

Design Philosophy: Process vs Structure

The Core Problem

When someone follows installation instructions, they experience **process** (step 1, step 2, ...)

But the instructions assume a specific **file structure** (files in certain directories, certain files)

These two things are **deeply linked but often documented separately**, creating confusion:

Scenario:

> "Follow these steps to install secret-server"

>

> Step 1: Copy web_server.py to your phone

> Step 2: Install requirements.txt

> Step 3: Run web_server.py

>

> ■ But where should web_server.py live? ~/secret-server/ or ~/app/ or ~/src/secret-server/?

> ■ When you "install requirements," does it auto-find the file in your chosen location?

> ■ How do you know if you've followed the process *correctly* if the structure is ambiguous?

The Solution: Coupled Documentation

1. **Explicit Target State**

Document the **final file structure** that must exist after successful installation:

...

PHONE

~/ (home directory)

■■■ secret-server/

■ ■■■ web_server.py

■ ■■■ main.py

■ ■■■ requirements.txt

■ ■■■ lib/

■ ■ ■■■ auth.py

■ ■ ■■■ crypto.py

■ ■ ■■■ network.py

■ ■ ■■■ ...

■ ■■■ db/

■ ■ ■■■ sys_adm/

■ ■ ■■■ auth.json

■ ■ ■■■ apitest/

■ ■ ■■■ ...

■ ■■■ start_server.sh

■

■■■ android_mnt/ (symlink or remount point on Linux)

■

■■■ .termux/

■■■ boot/

■■■ start_server (startup hook)

...

This is **non-negotiable**. Everything else assumes this structure.

2. **Process That References Structure**

Every step in the installation should **explicitly point back** to where files go:

...

PHASE 3: Copy Files

```
■■ Copy web_server.py → PHONE ~/secret-server/web_server.py
■■ Copy main.py → PHONE ~/secret-server/main.py
■■ Copy lib/ → PHONE ~/secret-server/lib/
■■ Copy db/ → PHONE ~/secret-server/db/
■■ Verify: PHONE$ ls ~/secret-server/
Expected output: main.py web_server.py lib/ db/ ...
```

...

****No ambiguity.**** Process = explicit paths + verification steps.

3. ****Validation Code****

Include a ****structure validator**** that users can run:

```
```python
def validate_structure(base_path="~/secret-server"):
 """Check if target directory matches expected structure."""
 checks = {
 "web_server.py": "Web server entry point",
 "main.py": "Main application module",
 "requirements.txt": "Python dependencies",
 "lib/auth.py": "Authentication module",
 "lib/crypto.py": "Encryption module",
 "lib/network.py": "Network utilities",
 "db/": "Database directory",
 }

 for path, description in checks.items():
 full_path = f"{base_path}/{path}"
 if exists(full_path):
 print(f"✓ {description}: {path}")
 else:
 print(f"X MISSING: {description}: {path}")
 return False

 return True
```
```

****Usage:****

```
```bash
python validate_structure.py ~/secret-server
Output:
✓ Web server entry point: web_server.py
✓ Main application module: main.py
... etc
```
```

Users can run this ****at any point**** to check: "Did I follow the process correctly?"

Benefits

1. ****Removes ambiguity**** – Process and structure are locked together
2. ****Self-verifying**** – Users can check their work
3. ****Scalable**** – New developers see the expected target state immediately
4. ****Debuggable**** – If something breaks, the structure is the diagnostic first step
5. ****Testable**** – Automated tests can verify structure matches expected state

Application

This philosophy is embedded in:

- ****VIRGIN_INSTALL_CHECKLIST.md**** – Process with explicit paths + verify steps
- ****[Directory structure docs]**** – target-state diagrams
- ****Validation scripts**** – automated structure checking
- ****Architecture documentation**** – why the structure is organized this way

Forward Note

As new projects emerge (auth-server, hub-manager, etc.), they should:

1. Define their target file structure first
2. Then write process documentation that references that structure
3. Include validation code
4. Test the process on virgin systems

This inverts the typical flow: ****structure → process → validation****, not process → "hope the