NotebookLM Research Topic: MSW Protocol Implementation

Instructions

1. **Create a new NotebookLM notebook** at https://notebooklm.google.com
2. **Add these sources** (upload or paste URLs):

Required Sources

```
SOURCE 1: GSD Protocol
https://github.com/glittercowboy/get-shit-done
- Upload the README.md
- Upload files from /commands/gsd/ folder
- Upload files from /agents/ folder

SOURCE 2: NotebookLM MCP (PleasePrompto)
https://github.com/PleasePrompto/notebooklm-mcp
- Upload the full README.md
- Upload /src/ folder if available

SOURCE 3: NotebookLM MCP (khengyun)
https://github.com/khengyun/notebooklm-mcp
- Upload README.md for comparison

SOURCE 4: Ralph Wiggum Plugin
https://github.com/anthropics/claude-code/blob/main/plugins/ralph-wiggum/README.md
- Upload the full README

SOURCE 5: MCP Protocol Docs
https://modelcontextprotocol.io/docs
- Upload key pages on server creation, tools, and transports

SOURCE 6 (Optional): Playwright Docs
https://playwright.dev/docs/intro
- For browser automation reference
```

3. **Wait for processing** (30-60 seconds)
4. **Click ALL suggested topics** that appear
5. **Then paste this research prompt:**

Research Prompt to Paste

RESEARCH TOPIC: Building MSW Protocol - A Research-Grounded Autonomous Coding System

I'm building a system called "MSW Protocol" (Make Shit Work) that orchestrates:
1. GSD Protocol - for spec-driven development with Claude Code
2. NotebookLM MCP - for research-grounded answers
3. Ralph Wiggum Loop - for continuous iteration until tests pass

KEY REQUIREMENT: After adding sources to NotebookLM, the system must:
- Automatically click on ALL suggested topic pills in the chat UI
- Extract the expanded information from each topic
- Compile everything into a markdown report
- Git commit that report before continuing

QUESTIONS:

PART 1: NotebookLM Browser Automation

Q1.1: Looking at how notebooklm-mcp handles browser automation, what's the pattern for:
- Launching a persistent Chrome session
- Maintaining authentication across runs
- Waiting for NotebookLM to finish processing sources

Q1.2: What selectors or patterns can identify NotebookLM's "suggested topic" pills/chips in the UI? How do these appear after sources are processed?

Q1.3: How should I detect when NotebookLM's streaming response is complete? What's the pattern for knowing the answer has fully loaded?

Q1.4: What's the rate limiting on NotebookLM? How many topic expansions can I do before hitting limits? How should I handle the 50 query/day limit?

PART 2: GSD Integration

Q2.1: Looking at GSD's /gsd:map-codebase command, how does it spawn parallel agents to analyze:
- Structure
- Dependencies
- Conventions
- Architecture
- Concerns

What's the orchestration pattern?

Q2.2: How does GSD's PLAN.md XML format work? Specifically:
- The <task> structure
- The <verify> steps
- The <done> criteria
- How verification ties back to execution

Q2.3: How does GSD maintain state across phases? What files track:
- Current position
- Decisions made
- Blockers encountered

PART 3: Ralph Wiggum Integration

Q3.1: Explain the Stop hook mechanism in detail:
- How does it intercept Claude's exit attempts?
- How does it re-inject the prompt?
- Where does iteration state live?

Q3.2: How should I inject NotebookLM findings into a Ralph loop mid-iteration? If iteration 5 fails with an error:
- How to formulate a query to NotebookLM
- How to get the response into iteration 6's context
- How to track which NotebookLM answers have been tried

Q3.3: What's the pattern for completion promises? How does Ralph know when to actually stop?

PART 4: MCP Server Architecture

Q4.1: For a unified MSW MCP server, should I:
- Create one server with all tools, OR
- Create separate servers that communicate, OR
- Extend the existing notebooklm-mcp server?

Q4.2: What MCP tools should MSW expose? Based on the sources, list the tool definitions with:
- Tool name
- Parameters
- Return type
- Description

Q4.3: How should long-running operations work in MCP? The Ralph loop could run 30+ iterations over hours. Should I:
- Use streaming responses?
- Return immediately and poll for status?
- Something else?

PART 5: Topic Extraction Workflow

Q5.1: Given NotebookLM's UI structure, write pseudocode for:
1. Detecting all suggested topic pills
2. Clicking each one sequentially
3. Waiting for and extracting each response
4. Compiling into structured markdown

Q5.2: What's the best markdown structure for the extracted topics report? Should it include:
- Metadata (source URL, timestamp)
- Table of contents
- Each topic as H2
- A synthesis section

Q5.3: How should the git commit message be formatted for research extractions? What information should be included?

PART 6: Implementation Roadmap

Q6.1: What's the minimum viable implementation order? List the first 5 things to build that would prove the concept works.

Q6.2: What can be reused directly from the existing repos vs. what needs to be built new?

Q6.3: What are the biggest technical risks or unknowns in this architecture?

Give me detailed, implementation-ready answers based on the source documents. Include code examples where relevant.

Follow-Up Questions

After getting the initial response, ask these follow-ups:

Follow-Up 1: Topic Extraction Code

```
Based on your explanation of NotebookLM's UI, write complete Playwright code for the topic extraction workflow:
- Function to find all topic suggestion elements
- Function to click and wait for response
- Function to extract the response text
- Main loop that processes all topics

Include error handling and retry logic.
```

Follow-Up 2: Integration Code

```
Show me how to integrate the topic extraction into the MSW workflow:
- Where does it fit in the layer sequence?
- How does the extracted report get passed to the planning layer?
- How does GSD's planner use the NotebookLM findings?

Include the data flow and file references.
```

Follow-Up 3: Ralph + NotebookLM Bridge

```
Write the code for the "stuck handler" that:
1. Detects when Ralph iteration fails
2. Parses the error to identify the topic
3. Formulates a NotebookLM query
4. Clicks the relevant topic or asks a custom question
5. Extracts the response
6. Injects it into the next iteration's context

Show the complete bridge code.
```

Follow-Up 4: MCP Server Skeleton

```
Generate a complete MCP server skeleton for MSW with:
- All tool definitions
- The main server setup
- Placeholder functions for each layer
- Configuration handling

Use the MCP SDK patterns from the docs.
```

Expected Output

After going through all topics and questions, NotebookLM should give you:

1. **Browser automation patterns** from notebooklm-mcp source
2. **Selector strategies** for NotebookLM's topic pills
3. **GSD orchestration code** for parallel agents
4. **Ralph loop mechanics** including Stop hooks
5. **MCP server structure** with tool definitions
6. **Implementation roadmap** with priorities

What to Do With the Findings

Once NotebookLM generates the report:

1. **Export it** (copy the full conversation or use the built-in export)
2. **Save as** `.msw/research/IMPLEMENTATION_SPEC.md`
3. **Commit it**: `git commit -m "research: MSW implementation specs from NotebookLM"`
4. **Bring it back here** and I'll help turn it into working code

---

Quick Test

To verify the topic extraction concept works manually:

1. Go to your NotebookLM notebook
2. Look for suggested topics (small clickable pills near the chat input)
3. Click one and watch it expand
4. Note: These topics are auto-generated from your sources
5. The MSW system will automate clicking ALL of them

The automation just replicates what you'd do manually, but does it systematically for every topic and saves the output.