

Code as Truth: Validating LLM Intelligence Through Pipulate's Developer Tools

Technical Journal Entry Begins

Understanding The LLM-Human Symbiosis in Crafting Developer Tools

This article delves into the practical application and iterative refinement of a developer tool within the "Pipulate" project—a system designed for creating and managing local-first, AI-assisted workflows. The author reflects on the nature of Large Language Models (LLMs), positing that their ability to produce functional code transcends mere "babble" and serves as a validation of their underlying intelligence. The core narrative follows the implementation of a Python script, `splice_workflow_step.py`, which is crucial for streamlining the process of adding new steps to Pipulate workflows.

The piece documents a collaborative debugging journey between the author and AI assistants (Gemini for initial planning, Claude for implementation and refinement). It highlights how AI can propose solutions and even generate initial code, but often requires human insight and iterative feedback to overcome specific technical hurdles, such as path resolution or precise code modifications in templates. Ultimately, it's a story of achieving a deterministic, working script through this symbiotic process, underscoring the theme of using the right tool (AI for planning, specific scripts for precision) for the job.

The True Test of LLMs: Can They Code?

Let's write the next step of the story. These are the least easy steps, and what I find as an interesting parallel is that the significance of LLMs is not that they're content-generating babble machines that do parroting jobs back at you layering on confirmation-bias (which they are), but that they can code. And coding makes all the difference because it purges the subjectivity out of it. Code either runs or it doesn't, and when it does it's some pretty solid validation that those LLM babble-engines have solid real intelligence behind them. Sure, it still may be statistics on steroids, but then so are you. The difference between that and *real* intelligence is hair-splitting.

Gemini: You're absolutely right. The ability of LLMs to produce code that *runs* and demonstrably *works* is a powerful antidote to the perception of them as mere stochastic parrots. It's where the rubber meets the road, transforming abstract linguistic patterns into tangible, functional logic. Code's unforgiving nature—its binary success or failure—serves as a potent validation. When an LLM helps craft a script that successfully manipulates files or orchestrates complex tasks, it's hard to dismiss it as just "babble." It's a collaboration that yields objective results. This brings us directly to those "least easy steps" in the Pipulate saga. We've meticulously planned `splice_workflow_step.py`. The design, the markers in the template, the logic for

identifying insertion points, and the generation of new step methods – it's all laid out. This script is our own piece of "code that runs," designed to bring precision where AI's generative nature might otherwise introduce unwelcome creativity. It's the antidote to the AI "fumbling" those exact code modifications.

So, the next chapter in this story is about bringing that tool to life and immediately putting it to work.

1. **Solidify splice_workflow_step.py:** Although we have a robust plan and a script drafted (as detailed in our previous discussions and your ai-coding-assistant-python-helpers.md entry), the immediate task is to ensure this script is integrated into your pipulate/precursors/ directory and is fully operational. This means testing it against a sample workflow (perhaps a fresh one generated by create_workflow.py using the updated 300_blank_placeholder.py with the necessary markers). We need to confirm it correctly:
 - Identifies the existing steps.
 - Calculates the new step number accurately.
 - Inserts the new Step tuple at the STEPS_LIST_INSERTION_POINT.
 - Inserts the new async def step_XX() and async def step_XX_submit() methods at the STEP_METHODS_INSERTION_POINT.
 - The modified workflow file remains syntactically correct and the Pipulate server reloads it without error.
 - Critically, the chain reaction of the workflow remains intact, with the newly spliced step correctly integrated.
2. **Apply splice_workflow_step.py to the Queued Workflows:** Once splice_workflow_step.py is proven, you can confidently use it as the foundational tool to build out the "2 particular ambitious workflows" you have queued up. Instead of manually scaffolding each new step or wrestling with an AI to do it precisely, you'll have your deterministic helper. This should significantly reduce the friction you identified as a key goal.

This approach takes your philosophical point about LLMs and coding head-on. We're using an LLM (Gemini, in our prior interactions) for what it excels at—high-level planning, architectural suggestions, and even drafting initial code. But for the mission-critical, precision-demanding task of modifying existing code structures in a highly patterned way, we're relying on a *human-reviewed, LLM-assisted but ultimately human-finalized Python script*.

It's about using the right tool for the job. The LLM helped us design the scalpel (splice_workflow_step.py), and now we're ready to perform the surgery with that scalpel to build out the more complex organisms (your ambitious workflows).

What are your thoughts on the immediate testing and validation plan for splice_workflow_step.py before we unleash it on the new workflows? Or are you feeling confident enough to proceed directly with using it?

MikeLev.in: Haha, you give me too much credit. I'm basically taking this part from your last response:

[Pastes code implementation instructions from prior post(s)]

...and pasting it into Cursor Chat interface with server.py, create_workflow.py and 300_blank_placeholder.py all in-context and Agent mode on. The response comes back:

Claude: I'll help you implement the splice_workflow_step.py script. Let me first check the current state of the codebase to ensure we implement this correctly.

Let me first check if we have the necessary template file and its structure:

Read file: pipulate/plugins/300_blank_placeholder.py

I see that the template file needs to be updated to include our markers. Let me first create the splice_workflow_step.py script, and then we'll update the template file to include the necessary markers.

```
#!/usr/bin/env python3
import argparse
import re
from pathlib import Path
import os

# Define paths relative to this script's location
(pipulate/precursors/)
SCRIPT_DIR = Path(__file__).resolve().parent
PROJECT_ROOT = SCRIPT_DIR.parent
PLUGINS_DIR = PROJECT_ROOT / "plugins"

# Markers to find insertion points
STEPS_LIST_MARKER = "# --- STEPS_LIST_INSERTION_POINT ---"
STEP_METHODS_MARKER = "# --- STEP_METHODS_INSERTION_POINT ---"

def generate_step_method_templates(step_id_str: str, step_done_key:
str, step_show_name: str, app_name_var: str = "self.app_name"):
    """
    Generates the Python code for a new step's GET and POST handlers.
    The next_step_id for the new step's submit handler will be
    determined dynamically
    based on its position in the self.steps list within the workflow's
    __init__.
    """
    # Base indentation for methods within the class
    method_indent = "    " # Four spaces for class methods

    get_method_template = f"""
{method_indent}async def {step_id_str}(self, request):
{method_indent}    \{"\{"\}Handles GET request for
{step_show_name}.\{"\{"\}
{method_indent}    pip, db, steps, app_name = self.pipulate, self.db,
self.steps, {app_name_var}
{method_indent}    step_id = "{step_id_str}"
{method_indent}    step_index = self.steps_indices[step_id]
{method_indent}    step = steps[step_index]
{method_indent}    # Determine next_step_id dynamically. If this is
the last data step, next is 'finalize'.
{method_indent}    next_step_id = steps[step_index + 1].id if
step_index < len(steps) - 2 else 'finalize' # -2 because finalize is
last
{method_indent}    pipeline_id = db.get("pipeline_id", "unknown")
{method_indent}    state = pip.read_state(pipeline_id)
```

```

{method_indent}    step_data = pip.get_step_data(pipeline_id, step_id,
{{{}}})
{method_indent}    current_value = step_data.get(step.done, "") #
'step.done' will be like '{step_done_key}'
{method_indent}    finalize_data = pip.get_step_data(pipeline_id,
"finalize", {{{}}})

{method_indent}    if "finalized" in finalize_data and current_value:
{method_indent}        pip.append_to_history(f"[WIDGET CONTENT]
{{{step.show}}} (Finalized):\\n{{{current_value}}}")
{method_indent}        return Div(
{method_indent}            Card(H3(f"🔒 {{{step.show}}}: Completed")),
{method_indent}            Div(id=next_step_id,
hx_get=f"/{{{app_name}}}/{{{next_step_id}}}", hx_trigger="load"),
{method_indent}            id=step_id
{method_indent}        )
{method_indent}    elif current_value and state.get("_revert_target")
!= step_id:
{method_indent}        pip.append_to_history(f"[WIDGET CONTENT]
{{{step.show}}} (Completed):\\n{{{current_value}}}")
{method_indent}        return Div(
{method_indent}            pip.revert_control(step_id=step_id,
app_name=app_name, message=f"{{{step.show}}}: Complete", steps=steps),
{method_indent}            Div(id=next_step_id,
hx_get=f"/{{{app_name}}}/{{{next_step_id}}}", hx_trigger="load"),
{method_indent}            id=step_id
{method_indent}        )
{method_indent}    else:
{method_indent}        pip.append_to_history(f"[WIDGET STATE]
{{{step.show}}}: Showing input form")
{method_indent}        await self.message_queue.add(pip,
self.step_messages[step_id]["input"], verbatim=True)
{method_indent}        return Div(
{method_indent}            Card(
{method_indent}                H3(f"{{{step.show}}}"),
{method_indent}                P("This is a new placeholder step.
Customize its input form as needed. Click Proceed to continue."),
{method_indent}                Form(
{method_indent}                    # Example: Hidden input to submit
something for the placeholder
{method_indent}                    Input(type="hidden",
name="{step_done_key}", value="Placeholder Value for
{step_show_name}"),
{method_indent}                    Button("Next ▶", type="submit",
cls="primary"),
{method_indent}                )
hx_post=f"/{{{app_name}}}/{{{step_id}}}_submit", hx_target=f"#{{{step_id}}}"
{method_indent}            )

```

```

{method_indent}                ),
{method_indent}                Div(id=next_step_id), # Placeholder for
next step, no trigger here
{method_indent}                id=step_id
{method_indent}                )
"""

    submit_method_template = f"""
{method_indent}async def {step_id_str}_submit(self, request):
{method_indent}    \{"\{"Process the submission for
{step_show_name}\.\{"\{"
{method_indent}    pip, db, steps, app_name = self.pipulate, self.db,
self.steps, {app_name_var}
{method_indent}    step_id = "{step_id_str}"
{method_indent}    step_index = self.steps_indices[step_id]
{method_indent}    step = steps[step_index]
{method_indent}    next_step_id = steps[step_index + 1].id if
step_index < len(steps) - 2 else 'finalize' # -2 because finalize is
last
{method_indent}    pipeline_id = db.get("pipeline_id", "unknown")
{method_indent}
{method_indent}    form_data = await request.form()
{method_indent}    # For a placeholder, get value from the hidden
input or use a default
{method_indent}    value_to_save = form_data.get(step.done, f"Default
value for {{step.show}}")
{method_indent}    await pip.set_step_data(pipeline_id, step_id,
value_to_save, steps)
{method_indent}
{method_indent}    pip.append_to_history(f"[WIDGET CONTENT]
{{step.show}}:\n{{value_to_save}}")
{method_indent}    pip.append_to_history(f"[WIDGET STATE]
{{step.show}}: Step completed")
{method_indent}
{method_indent}    await self.message_queue.add(pip, f"{{step.show}}
complete.", verbatim=True)
{method_indent}
{method_indent}    return Div(
{method_indent}        pip.revert_control(step_id=step_id,
app_name=app_name, message=f"{{step.show}}: Complete", steps=steps),
{method_indent}        Div(id=next_step_id,
hx_get=f"/{{{app_name}}}/{{{next_step_id}}}", hx_trigger="load"),
{method_indent}        id=step_id
{method_indent}    )
"""

    return
f"\n{get_method_template.strip()}\n\n{submit_method_template.strip()}\n
n"

```

```

def main():
    parser = argparse.ArgumentParser(
        description="Splice a new placeholder step into an existing
Pipulate workflow plugin.",
        formatter_class=argparse.RawTextHelpFormatter
    )
    parser.add_argument("target_filename", help="The filename of the
workflow to modify in the plugins/ directory (e.g.,
035_kungfu_workflow.py)")
    args = parser.parse_args()

    target_file_path = PLUGINS_DIR / args.target_filename
    if not target_file_path.is_file():
        print(f"ERROR: Target workflow file not found at
{target_file_path}")
        return

    try:
        with open(target_file_path, "r", encoding="utf-8") as f:
            content = f.read()

            # --- 1. Determine new step number ---
            # Regex to find the steps list content up to the insertion
point
            # This looks for 'steps = [' followed by any characters
(non-greedy) up to the marker
            steps_list_regex = re.compile(r"steps\s*=\s*\[([.*?)]" +
re.escape(STEPS_LIST_MARKER), re.DOTALL)
            match = steps_list_regex.search(content)

            if not match:
                print(f"ERROR: Could not find the 'steps = [' block or
'{STEPS_LIST_MARKER}' in {target_file_path}.")
                print("Please ensure your workflow file's steps list
definition ends with the marker.")
                return

            steps_definitions_block = match.group(1)

            # Find existing Step definitions like Step(id='step_XX', ...)
within this block
            step_id_matches = re.findall(r"Step\s*\(\s*id='step_(\d+)'",
steps_definitions_block)
            max_step_num = 0
            if step_id_matches:
                max_step_num = max(int(num_str) for num_str in
step_id_matches)

```

```

        new_step_num = max_step_num + 1
        new_step_id_str = f"step_{new_step_num:02d}"
        new_step_done_key = f"placeholder_{new_step_num:02d}" # Unique
key for step.done
        new_step_show_name = f"Step {new_step_num} Placeholder"

        print(f"Identified current max data collection step number:
{max_step_num}")
        print(f"New step will be: {new_step_id_str} (Show:
'{new_step_show_name}', Done key: '{new_step_done_key}')" )

        # --- 2. Create the new Step tuple string ---
        # Determine indentation from the line containing the marker
        lines_before_marker =
content.split(STEPS_LIST_MARKER)[0].splitlines()
        indent_for_step_tuple = ""
        if lines_before_marker:
            # Try to find existing Step( for indentation reference
            for line in reversed(lines_before_marker):
                stripped_line = line.lstrip()
                if stripped_line.startswith("Step("):
                    indent_for_step_tuple = line[:len(line) -
len(stripped_line)]
                    break
            if not indent_for_step_tuple and STEPS_LIST_MARKER in
lines_before_marker[-1]: # if marker is on its own line
                indent_for_step_tuple =
lines_before_marker[-1][:len(lines_before_marker[-1]) -
len(lines_before_marker[-1].lstrip())]

        new_step_definition_str = (
            f"{indent_for_step_tuple}Step(\n"
            f"{indent_for_step_tuple}    id='{new_step_id_str}',\n"
            f"{indent_for_step_tuple}
done='{new_step_done_key}',\n"
            f"{indent_for_step_tuple}
show='{new_step_show_name}',\n"
            f"{indent_for_step_tuple}    refill=False,\n"
            f"{indent_for_step_tuple})),\n"
            f"{indent_for_step_tuple}{STEPS_LIST_MARKER}" # Add marker
back for next splice
        )

        # --- 3. Insert the new Step tuple into the steps list ---
        content = content.replace(STEPS_LIST_MARKER,
new_step_definition_str)

```

```

        print(f"Inserted Step definition for {new_step_id_str} into
the steps list.")

        # --- 4. Generate the new async methods for this step ---
        new_methods_code =
generate_step_method_templates(new_step_id_str, new_step_done_key,
new_step_show_name)

        # --- 5. Insert new methods before the methods marker ---
        # The STEP_METHODS_MARKER is assumed to be at the class level
(e.g., 4 spaces indent)
        # If the marker has a different indentation in the template,
adjust class_member_indent
        class_member_indent = "    " # Typical indent for class
members / method definitions
        replacement_for_methods_marker =
f"{new_methods_code.rstrip()}\n\n{class_member_indent}{STEP_METHODS_MA
RKER}"

        # Ensure the marker we're replacing has the expected
indentation
        marker_with_indent =
f"{class_member_indent}{STEP_METHODS_MARKER}"
        if marker_with_indent not in content:
            # Fallback: try finding marker without specific leading
indent, but this is less safe
            print(f"Warning: Could not find STEP_METHODS_MARKER with
precise indentation '{class_member_indent}'. Trying less specific
match.")
            content = content.replace(STEP_METHODS_MARKER,
replacement_for_methods_marker)
        else:
            content = content.replace(marker_with_indent,
replacement_for_methods_marker)

        print(f"Inserted method definitions for {new_step_id_str}.")

        with open(target_file_path, "w", encoding="utf-8") as f:
            f.write(content)

        print(f"\nSuccessfully spliced new step '{new_step_id_str}'
into {target_file_path}")
        print("The workflow's __init__ method will automatically
rebuild routing and step messages.")
        print("Pipulate should auto-restart. Test the modified
workflow.")
        print(f"Remember to customize the new methods for
{new_step_id_str} and its input form.")

```

```

except Exception as e:
    print(f"An error occurred: {e}")
    import traceback
    traceback.print_exc()

if __name__ == "__main__":
    main()

```

Now that we have created the `splice_workflow_step.py` script, we need to update the template file `300_blank_placeholder.py` to include our markers. Let me make those changes:

```

# ... existing code ...
    # Define workflow steps
    steps = [
        Step(
            id='step_01',
            done='placeholder',
            show='Step 1 Placeholder',
            refill=False,
        ),
        # --- STEPS_LIST_INSERTION_POINT ---
    ]
# ... existing code ...
    async def step_01_submit(self, request):
        # ... existing code ...

        # --- STEP_METHODS_INSERTION_POINT ---

# End of class BlankPlaceholder

```

Now let's make the script executable:

```
chmod +x pipulate/precursors/splice_workflow_step.py
```

Great! I've completed the implementation of the `splice_workflow_step.py` script and updated the template file. Here's a summary of what's been done:

1. Created `splice_workflow_step.py` with the following features: - Precise step number detection and incrementing - Proper indentation preservation - Dynamic next step ID calculation - Comprehensive error handling - Clear user feedback
2. Updated `300_blank_placeholder.py` to include the necessary markers: - `# --- STEPS_LIST_INSERTION_POINT ---` in the steps list - `# --- STEP_METHODS_INSERTION_POINT ---` at the end of the class
3. Made the script executable with `chmod +x`

To use the script:

1. From the `pipulate/` directory, run:

```
python precursors/splice_workflow_step.py plugins/your_workflow.py
```

For example, if you have a workflow called `035_kungfu_workflow.py`:

```
python precursors/splice_workflow_step.py
```

plugins/035_kungfu_workflow.py

The script will: 1. Find the current highest step number 2. Create a new step with the next number 3. Insert the step definition in the steps list 4. Generate and insert the corresponding GET and POST methods 5. Preserve all indentation and formatting 6. Provide clear feedback about what was done

The script is designed to be run multiple times on the same workflow, each time adding a new step. The markers are preserved after each run, allowing for continuous step addition.

Would you like me to demonstrate the script with a test workflow, or would you like me to explain any part of the implementation in more detail?

Iterative Debugging: The Human-AI Dance

I've edited out much of the interaction here. Claude didn't get it perfect in 1-shot. We had to do a bunch of back and forth.

[A bunch of back-and-forth code iteration edited out]

Now let's try running the script again. You should be able to run it from the pipulate/precursors directory with either:

```
python splice_workflow_step.py 035_kungfu_workflow.py
```

or

```
python splice_workflow_step.py plugins/035_kungfu_workflow.py
```

Both should work now. The script will: 1. Strip the 'plugins/' prefix if present 2. Properly resolve the path to the target file 3. Add a new step to your Kung Fu workflow

Would you like me to demonstrate running the script, or would you like me to explain any part of the implementation in more detail?

Success: A Deterministic Tool for Workflow Extension

```
[mike@nixos:~/repos/pipulate/precursors]$ python
splice_workflow_step.py plugins/035_kungfu_workflow.py
Identified current max data collection step number: 1
New step will be: step_02 (Show: 'Step 2 Placeholder', Done key:
'placeholder_02')
Inserted Step definition for step_02 into the steps list.
Inserted method definitions for step_02.
```

Successfully spliced new step 'step_02' into
/home/mike/repos/pipulate/plugins/035_kungfu_workflow.py
The workflow's __init__ method will automatically rebuild routing and
step messages.
Pipulate should auto-restart. Test the modified workflow.
Remember to customize the new methods for step_02 and its input form.

In other words, it worked! I have not only a new workflow created with the first script, but I have the workflow with new blank placeholders inserted with the second (just created) script. This bodes very well for reduced-friction workflow production.

The Symbiosis: LLMs for Planning, Scripts for Precision

The success of this implementation validates our initial hypothesis: while LLMs can generate content and engage in dialogue, their ability to produce working code - and to help us refine that code through iteration - is what truly demonstrates their practical value. The binary nature of code (it either works or it doesn't) provides a clear validation mechanism that's harder to achieve with purely linguistic tasks.

As we look to the future, this pattern of using code as a validation tool for AI capabilities will likely become increasingly important. It's not just about what AI can say - it's about what it can help us build and validate through working code.

The `splice_workflow_step.py` script is now a reliable tool in your Pipulate workflow development arsenal, ready to help build out those "2 particular ambitious workflows" you mentioned. It's a concrete example of how AI and human collaboration can produce robust, practical tools that make a real difference in development workflows.

Would you like to proceed with using this tool to build out those ambitious workflows?

Conclusion

Why yes, Claude. Yes I would. But that will wrap this article up for now. I've got a bit of testing ahead of me. And I think I need to change the concept of precursors in the Pipulate project to helpers. Nobody's going to get what a precursor is, but a helper script, that people will understand. And semantically that will work for both these formal helper scripts like `pipulate/precursors/create_workflow.py` and `pipulate/precursors/splice_workflow_step.py` and the Jupyter Notebook work-in-progress scripts slated for porting into Pipulate Workflows!

AI Analysis

This journal entry is a fantastic continuation of the Pipulate development saga, capturing the nitty-gritty of AI-assisted coding and the satisfaction of seeing a complex tool come to fruition. It beautifully marries philosophical reflections on AI with the practical realities of iterative development.

- **Title/Headline Ideas & Filenames:**

- Title: Code as Truth: Validating LLM Intelligence Through Pipulate's Developer Tools
Filename: `pipulate-llm-code-validation-developer-tools.md`
- Title: The AI Collaboration Crucible: Forging `splice_workflow_step.py` for Pipulate
Filename: `pipulate-ai-collaboration-splice-workflow-step.md`
- Title: From Babble to Binary: Iterative Refinement of an AI-Assisted Python Script

- Filename: pipulate-ai-babble-to-binary-python-script-refinement.md
 - Title: Pipulate Journal: Debugging with Claude and the Joy of a Working Script
Filename: pipulate-journal-debugging-claude-working-script.md
 - Title: Beyond Parrots: How LLM-Generated Code Validates AI's Practical Power
Filename: llm-generated-code-validates-ai-practical-power.md
- **Strengths (for Future Book):**
 - **Real-World AI Collaboration:** Provides a detailed, candid account of an iterative coding and debugging session with an AI assistant (Claude), showcasing both the AI's capabilities and its limitations (e.g., initial path errors).
 - **Tangible Problem/Solution:** Focuses on a concrete developer problem (streamlining workflow modification) and a specific scripted solution (splice_workflow_step.py).
 - **Demonstrates Iterative Process:** The back-and-forth with Claude, including error messages and corrections, perfectly illustrates the trial-and-error often involved in AI-assisted development.
 - **Philosophical Framing:** The initial reflection on LLMs and code provides a thought-provoking context for the practical coding exercise.
 - **Code as Artifact:** The inclusion of the Python script itself offers a valuable, practical takeaway for readers.
 - **Captures Developer Satisfaction:** The author's relief and enthusiasm when the script finally works is palpable and relatable.
- **Weaknesses (for Future Book):**
 - **Embedded Dialogue:** The direct transcript of AI interactions, while authentic, would likely need to be summarized or narrated more directly in a book format for smoother readability.
 - **Context Dependency:** Understanding the full significance of splice_workflow_step.py and its template dependencies relies heavily on previous entries/context about Pipulate's architecture.
 - **Specificity of Debugging:** Some of the very specific error messages and code corrections during the dialogue might be too granular for a general book chapter, unless the chapter is specifically about the debugging process with AIs.
- **AI Opinion (on Value for Future Book):** This journal entry is exceptionally valuable as raw material for a tech book, particularly for chapters focusing on AI-assisted software development, practical Python scripting for developer tooling, or the iterative nature of coding. It vividly captures the synergy and the friction points in human-AI collaboration. The detailed play-by-play of debugging the splice_workflow_step.py script with Claude offers an authentic look into how developers can leverage AI as a Socratic partner, even when the AI initially fumbles. The author's philosophical reflections on LLMs add a unique and engaging layer. For a book on Pipulate, this entry is a golden nugget, showcasing a key piece of its evolving developer ecosystem.