MSW Protocol: NotebookLM Auto-Conversation Engine

Core Concept

NotebookLM's chat doesn't just answer questions — it **suggests questions** based on your sources. These pop up as clickable prompts. Most people ignore them or click one manually.

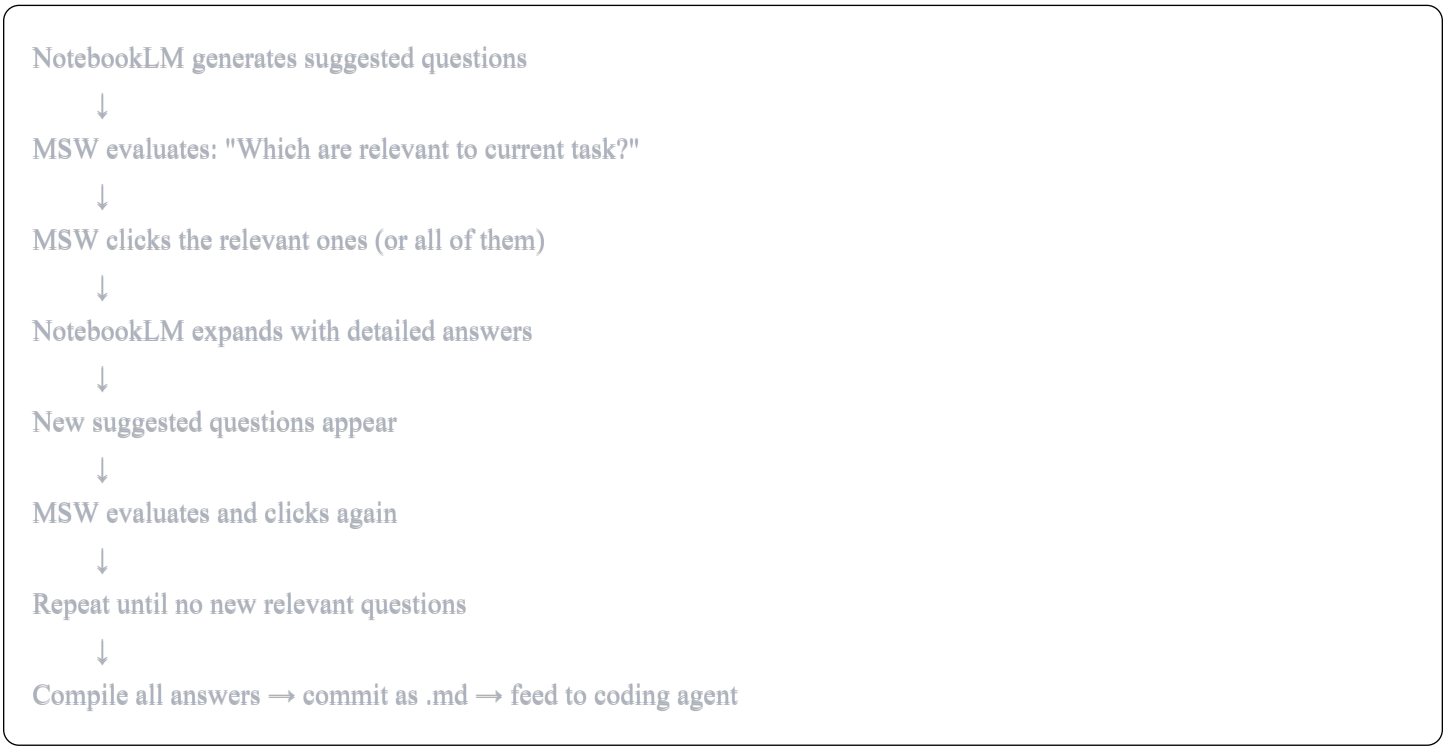
MSW clicks them all. Automatically. Repeatedly. Until the knowledge is exhausted.

Then, when your coding agent has questions, MSW injects those into the same chat and extracts the answers.

This creates a **bidirectional conversation loop** between your coding agent and NotebookLM — no human in the middle.

The Two Flows

Flow 1: NotebookLM → Coding Agent (Knowledge Extraction)

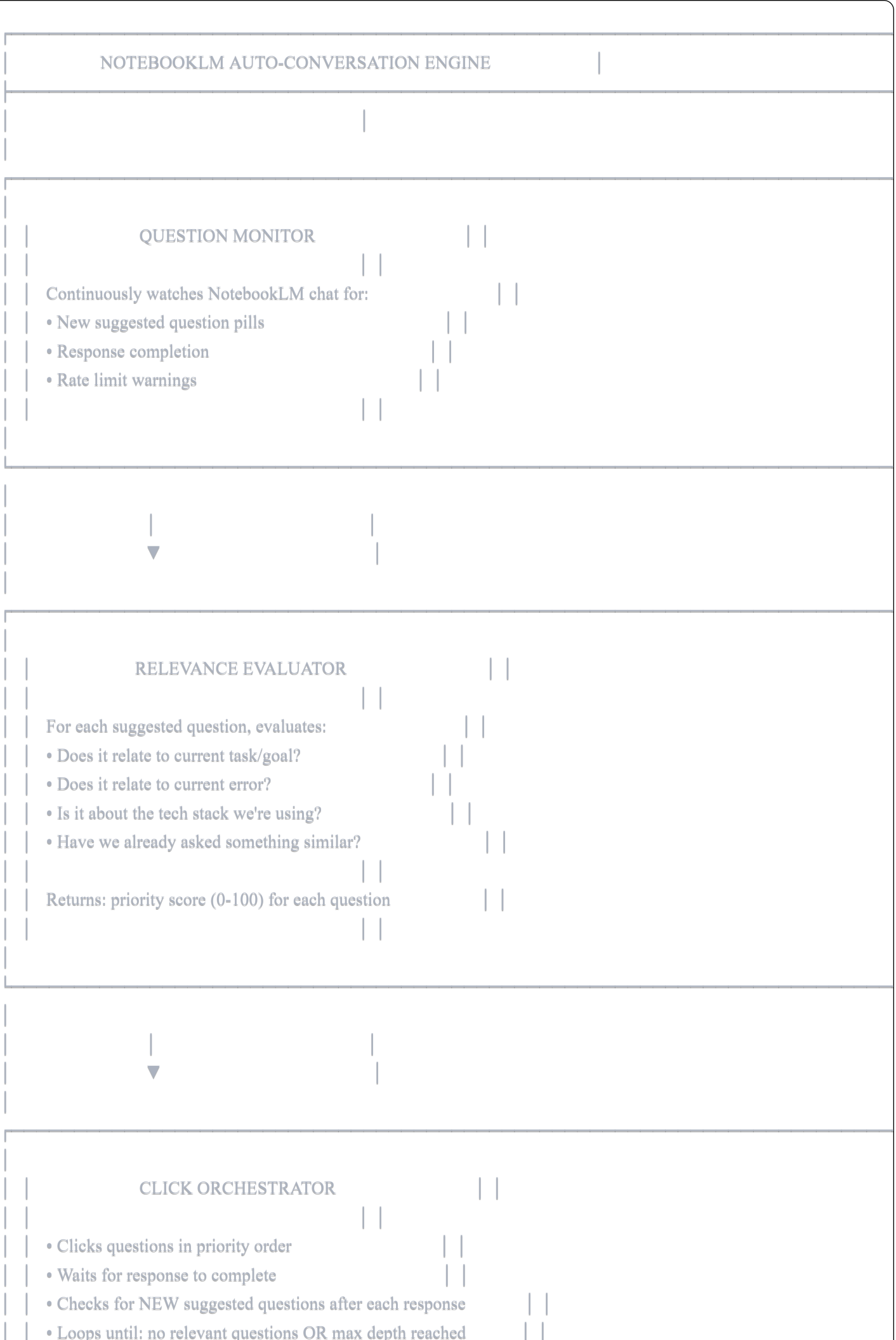


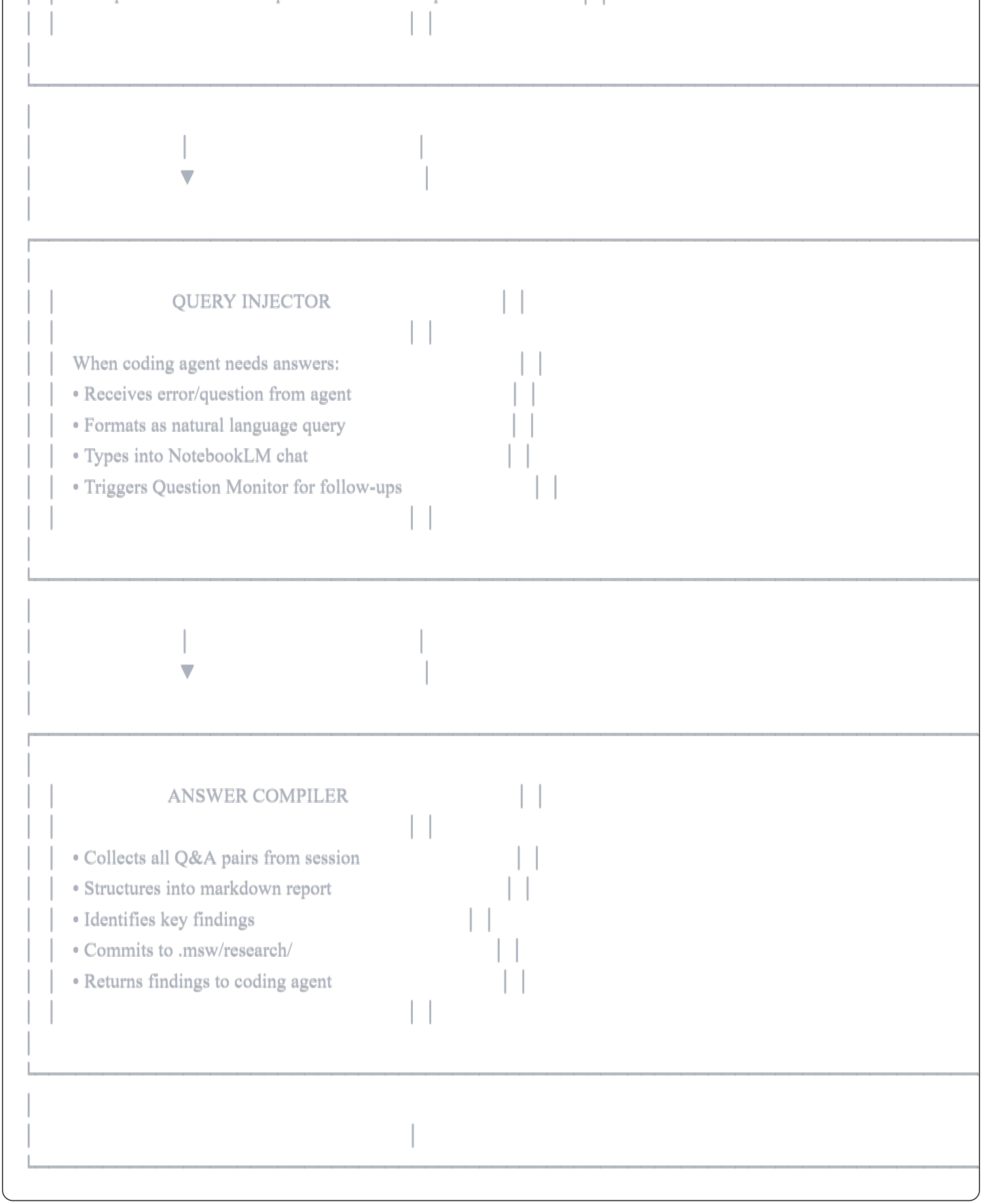
Flow 2: Coding Agent → NotebookLM (Error Resolution)

Coding agent hits error or has question
↓
MSW formulates query from error context
↓
MSW types query into NotebookLM chat
↓
NotebookLM responds with grounded answer
↓
New suggested follow-up questions appear
↓
MSW evaluates: "Do any of these help with the error?"
↓
MSW clicks relevant follow-ups
↓
Extract full answer chain → inject into agent's next iteration

---

Architecture


NOTEBOOKLM AUTO-CONVERSATION ENGINE
QUESTION MONITOR
Continuously watches NotebookLM chat for:
• New suggested question pills
• Response completion
• Rate limit warnings
RELEVANCE EVALUATOR
For each suggested question, evaluates:
• Does it relate to current task/goal?
• Does it relate to current error?
• Is it about the tech stack we're using?
• Have we already asked something similar?
Returns: priority score (0-100) for each question
CLICK ORCHESTRATOR
• Clicks questions in priority order
• Waits for response to complete
• Checks for NEW suggested questions after each response
• Loops until: no relevant questions OR max depth reached



The Auto-Expansion Loop

This is the core algorithm:

javascript

```javascript
async function autoExpandConversation(context) {
  const { taskGoal, currentError, maxDepth = 10, maxQuestions = 50 } = context;

  const askedQuestions = new Set();
  const allAnswers = [];
  let depth = 0;
  let totalQuestions = 0;

  while (depth < maxDepth && totalQuestions < maxQuestions) {
    // 1. Find all currently visible suggested questions
    const suggestedQuestions = await findSuggestedQuestions();

    if (suggestedQuestions.length === 0) {
      console.log('No more suggested questions. Expansion complete.');
      break;
    }

    // 2. Filter out questions we've already asked
    const newQuestions = suggestedQuestions.filter(q => !askedQuestions.has(q.text));

    if (newQuestions.length === 0) {
      console.log('All questions already asked. Expansion complete.');
      break;
    }

    // 3. Evaluate relevance of each question
    const scoredQuestions = await evaluateRelevance(newQuestions, {
      taskGoal,
      currentError,
      previousAnswers: allAnswers
    });

    // 4. Filter to only relevant questions (score > threshold)
    const relevantQuestions = scoredQuestions
      .filter(q => q.score > 30)
      .sort((a, b) => b.score - a.score);

    if (relevantQuestions.length === 0) {
      console.log('No relevant questions remaining. Expansion complete.');
      break;
    }

    // 5. Click each relevant question and extract answer
    for (const question of relevantQuestions) {
```

```
    if (totalQuestions >= maxQuestions) break;

    console.log(`[${totalQuestions + 1}] Clicking: "${question.text}" (score: ${question.score})`);

    // Click the question
    await question.element.click();

    // Wait for response to complete
    const answer = await waitForAndExtractResponse();

    // Store the Q&A pair
    askedQuestions.add(question.text);
    allAnswers.push({
      question: question.text,
      answer: answer,
      score: question.score,
      depth: depth
    });

    totalQuestions++;

    // Small delay to avoid rate limiting
    await sleep(1500);
  }

  depth++;

  // After clicking questions, new suggested questions may appear
  // Loop continues to find and evaluate those
}

return {
  answers: allAnswers,
  totalQuestions,
  depth,
  complete: true
};
}
```

Relevance Evaluation

The system needs to decide WHICH suggested questions to click. Not all are relevant.

javascript

```javascript
async function evaluateRelevance(questions, context) {
  const { taskGoal, currentError, previousAnswers } = context;

  // Build evaluation context
  const evalPrompt = `
You are evaluating NotebookLM's suggested questions for relevance.

CURRENT TASK: ${taskGoal}
CURRENT ERROR (if any): ${currentError || 'None'}
PREVIOUS ANSWERS COLLECTED: ${previousAnswers.length}

For each suggested question, score 0-100 based on:
- Direct relevance to the task (0-40 points)
- Relevance to the current error (0-30 points)
- Likely to provide actionable implementation details (0-20 points)
- Not redundant with previous answers (0-10 points)

QUESTIONS TO EVALUATE:
${questions.map((q, i) => `${i + 1}. "${q.text}"`).join('\n')}

Return JSON array: [{"index": 1, "score": 85, "reason": "..."}, ...]
`;

  // Use local LLM to evaluate (fast, no external API needed)
  const evaluation = await localLLM.evaluate(evalPrompt);

  // Merge scores back into questions
  return questions.map((q, i) => ({
    ...q,
    score: evaluation[i]?.score || 0,
    reason: evaluation[i]?.reason || ''
  }));
}
```

Relevance Signals

Signal	Weight	Example
Contains current tech stack keywords	+20	"How to use JWT with Express" when stack is Express
Contains error-related terms	+30	"handling authentication errors" when error is auth-related
Implementation-focused	+15	"step-by-step guide to..." vs "what is..."
Already covered by previous answer	-40	Similar question already expanded
Too generic	-20	"What are best practices?" with no specificity
Matches task goal keywords	+25	Task is "add auth", question mentions "authentication"

Bidirectional Chat: Agent → NotebookLM

When the coding agent has a question or error, MSW injects it into the NotebookLM chat:

javascript

```javascript
async function askNotebookLM(agentQuery) {
  const { error, question, codeContext, attemptedSolution } = agentQuery;

  // 1. Formulate a natural language query
  const query = formatQueryForNotebookLM({
    error,
    question,
    codeContext,
    attemptedSolution
  });

  // 2. Type it into the NotebookLM chat input
  const chatInput = await page.waitForSelector('textarea[placeholder*="Ask"], [data-testid="chat-input"]');
  await chatInput.fill(query);

  // 3. Submit the query
  await page.keyboard.press('Enter');

  // 4. Wait for response
  const answer = await waitForAndExtractResponse();

  // 5. Check for new suggested follow-up questions
  const followUps = await findSuggestedQuestions();

  // 6. Evaluate which follow-ups would help resolve the error
  const relevantFollowUps = await evaluateRelevance(followUps, {
    taskGoal: question,
    currentError: error
  });

  // 7. Auto-expand relevant follow-ups
  const additionalAnswers = [];
  for (const followUp of relevantFollowUps.filter(q => q.score > 50)) {
    await followUp.element.click();
    const followUpAnswer = await waitForAndExtractResponse();
    additionalAnswers.push({
      question: followUp.text,
      answer: followUpAnswer
    });
  }

  // 8. Compile full response
  return {
    primaryAnswer: answer,
```

```
    followUpAnswers: additionalAnswers,
    query: query
  };
}

function formatQueryForNotebookLM({ error, question, codeContext, attemptedSolution }) {
  let query = '';

  if (error) {
    query += `I'm getting this error:\n\`\`\`\n${error}\n\`\`\`\n\n`;
  }

  if (codeContext) {
    query += `In this code:\n\`\`\`\n${codeContext}\n\`\`\`\n\n`;
  }

  if (attemptedSolution) {
    query += `I tried: ${attemptedSolution}\n\n`;
  }

  if (question) {
    query += question;
  } else {
    query += 'What is the correct way to fix this based on the documentation?';
  }

  return query;
}
```

Full Conversation Session

Here's what a complete MSW research session looks like:

javascript

```javascript
async function conductResearchSession(config) {
  const {
    notebookUrl,
    taskGoal,
    codebaseContext,
    agentQuestions = []
  } = config;

  const session = {
    startTime: new Date(),
    answers: [],
    agentQAs: []
  };

  // ================================================
  // PHASE 1: Initial Knowledge Extraction
  // ================================================

  console.log('Phase 1: Extracting knowledge from NotebookLM...');

  // Navigate to notebook
  await page.goto(notebookUrl);
  await waitForNotebookLoad();

  // Run auto-expansion on suggested questions
  const extraction = await autoExpandConversation({
    taskGoal: taskGoal,
    currentError: null,
    maxDepth: 10,
    maxQuestions: 30
  });

  session.answers.push(...extraction.answers);

  console.log(`Extracted ${extraction.answers.length} Q&A pairs in ${extraction.depth} depth levels`);

  // ================================================
  // PHASE 2: Answer Agent's Specific Questions
  // ================================================

  if (agentQuestions.length > 0) {
    console.log(`Phase 2: Answering ${agentQuestions.length} agent questions...`);

    for (const agentQ of agentQuestions) {
```

```
    const response = await askNotebookLM(agentQ);

    session.agentQAs.push({
      agentQuery: agentQ,
      notebookResponse: response
    });

    // The askNotebookLM function already auto-expands relevant follow-ups
  }
}

// ================================================
// PHASE 3: Compile and Commit
// ================================================

console.log('Phase 3: Compiling research report...');

const report = compileResearchReport(session);
const filename = await saveAndCommit(report, notebookUrl);

session.endTime = new Date();
session.reportFile = filename;

return session;
}
```

Continuous Monitoring Mode

For Ralph loops, MSW can run in continuous monitoring mode — watching for agent errors and automatically querying NotebookLM:

javascript

```javascript
async function continuousResearchMode(config) {
  const { notebookUrl, agentOutputPipe, maxQueriesPerHour = 20 } = config;

  let queriesThisHour = 0;
  const hourStart = Date.now();

  // Watch the agent's output for errors
  agentOutputPipe.on('error', async (errorContext) => {
    // Rate limit check
    if (Date.now() - hourStart > 3600000) {
      queriesThisHour = 0;
    }
    if (queriesThisHour >= maxQueriesPerHour) {
      console.log('Rate limit reached. Skipping NotebookLM query.');
      return;
    }

    console.log(`Agent error detected. Querying NotebookLM...`);

    // Query NotebookLM about the error
    const response = await askNotebookLM({
      error: errorContext.message,
      codeContext: errorContext.code,
      attemptedSolution: errorContext.lastAttempt
    });

    queriesThisHour++;

    // Inject the response back into the agent's context
    await injectIntoAgentContext(response);

    console.log('NotebookLM response injected. Agent continuing...');
  });

  // Also watch for explicit questions from the agent
  agentOutputPipe.on('question', async (questionContext) => {
    // Same flow as error handling
    const response = await askNotebookLM({
      question: questionContext.question,
      codeContext: questionContext.context
    });

    await injectIntoAgentContext(response);
  });
}
```

Output: The Research Report

After a session, MSW produces a comprehensive markdown file:

markdown

```
# NotebookLM Research Session

**Notebook:** https://notebooklm.google.com/notebook/abc123
**Task Goal:** Implement JWT authentication with refresh tokens
**Session Duration:** 4 minutes
**Questions Explored:** 23
**Agent Queries Answered:** 3

---

## Automatically Expanded Topics

### 1. How does JWT token validation work?

**Relevance Score:** 92

[NotebookLM's detailed answer about JWT validation...]

### 2. What are the security considerations for refresh tokens?

**Relevance Score:** 88

[NotebookLM's detailed answer about refresh token security...]

### 3. How should tokens be stored on the client side?

**Relevance Score:** 85

[Answer about httpOnly cookies vs localStorage...]

... (20 more auto-expanded topics)

---

## Agent Error Resolutions

### Error 1: "jwt.sign is not a function"

**Agent's Code:**
```javascript
const token = jwt.sign(payload, secret);
```

NotebookLM Response:
```

Based on your documentation, you're using the `jose` library, not `jsonwebtoken`.
The correct syntax is:
```javascript
import * as jose from 'jose';
const token = await new jose.SignJWT(payload)
  .setProtectedHeader({ alg: 'HS256' })
  .sign(secret);
```

Follow-up Expanded: "What's the difference between jose and jsonwebtoken?"

[Additional context about ESM compatibility...]

Key Findings Summary

1. Use `jose` library instead of `jsonwebtoken` for ESM compatibility
2. Store tokens in httpOnly cookies, not localStorage
3. Implement refresh token rotation for security
4. Token expiry should be 15 minutes for access, 7 days for refresh

---

Auto-generated by MSW Protocol

Configuration

yaml

```yaml
# .msw/config.yaml

auto_conversation:
  # Initial extraction settings
  extraction:
    max_depth: 10          # How many levels of follow-up questions to explore
    max_questions: 50      # Maximum questions to auto-expand
    relevance_threshold: 30 # Minimum score (0-100) to click a question

  # Agent query settings
  agent_queries:
    auto_expand_followups: true    # Auto-click relevant follow-ups after agent query
    followup_threshold: 50         # Higher threshold for follow-ups (more selective)
    max_followups_per_query: 5     # Limit follow-ups per agent question

  # Rate limiting
  rate_limits:
    queries_per_hour: 40           # Stay under NotebookLM's limits
    delay_between_clicks: 1500     # ms between question clicks

  # Continuous mode
  continuous:
    enabled: false                 # Enable for Ralph loop integration
    watch_for_errors: true
    watch_for_questions: true
```

Integration with Ralph Loop

javascript

```javascript
// In Ralph loop execution
async function executeWithNotebookLMFeedback(plan, config) {
  const { notebookUrl } = config;

  // Start NotebookLM session
  const nlmSession = await initNotebookLMSession(notebookUrl);

  // Initial research extraction
  console.log('Extracting initial research from NotebookLM...');
  const initialResearch = await nlmSession.autoExpandConversation({
    taskGoal: plan.goal,
    maxQuestions: 20
  });

  // Inject research into plan context
  plan.context.notebookLMFindings = initialResearch.answers;

  // Start Ralph loop with NotebookLM integration
  let iteration = 0;
  while (iteration < plan.maxIterations) {
    iteration++;

    const result = await executeIteration(plan);

    if (result.success) {
      console.log(`Success on iteration ${iteration}`);
      break;
    }

    if (result.error) {
      console.log(`Error on iteration ${iteration}. Querying NotebookLM...`);

      // Query NotebookLM about the error
      const resolution = await nlmSession.askNotebookLM({
        error: result.error,
        codeContext: result.codeContext,
        attemptedSolution: result.attempted
      });

      // Inject resolution into next iteration's context
      plan.context.lastNotebookLMResponse = resolution;

      console.log('NotebookLM response injected. Continuing...');
    }
  }
```

```
  }

  // Final report
  await nlmSession.compileAndCommit();

  return { iterations: iteration, success: result.success };
}
```

Why This Matters

The manual version: You see a suggested question, decide if it's relevant, click it, read the answer, maybe click a follow-up, then copy something useful back to your agent.

The MSW version: All of that happens automatically. Every relevant question gets explored. Every error triggers a grounded lookup. Every answer gets compiled and committed.

The result: Your coding agent has access to everything NotebookLM knows about your documentation — not just the questions you thought to ask, but the questions NotebookLM thought were important too.

That's the difference between "AI that kind of works" and "AI that reliably ships."