MSHELL Interlang Guide (C ,C++, Rust, Go, Python, Lua, Bash)

mshell Inter-Language Data Exchange & LLM Directives

Complete Reference Guide

Table of Contents

Overview

mshell extends standard Markdown by adding inter-language data exchange and inline LLM directives. A single `.md` file can contain code blocks in multiple languages that pass data between each other, invoke LLM models to process or generate content, and produce visualizations — all executed sequentially in one document.

What makes mshell Markdown special:

- Code blocks in different languages share data through session variables
- LLM directive blocks call AI models inline
- Variables persist across all blocks in the same document execution
- All 7 languages + 3 LLM models + 3 vendors (Ollama, OpenAI, Claude) supported

---

Core Concepts

Session Context

When mshell executes a Markdown document it creates a session context — a temporary directory `/tmp/mshell_ctx_<pid>/` that stores variable files. Each variable is a plain text file containing the stdout of the block that wrote it.

Execution Order

Blocks execute strictly top to bottom. If block B reads a variable written by block A, block A must appear first in the document.

Synchronous Execution

Blocks with output variables (`>var`) always run synchronously — mshell waits for completion before proceeding to the next block. This guarantees data is available when the next block needs it.

Supported Languages

Fence tag	Language	Notes
bash	Bash	Full shell, PATH set automatically
python or py	Python 3	Auto-installs missing packages via pip
c	C	gcc std=c11, libraries auto-detected
cpp or c++	C++	g++ std=c++17, libraries auto-detected
rust or rs	Rust	rustc for simple, cargo for external crates
go or golang	Go	go build, mod init/tidy automatic
lua	Lua	lua5.4, luarocks path auto-configured
mshell	mshell commands	Native mshell command block

Variable Syntax

Variables are declared as attributes on the code fence line.

Write stdout to variable:

```
```lang >varname
```

Read variable (available as MSH_VAR_varname env, pointing to file path):
```

```lang <varname
```

Both read and write:

```lang <invar >outvar
```

Output variable

Captures the block stdout and saves it to session context.

```
echo "Hello World"
date
```

Input variable

Makes the variable available as environment variable `MSH_VAR_varname` which contains a **file path** to the data file.

```
cat $MSH_VAR_result
```

Important: `MSH_VAR_varname` is always a file path. Always read the file:

```
# correct:
cat $MSH_VAR_myvar

# wrong - gives path string, not data:
echo $MSH_VAR_myvar
```

LLM Directives

LLM directives are HTML comment blocks that invoke AI models inline.

Syntax:

```
<!--@N <input_var >output_var
prompt text here
-->
```

Where `N` is the model number: `1`, `2`, or `3`.

Directive components

Component	Description
`<!--@1`	Call model 1
`<!--@2`	Call model 2
`<!--@3`	Call model 3
`<varname`	Read variable as context, prepended to prompt
`>varname`	Save LLM response to this variable
prompt text	Everything after the first line until `-->`

Models configuration in .mshellrc

For OpenAI:

```
model1=gpt-4o
model2=gpt-4o-mini
model3=o1-mini
```

For Ollama:

```
model1=qwen2.5-coder:7b
model2=llama3.2:3b
model3=deepseek-r1:7b
```

Exec Mode

Exec mode (`<!--@Nx-->`) calls the LLM in execution mode — the model's response is parsed for code blocks and executed immediately. Equivalent to the `ollamaNexec` command but inline in a Markdown document.

Syntax:

```
<!--@1x
prompt
-->

<!--@2x
prompt
-->

<!--@3x
prompt
-->
```

Exec mode requires a code fence trigger

Because exec mode directives contain no code fences, add a stub block in the same document to trigger the markdown parser. Without it the document is processed line-by-line as shell commands.

Always add a stub:

````
<!--@1x
...your prompt...
-->

```bash
echo ""
```

### Prompt template for exec mode

Always tell the model to wrap its code in a markdown fence:

Write python code in a `python` markdown block.

This works reliably across all model sizes.

### When passing variable data to exec mode LLM

If you pass a variable with `<varname`, tell the LLM it contains a file path:

MSH_VAR_data environment variable contains a FILE PATH. Read that file.

---

## Language Block Reference
````

Bash

Full bash, PATH includes all standard system paths.

```bash >output
for i in $(seq 1 5); do
    echo "Item $i: $((i * i))"
done
```

Python

Python 3 with automatic package installation. Missing packages detected from import statements and installed via pip.

```
import numpy as np
data = [23, 45, 12, 67, 34, 89, 11, 56]
print(f"mean={np.mean(data):.2f}, std={np.std(data):.2f}")
```

C

Compiled with gcc std=c11. Libraries auto-detected from include directives (-lm, -lpthread, -lGL etc.).

```
#include <stdio.h>
#include <math.h>
int is_prime(int n) {
    if (n < 2) return 0;
    for (int i = 2; i <= (int)sqrt(n); i++)
        if (n % i == 0) return 0;
    return 1;
```

```
}
int main() {
    for (int i = 2; i <= 30; i++)
        if (is_prime(i)) printf("%d\n", i);
    return 0;
}
```

Rust

Simple programs use rustc. Programs with external crates use cargo with cargo add automatically.

```
fn fib(n: u64) -> u64 {
    match n { 0 => 0, 1 => 1, _ => fib(n-1) +
fib(n-2) }
}
fn main() {
    for i in 0..10 { println!("{}", fib(i)); }
}
```

Go

Programs automatically get go mod init and go mod tidy.

```
package main
import (
    "fmt"
    "sort"
)
func main() {
    nums := []int{5, 2, 8, 1, 9, 3}
    sort.Ints(nums)
```

```
    for _, n := range nums { fmt.Println(n) }
}
```

Lua

Lua 5.4 with luarocks path configured automatically.

```
local m = {{1,2,3},{4,5,6},{7,8,9}}
for i = 1, 3 do
    for j = 1, 3 do io.write(m[i][j] .. " ") end
    io.write("\n")
end
```

Reading Variables in Each Language

Language	How to read MSH_VAR_name
Bash	`cat $MSH_VAR_name`
Python	`open(os.environ['MSH_VAR_name']).read()`
Rust	`fs::read_to_string(env::var("MSH_VAR_name").unwrap())`
Go	`os.ReadFile(os.Getenv("MSH_VAR_name"))`
Lua	`io.open(os.getenv("MSH_VAR_name"), "r"):read("*all")`

Python:

```
import os
content = open(os.environ['MSH_VAR_myvar']).read()
lines = content.strip().split('\n')
```

Rust:

```
use std::env;
use std::fs;
fn main() {
    let path = env::var("MSH_VAR_myvar").unwrap();
    let content = fs::read_to_string(path).unwrap();
    for line in content.lines() { println!("{}",
line); }
}
```

Go:

```
package main
import (
    "fmt"
    "os"
    "strings"
)
func main() {
    path := os.Getenv("MSH_VAR_myvar")
    data, _ := os.ReadFile(path)
    lines :=
strings.Split(strings.TrimSpace(string(data)), "\n")
    fmt.Println(lines)
}
```

Lua:

```
local path = os.getenv("MSH_VAR_myvar")
local f = io.open(path, "r")
local content = f:read("*all")
```

```
f:close()
```

Tested Examples

All 28 examples below were tested and verified working across OpenAI, Ollama, and Claude vendors.

Test 1 — Basic bash block

No variables, no LLM. A plain bash block.

```
echo "Hello from bash"
date
```

Test 2 — Variable passing between bash blocks

Write stdout to a variable, read it in the next block.

```
echo "hello from bash block"
echo "line two"

echo "=== received ==="
cat $MSH_VAR_data
echo "=== done ==="
```

Test 3 — Bash to LLM to Bash

Pass bash output as context to LLM, save answer, use in next block.

```
echo "2 + 2 = ?"

<!--@1 <data >answer
Give a one-line answer to this math question.
-->

echo "LLM said:"
cat $MSH_VAR_answer
```

Test 4 — Bash to Python

Bash produces a list of numbers, Python computes sum and average.

```
echo "10 20 30 40 50"

import os
nums = open(os.environ['MSH_VAR_data']).read().split()
total = sum(int(x) for x in nums)
print(f"Sum: {total}, Average: {total/len(nums)}")
```

Test 5 — Python to Bash

Python generates items, Bash displays them.

```
for i in range(5):
    print(f"item_{i}: {i*i}")

echo "=== Python output ==="
cat $MSH_VAR_result
```

Test 6 — C to Python

C computes squares, Python reads the list and processes it.

```
#include <stdio.h>
int main() {
    for (int i = 1; i <= 5; i++)
        printf("%d\n", i * i);
    return 0;
}

import os
data = open(os.environ['MSH_VAR_numbers']).read()
vals = [int(x) for x in data.split()]
print(f"Squares from C: {vals}")
print(f"Total: {sum(vals)}")
```

Test 7 — C to Python to LLM to Bash

Four-stage pipeline: C generates primes, Python computes stats, LLM gives insight, Bash prints report.

```
#include <stdio.h>
int main() {
    int primes[] = {2,3,5,7,11,13,17,19,23,29};
    for (int i = 0; i < 10; i++)
        printf("%d\n", primes[i]);
    return 0;
}
```

```
import os
nums = [int(x) for x in
open(os.environ['MSH_VAR_raw']).read().split()]
print(f"Count: {len(nums)}, Sum: {sum(nums)}, Max:
{max(nums)}")
```

```
<!--@1 <stats >conclusion
These are statistics about prime numbers. Give a one
sentence insight.
-->
```

```
echo "=== LLM insight ==="
cat $MSH_VAR_conclusion
```

Test 8 — Bash to Lua

Bash passes a list of numbers, Lua computes sum, count, and average.

```
echo "10 20 30 40 50"

local f = io.open(os.getenv("MSH_VAR_data"))
local nums = {}
for n in f:read("*a"):gmatch("%d+") do
    table.insert(nums, tonumber(n))
end
f:close()
local sum = 0
for _, v in ipairs(nums) do sum = sum + v end
print(string.format("Sum: %d, Count: %d, Avg: %.1f",
sum, #nums, sum/#nums))
```

Test 9 — Lua to Bash

Lua generates items, Bash displays them.

```
for i = 1, 5 do
    print(string.format("lua_item_%d = %d", i, i*i*i))
end

echo "=== Lua output ==="
cat $MSH_VAR_result
```

Test 10 — Bash to Rust

Bash passes a number, Rust computes its factorial.

```
echo "5"
```

```
use std::env;
use std::fs;
fn main() {
    let path = env::var("MSH_VAR_data").unwrap();
    let n: u64 =
fs::read_to_string(path).unwrap().trim().parse().unwrap();
    let fact: u64 = (1..=n).product();
    println!("{}! = {}", n, fact);
}
```

Test 11 — Rust to Python

Rust finds primes, Python reads and summarizes the list.

```
fn main() {
    for n in 2u32..30 {
        let is_prime = (2..n).all(|i| n % i != 0);
        if is_prime { println!("{}", n); }
    }
}
```

```
import os
data = open(os.environ['MSH_VAR_primes']).read()
nums = [int(x) for x in data.split()]
print(f"Primes up to 30: {nums}")
print(f"Count: {len(nums)}")
```

Test 12 — Bash to Go

Bash passes a number, Go computes the sum of all its divisors.

```
echo "100"
```

```
package main
import (
    "fmt"
    "os"
    "strconv"
    "strings"
)
func main() {
    path := os.Getenv("MSH_VAR_data")
    b, _ := os.ReadFile(path)
    n, _ := strconv.Atoi(strings.TrimSpace(string(b)))
    sum := 0
    for i := 1; i <= n; i++ {
        if n%i == 0 { sum += i }
    }
    fmt.Printf("Divisors sum of %d = %d\n", n, sum)
}
```

Test 13 — LLM generates data, Python processes

LLM generates city population data, Python extracts numbers and computes totals.

```
<!--@1 >dataset
Generate a list of 5 random city names with
populations, one per line, format: "City: N"
-->

import os, re
data = open(os.environ['MSH_VAR_dataset']).read()
nums = [int(x) for x in re.findall(r'\d+', data)]
print(f"Total population: {sum(nums):,}")
print(f"Average: {sum(nums)//len(nums):,}")
```

Test 14 — C generates, LLM explains, Bash reports

C outputs Fibonacci numbers, LLM identifies the pattern, Bash prints the result.

```
#include <stdio.h>
int main() {
    int fib[] = {1,1,2,3,5,8,13,21,34,55};
    for (int i = 0; i < 10; i++)
        printf("%d\n", fib[i]);
    return 0;
}

<!--@1 <rawdata >explanation
What pattern do these numbers follow? One sentence
only.
```

```
-->

echo "=== C computed, LLM explained ==="
cat $MSH_VAR_explanation
```

Test 15 — LLM writes Python code, Python executes it

LLM generates code into a variable, Bash inspects it, Python runs it with exec().

```
<!--@1 >code
Write only python code, no explanation, no markdown.
Print first 10 fibonacci numbers using a generator.
-->

echo "=== LLM wrote this code ==="
cat $MSH_VAR_code

import os
exec(open(os.environ['MSH_VAR_code']).read())
```

Test 16 — Python to LLM filter to Python

Python generates a list, LLM filters it by a condition, Python processes the filtered result.

```
data = ["apple 5", "banana 12", "cherry 3", "date 8",
"elderberry 1"]
for item in data:
    print(item)

<!--@1 <raw >filtered
From this list keep only items with number > 5. Return
only the filtered lines, nothing else.
-->

import os
lines =
open(os.environ['MSH_VAR_filtered']).read().strip().split('\n
print(f"Filtered {len(lines)} items:")
for line in lines:
    print(f"  {line}")
```

Test 17 — Lua to LLM translate to Bash

Lua generates metrics as key=value pairs, LLM converts to human-readable summary, Bash displays.

```
local data = {cpu=87, memory=62, disk=45, network=23}
for k, v in pairs(data) do
    print(k .. "=" .. v)
end

<!--@1 <report >summary
These are server metrics as key=value pairs. Write a
one-line human readable status summary.
-->
```

```
echo "Server status: $(cat $MSH_VAR_summary)"
```

Test 18 — Two LLM calls in pipeline

Bash writes a question, first LLM answers it, second LLM turns the answer into a haiku.

```
echo "What is the boiling point of water in Celsius?"

<!--@1 <question >answer
-->

<!--@1 <answer >haiku
Turn this fact into a haiku. Return only the haiku lines.
-->

echo "=== Haiku ==="
cat $MSH_VAR_haiku
```

Test 19 — LLM generates matplotlib code, Python executes

LLM writes plot code into a variable, Python runs it with exec(). Chart saved to file.

```
<!--@1 >plotcode
```

```
Write only python code using matplotlib. No
explanation, no markdown fences.
Plot a bar chart of these values: Mon=5, Tue=8, Wed=3,
Thu=9, Fri=6.
Save to /tmp/chart.png, do not call plt.show().
-->

import os
exec(open(os.environ['MSH_VAR_plotcode']).read())
print("Chart saved to /tmp/chart.png")

ls -la /tmp/chart.png
```

Test 20 — LLM generates seaborn heatmap

LLM writes seaborn code, Python executes it.

```
<!--@1 >plotcode
Write only python code using seaborn and matplotlib. No
explanation, no markdown.
Create a heatmap of a 5x5 correlation matrix with
random data (seed=42).
Save to /tmp/heatmap.png, do not call plt.show().
-->

import os
exec(open(os.environ['MSH_VAR_plotcode']).read())
print("Saved /tmp/heatmap.png")
```

Test 21 — C data to LLM plot code to Python

C computes sine wave x/y pairs, LLM writes matplotlib code to read the file and plot, Python executes. Note the prompt tells LLM that MSH_VAR_data is a file path.

```
#include <stdio.h>
#include <math.h>
int main() {
    for (int i = 0; i <= 20; i++)
        printf("%.2f %.2f\n", i * 0.314, sin(i *
0.314));
    return 0;
}

<!--@1 <data >plotcode
Write only python code. No explanation, no markdown
fences.
MSH_VAR_data environment variable contains a FILE PATH.
Read the file to get x y pairs.
Plot a sine wave line chart, save to /tmp/sine.png, no
plt.show().
-->

import os
exec(open(os.environ['MSH_VAR_plotcode']).read())
print("Saved /tmp/sine.png")
```

Test 22 — Exec mode: histogram

LLM generates code and mshell executes it immediately. The stub bash block triggers the markdown parser.

````
<!--@1x
Write python code in a python markdown code block (```python ... ```).
The code should:
1. Generate 50 random numbers from normal distribution
2. Plot a histogram, save to /tmp/histogram.png
3. Print mean and std
Use matplotlib, numpy.
-->

echo ""
````

Test 23 — Go data to LLM to Plotly HTML

Go generates monthly data, LLM writes plotly code to read the file and create a chart, Python executes. Result saved as interactive HTML.

````
package main
import "fmt"
func main() {
    for i := 1; i <= 12; i++ {
        fmt.Printf("Month%d %d\n", i, 100 + i*15 + (i%3)*20)
    }
}

<!--@1 <data >plotcode
````

```
Write only python code using plotly. No explanation, no markdown fences.
MSH_VAR_data environment variable contains a FILE PATH. Read that file to get month name and value pairs.
Create a line chart, save to /tmp/plotly_chart.html, no show().
-->

import os
exec(open(os.environ['MSH_VAR_plotcode']).read())
print("Saved /tmp/plotly_chart.html")
```

Test 24 — Exec mode: matplotlib animated window

LLM generates animated sine wave code, mshell executes it, window opens live.

```
<!--@1x
Write python code in a ```python``` markdown block.
Animated sine wave in matplotlib window.
Use plt.show(). Use matplotlib, numpy.
-->

echo ""
```

Test 25 — Exec mode: Plotly 3D surface in browser

LLM generates interactive 3D surface plot, mshell executes it, browser opens automatically.

```
<!--@1x
Write python code in a ```python``` markdown block.
Write python code using plotly that creates an interactive 3D surface plot
of z = sin(x) * cos(y). Call fig.show() at the end.
-->

echo ""
```

Test 26 — Exec mode: Tkinter analog clock

LLM generates a live analog clock with tkinter, mshell executes it, window opens.

```
<!--@1x
Write python code in a ```python``` markdown block.
Write python code using tkinter and math that draws an animated clock
that updates every second. Use mainloop().
-->

echo ""
```

Test 27 — Exec mode: C OpenGL spinning triangle

LLM generates C OpenGL code, Python writes it to file, compiles with gcc, and runs. OpenGL window opens.

```
<!--@1x
Write python code in a ```python``` markdown block.
Write C code that opens an OpenGL window using GLUT and draws a spinning colored triangle.
Write the C code to a file, compile with gcc -lGL -lGLU -lglut, run the result. Use subprocess.
-->

echo ""
```

Test 28 — Exec mode: Pygame bouncing balls

LLM generates pygame animation, mshell executes it, animation window opens.

```
<!--@1x
Write python code in a ```python``` markdown block.
Write python code using pygame that shows bouncing colored balls animation.
Run it with pygame.display and event loop.
-->

echo ""
```

Best Practices

Variable naming

Use descriptive names that reflect the content:

```
>raw_data        — initial data from source
>filtered        — after filtering or processing
>stats           — computed statistics
>analysis        — LLM analysis result
>plotcode        — LLM-generated visualization code
```

Data format between languages

For inter-language compatibility use simple line-based formats:

```
# One value per line
42.5
31.2
56.8

# Key=value pairs (easy to parse in any language)
cpu=87
memory=62

# Space-separated columns
2026-01-01 120.5
2026-01-02 135.2
```

Avoid complex nested JSON — harder to parse uniformly across all languages.

Prompt tips for code generation

Always specify these in prompts:

- What library to use: `using matplotlib`, `using numpy`
- No extra text: `Return only the code, no explanation`
- When variable is a file path: `MSH_VAR_X contains a FILE PATH. Read that file.`
- Where to save output: `Save to /tmp/output.png`

Block ordering

Always place producer blocks before consumer blocks.

CORRECT order — producer writes first, consumer reads after:

```
...

...
```

WRONG — consumer runs before producer, variable does not exist yet:

```
...

...
```

Error handling

Check that variables exist before using them:

```
import os, sys
var_path = os.environ.get('MSH_VAR_mydata')
```

```
if not var_path:
    print("Error: mydata not set", file=sys.stderr)
    sys.exit(1)
content = open(var_path).read()
```

Troubleshooting

Variable not found

Symptom: `[msh_ctx] Warning: variable 'myvar' not found`

Cause: The producing block has not run yet or it failed silently.

Fix: Check that the block with `>myvar` appears before the block with `<myvar`. Verify the producing block runs without errors.

LLM treats variable as data instead of file path

Symptom: Python error like `ValueError: could not convert string to float: '/tmp/mshell_ctx_1234/data'`

Cause: LLM-generated code uses `os.environ['MSH_VAR_data']` directly as data instead of reading the file at that path.

Fix: Add to the LLM prompt: `MSH_VAR_data environment variable contains a FILE PATH. Read that file.`

Exec mode: LLM output visible but nothing executes

Symptom: LLM response text appears in terminal but no code runs.

Cause: Document has no code fences so mshell processes it line-by-line as shell commands instead of using the markdown parser.

Fix: Add a stub block anywhere in the document:

```
echo ""
```

CRLF line endings

Symptom: `command not found: date\r`

Cause: Markdown file has Windows line endings.

Fix: `dos2unix myfile.md` or configure your editor to save with Unix line endings.

Python package not auto-installed

Symptom: `ModuleNotFoundError`

Cause: Package name differs from the import name, e.g. `import cv2` but package is `opencv-python`.

Fix: Add a manual install block before the python block:

```
pip3 install opencv-python --break-system-packages -q
```

C compilation fails

Symptom: `=== C Compilation failed ===`

Cause: Missing library development headers.

Fix: `sudo apt install lib<name>-dev`. The compiler error appears in the output before the failure message.

Rust first run is slow

Symptom: Long pause on first execution with external crates.

Cause: cargo downloads and compiles dependencies on first use.

Fix: Normal behavior. Subsequent runs use cached builds. For fast compilation write self-contained code without external crates and use rustc directly.

Quick Reference

```
VARIABLE SYNTAX
  ```lang >outvar          write stdout to outvar
  ```lang <invar           read invar as MSH_VAR_invar
(file path)
  ```lang <invar >outvar   both

READ VARIABLE — MSH_VAR_name is always a FILE PATH
  bash:    cat $MSH_VAR_name
  python:  open(os.environ['MSH_VAR_name']).read()
  rust:
fs::read_to_string(env::var("MSH_VAR_name").unwrap())
  go:      os.ReadFile(os.Getenv("MSH_VAR_name"))
  lua:     io.open(os.getenv("MSH_VAR_name"),
"r"):read("*all")

LLM DIRECTIVES
  <!--@1 >out
  prompt
  -->                      model 1, save response to
out

  <!--@2 <in >out
  prompt
  -->                      model 2, read in, save to
out

  <!--@3 <in
  prompt
  -->                      model 3, read in, no save

  <!--@1x
```
```

```
  prompt
  -->                          model 1 exec mode:
generates and runs code

EXEC MODE — always add stub block when no other ``` in
document:
  ```bash
  echo ""
```

EXEC MODE PROMPT TEMPLATE

Write python code in a `python` markdown block.

[your instructions here]

WHEN LLM READS A VARIABLE FILE

MSH_VAR_data environment variable contains a FILE PATH. Read that file.

```
---

*mshell v1.4.1 — Inter-Language Exchange & LLM
Directives*

*Session context: /tmp/mshell_ctx_pid — variables
persist for the duration of the mshell process*



---
```
```