

AI Agents: Industrial Software Engineering

Software Engineering Best Practices for AI Agents

Hacking vs Engineering



The Paradox of Progress

As an industry, we are in cross-roads. No authoritative answer.

Embrace uncertainty. Gain confidence thru experiments for YOU, what works

What has changed?

Written code vs LLM Reasoning

LLM outputs are non-deterministic

Prompt Engineering: How to *program* LLM?

Vibe coding: How much to rely on?

What has NOT changed?

Don't break things

Always be fast

More code → more bugs

Developer optimism

Murphy's Law

What do we gonna learn

Ship with confidence

Embrace experiments

Speed as a feature

Ship with confidence

Testing & Evaluation Strategy

Deterministic vs Non-deterministic Testing

🎯 Deterministic Tests

```
import re

def extract_email(text: str) -> str | None:
    match = re.search(r"[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-z]{2,}", text)
    if match:
        return f"The email id: {match.group(0).lower()}"
    return "Email not found"

def test_extract_email():
    response = extract_email("Contact me at Adam@example.com for details.")
    assert response == "The email id: adam@example.com"
```

- Fast execution
- Reliable CI/CD
- Easy debugging

Deterministic vs Non-deterministic Testing



Non-deterministic Tests

```
1  llm = DummyLLM()
2  # Prompt:    response = llm.run("Extract the email address from: Contact me at user@example.com for details. \
3  # When you cannot extract, say 'Email not found'. if you can find .... ")
4
5
6  @pytest.mark.slow
7  def test_extract_email_llm():
8      response = llm.extract_email("Contact me at Adam@example.com for details.")
9
10     email = extract_email(response)
11     assert email == "The email id: user@example.com"
```

- Mostly works. But not deterministic.
- Costly to run
- Slower execution

Deterministic vs Non-deterministic Testing

🎭 Use Mocks

```
@pytest.fixture
def mock_llm():
    return Mock(invocation={
        "content": "Create a calendar event.",
        "tool_calls": [
            {
                "name": "calendar.create",
                "args": {"title": "Sync with Alice", "start": "2025-10-01T15:00", "duration_min": 30}
            }
        ],
    })

@pytest.fixture
def mock_calendar():
    return Mock(create={"id": "evt_123", "title": "Sync with Alice", "start": "2025-10-01T15:00"})

def test_calendar_agent(mock_llm, mock_calendar):
    agent = CalendarAgent(llm=mock_llm, tools={"calendar.create": mock_calendar})
    msg = agent.handle("Book a 30-min sync with Alice tomorrow at 3pm")
    mock_llm.invoke.assert_called_once()
    mock_calendar.create.assert_called_once_with(
        title="Sync with Alice", start="2025-10-01T15:00", duration_min=30
    )
    assert "evt_123" in msg and "Sync with Alice" in msg
```



Non-deterministic Testing aka Evals

⚖️ Evals: LLM as judge

```
def judge_score(query:str, expected:Dict[str,Any], got:Dict[str,Any])→Dict[str,Any]:  
    client = OpenAI()  
    rubric = (  
        "You are an evaluator. Score 0..1 JSON: {\"score\":float,\"reason\":string}.\n"  
        "Full credit if tool args match the user's intent, fields present and plausible."  
    )  
    prompt = f"""  
User: {query}  
Expected args: {json.dumps(expected)}  
Got args: {json.dumps(got)}  
Return ONLY JSON with keys score, reason.  
"""  
  
    r = client.chat.completions.create(  
        model=os.getenv("OPENAI_MODEL","gpt-4o-mini"),  
        messages=[{"role":"system","content":rubric}, {"role":"user","content":prompt}],  
        temperature=0  
    )  
    return json.loads(r.choices[0].message.content.strip())
```



Non-deterministic Testing aka Evals

⚖️ Evals: Have a large eval

```
EVALS = [
    ("Book 30-min sync with Alice tomorrow 3pm PST",
     {"title": "Sync with Alice", "start": "2025-10-01T15:00", "duration_min": 30}),
    ("Schedule 1h review with Bob on Oct 2 at 10am",
     {"title": "Review with Bob", "start": "2025-10-02T10:00", "duration_min": 60}),
    ("Add standup today 9am, 15min",
     {"title": "Daily Standup", "start": "2025-09-27T09:00", "duration_min": 15}),
]
```



```
@pytest.mark.parametrize("query,expected", EVALS)
def test_calendar_agent_eval(query, expected):
    if not os.getenv("OPENAI_API_KEY"): pytest.skip("no key")
    tool, agent = CalendarTool(), CalendarAgent(ReallLM(), tool=CalendarTool())
    _ = agent.handle(query)
    got = tool.last
    verdict = judge(query, expected, got)
    assert verdict["score"] >= 0.8, verdict["reason"]
```



Sidenote: Dependency Injection & `__call__`

```
class DummyNode:
    def __init__(self, llm=None, tools: Dict[str, Any]):
        self.llm = llm
        self.tools = tools

    def __call__(self, state: Dict[str, Any]) → Dict[str, Any]:
        # Do something with self.llm and self.tools

# --- LangGraph wiring ---
builder = StateGraph(dict)
dummy_node = DummyNode(llm=real_llm, tools=real_tools)
builder.add_node("dummy", dummy_node)
builder.add_edge(START, "dummy")
builder.add_edge("dummy", END)
graph = builder.compile()

## Tests

DummyNode(mock_llm, mock_tools)
```



Test pyramid

- Nodes testing: Both deterministic & non-deterministic
- Large node level tests
- Less graph level E2E tests
- Large Evalset > Higher confidence



Embrace experiments

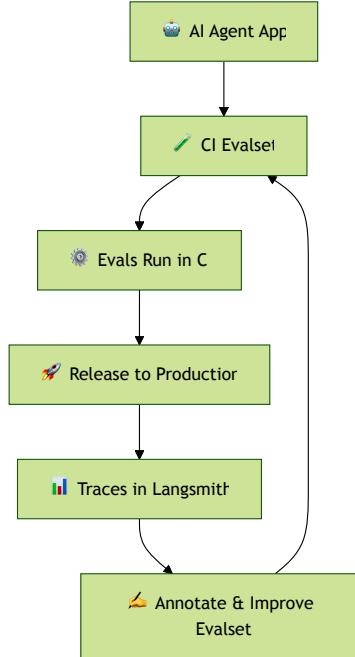


Establish feedback loop



Grow agents, don't construct them

- Start with base evalset.
- Establish traces in production
- Collect user feedback
- Monitor and annotate to improve evalset
- Repeat





Experiment setups: Test in production

- A/B testing: Route traffic to parallel setups
- Safeguard: User feedback based (e.g. % 🤔 disables the experiment in production)
- Bayesian: Use user feedback and dynamically switch
- User cohort-based testing
- Query based routing: Supervisor agent patterns
- CI: Evalset against experiments



Speed as a feature



KV Cache

Which will perform better?

Version A

System:

You are SQLCopilot v3.1. (...800-token policies & examples)
Policies: ...

User:

[today=2025-09-27, org=Acme-42]

SCHEMA:

tables: users(id, ...), orders(id, ...), -- (unordered)

TASK:

Write a SQL for top spenders last 7d.



Version B

System:

You are SQLCopilot v3.1. (...800-token policies & examples)
Policies: ...

Style guide: ...

SCHEMA (canonical order, stable formatting):

users(id, ...)
orders(id, ...)

User

today=2025-09-27

org=Acme-42

TASK:

Write a SQL for top spenders last 7d.



KV Cache

🐌 Version A

```
# Baseline (low reuse)
System:
  You are SQLCopilot v3.1. (...800-token policies & exemplars)
  Policies: ...
User:
  [today=2025-09-27, org=Acme-42]
SCHEMA:
  tables: users(id, ...), orders(id, ...), -- (unordered)
TASK:
  Write a SQL for top spenders last 7d.
```

⚡ Version B

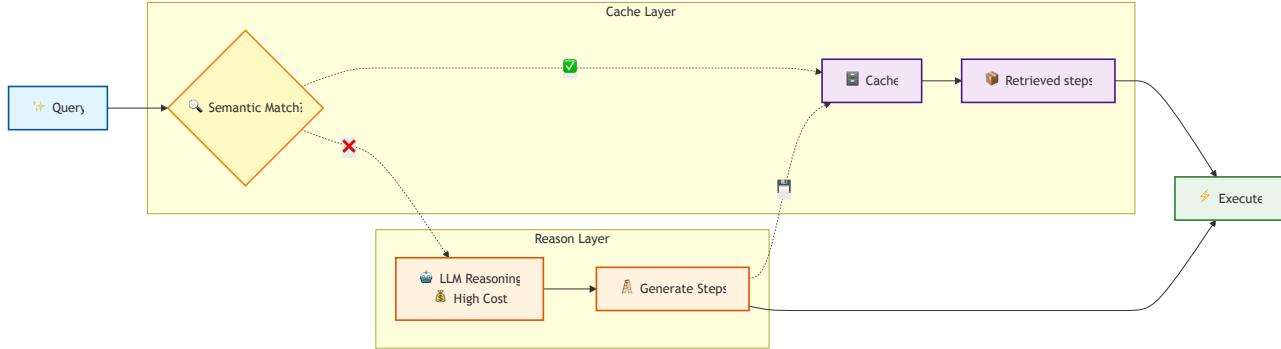
```
System:
  You are SQLCopilot v3.1. (...800-token policies & exemplars)
  Policies: ...
Style guide: ...
SCHEMA (canonical order, stable formatting):
  users(id, ...)
  orders(id, ...)
User
  today=2025-09-27
  org=Acme-42
TASK:
  Write a SQL for top spenders last 7d.
```

- Freeze the header: identical system prompt
- Push volatility to the tail (User content in above example)
- Avoid drift: Consistent ordering (table order in schema)
- Chunk long exemplars: keep few-shot examples stable



Reasoning cache

- Store the cached steps
- Semantic matching based on embeddings
- Iterate on cache-hit ratio



Questions?