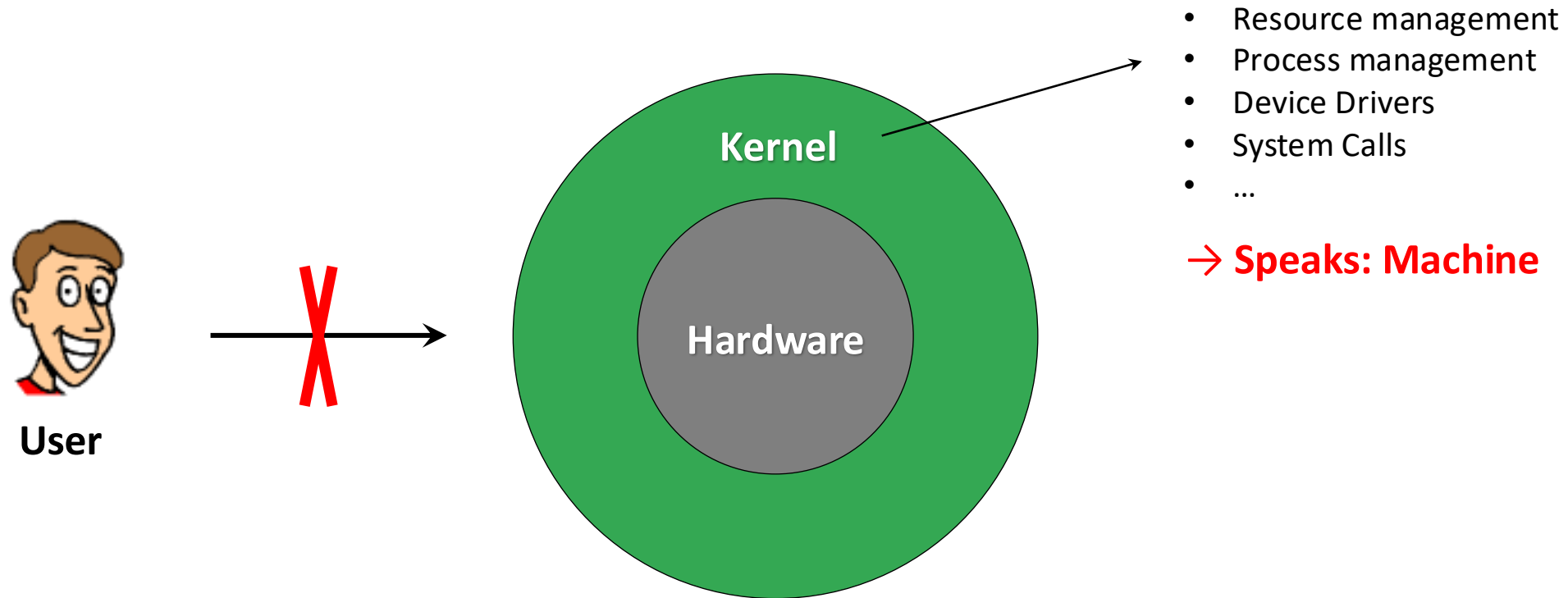
You log in and type commands yourself, in real time.

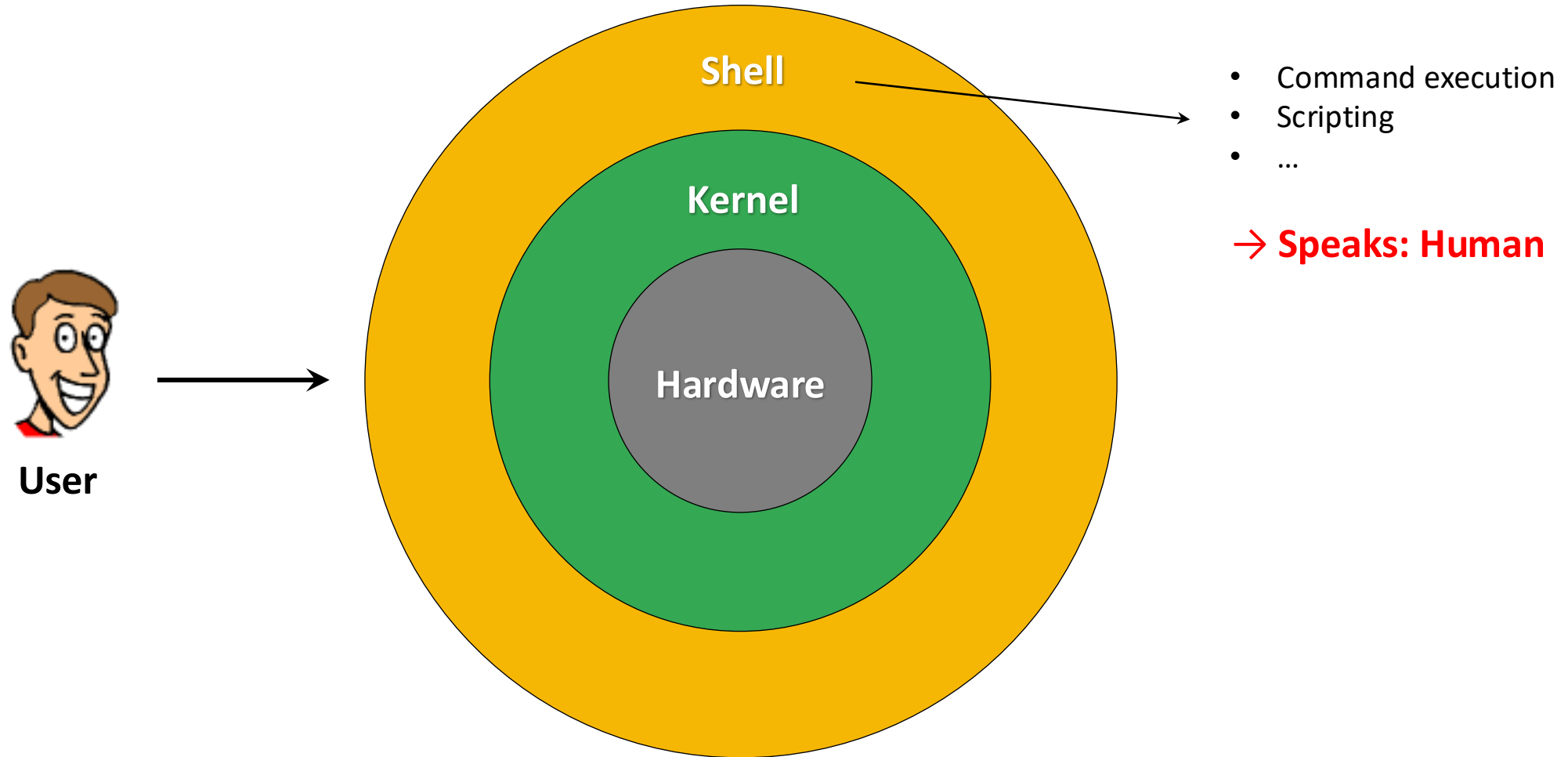


Batch Job

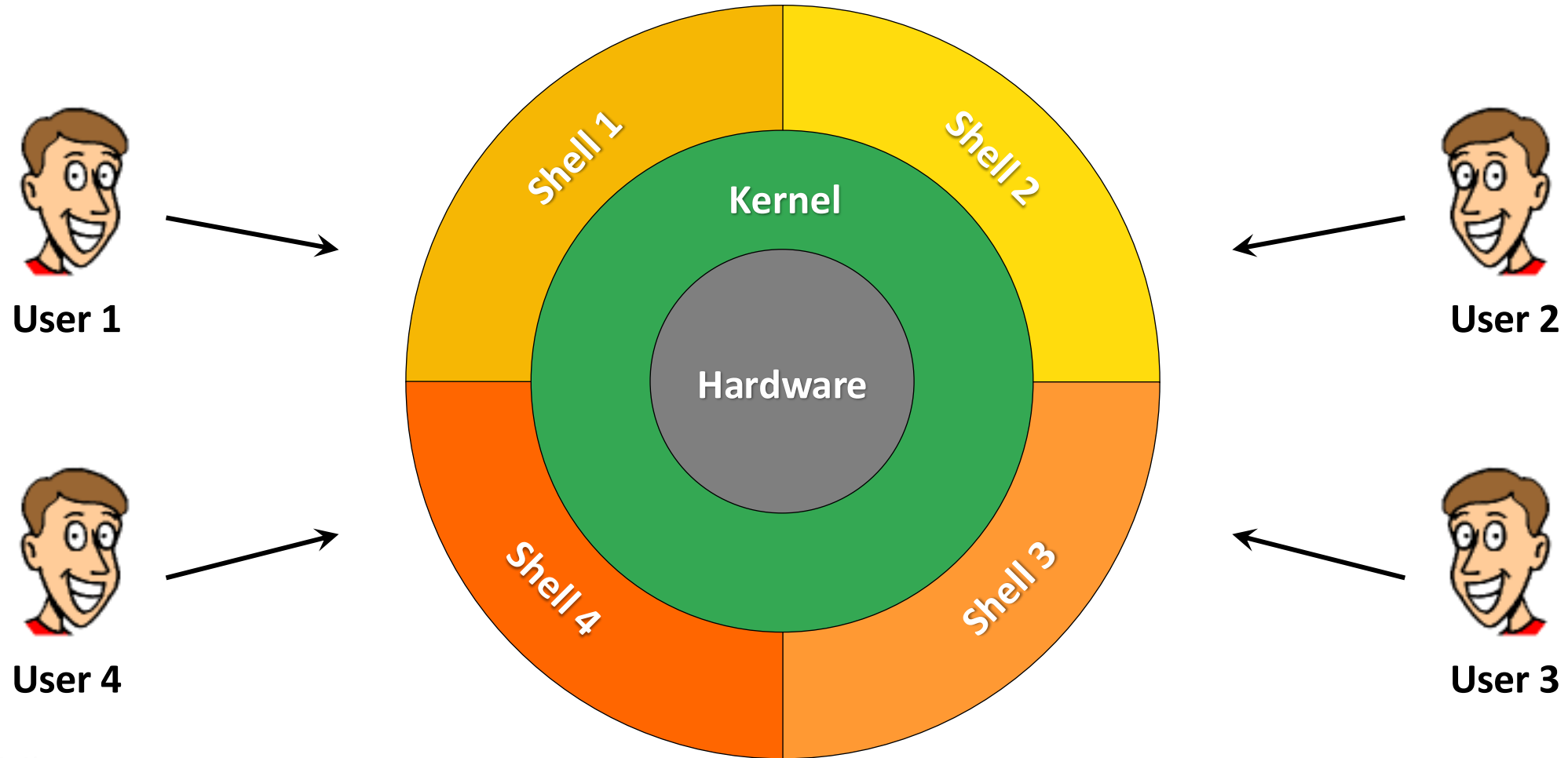
You submit a script and Slurm runs it for you, unattended.

In both cases, you're reaching a Linux system through Shell.





Scenario 1: Multiple Shells, One Kernel





Every user gets their own Shell session

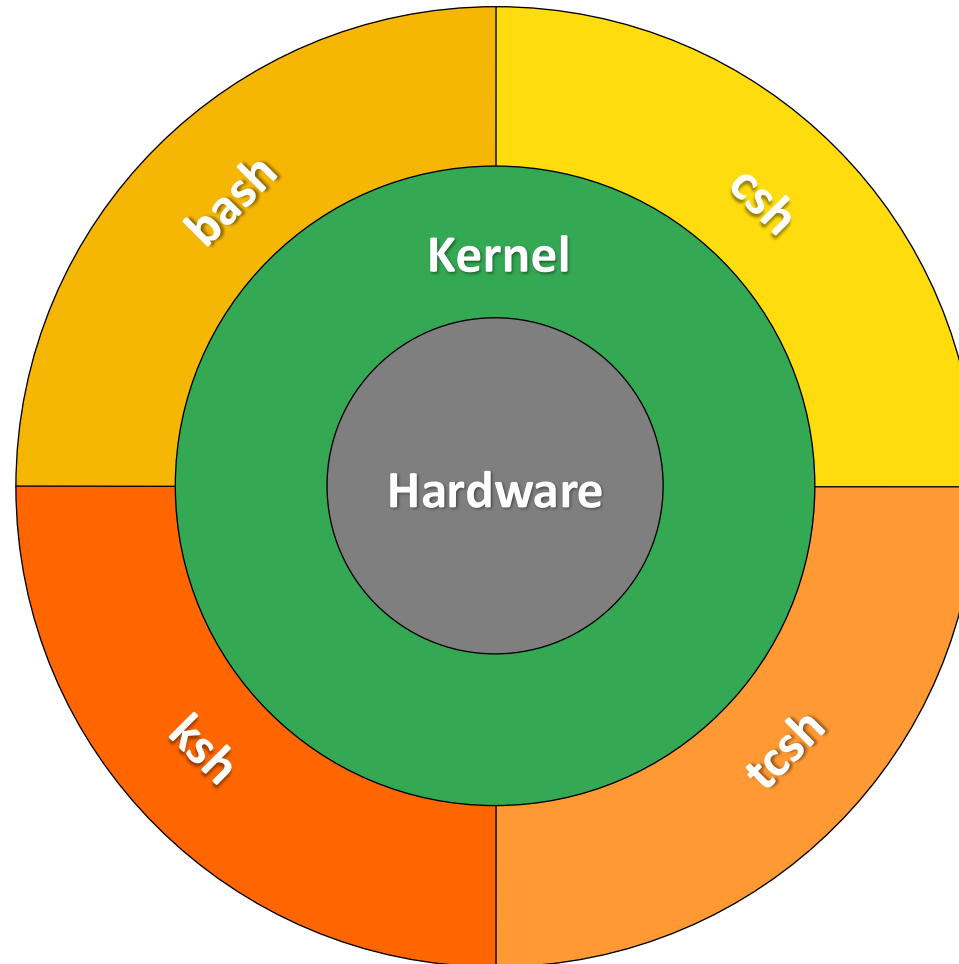
- One kernel manages the whole machine
- Each login (or job) spins up its own Shell
- A Shell can even launch another Shell inside it — a subshell (more on this in Module 3)

Working Definition

“Shell”

A user interface to access UNIX-like systems (e.g., Linux) by executing commands.

That's it — everything today builds on this one idea.



Most of these are supported on our clusters (except in red) — use whichever you like and set your own default.

sh

Original Bourne Shell

/bin/sh

bash

Bourne Again Shell

/bin/bash

csh

C Shell

/bin/csh

tcsh

TENEX C Shell

/bin/tcsh

ksh

KornShell

/bin/ksh

zsh

Z Shell — macOS default

/bin/zsh

dash

Debian Almquist Shell

/bin/dash

fish

Friendly Interactive Shell

/bin/fish



bash is the default Shell on all LSU/LONI clusters — that's what we'll use for every example today.

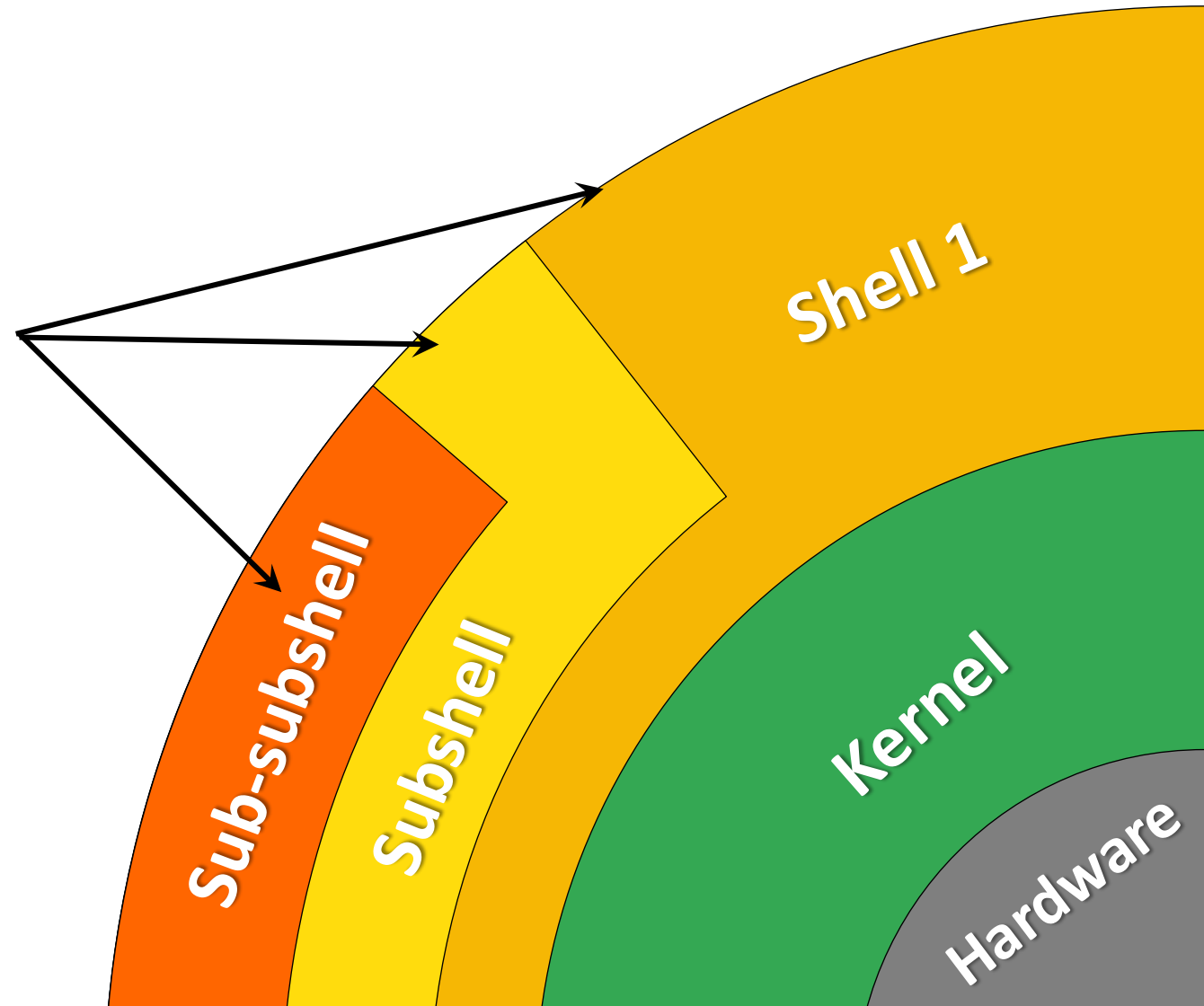


A Shell can launch another Shell inside it.

We'll see the details later — for now, just remember: you can have Shells within Shells.



User 1



Introduction

- What's Shell?
- ▶ **What Shell Offers You?**

Basic Knowledge

- Interactive vs Non-interactive (Shell Script)
- Basic Commands & Syntax
- Variables
- Arrays
- Arithmetic Operations

Beyond Basics

- Subshells
- Flow Control
- Advanced Text Processing Commands

Bonus

- Where to Get Help



Typing commands one by one

```
$ cd /work/$USER  
$ ls -l  
$ queue -u $USER
```

Quick, simple, one task at a time.



...or a full automated script

```
#!/bin/bash  
urls=("a.gz" "b.gz" "c.gz")  
for f in "${urls[@]}"; do  
    wget "$f" &  
done  
wait  
echo "All downloads complete!"
```

download.sh — a realistic mini-workflow



What Shell Offers You

Shell Scripting: a practice to automate tasks with Shell commands.

\$x

Variables

Store and reuse values,
paths, and command output

{ }

Control Structures

if / elif / else for
branching logic

↻

Loops

for, while — iterate over
files, arrays, sequences

**>_
_**

Commands

Run any Linux command,
pipe and redirect output

f ()

Functions

Reusable command blocks
to keep scripts clean

≈

Text Processing

grep, sed — search and
transform files at scale

⚡

Automation

Chain it all together to
automate your HPC workflow

\$x

Variables

Store and reuse values,
paths, and command output

{ }

Control Structures

if / elif / else for
branching logic

↻

Loops

for, while — iterate over
files, arrays, sequences

**>_
_**

Commands

Run any Linux command,
pipe and redirect output

All of these work in both interactive terminals and batch job scripts.

f ()

Functions

Reusable command blocks
to keep scripts clean

≈

Text Processing

grep, sed — search and
transform files at scale

⚡

Automation

Chain it all together to
automate your HPC workflow

Shebang

```
#!/bin/bash
```

Commands

```
# — Variables —  
RESULTS_DIR="./outputs"  
REPORT="summary.txt"
```

Variables

```
# — Commands —  
echo "Started: $(date)"  
mkdir -p "$RESULTS_DIR"
```

Loops & Control Structures

```
# — Function —  
check_job() {  
    grep -q "RESULT:" "$1" && return 0 || return 1  
}
```

Functions

```
# — Loop + Control Structure —  
passed=0  
for file in "$RESULTS_DIR"/job_*.out; do  
    if check_job "$file"; then  
        echo "[PASS] $file"  
        passed=$((passed + 1))  
    else  
        echo "[FAIL] $file"  
    fi  
done
```

```
# — Text Processing —  
sed -i "s|/scratch/old|/work/project|g" "$RESULTS_DIR"/*.out  
grep "RESULT:" "$RESULTS_DIR"/*.out > "$REPORT"
```

Grep & sed

```
# — Automation —  
echo "$passed jobs passed. Report saved to $REPORT"
```

Shebang

```
#!/bin/bash
```

Commands

```
# — Variables —  
RESULTS_DIR="./outputs"  
REPORT="summary.txt"
```

Variables

```
# — Commands —  
echo "Started: $(date)"  
mkdir -p "$RESULTS_DIR"
```

```
# — Function —  
check_job() {  
    grep -q "RESULT:" "$1" && return 0 || return 1  
}
```

Functions

Loops & Control Structures

```
    echo "[PASS] $file"  
    passed=$((passed + 1))  
else  
    echo "[FAIL] $file"  
fi  
done
```

```
# — Text Processing —  
sed -i "s|/scratch/old|/work/project|g" "$RESULTS_DIR"/*.out  
grep "RESULT:" "$RESULTS_DIR"/*.out > "$REPORT"
```

Grep & sed

```
# — Automation —  
echo "$passed jobs passed. Report saved to $REPORT"
```

Isn't it basically a programming language?

Question	Why would I need...	...if I can just use
Everyday glue work	Shell	Python / C++ / Fortran?
Heavy computation	Python / C++ / Fortran	Shell?



a) Why Shell over another language?

Shell is a “quick and dirty” way to get things done.
Example: replace “/ddnB/work” with “/work” in every file under ~/mycode/ — one sed one-liner beats a whole Python script.



b) Why another language over Shell?

Shell is highly inefficient for heavy calculation. A Pi-estimation benchmark in C finishes almost instantly; the same logic in Shell can take far longer.

Example 1: Same Task, Two Approaches

Question	Why would I need...	...if I can just use
Everyday glue work	Shell	Python / C++ / Fortran?
Heavy computation	Python / C++ / Fortran	Shell?

Task: replace all occurrences of “/ddnB/work” with “/work” in every file under ~/mycode/ and its subfolders.

pathSwap.py

```
import os, re
for root,_,files in os.walk("mycode"):
    for fn in files:
        p = os.path.join(root, fn)
        s = open(p).read()
        s = s.replace("/ddnB/work","/work")
        open(p,"w").write(s)
```

pathSwap.sh

```
#!/bin/bash
find mycode -type f | xargs \
    sed -i 's|/ddnB/work|/work|g'
```

One line vs. seven — Shell wins for quick file & text manipulation.

Shell is a “**quick and dirty**” way to get things done!

Question	Why would I need...	...if I can just use
Everyday glue work	Shell	Python / C++ / Fortran?
Heavy computation	Python / C++ / Fortran	Shell?

Benchmark: estimating π to 10,000 iterations — same algorithm, two implementations.

```
[kasetti@qbd1 1.2-WhatCanShellDo]$ time ./pi_c 10000
niter=10000
count in circle:7852
Pi: 3.140800

real    0m0.003s
user    0m0.000s
sys     0m0.001s
```

```
[kasetti@qbd1 1.2-WhatCanShellDo]$ time ./pi_shell.sh 10000  
niter=10000  
count in circle:7866  
Pi: 3.1464000000000000000000000000  
  
real    1m7.762s  
user    0m27.100s  
sys     0m46.509s
```

Shell (100x+ slower)

Shell (100x+ slower)

Shell is **highly inefficient** for heavy calculation!



RULE OF THUMB

**Anything you wish would run faster —
you should NOT use Shell.**



Shell scripting IS for...

- Automating job workflows (env setup, calling executables)
- Pre/post-processing — trimming data, editing configs in batch



Shell scripting is NOT for...

- Heavy calculation (anything you wish ran faster)
- Replacing a language / software you already know well



We DO expect you to...

- Leave familiar with Shell's basic usage
- Be able to use it to streamline your job workflow



We do NOT expect you to...

- Be a Linux or Shell expert by end of today



Basic Knowledge

Interactive vs. non-interactive, commands, variables, arrays, and arithmetic.

Introduction

- What's Shell?
- What Shell Offers You?

Basic Knowledge

- ▶ **Interactive vs Non-interactive (Shell Script)**
- Basic Commands & Syntax
- Variables
- Arrays
- Arithmetic Operations

Beyond Basics

- Subshells
- Flow Control
- Advanced Text Processing Commands

Bonus

- Where to Get Help



Interactive

- Runs in the terminal
- You interact in real time
- Type commands one by one
- E.g., every time you log in



Non-Interactive (Script)

- A pre-written script file
- No interaction while running
- Executes line-by-line, by itself



Interactive

```
[kasetti@qbd1 ShellScripting]$ ls
1.1-ShellExamples  2.1-InteractiveVsNonInteractive  2.5-Arithmetic  3.2-FlowControl
1.2-WhatCanShellDo  2.2-BasicCommands              3.1-Subshells   3.3-TextProcessing
[kasetti@qbd1 ShellScripting]$
[kasetti@qbd1 ShellScripting]$ date
Wed Feb 11 06:17:39 CST 2026
[kasetti@qbd1 ShellScripting]$
[kasetti@qbd1 ShellScripting]$ echo $SHELL
/bin/bash
[kasetti@qbd1 ShellScripting]$
[kasetti@qbd1 ShellScripting]$ cd 1.1-ShellExamples/
[kasetti@qbd1 1.1-ShellExamples]$
[kasetti@qbd1 1.1-ShellExamples]$ ls
countFiles.sh  helloworld.sh  parallelDownload.sh  pi_c  pi_c.sbatch
[kasetti@qbd1 1.1-ShellExamples]$
[kasetti@qbd1 1.1-ShellExamples]$ ./pi_c 10000000
niter=10000000
count in circle:7854959
Pi: 3.141984
[kasetti@qbd1 1.1-ShellExamples]$
```



Non-Interactive (Script)

```
#!/bin/bash

# Variables
DATA_DIR=$1
MAX_FILES=3
count=0

echo "Starting the script..."
echo "Looking for files in $DATA_DIR"

# Check if directory exists
if [[ ! -d $DATA_DIR ]]; then
    echo "Directory $DATA_DIR does not exist"
    exit 1
fi
```

Shell scripting works the same way in both modes — a few features may differ slightly.



helloworld.sh

```
#!/bin/bash
```

```
date
```

```
echo "Hello, HPC world!"
```

```
echo "Today is $(date)"
```

#!/bin/bash

The “shebang” — tells the system which Shell should run this script.

date, echo ...

The commands to run, one per line — executed top to bottom, just like typing them interactively.



Don't forget - run “chmod u+x helloworld.sh” first so the file is executable.

Four Ways to Run a Script

Method		Example	Remarks			
			Must be executable?	Which Shell?	Start subshell?	Others
1	Use full path (Most common)	\$./helloworld.sh \$ /path/to/helloworld.sh	Yes	Shebang (if exist) or default Shell	Yes	-
2	Use specific Shell	\$ bash helloworld.sh \$ cs h helloworld.sh	No	Specified Shell	No	-
3	Use "source" or "."	\$ source helloworld.sh \$. helloworld.sh	No	Current Shell	No	-
4	Run as Shell command	\$ helloworld.sh	Yes	Shebang (if exist) or default Shell	Yes	Parent directory must be included in \$PATH environment variable

Not executable? `chmod u+x helloworld.sh` (method 1)

- What is this?

➔ Anything you learned about Shell today, applies to your batch job files!

```
#!/bin/bash
#SBATCH -A loni_loniadmin1
#SBATCH -p single
#SBATCH -t 1:00:00
#SBATCH -N 1
#SBATCH -n 12

# Time stamp at the beginning
date

# Run the code
./pi_c 1000000

# Time stamp at the end
date
```

Introduction

- What's Shell?
- What Shell Offers You?

Basic Knowledge

- Interactive vs Non-interactive (Shell Script)
 - ▶ **Basic Commands & Syntax**
- Variables
- Arrays
- Arithmetic Operations

Beyond Basics

- Subshells
- Flow Control
- Advanced Text Processing Commands

Bonus

- Where to Get Help



File

ls List files at a given path
cp / mv Copy / move files
rm Removes files
find Search for files



Directory

cd Change directory
mkdir Create a directory
pwd Print working directory



Display

cat Print entire file
head / tail First / last lines
more / less Page through a file



System

echo Print strings
date Current date & time

Character	Description	Example
#	Comment: Anything follows in the same line will not be executed.	\$ date # Print time stamp
;	Command separator: Allows multiple commands in one line.	\$ module purge; module load python
	Pipeline: Use output of first command as input of the second.	\$ queue -u \$USER wc -l
>	Redirect (Output): Redirect standard output / error to file. This method overwrites the file.	\$./testoutput > out.txt \$./testoutput 1> out.txt 2> err.txt
>>	Redirect (Output): Redirect standard output / error to file. This method appends to the file.	\$./testoutput >> out.txt \$./testoutput 1>> out.txt 2>> err.txt
<	Redirect (Input): Read input from a file instead of standard input.	\$./testinput < input.txt
&	Send to background: Send a command to background, and do not wait for it to finish.	\$ sleep 10 &

[1] [ShellScripting/2.2-BasicCommands/testoutput](#)
[2] [ShellScripting/2.2-BasicCommands/testinput](#)



Introduction

- What's Shell?
- What Shell Offers You?

Basic Knowledge

- Interactive vs Non-interactive (Shell Script)
- Basic Commands & Syntax
 - ▶ **Variables**
- Arrays
- Arithmetic Operations

Beyond Basics

- Subshells
- Flow Control
- Advanced Text Processing Commands

Bonus

- Where to Get Help

	To assign	To access	To delete
Syntax	var=value	\$var	unset var
Example	workdir="/work/\$USER"	cd \$workdir	unset workdir



3 Rules to Remember

- All Shell variables are strings — no int/float/bool type
- No spaces allowed around the “=” in assignment
- Use \${ } to explicitly mark a variable name, e.g. \${var}



Why the { } braces?

- Needed when text follows the variable directly
- e.g. echo "\${var}_backup" vs. echo "\$var_backup"
- Without braces, Shell reads a variable named var_backup

- Allowed characters: letters (a-z, A-Z), numbers (0-9), underscore (_)
- Must begin with a letter or underscore
- No other special characters (# @ % \$ etc.)
- Case sensitive: VAR ≠ var



Allowed

varname
var_name
_varName
var123



Not Allowed

123var
#var
var@name
var-123

	Local	Global
Syntax	var=value	export VAR=value
Scope	Exists only in current shell	Copied to all subshells
Convention	lowercase	UPPERCASE



Environment variables

Special (usually global) variables that Shell or other programs read to regulate behavior — change them with care.

Programs may have their own too (Conda / Python / R / MPI ...)

Variable		Functionality
Shell	USER	Username.
	PWD	Full path to current directory.
	HOME	Full path to user's home directory.
	SHELL	Default Shell
	PATH	A list of paths to look for executables as Shell commands (separated by ":").
	LD_LIBRARY_PATH	A list of paths to look for shared libraries (separated by ":").
Slurm	SLURM_JOB_ID	Slurm job ID.
	SLURM_JOB_NODELIST	A list of nodes required for current job (useful for MPI).
OpenMP	OMP_NUM_THREADS	Number of threads per process for OpenMP.
...		

“ ”

Double quotes

Allows **\$variable expansion** and **command substitution**; preserves everything else literally.

```
$ echo "echo $USER"  
echo kasetti
```

' '

Single quotes

Preserves the literal value of ALL characters inside — **no expansion at all**.

```
$ echo 'echo $USER'  
echo $USER
```

` `

Backticks

Command substitution — runs the command and substitutes its output. (Modern style: use `$(...)` instead — more slide ahead!)

```
$ echo `echo $USER`  
kasetti
```

Introduction

- What's Shell?
- What Shell Offers You?

Basic Knowledge

- Interactive vs Non-interactive (Shell Script)
- Basic Commands & Syntax
- Variables
- ▶ **Arrays**
- Arithmetic Operations

Beyond Basics

- Subshells
- Flow Control
- Advanced Text Processing Commands

Bonus

- Where to Get Help

Same basic logic as arrays in any language — indexed from 0, with a few Shell-specific twists.

	To assign	To access	To delete
Entire array	myAry=("Alice" "Bob" "Charlie")	echo \${myAry[@]}	unset myAry
One element	myAry[1]="Brian"	echo \${myAry[1]}	unset myAry[1]

Bonus: get array length with `${#myAry[@]}`



Real use case: parallel job submission

```
inputParams=("case1" "case2" "case3")
parallel myexec ::: ${inputParams[@]}
```



"I'm not using Shell for heavy calculation anyways! What can I possibly need arrays for?"



GNU Parallel — coming next month

A tool that runs many processes at once, each with different parameters.
Shell arrays are the typical way to feed those parameters in.

```
$ parallel myexec ::: ${inputParams[@]}
```

Introduction

- What's Shell?
- What Shell Offers You?

Basic Knowledge

- Interactive vs Non-interactive (Shell Script)
- Basic Commands & Syntax
- Variables
- Arrays
- ▶ **Arithmetic Operations**

Beyond Basics

- Subshells
- Flow Control
- Advanced Text Processing Commands

Bonus

- Where to Get Help



Wait a minute!

Didn't you say Shell doesn't support number types, and we shouldn't use it for heavy calculation?

Correct! But sometimes, arithmetic is still needed.

- What does **NOT** work:

```
$ a=10
$ b=$a/3+2
$ echo $b      # what do you get?
10/3+2         # ...not math! just text.
```

Shell treats every variable as a string. It has no idea \$b was meant to be arithmetic.

- What **DOES** work (assuming **a=10**):

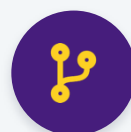
Method		Example	Remarks
1	<code>\$((...))</code> (Most common)	<code>\$ echo <code>\$(((\$a/3+2))</code></code>	- Evaluate everything inside the braces. Integers only!
2	<code>let</code> (Slightly more advanced)	<code>\$ let b=\$a/3+2</code> <code>\$ let b=a/3+2</code> <code>\$ let b++</code>	- Evaluate assignment w/ arithmetic calculation. "\$" can be emitted. Integers only!
3	<code>expr</code> (Legacy, most limited)	<code>\$ expr \$a / 3 + 2</code>	- Strictly limited to "ARG1 OPERATION ARG2" format. Integers only!
4	<code>bc</code> (Most powerful)	<code>\$ bc</code> <code>scale=3</code> <code>a=10;a/3+2</code> <code>\$ bc < bcExample.txt</code> <code>\$ echo "\$a/2+3" bc</code>	Interactive and non-interactive mode. - Does NOT support Shell syntax (namely, "\$" for variables). - Unassigned variables treated as 0. scale variable determines number of decimals. - Supports float number!

A few habits that weren't always emphasized in older Shell tutorials — worth adopting from day one.



Fail fast with `set -euo pipefail`

Add this near the top of scripts so errors, unset variables, and pipeline failures stop the script instead of silently continuing.



Prefer `$(...)` over backticks

`$(command)` nests cleanly and is easier to read than ``command`` — the modern standard for command substitution.



Lint with ShellCheck

`shellcheck myscript.sh` catches quoting bugs, unset variables, and portability issues before you run the script.



Quote your variables

Always write `"$var"` instead of `$var` to avoid word-splitting and glob surprises with paths that contain spaces.



Interactive vs. Non-interactive (Shell Script)



Basic Commands & Syntax



Variables



Arrays



Arithmetic Operations

- Get some water
- Use restroom
- Ask questions

Don't forget — the examples are at:

<https://www.hpc.lsu.edu/static/training/weekly-materials/Downloads/ShellScripting.zip>



Beyond Basics

Subshells, flow control, and advanced text processing.

Introduction

- What's Shell?
- What Shell Offers You?

Basic Knowledge

- Interactive vs Non-interactive (Shell Script)
- Basic Commands & Syntax
- Variables
- Arrays
- Arithmetic Operations

Beyond Basics

- ▶ **Subshells**
- Flow Control
- Advanced Text Processing Commands

Bonus

- Where to Get Help



Definition

A child process launched by an existing Shell.



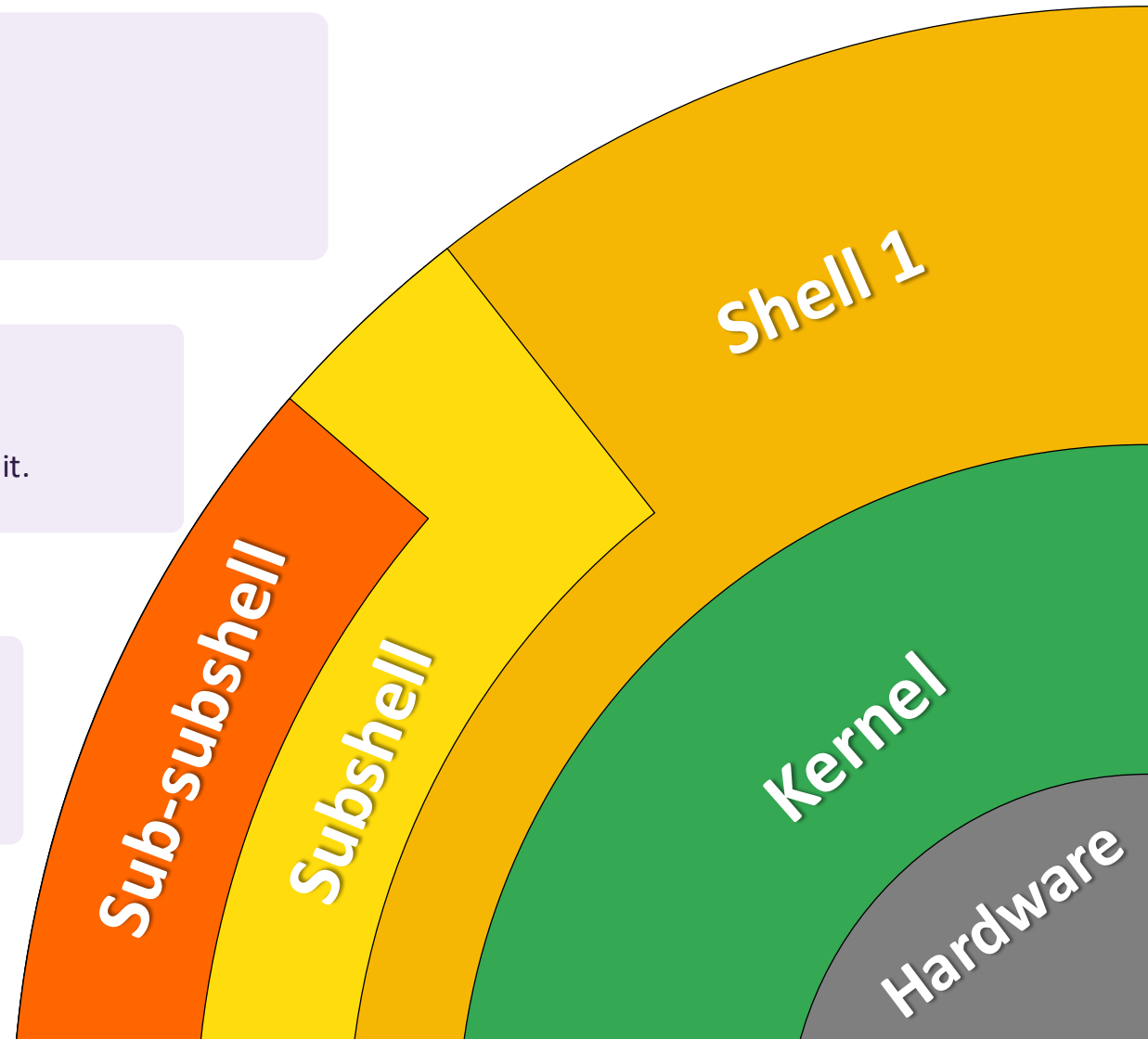
Similarity

Still a Shell! Everything we've covered works the same way inside it.



Difference

An isolated environment from its parent — a “sandbox” Shell.



a) Launch a subshell

Method		Example	Remarks
1	Run a Shell script	\$./subshell.sh \$ bash subshell.sh	- Can launch different Shell types - Check subshell level: \$SHLVL
2	Explicitly launch an interactive subshell	\$ bash	
3	Use command grouping "(...)"	\$ (echo "I am in subshell!")	- Launches the same Shell type - Does NOT change \$SHLVL

- What does **NOT** launch a subshell?
 - **source** subshell.sh
 - Commonly used for **environment setting** scripts (You WANT it to set up current Shell)
 - source setenv.sh

[1] *ShellScripting/3.1-Subshells/subshell.sh*

[2] *ShellScripting/3.1-Subshells/setenv.sh*





	var="Hello" (local)	export var="Hello" (global)
Level 1 (defined here)	✓ Exists	✓ Exists
Level 2 (subshell)	✗ Does not exist	✓ Copied down
Level 3 (sub-subshell)	✗ Does not exist	✓ Copied down

Local variables stay put; exported (global) variables travel down into every subshell.

Introduction

- What's Shell?
- What Shell Offers You?

Basic Knowledge

- Interactive vs Non-interactive (Shell Script)
- Basic Commands & Syntax
- Variables
- Arrays
- Arithmetic Operations

Beyond Basics

- Subshells
- ▶ **Flow Control**
- Advanced Text Processing Commands

Bonus

- Where to Get Help



Flow Control

- Condition — if statement
 - Loop — for loop
 - Loop — while loop
 - Functions

```
if [ condition ]; then
    # do something
elif [ condition2 ]; then
    # do something else
else
    # fallback
fi
```

Key Points

- elif and else are both optional
- Strict spaces required inside [condition]
- Use [[]] for modern features — regex, logic operators, and more

Comparison	Integer	String
Equal / Not equal	[\$a -eq 0] / [\$a -ne 0]	[\$a == \$b] / [\$a != \$b]
Greater / Less than	[\$a -gt 0] / [\$a -lt 0]	—
Zero / non-zero length	—	[-z \$a] / [-n \$a]

File Test	Syntax
Exists	[-e myfile]
Is a regular file / directory	[-f myfile] / [-d /home/\$USER]
Not zero size	[-s myfile]
Read / write / execute permission	[-r] / [-w] / [-x myfile]

Logic	[]	[[]]
NOT	[! -e myfile]	[[! -e myfile]]
AND	[-f a] && [-s a]	[[-f a && -s a]]
OR	[-f a] [-f b]	[[-f a -f b]]



[] single brackets

Supported by more Shells — use when you need portability across Shell types.



[[]] double brackets

Best supported by bash — use when you need versatility (regex, &&, ||).

```
for arg in "${myAry[@]}"
do
    # do something with $arg
done
```

Run the loop body once for each element in a list.

Source	Example
User-defined array	myAry=("Alice" "Bob"); for arg in \${myAry[@]}
Generated sequence	for arg in `seq 1 4`
Output of a command	for arg in `ls \$HOME`



Try it: loop over your submitted jobs

```
for jobid in $(squeue -u $USER -h -o %A); do
    scontrol show job $jobid | grep RunTime
done
```

```
while [ condition ]  
do  
    # do something  
done
```

```
counter=0  
while [ $counter -lt 10 ]  
do  
    echo "Counter is now $counter"  
    let counter++  
done
```



Make sure there's always an escape condition.

If the condition never becomes false — e.g., you forget “let counter++” — the loop runs forever and your job never finishes. Always double-check the increment/exit logic before submitting.

[1] *ShellScripting/3.2-FlowControl/while.sh*



```
# Define
function_name () {
    # do something
}

# Call – no parentheses
function_name arg1 arg2
```

Arguments are accessed as \$1, \$2 ... \${10}, or \$@ for all of them.

Remarks	Example
All variables are global by default	<pre>\$ myFunc1 () { var="Bob" } \$ var="Alice"; myFunc1 ; echo \$var</pre>
Local variables must be explicitly declared	<pre>\$ myFunc2 () { local var="Bob" } \$ var="Alice"; myFunc2 ; echo \$var</pre>
Does NOT support return (Use global variable if needed)	<pre>\$ myAdd () { result=\$((\$1+\$2)) } \$ myAdd 10 20 ; echo \$result 30</pre>



Condition — if statement



Loop — for loop



Loop — while loop



Functions

Introduction

- What's Shell?
- What Shell Offers You?

Basic Knowledge

- Interactive vs Non-interactive (Shell Script)
- Basic Commands & Syntax
- Variables
- Arrays
- Arithmetic Operations

Beyond Basics

- Subshells
- Flow Control
- ▶ **Advanced Text Processing Commands**

Bonus

- Where to Get Help



grep

→ *search*



sed

→ *edit*

Search for patterns (formatted strings) in an input stream — files or a pipe.

```
grep <options> <search pattern> <files>
```

Description	Example
Search for lines contain given string in a file	\$ grep "Sales" employee1.txt
Search for lines do NOT contain given string in a file	\$ grep -v "Sales" employee1.txt
Search all files for lines contain given string in the directory	\$ grep "Sales" *
List files that do NOT contain given string in the directory	\$ grep -L "Sales" *
Search for strings in a pipe	\$ squeue grep \$USER

a) grep

ii. Useful options

Option	Description
-i	Ignore cases.
-r,-R	Search recursively.
-v	Invert match (return those do NOT match pattern)
-l	List names of the files that match the pattern.
-L	List names of the files that do NOT match the pattern.
-n	Print line number with output lines.
...	



Simple strings

`grep "ERROR" cluster.log` — matches the exact text "ERROR".



Regular expressions

Formatted strings that match beyond fixed text — patterns, not just words.

a) grep

iii. Pattern

Metacharacter		Matches	Example
Anchor	^	Beginning of a line.	^Name (Beginning of a line followed by "Name")
	\$	End of a line.	Salary\$ ("Salary" followed by end of a line)
Substitution	.	Any single character	a.e (E.g., "age", "ame", "a#e", "a1e",...)
Repetition	*	Preceding char. repeats 0 or more times	50* (E.g., "5", "50", "500",...)
	+	Preceding char. repeats 1 or more times	50+ (E.g., "50", "500",...)
	?	Preceding char. repeats 0 or 1 times	50? (E.g., "5", "50")
	{n,m}	Preceding char. repeats n to m times	50{1,3} (E.g., "50", "500", "5000")
Or	[]	Any single character inside	[0-9] (E.g., any single number character)
	[^]	Any single character NOT inside	[^0-9] (E.g., any single character but a number)
		Either pattern	Sales Technology (E.g., "Sales" or "Technology")
...			

A powerful “stream editor” for text transformation on an input stream — files or a pipe.

```
sed <options> <script> <files>
```

All **patterns** support regular expression

Function	Usage	Description
Substitution	\$ sed 's/pattern/replacement/flags' file	For each line, replace matched “pattern” with “replacement”, and print out results.
	\$ sed 's/\$[0-9]*/\$9000/' employee2.txt	Replace only the first match of each line.
	\$ sed 's/\$[0-9]*/\$9000/g' employee2.txt	“Greedy” mode, replace all matches of each line.
Deletion	\$ sed '/pattern/d' file	Delete lines with matched pattern, and print results.
	\$ sed '/Sales/d' employee2.txt	Delete all lines matches “Sales”.
	\$ sed '2,4d' employee2.txt	Remove line 2 through 4.
Insertion	\$ sed '/pattern/ i\nnewline' file # Insert before \$ sed '/pattern/ a\nnewline' file # Insert after	Insert / Append new line at specific location, and print results.
	\$ sed '/Alice/ i\nnewline'	Insert before lines matches “Alice”.
	\$ sed '3 a\nnewline'	Append to line 3.
...		

b) sed

ii. Other common usage

Usage	Example	Description
\$ sed -i <script> file	\$ sed -i 's/\${[0-9]*/\$9000/' employee2.txt	Change file in-place instead of printing results.
\$ sed -e <script1> -e <script2> file	\$ sed -e 's/\${[0-9]*/\$9000/' \ -e 's/Rep/Assistant/' employee2.txt	Execute multiple scripts.
\$ cmd sed <options> <script>	\$ conda env list sed '/^#/d'	Parsing piped output instead of file.



grep searches, sed edits.

If you don't remember every flag, that's fine — just remember these two names exist for these two jobs.

Bonus: Where to Get Help?

Introduction

- What's Shell?
- What Shell Offers You?

Basic Knowledge

- Interactive vs Non-interactive (Shell Script)
- Basic Commands & Syntax
- Variables
- Arrays
- Arithmetic Operations

Beyond Basics

- Subshells
- Flow Control
- Advanced Text Processing Commands

Bonus

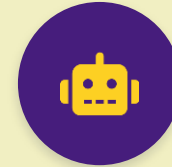
- ▶ **Where to Get Help**



1. HPC User Services

Email: sys-help@loni.org
Phone: +1 (225) 578-0900

Real humans who know our clusters, our storage,
and your workflows.



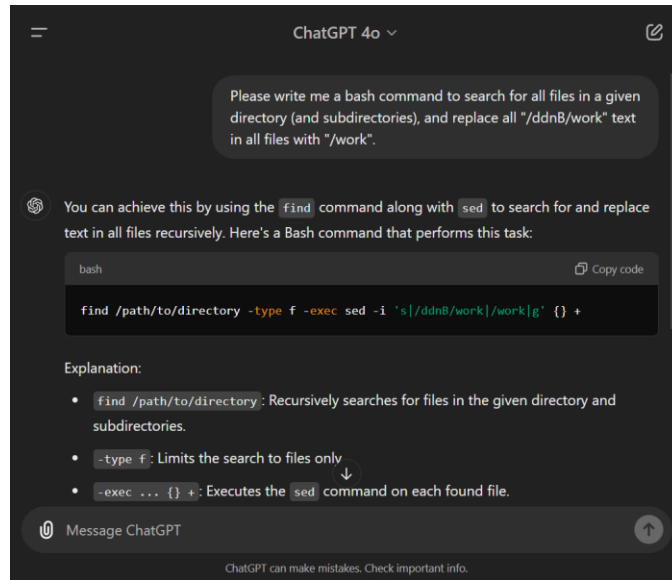
2. Generative AI

Claude, ChatGPT, GitHub Copilot — and MikeGPT,
LSU's own assistant trained on LSU public data.

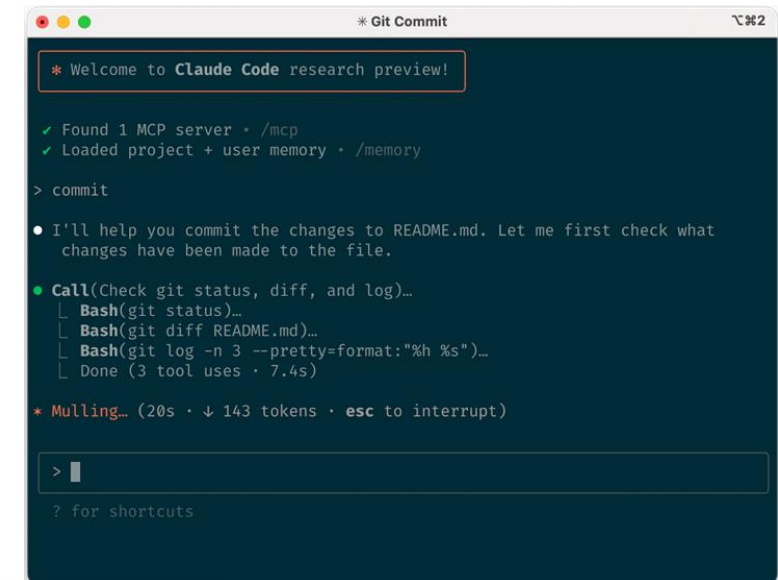
mikegpt.lsu.edu

Great for quick, disposable snippets; not a
replacement for understanding what the code does.

ChatGPT

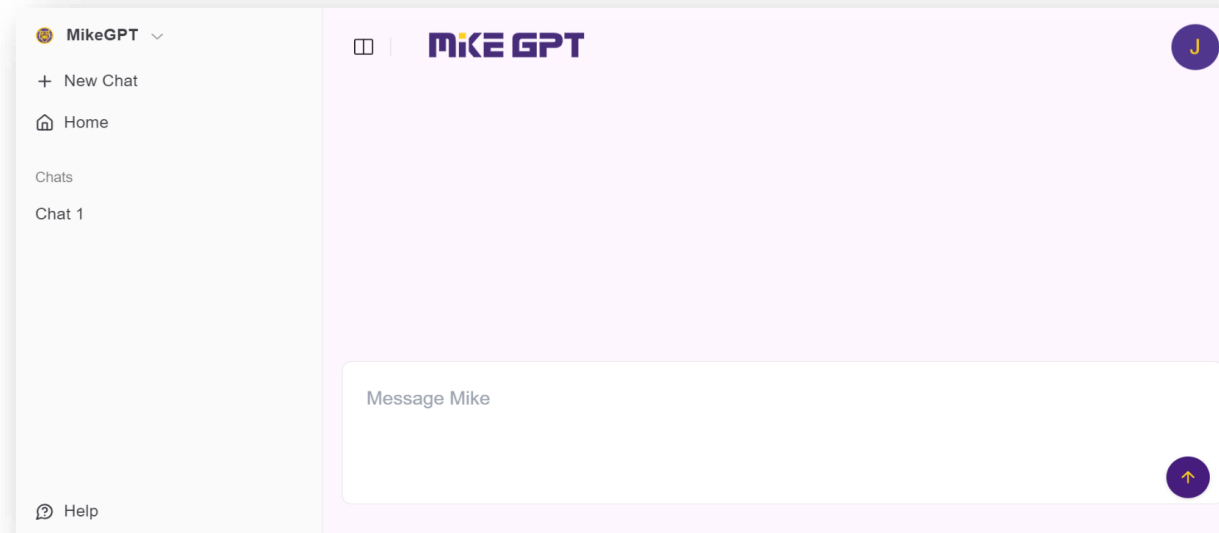


Claude



MikeGPT

(LSU's own GPT chat box! Trained with LSU public realm data)



Why AI Pairs Well With Shell Scripting?

Generative AI is...	Shell scripting needs...
Good at quick, “dirty” answers	A quick, dirty answer — that's often all you need
Not great at citing reliable sources	You rarely need a citation for a one-liner
Strong at short, focused code	One clean line, not a sprawling codebase



1. Ask the right question

Think through the task yourself first, then ask AI precisely.



2. TEST, TEST, TEST

Run it in a safe sandbox — especially if it's destructive.



3. Adopt into your workflow

Once verified, fold it into your real scripts and jobs.



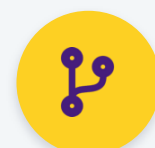
Introduction

What Shell is, and its sweet spot vs. limits



Basic Knowledge

Commands, variables, arrays, arithmetic



Beyond Basics

Subshells, flow control, grep & sed



Bonus

HPC User Services + generative AI, done right



Take-Home Message

Anything you wish to

- ✓ Run faster; you should NOT use Shell.
- ✓ Automate, pre(post) processing; you SHOULD use Shell.

Thank You!

Questions welcome anytime — that's what we're here for.



sys-help@loni.org



+1 (225) 578-0900



hpc.lsu.edu/training



Keep Practicing

Download the exercises and try each example yourself on the cluster:

<https://www.hpc.lsu.edu/static/training/weeklymaterials/Downloads/ShellScripting.zip>