Week 7: Evals - Testing Your Agent Before Demo Day

01. Session Goals

- Understand why you need to test your agent before trusting it
- Create a "golden dataset" of 5 expected input/output pairs
- Run your agent against real test cases and compare results
- Learn why generic evals don't work and domain-specific ones do

02. Block 1: Theory - Why Test Your Agent? (30 min)

The Problem
You built an agent. It works in your demos. But does it actually work?

Traditional software: input → same output every time → easy to test

AI agents: input → different output each time → how do you test that?

The Solution: A Golden Dataset
A "golden dataset" is just a fancy name for a simple thing:

A list of inputs you expect your agent to handle, paired with the outputs you'd consider "good enough."

That's it. No infrastructure. No complex tooling. Just:

Input	What Good Looks Like
"Score this lead: VP Engineering at 500-person tech company"	Score 70-90, mentions seniority, mentions company size
"Summarize this 3-page PDF"	Captures main points, under 200 words, no hallucinations
"Classify this support ticket"	Correct category, appropriate priority

Why Off-the-Shelf Evals Don't Work
You might think: "Can't I just use some eval framework that scores my agent automatically?"

No. Here's why (from Hamel Husain's eval research (https://hamel.dev/blog/posts/evals-faq/)):

Off-the-Shelf Eval	Why It Fails
"Helpfulness score 1-10"	What does "helpful" mean for YOUR use case?
"Coherence rating"	Your customer support agent could be coherent but wrong
"Safety check"	Passes safety but gives bad business advice
"Factual accuracy"	Checks facts but misses the actual task

Generic metrics create false confidence. Your agent could score 95% on a generic eval and still be useless for your actual workflow.

The only eval that matters is: Does it do what YOU need it to do?

The Right Approach

1. Build evals WHILE building, not after
   - Don't wait until you're "done" to create test cases
   - Every time you build a new capability, immediately create 3 eval questions
   - Example: Build an MCP tool → Immediately create 3 queries to test it
   - This catches problems early when they're easy to fix
2. Start with error analysis, not infrastructure
   - Run your agent 20-50 times on real inputs
   - Manually look at the outputs
   - Ask: "Would I accept this if a human produced it?"
   - Spend 30 minutes doing this before building anything
3. Define YOUR pass/fail criteria
   - What makes an output "good enough" for your use case?
   - Be specific. Not "good summary" but "captures the 3 main points"
   - Write it down. This becomes your eval criteria.
4. Use the Three-Query Pattern
   - Create at least 3 test queries for each capability:
   - 2 basic queries: Simple, should definitely work (e.g., "How many companies are in the database?")
   - 1 complex query: Tests reasoning and synthesis (e.g., "Which AI startups are most likely to raise Series B?")
   - The complex query reveals whether your agent can think, not just retrieve
5. Run and compare
   - Run your agent on each input
   - Compare actual output to expected
   - Note: You're looking for "close enough," not "exact match"

Test Prediction, Not Just Retrieval

The best evals test whether your agent can reason and synthesize, not just fetch data.

Eval Type	Bad Example	Good Example
Data retrieval	"List all Series A companies"	"Which companies are most likely to raise Series B?"
Research	"Find info about Acme Corp"	"Would Acme Corp be a good acquisition target? Why?"
Analysis	"What's the average deal size?"	"Is the market heating up or cooling down? Support your answer."

Prediction evals are powerful because:

- They require the agent to synthesize multiple data points
- They expose gaps in reasoning, not just gaps in retrieval
- They're closer to real business questions

Example from the startup funding database:

```
Input: "Rank the AI coding tools by likelihood of getting Series B. Explain your reasoning."

Good output must:
- Consider funding amount vs. industry median
- Factor in time since founding to Series A
- Weigh investor track record
- Provide explicit reasoning for each ranking
```

Context Management Criteria (For Data Agents)

If your agent queries databases or large datasets, add these pass/fail criteria:

Criterion	Pass	Fail
Acknowledges limits	"Showing top 100 of 45,000 results"	Presents limited results as complete
Uses appropriate limits	Adds LIMIT clause for exploration	Returns unbounded results
Tracks truncation	Notes when previous results were limited	Forgets and makes claims based on incomplete data
Aggregates first	Starts with GROUP BY, then drills down	Tries to load entire dataset

These criteria prevent your agent from making confident claims based on incomplete data.

Use Binary Pass/Fail, Not Scales

From Hamel's research: Don't use 1-10 scales. Use pass/fail.

Approach	Problem
"Rate this output 1-10"	What's the difference between a 6 and a 7? Nobody knows.
"Score helpfulness 1-5"	Scores drift over time. Hard to aggregate.
"Pass or fail?"	Clear, actionable, comparable across runs.

Binary judgments force you to decide: "Is this good enough or not?"

If you find yourself wanting to give something a "6 out of 10," ask yourself: would you accept this from a human team member? Yes = pass. No = fail.

If You Use LLM-as-Judge, Require Reasoning
Sometimes you need an LLM to evaluate outputs (complex judgments, scaled evaluation). If you do:

Always require reasoning BEFORE the verdict.

```
Bad:  "Pass: true"
Good: "The output correctly identified the company size (500 employees) and
       mentioned the VP title as a seniority indicator. However, it failed
       to note the technology industry fit. Verdict: FAIL"
```

Why reasoning first?

- Forces the LLM to think before judging
- Lets you debug bad judgments
- Catches cases where the verdict doesn't match the reasoning

Don't Aim for 100% Pass Rate
From Hamel's research:

> "A 70% pass rate might indicate you're testing meaningful things. A 100% pass rate might mean your tests are too easy."

If every test passes, your golden dataset probably isn't challenging enough.

Calibrate Domain Specificity
Your eval questions need to be domain-specific, but not TOO specific.

Too Generic	Just Right	Too Specific
"Does it return data?"	"Does it return funding data with correct schema?"	"Does it return exactly 47 rows for Q3 2024?"
"Is it helpful?"	"Does it explain the trend direction with evidence?"	"Does it mention the exact words 'market cooling'?"
"Does it work?"	"Does it handle missing data gracefully?"	"Does it throw error code ERR_NULL_12?"

The sweet spot: Questions that test your specific domain logic but don't break when underlying data changes.

03. Block 2: Lab 1 - Create Your Golden Dataset (45 min)

Task: Build 5 Input/Output Pairs for Your Agent
Pick the agent you've been building throughout this course. Create a golden dataset to test it.

Step 1: Create a simple file to store your test cases:

```
mkdir -p agents/my-agent-evals
cd agents/my-agent-evals
```

Create `golden-dataset.md`:

```
# Golden Dataset for [Your Agent Name]

## Test Case 1: [Name]
**Input:**
[What you'll give the agent]

**Expected Output:**
[What "good" looks like - be specific]

**Pass Criteria:**
- [ ] Criterion 1
- [ ] Criterion 2
- [ ] Criterion 3

---

## Test Case 2: [Name]
...
```

Step 2: Fill in 5 test cases

Think about:

- 2-3 "happy path" cases (normal inputs you expect)
- 1-2 edge cases (unusual but valid inputs)
- 1 potential failure case (what should it refuse or handle gracefully?)

Examples by domain:

Domain	Happy Path	Edge Case	Failure Case
GTM/Sales	Score a well-documented lead	Lead with missing company info	Obvious spam/fake lead
Developer Tools	Review a clean PR	PR with 50+ files changed	PR with merge conflicts
Content/Marketing	Summarize a blog post	Summarize a 50-page whitepaper	Summarize an image-only PDF
Customer Support	Classify a billing question	Ticket in another language	Abusive/threatening message
Operations	Process a standard invoice	Invoice with multiple currencies	Invoice missing required fields
Data Analytics	Profile a clean CSV	CSV with mixed data types	Corrupted or empty file

Data Analytics Example (Using startup-funding.db):

Here's a preview of the golden dataset for a data analysis agent. See `evals/week7-golden-dataset.md` for the complete 8-eval set with expected outputs and pass criteria.

Test	Input	Pass Criteria
Basic retrieval	"How many startups are in the database?"	Returns exactly 200
Aggregation	"Average funding by stage?"	Pre-Seed ~$1.76M, Seed ~$6M, Series A ~$24.6M, B ~$62M, C ~$192M
Multi-table join	"Top 5 investors by portfolio size?"	Intel Capital #1 with 15 companies
Trend analysis	"Is funding heating up or cooling?"	Notes 2021-2023 growth, 2024 plateau
Prediction	"Which Series A companies will raise B next?"	Ranks with reasoning, cites amount + investor + timing
Context management	"List all 2024 funding rounds"	States "showing X of 91" if limited
Edge case	"Compare Cursor vs Replit"	Notes data asymmetry (1 round vs 2), caveats incompleteness

Step 3: Run your agent on each input

For now, do this manually:

1. Open Claude Code
2. Trigger your agent/skill with test input #1
3. Copy the output
4. Compare to your expected output
5. Mark pass/fail for each criterion
6. Repeat for all 5 test cases

Step 4: Record results

Add a results section to your file:

```
## Results

| Test Case | Pass/Fail | Notes |
|-----------|-----------|-------|
| 1: Happy path lead | ✓ Pass | Score was 82, within expected range |
| 2: Missing data | ✓ Pass | Correctly noted missing info |
| 3: Edge case | ✗ Fail | Timed out on large input |
| 4: ... | ... | ... |
| 5: ... | ... | ... |

**Pass Rate:** 4/5 (80%)

**What I Learned:**
- Agent handles normal cases well
- Struggles with very large inputs
- Need to add timeout handling
```

Success Criteria

- [ ] 5 test cases documented
- [ ] Each has input, expected output, and pass criteria
- [ ] All 5 have been run through your agent
- [ ] Results recorded with notes

---

04. BREAK (10 min)

---

05. Block 3: Theory - Automating Your Evals (30 min)

When to Automate
Manual testing is fine for 5 test cases. But what about:

- 50 test cases?
- Running after every code change?
- Comparing different prompts?

That's when you want automation.

Two Options for Running Evals at Scale
Option 1: Workshop Eval Runner Script (Recommended)

This workshop includes a ready-to-use eval runner at `scripts/run-funding-evals.py`. It demonstrates:

- Streaming output so you see Claude's work in real-time
- Tool call visibility (shows SQL queries and results)
- Boolean pass/fail scoring with string matching
- JSON result export for analysis

See the detailed documentation in Block 3, Lab 2 below.

Option 2: Custom Script with Claude Agent SDK

For custom eval needs, here's the minimal approach using the Claude Agent SDK (same patterns from Week 6) with parallel execution:

```
// src/eval-runner.ts
import { query } from "@anthropic-ai/claude-agent-sdk";

interface TestCase {
  id: string;
  input: string;
  mustMention: string[];
}

interface EvalResult {
  id: string;
  passed: boolean;
  output: string;
  notes: string[];
}

// Your golden dataset
const testCases: TestCase[] = [
  {
    id: "1",
    input: "Score this lead: VP Engineering at 500-person tech company",
    mustMention: ["seniority", "company size"]
  },
  {
    id: "2",
    input: "Score this lead: Marketing Coordinator at 10-person startup",
    mustMention: ["small company", "junior"]
  },
  // Add more test cases...
];

async function runSingleEval(testCase: TestCase): Promise<EvalResult> {
  const result = await query({
    prompt: testCase.input,
    options: {
      maxTurns: 3,
    }
  });

  const output = result.text;
  const notes: string[] = [];
  let passed = true;

  // Check if expected terms are mentioned
  for (const term of testCase.mustMention) {
    if (!output.toLowerCase().includes(term.toLowerCase())) {
      passed = false;
      notes.push(`Missing: ${term}`);
    }
  }

  return { id: testCase.id, passed, output, notes };
}

async function runAllEvals(): Promise<void> {
  console.log(`Running ${testCases.length} test cases in parallel...\n`);

  // Run all test cases in parallel
  const results = await Promise.all(
    testCases.map(tc => runSingleEval(tc))
```

Run it:

```
npm install @anthropic-ai/claude-agent-sdk typescript ts-node
npx ts-node src/eval-runner.ts
```

The Key Insight
Your eval is only as good as your pass/fail criteria.

"Does the output contain the word 'score'?" - Bad eval, too simple

"Is the score between 70-90 AND does it mention company size?" - Better, domain-specific

Spend more time on defining good criteria than on automation infrastructure.

---

06. Block 4: Lab 2 - Iterate on Your Agent (30 min)

Task: Fix One Failing Test Case
From your Lab 1 results, pick a test case that failed (or barely passed). Fix it.

Step 1: Analyze the failure

- What did the agent output?
- What did you expect?
- Why is there a gap?

Common issues:

- Prompt doesn't give enough guidance
- Missing examples in the skill
- Edge case not handled
- Wrong tool being used

Step 2: Make one change

Don't change everything at once. Make ONE adjustment:

- Add a clarifying instruction to your prompt
- Add an example to your skill
- Constrain the output format
- Handle the edge case explicitly

Step 3: Re-run the test

Did it pass now? If not, try a different fix.

Step 4: Check you didn't break other tests

Run all 5 test cases again. Sometimes fixing one thing breaks another.

The Iteration Loop

```
Run tests -> See failure -> Analyze why -> Make ONE change -> Run again
```

Repeat until your agent behaves the way you expect.

Group Discussion (10 min)
Share with your table:

- What test case failed for you?
- What did you change to fix it?
- Did fixing it break anything else?

---

07. Wrap-Up (15 min)

Key Takeaways
1. Golden dataset = expected inputs + outputs - Nothing fancy, just what you expect your agent to do
2. Generic evals don't work - "Helpfulness" scores tell you nothing about your specific use case
3. Start with error analysis - Manually review 20-50 outputs before building automation
4. 5 test cases is enough to start - You can always add more later
5. Iterate on failures - The point of testing is to find and fix problems

Homework
Part 1: Expand Your Golden Dataset

Grow your golden dataset from 5 to 10 test cases:

- Add more edge cases you discovered
- Include inputs that caused problems
- Cover the full range of what your agent should handle

Part 2: Automate Your Evals

Either:

- Adapt the workshop eval runner (`scripts/run-funding-evals.py`) for your use case
- Write a simple script using the Claude Agent SDK (see Block 3)

Run your 10 test cases automatically and save the results.

Part 3: Document Your Findings

Create `eval-report.md` with:

- Your 10 test cases (input + expected output)
- Results from automated run
- What you changed to fix failures
- Final pass rate and notes

Part 4: Prepare Your Demo (Week 8)

Next week is demo day. Start preparing now:

1. Pick your best agent - The one that showcases your learning
2. Prepare a 5-minute demo covering:
   - The problem you're solving (30 sec)
   - Your solution architecture (1 min)
   - Live demo with real data (2.5 min)
   - What you learned (1 min)
3. Record a backup video - In case of live demo issues
4. Test your eval results are ready to show - Demonstrating that you tested your agent is impressive

Next Week Preview

Week 8: Demos - Present your projects and learn from each other

---

08. Facilitator Notes

Philosophy Shift

This week is intentionally less technical than the original. The goal is:

- Make evals accessible to non-developers
- Focus on the THINKING (what makes a good test) not the TOOLING
- Get people comfortable with manual evaluation before automation

Common Questions

"How do I judge if the output is 'close enough'?"

You're the domain expert. If you would accept this from a human team member, it passes.

"My agent gives different outputs each time. How do I test that?"

Test for criteria, not exact matches. "Does it mention X?" rather than "Does it output exactly Y?"

"5 test cases seems too few."

It's a starting point. Quality over quantity. 5 thoughtful tests beat 50 generic ones.

"Should I use LLM-as-judge?"

Not for this course. It adds complexity and cost. Start with simple rule-based checks.

"When should I create eval questions?"

Immediately when you build something new. Just built an MCP tool? Create 3 eval questions right then. Wrote a new skill? Add 3 test cases before moving on. This habit catches problems early.

"What makes a good prediction eval?"

It should require synthesis across multiple data points. "Which company will raise Series B next?" is better than "List companies that raised Series A" because it tests reasoning, not retrieval.

Timing

- Block 1 (Theory): Focus on the "why" - don't rush
- Block 2 (Lab 1): Give full 45 min - creating good test cases takes time
- Block 3 (Theory): Show both options, but emphasize simplicity
- Block 4 (Lab 2): Hands-on iteration is the most valuable part

If People Finish Early

Have them:

- Add more test cases
- Help a neighbor debug their agent
- Start on the automation script
- Begin demo prep

Resources

- Hamel Husain's Eval FAQ (https://hamel.dev/blog/posts/evals-faq/) - The article this session draws from
- Claude Agent SDK Docs (https://docs.anthropic.com/en/docs/agents-and-tools/claude-agent-sdk/overview) - For custom automation scripts
- Claude Code CLI Docs (https://docs.anthropic.com/en/docs/claude-code) - For understanding CLI flags

Workshop Eval Resources:

- \`data/evals/funding-analysis-evals.json\` - Machine-readable eval set (16 test cases)
- \`evals/week7-golden-dataset.md\` - Detailed documentation with expected outputs
- \`scripts/run-funding-evals.py\` - Eval runner script (see documentation below)

---

09. Appendix: Eval Runner Script Documentation

Overview

The workshop includes \`scripts/run-funding-evals.py\`, a Python script that runs evals against the startup funding database using Claude Code CLI. It demonstrates best practices for eval automation.

How to Run

```
# Run all evals
python3 scripts/run-funding-evals.py

# Run only easy/medium/hard evals
python3 scripts/run-funding-evals.py --filter=easy
python3 scripts/run-funding-evals.py --filter=hard

# Run a specific eval by ID
python3 scripts/run-funding-evals.py --id=basic-001

# Dry run - see evals without executing
python3 scripts/run-funding-evals.py --dry-run

# Verbose mode - show criteria details for all results
python3 scripts/run-funding-evals.py --verbose
```

What You'll See

The script streams Claude's output in real-time, including tool calls:"

```
--------------------------------------------------
[basic-001] Basic Count
--------------------------------------------------

+- Bash
| sqlite3 data/startup-funding.db "SELECT COUNT(*) as startup_count FROM startups;"
| 200
+-
**SQL Query:**
```

SELECT COUNT(*) as startup_count FROM startups;

```

**Answer:** There are **200 startups** in the database.

-> ✓ PASS (8s)
```

How It Works

1. Uses Claude Code CLI with Streaming

The script uses proper CLI flags for real-time output:

```
cmd = [
    'claude', '-p', prompt,           # Direct prompt (no piping)
    '--output-format', 'stream-json', # Newline-delimited JSON events
    '--verbose',                      # Required for stream-json with -p
    '--allowedTools', 'Bash(sqlite3:*),Read'  # Auto-approve DB queries
]
```

2. Parses Stream Events

The `stream-json` format emits events as newline-delimited JSON:

Event Type	Contains	Script Action
`assistant`	Text content, tool_use	Print text, show tool calls
`user`	tool_result	Show query results
`result`	Final output	Extract for scoring

3. Boolean Pass/Fail Scoring

Each eval has pass criteria checked with simple string matching:

Criterion Pattern	How It's Checked
`"Returns exactly 200"`	`"200" in output`
`"Does NOT hardcode"`	`"hardcode" not in output.lower()`
`"Uses COUNT(*)"`	`"count(*)" in output.lower()`
Other patterns	Marked as "needs review"

4. Results Export

Results are saved to `output/eval-results.json`:

```
{
  "timestamp": "2026-01-21T06:41:27Z",
  "eval_set": "Startup Funding Analysis Evals",
  "summary": {"passed": 12, "failed": 2, "review": 2, "total": 16},
  "results": [
    {
      "id": "basic-001",
      "name": "Basic Count",
      "passed": true,
      "output": "SELECT COUNT(*) ... **200 startups**",
      "duration_ms": 8786,
      "criteria_results": [...]
    }
  ]
}
```

Eval JSON Format

Evals are defined in `data/evals/funding-analysis-evals.json`:

```
{
  "name": "Startup Funding Analysis Evals",
  "evals": [
    {
      "id": "basic-001",
      "name": "Basic Count",
      "category": "retrieval",
      "difficulty": "easy",
      "input": "How many startups are in the database?",
      "pass_criteria": [
        "Returns exactly 200",
        "Uses COUNT(*) or equivalent"
      ]
    }
  ],
  "scoring": {
    "expected_pass_rates": {
      "easy": 0.95,
      "medium": 0.80,
      "hard": 0.60
    }
  }
}
```

Adapting for Your Agent

To use this pattern for your own agent:

1. Create your eval JSON with inputs and pass criteria
2. Modify the prompt template in `run_eval()` to match your agent's context
3. Update `--allowedTools` to match what your agent needs
4. Adjust scoring logic for your criteria patterns

Key Design Decisions

Decision	Why
Stream-json output	See progress during long evals instead of waiting
Tool visibility	Debug what queries the agent is running
Boolean scoring	Clear pass/fail, no ambiguous scales
String matching	Simple, fast, no LLM-as-judge complexity
JSON export	Enables trend analysis across runs