

# AI for Scripting

23 June 2025 23:20

## Leveraging GitHub Copilot for Bash Scripting

### 1. Introduction

- **Objective:** Use AI (GitHub Copilot) to accelerate writing and learning Bash scripts.
- **Prerequisite:** Virtual machines (scriptbox, web01, web02, web03) already up and running.
- **Setup:** Place your scripts folder (downloaded from lecture resources) into your VM directory and open it in VS Code.

### 2. VS Code Extensions

- **GitHub Copilot:** Enables AI code suggestions.
- **Bash IDE Extension:** Provides Bash-specific syntax support.

### 3. Using Copilot in a Script

1. **Open** firstscript.sh.
2. **Navigate** to the end of the file and press **Enter**—Copilot begins suggesting code.
3. **Accept** suggestions with **Tab**, then **Enter** to move to the next line.
  - Adds sections like **CPU Utilization**, **System Information**, **Network Information**.
  - Copilot matches your existing formatting (e.g., number of hash separators).
4. **Review** the AI's suggestions to learn new commands (e.g., `top -b -nl 1 | grep "Cpu(s)"`, `ps aux --sort=-%mem | head -n 10`).

### 4. Common AI-Generated Sections

- **CPU Utilization:** using `top` and `grep`.
- **System Information:** `uname -a`.
- **Network Information:** `ifconfig` or `ip -a`.
- **Process Listing:** `ps aux | sort ....`
- **Open Files:** `lsdf`.
- **Running Services:** `systemctl list-units --type=service`.

### 5. Reviewing & Refining with Copilot

1. **Select** all lines in VS Code.
2. Click the **Copilot** icon → **Review using Copilot**.
3. **Iterate** through suggestions:
  - Correct typos.
  - Replace deprecated commands (e.g., `ifconfig` → `ip -a`).
  - Improve efficiency (e.g., use `ps --sort` instead of raw `ps aux`).
4. **Apply** or **Discard** each suggestion based on relevance.

### 6. Best Practices

- **Always review** AI-generated code before execution.

- **Learn from suggestions:** Understand new commands and patterns introduced by Copilot.
- **Maintain consistency:** Copilot adapts to your formatting style when you accept its proposals.

*Tip:* Treat Copilot as a co-pilot: it accelerates coding but requires your judgment and Bash knowledge to ensure correctness and security.

# Leveraging GitHub Copilot: Advanced Bash Scripting

## 1. Preparing the VM Environment

- **Sync Folder:**
  - Place your scripts ZIP (from lecture resources) into your VM's /vagrant directory.
  - Extract to see the scripts/ folder inside your VM.
- **Editing & Saving:**
  - Open the VM folder in VS Code.
  - Always **save** your edits (Ctrl+S) before testing in the VM.

## 2. Installing VS Code Extensions

- **GitHub Copilot** → AI-powered code completion & chat.
- **Bash IDE Extension** → Bash syntax highlighting and support.

## 3. Invoking Inline Copilot Chat

- **Command:**
  - **Windows:** Ctrl+I
  - **macOS:** ⌘+I
- **Use:** Select code (or place cursor) then open the inline chat pane.

## 4. Improving Code with Copilot

1. **Prompt:** "Improve the code according to development best practices."
2. **AI Suggestions:**
  - Automatically inserts at top of script:
 

```
set -euo pipefail
```

    - -e: Exit on any non-zero status
    - -u: Error on unset variables
    - -o pipefail: Fail if any command in a pipeline fails
3. **Explaining a Suggestion:**
  - **Select** the set -euo pipefail line → **Ctrl+I** → type /explain → **Enter**
  - **View in Chat** for a detailed breakdown of each flag's purpose.

## 5. Introducing Functions via Copilot

1. **Definition:**

```
log() {
    echo "#####"
```

```

echo "$1"
echo "#####"
}

```

## 2. Usage:

- Call `log "Installing packages"` → prints header, message, and footer.

## 3. Refactoring Script:

- Copilot suggests grouping related tasks into functions (e.g., `install_dependencies`, `deploy_artifact`, `restart_service`).
- **Main Execution** section simply calls each function in sequence.

## 6. Saving & Testing

- **Save** your enhanced script in VS Code.
- **Test** in the VM's `/vagrant/scripts` directory:  
`bash websetup.sh`
- Observe improved robustness (immediate exit on errors) and cleaner structure via functions.

*Tip:* Use Copilot's inline chat to **ask for explanations**, **refactor into functions**, and **apply best practices** without leaving your editor.

# Leveraging ShellCheck & Copilot for Code Quality

## 1. Overview

- **Objective:** Use **ShellCheck** (via Bash IDE extension) alongside **GitHub Copilot** to identify and fix common Bash pitfalls in existing scripts.
- **Approach:** For each script, review Copilot/ShellCheck suggestions, apply fixes cautiously, and test on VMs.

## 2. vars\_websetup.sh Review

### 1. Globbing & Word Splitting Warning

- **Issue:** `yum install $PACKAGE`
- **Fix:** Wrap in quotes → `yum install "$PACKAGE"`
- **ShellCheck Code:** SC2086

### 2. cd \$TEMP\_DIR Safety

- **Issue:** Unchecked `cd` can leave you in wrong directory if `$TEMP_DIR` is unset.
- **Fix Suggestion:**  
`cd "$TEMP_DIR" || exit 1`

### 3. Usage Tips:

- Open inline chat (`Ctrl+I/⌘+I`) on warning → use `/fix` to auto-apply ShellCheck quick fix.
- Combine ShellCheck extension with Copilot for best results.

## 3. command\_subs.sh Review

- **Backticks vs. \$(...)**
  - **Issue:** Backticks `<code>`cmd`</code>` are harder to nest and read.
  - **Fix:** Replace with `$(cmd)`

- **ShellCheck Code:** suggestion to use `$(...)`
- **Apply:** Select the entire script → inline chat → /fix → accept to convert all backticks.

## 4. userInput.sh Review

- **read without -r**
  - **Issue:** read var interprets backslashes, mangling input.
  - **Fix:** Use `read -r var` to preserve backslashes.
  - **ShellCheck Article:** Open link for details on `read -r`.

## 5. monit.sh Review

- **Exit Code Check**
  - Suggests using **file test operator** `-f "$PID_FILE"` instead of checking `$?` after file existence tests.
- **Shebang Best Practices**
  - **Add** `set -euo pipefail` at top for robust error handling.
- **Refactoring:**
  - Move critical paths (e.g., `/var/run/httpd/httpd.pid`) into named variables.
  - Use functions for clarity.
- **Process:** Select code → inline chat → “Improve code as per best practices” → review & accept.

## 6. General Best Practices

1. **Always review** AI or lint suggestions before applying.
2. **Test** fixes in your VM environment to ensure correct behavior.
3. **Maintain readability** and consistent style after refactors.
4. **Combine tools:**
  - **ShellCheck** catches static analysis issues.
  - **Copilot** suggests structural improvements and refactorings.

# Leveraging Copilot for Advanced Remote Deployment

## 1. Introduction

- **Goal:** Improve and automate our remote web-setup framework using GitHub Copilot, functions, and arrays; then use Copilot to scaffold a new Tomcat setup project.
- **Context:** We already have `multios_websetup.sh` and `web_deploy.sh` in the `remote_websetup` folder.

## 2. Refactoring with Functions

1. **Invoke Copilot:**
  - **Select** all code in `multios_websetup.sh` → **Ctrl+A, Ctrl+I** → “Improve this code as per

development standards, and use functions.”

## 2. Local vs. Global Variables:

- Copilot introduces local declarations inside functions (e.g., local PACKAGE, local SVC) to limit variable scope.
- **Explanation:** Select the function code → /explain in chat → Copilot clarifies that local confines the variable to that function.

## 3. Main Function Pattern:

- Copilot wraps OS detection and setup steps into discrete functions.
- Defines a main() function that invokes each setup function in order, then ends with a call to main.

# 3. Fixing the Deployment Loop

## 1. Review web\_deploy.sh:

- Copilot suggests improvements, but default suggestions used a **while** loop that exited after the first host.

## 2. Override Suggestion:

- Prompt Copilot inline: “Use a for loop instead of a while loop.”
- Copilot adjusts the loop to:  

```
mapfile -t hosts < <(grep -v '^#' remhosts)
for host in "${hosts[@]}; do
...
done
```

## 3. Using Arrays:

- **mapfile -t hosts < <(grep -v '^#' remhosts)** reads non-comment lines into the hosts array.
- Iteration with "\${hosts[@]}" ensures each hostname is handled correctly, preserving spaces.

# 4. Best Practices & Error Handling

## • File Existence Checks:

- Before deployment, verify remhosts and multios\_websetup.sh exist:  

```
[[ -f remhosts ]] || { echo "Host file missing"; exit 1; }
[[ -f multios_websetup.sh ]] || { echo "Script missing"; exit 1; }
```

## • Push & Execute with Sudo:

- Ensure remote script runs with root privileges if needed:  

```
ssh "$USR@$host" "sudo bash /tmp/multios_websetup.sh"
```

## • Cleanup:

- Remove remote script after execution to keep targets tidy:  

```
ssh "$USR@$host" "rm /tmp/multios_websetup.sh"
```

# 5. Scaffold a Tomcat Setup Project with Copilot

## 1. Start New Chat Workspace: Inline chat → /new.

## 2. Prompt:

“Write Tomcat setup scripts based on OS—RPM-based (CentOS) and Ubuntu-based—using best practices. Include a deploy script that pushes and runs the correct setup script on hosts listed in a file.”

## 3. Generated Structure:

```
tomcat-setup-project/
├── hosts      # inventory file
├── README.md  # instructions
├── scripts/
│   └── setup_rpm.sh
```

```
|— setup_deb.sh
|— deploy_tomcat_setup.sh
```

#### 4. Deploy Script:

- Reads hosts file into an array.
- Loops over each host, detects OS (yum --help vs. apt --help), pushes the appropriate setup\_\*.sh via scp, and executes it via SSH.

## 6. Key Takeaways

- **Copilot** accelerates refactoring, function extraction, and array usage.
- **Inline chat** commands (/explain, /fix) help understand and correct suggestions.
- **Your expertise** guides Copilot: override default AI suggestions when you know a better approach (e.g., switching from while to for).
- **Testing** on VMs is essential before deploying to production.

*Conclusion:* Combining **strong Bash fundamentals** with **AI-assisted development** lets you build robust, maintainable automation frameworks—and even scaffold entire projects like Tomcat setup—with minimal boilerplate.