# Large Language Models for Software Engineers

Session 2: The Agent Loop, Tools, and Memory

Monarch Wadia · monarchwadia.com

SECTION 1

# Logistics & Goals

From the Single Call to the Agent Loop

# Where We Left Off: The Single Call

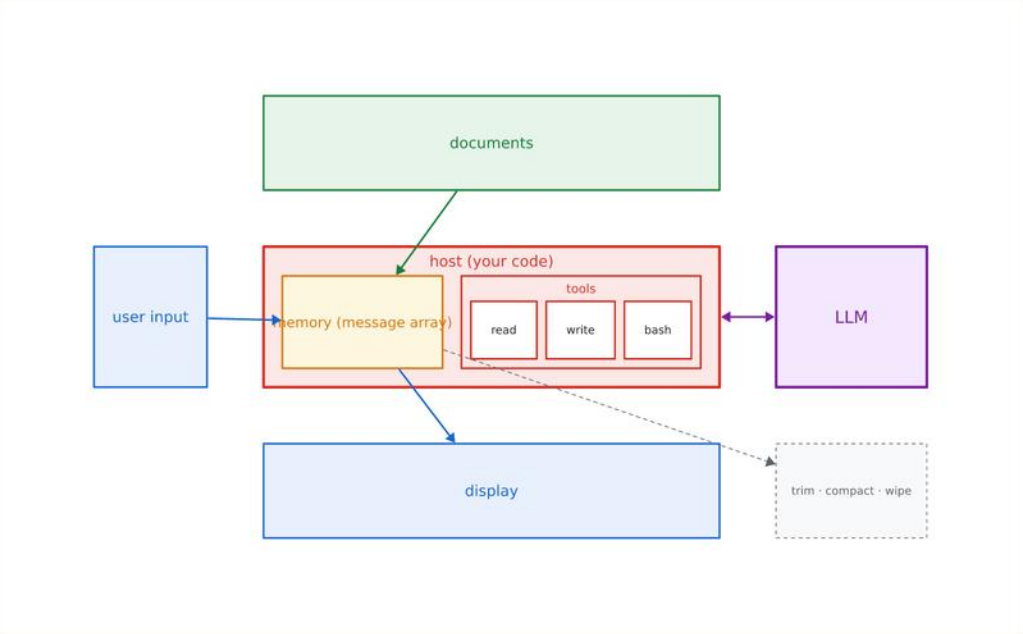
- **Stateless Engine** : The remote LLM retains zero state between requests (f(messages) -> message).
- **History in Application RAM** : The host application stores the message array and appends each turn.
- **Control vs. Data Plane** : The system prompt dictates persona, guardrails, and output schemas.
- **The Next Step** : Moving from manual single calls to automated multi-turn execution loops.

```
// Session 1: Stateless Single Call
var request = new ConverseRequest
{
    ModelId = "anthropic.claude-3-5-sonnet",
    System = new List<SystemContentBlock>
    {
        new() { Text = "You are a banking assistant." }
    },
    Messages = new List<Message>
    {
        new()
        {
            Role = ConversationRole.User,
            Content = new() { new() { Text = "Ping" } }
        }
    },
    InferenceConfig = new() { MaxTokens = 100 }
};

var response = await client.ConverseAsync(request);
Console.WriteLine(response.Output.Message.Content[0].Text);
```

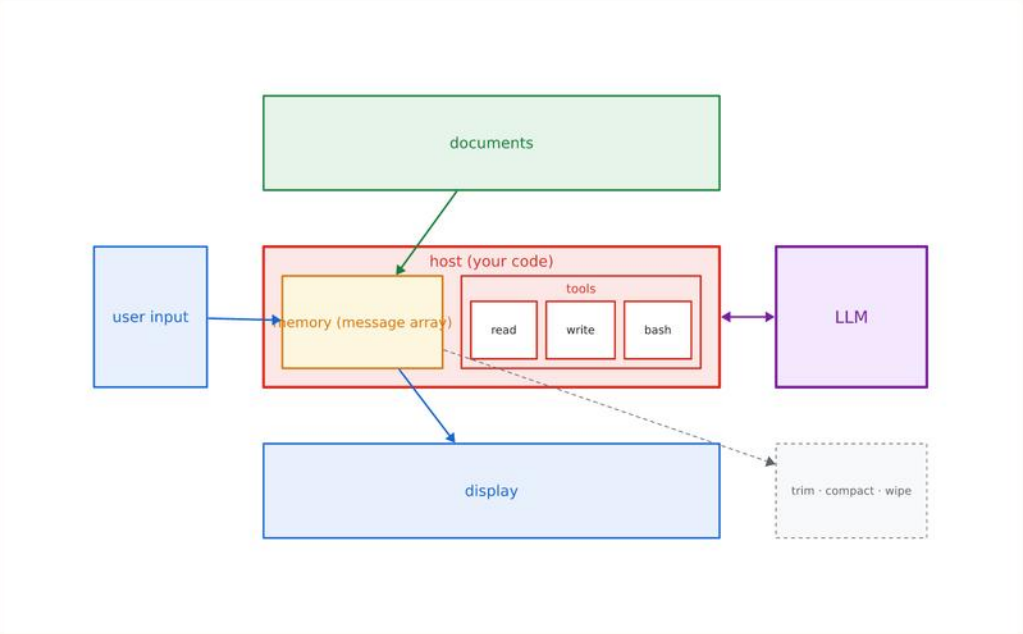
# Workshop Series Learning Outcomes

- **1. Treat LLMs as Infrastructure** : View models as stateless HTTP microservices with explicit schemas.
- **2. Build from First Principles** : Connect the Model (generation), Harness (logic & RAM), and Tools (APIs).
- **3. Control Developer Agent Tools** : Diagnose and optimize tools like Cursor, Claude Code, and Copilot.
- **4. Prevent Production Failures** : Guard against context overflow, format drift, and hallucination.



# Session 2 Learning Outcomes

- **1. Build the Agent Loop** : Implement the 5-step loop (input, memory, call, append, display).
- **2. Put Knowledge in the Loop** : Connect external documents via RAG and whole-file attachments.
- **3. Turn Output into Action** : Upgrade from brittle text scraping to the structured tool contract.
- **4. Equip the Toolbox** : Implement read tools, the surgical write idiom, and the shell.
- **5. Master Memory Management** : Control context growth with trim, compact, and wipe operations.



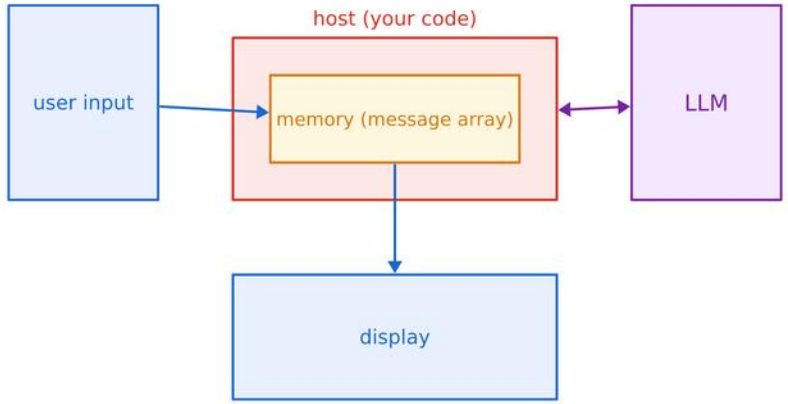
SECTION 2

# The Simplest Agent

The Loop, the Memory Array, and Content Blocks

# The Simplest Agent Loop

- **The Five Steps** : Capture user input, append to memory, call LLM, append response, display.
- **The Host (Your Code)** : Your application is the driver; it owns the loop and the state.
- **The Memory Array** : A plain in-memory list of messages passed on every API call.
- **The Remote Model** : Meets the transcript fresh on every invocation; holds no internal memory.



# The Loop in Code

- **The Skeleton** : The whole agent is one list, one HTTP call, and a while loop around them.
- **Harness Drives State** : The LLM never 'decides' to remember; your code mutates the array.
- **Deterministic Host** : Your host controls exit conditions, retry policies, and timeouts.

```
// The Full Agent Loop in C#
var memory = new List<Message>();

while (true)
{
    Console.Write("> ");
    var input = Console.ReadLine();
    if (string.IsNullOrEmpty(input)) break;

    // 1. Append user input
    memory.Add(new Message {
        Role = ConversationRole.User,
        Content = new() { new() { Text = input } }
    });

    // 2. Call LLM with full transcript
    var response = await client.ConverseAsync(
        new ConverseRequest { Messages = memory }
    );

    // 3. Append response & display
    var reply = response.Output.Message;
    memory.Add(reply);
    Console.WriteLine(reply.Content[0].Text);
}
```

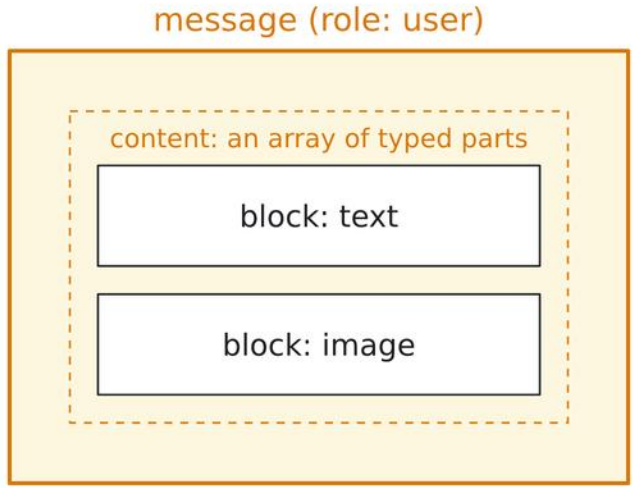
# The Array Is the Conversation

- **The Array Is the Mind** : The model keeps no diary and has zero memory of prior turns.
- **Total Amnesia on New Chat** : Starting a new chat simply initializes an empty array.
- **The Rewind Effect** : Editing an old message rewrites the array; future turns vanish instantly.

```
// The Conversation Transcript Array
[
  {
    "role": "user",
    "content": "add rate limiting to login"
  },
  {
    "role": "assistant",
    "content": "I will start by checking the route."
  },
  {
    "role": "user",
    "content": "make it 5 attempts max"
  },
  {
    "role": "assistant",
    "content": "Updated lockout threshold to 5."
  }
]
```

# Everything Is a Content Block

- **Message Anatomy** : A message content is an array of typed blocks (text, image, tool_result).
- **Multi-Modal Inputs** : Screenshots and attachments ride in through the exact same door.
- **Token Economics** : Image blocks consume significant token budgets; send only what is needed.
- **Unified Door** : Tool outputs and error messages enter the memory array as content blocks.



# Streaming & Partial Information

- **Real-Time Generation** : Models emit tokens sequentially as they are generated over HTTP.
- **Perceived Latency** : Streaming provides immediate visual feedback during long generations.
- **The Partial State Trap** : A half-streamed response is an incomplete sentence or truncated JSON.
- **The Golden Rule** : Treat streamed output as unverified until the end-of-turn signal arrives.

# The Flaw: The Loop Only Appends

- **The Hidden Flaw** : The basic loop only ever adds; nothing is ever removed automatically.
- **Compounding Cost** : The full array is re-sent on every turn, billing prior tokens repeatedly.
- **Context Ceiling** : Without active management, long conversations inevitably overflow the window.

```
// Turn 1: 150 tokens
messages = [ U1, A1 ]

// Turn 5: 1,800 tokens (resending everything!)
messages = [ U1, A1, U2, A2, U3, A3, U4, A4, U5, A5 ]

// Turn 20: 18,000 tokens (approaching budget ceiling)
messages = [ U1, A1, ... U20, A20 ]

// Flaw: Append-only growth
// Total Cost = Sum(Turn_i * Tokens_i)
// Without trimming, latency and costs explode.
```

SECTION 3

# Knowledge in the Loop

RAG, Attachments, and Cracking the Sealed Loop

# The Sealed Loop Problem

- **Training Cutoff** : The model knows only public training data up to its cutoff date.
- **Private Data Gap** : Internal policies, customer records, and codebases were never in the training set.
- **Cracking the Seal** : External knowledge must be actively injected into host memory by your code.

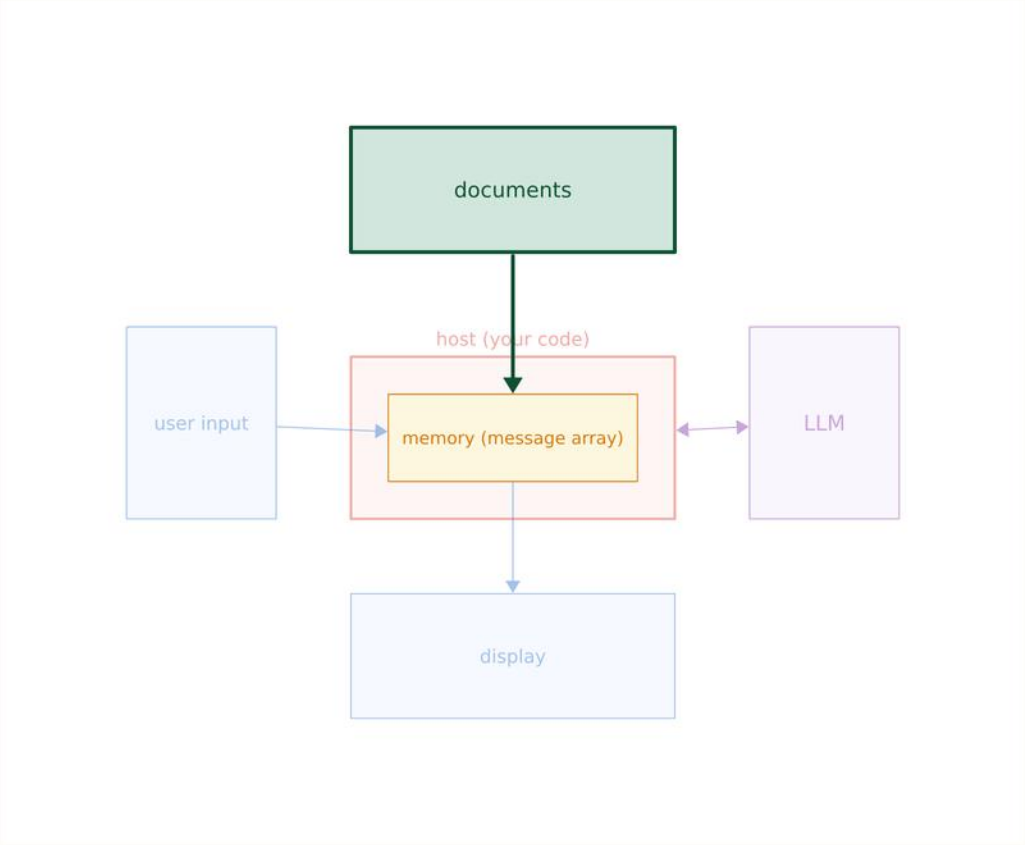
```
// Query to standard model without context:
{
  "role": "user",
  "content": "Can an annual subscriber get a refund after 2 months under our policy?"
}

// LLM Output (Unanchored Guess):
"Under standard SaaS industry practices, annual plans are non-refundable after 30 days..."
```

**Hallucination / Knowledge Gap: The model invents plausible generic answers because company policies were never in its training set.**

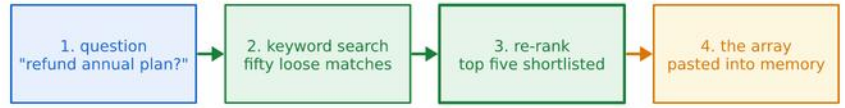
# Knowledge on Demand: RAG

- **Retrieval-Augmented Generation** : The host searches documents and pastes excerpts into memory.
- **Sliding the Page Across** : We do not retrain the model; we hand it the relevant page to read.
- **No Library Shelf in the LLM** : The model reads the injected context just like standard prompt text.



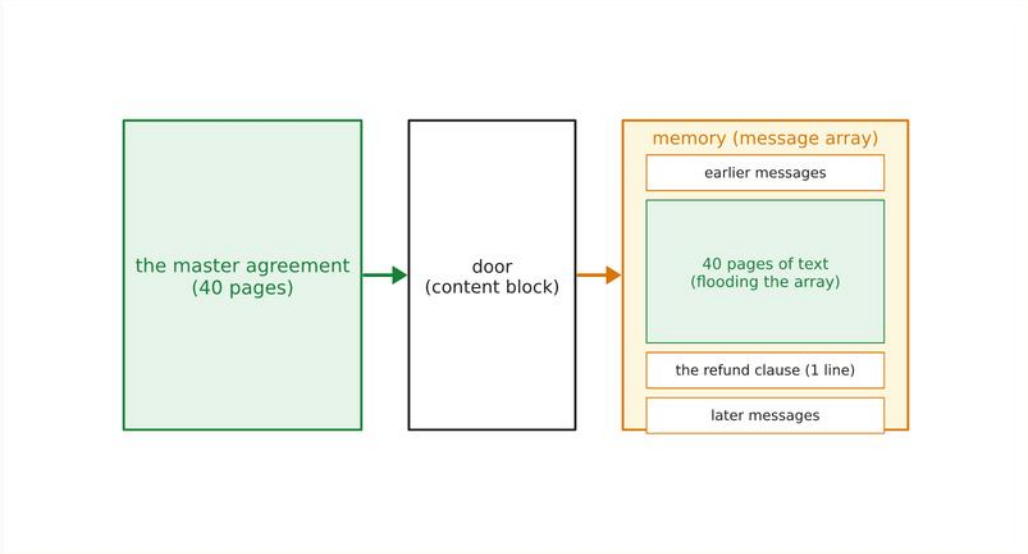
# Retrieve Wide, Read Narrow

- **Step 1: Broad Search** : Keyword, vector, or hybrid search pulls candidate document excerpts.
- **Step 2: Re-Ranking** : Score candidate passages against the query to select the best 3 to 5.
- **Search as an Upgraded Arrow** : Whether keyword, vector, or graph search, the loop interface is identical.



# Include It Whole: Attachments

- **Direct Attachment** : For concise documents (1-3 pages), attach the entire text or PDF directly.
- **Zero Search Latency** : Eliminates indexing, vector databases, and retrieval misses.
- **Non-Text Media** : Scanned agreements, architecture diagrams, and charts ride in as image blocks.



# Retrieval Traps: Trust & Flood

- **The Trust Flaw** : The model treats injected text as truth; stale policies yield confident stale answers.
- **The Flood Flaw** : Attaching massive binders dilutes attention and inflates latency and token cost.
- **Best Practice** : Curate document sources and inject only the precise excerpts required.

```
// Attaching a 50-page Master Agreement:
messages = [
  {
    "role": "user",
    "content": [
      { "type": "text", "text": "Find refund terms:" },
      { "type": "text", "text": "[... 45,000 words of legal definitions ...] Section 14.2: Refunds permitte
    ]
  }
]
```

**The Needle Drowned: 45,000 words of boilerplate dilutes model attention and bills full token costs on every subsequent turn.**

SECTION 4

# The First Way to Act

From JSON Output to the Tool Contract

# The Homemade Actor

- **The Naive Approach** : Prompting the model to emit ad-hoc JSON action commands in text.
- **Host Acts as Scraper** : The host inspects the text, parses JSON, executes, and replies.
- **Brittle Integration** : Mixing natural language reasoning with execution commands in one string.

```
// System Prompt:
If you need to run tests, output raw JSON:
{"action": "run_test", "path": "test_auth.py"}

// Host Code:
var reply = response.Output.Message.Content[0].Text;
if (reply.Contains(""action": "run_test""))
{
    var json = ExtractJson(reply);
    RunTest(json.Path);
}
```

# When the Seams Show

- **Conversational Leakage** : Models naturally prepend explanations ('Sure, I can help...').
- **Markdown Fences** : Delimiters like json break strict JSON parsers.
- **The Seam Problem** : String parsing against generative text is brittle in production.

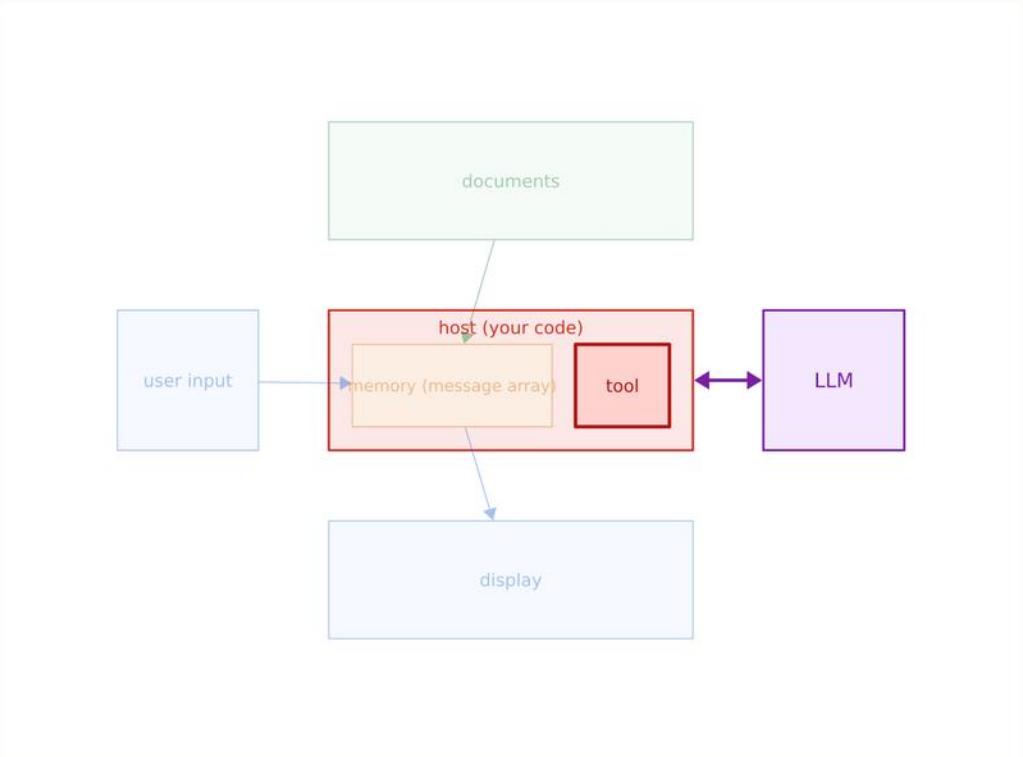
```
// Actual Model Output:
Sure! I would be happy to run those tests for you.
Here is the command:

```json
{
  "action": "run_test",
  "path": "tests/auth/test_login.py"
}
```
Let me know if you need anything else!
```

**Format Drift & Parsing Failure: Markdown code fences and conversational wrapper text broke System.Text.Json parser.**

# The First-Class Tool Contract

- **First-Class Tool API** : Declare available functions with explicit JSON schemas in the API request.
- **Separate Channels** : The model emits structured tool invocation blocks separate from message text.
- **The Host Decides** : The model proposes tool calls; the host validates and executes them.



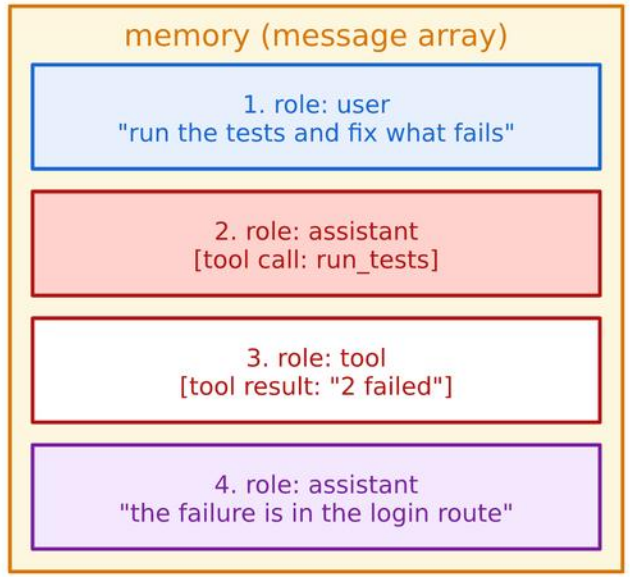
# Declaring Tool Schemas

- **Tool Name & Description** : Plain English explanations guide the model on when to select the tool.
- **Input Schema (JSON Schema)** : Defines exact required parameters, types, and constraints.
- **Type Safety** : Enables compile-time and runtime validation before executing local methods.

```
// Tool Schema Declaration (C# Bedrock)
var tool = new Tool
{
    ToolSpec = new ToolSpecification
    {
        Name = "read_file",
        Description = "Read content of a source file",
        InputSchema = new ToolInputSchema
        {
            Json = Document.FromObject(new
            {
                type = "object",
                properties = new
                {
                    path = new { type = "string" },
                    offset = new { type = "integer" }
                },
                required = new[] { "path" }
            })
        }
    }
};
```

# The Second Heartbeat

- **Turn 1 (Model)** : Returns tool_use content block with target tool name and arguments.
- **Turn 2 (Host)** : Executes local function (e.g., database query or file read).
- **Turn 3 (Result)** : Appends tool_result content block to history array and calls model again.
- **Running Errands** : A single user prompt may trigger a dozen internal tool roundtrips.



# Validate Before Acting

- **Never Trust Model Arguments** : Validate paths, types, permissions, and ranges in host code.
- **Graceful Feedback** : Return tool failure messages as content blocks so the model can adapt.
- **Security Boundary** : Your host code enforces access control; the model only holds a pencil.

```
// Model Requests Unauthorized Path:
{
  "name": "read_file",
  "input": { "path": "../../etc/shadow" }
}

// Host Defensively Validates:
if (!IsWithinProjectRoot(path))
{
    return new ToolResultContentBlock {
        Status = ToolResultStatus.Error,
        Content = new() { new() {
            Text = "Access denied: path outside sandbox."
        } }
    };
}
```

**Defensive Boundary: Host intercepts path traversal, rejects safely, and sends error block so model can re-aim.**

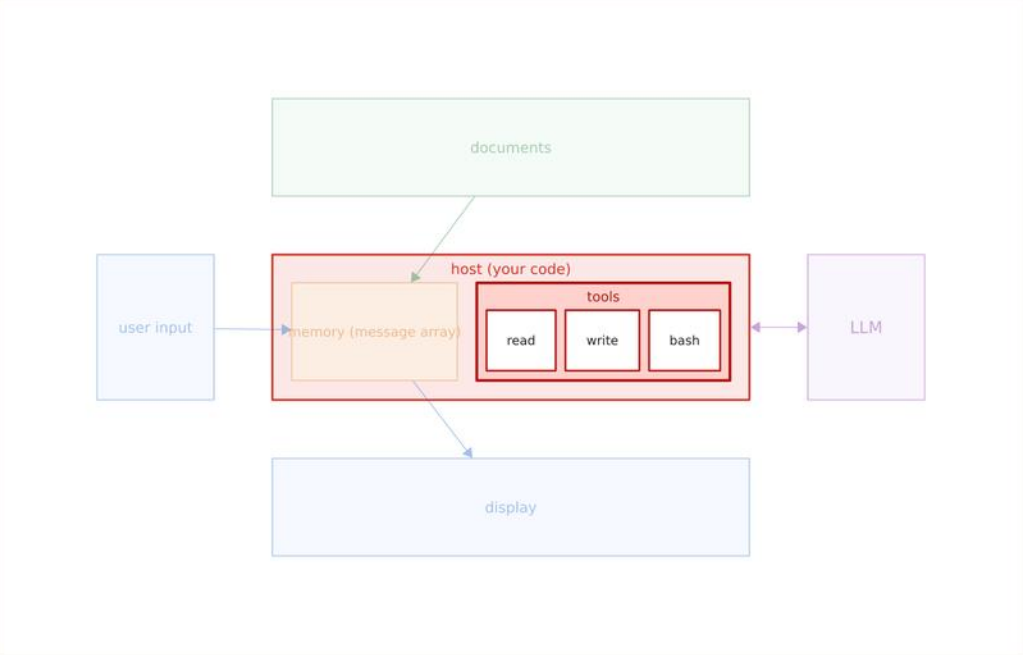
SECTION 5

# The Toolbox

Read Tools, the Surgical Write, and the Shell

# The Developer Bench

- **Toolbox Fans Out** : The single tool interface expands into specialized file system tools.
- **Tools Define Character** : A read tool makes a researcher; a write tool makes an editor; a shell makes an operator.
- **Standardized Foundation** : Every tool uses the exact same tool_use -> tool_result protocol.



# Read Tools: Exploring Codebases

- **Eyes for the Machine** : list_files answers 'what is here'; read_file inspects contents.
- **Read Selectively** : Always support line offsets and limits to prevent massive file dumps.
- **Context Discipline** : Reading whole files needlessly floods the history array.

```
// 1. List files in directory
tool_call: list_files(path: "src/auth/")
// Result: ["login.py", "tokens.py", "models.py"]

// 2. Read specific slice with offset & limit
tool_call: read_file(
    path: "src/auth/login.py",
    offset: 35,
    limit: 20
)
// Result: lines 35-55 only (not the whole 500-line file)
```

# The Surgical Write Idiom

- **The Surgical Pair** : oldstring specifies the exact target lines; newstring replaces them.
- **Cost Proportional to Diff** : Editing one line in a 1,000-line file costs only two lines of tokens.
- **Clear Human Review** : Produces clean, readable diffs rather than full-file replacements.
- **Preventing Corruption** : Avoids rewriting untouched code where generation errors could hide.

```
// The Surgical Write Request
{
  "tool": "edit_file",
  "file": "src/auth/login.py",
  "oldstring": "if (attempts > 5) LockAccount();",
  "newstring": "if (attempts > 3) LockAccount();"
}

// Benefit 1: Token cost proportional to diff (2 lines)
// Benefit 2: Reviewable 1-line diff in pull requests
// Benefit 3: Zero chance of corrupting untouched methods
```

# Safety Lock: Oldstring Validation

- **Proof of Context** : Requiring oldstring forces the model to prove what is currently in the file.
- **Ambiguity Protection** : If oldstring matches zero or multiple locations, the tool aborts safely.
- **Self-Correction** : The error feedback loop lets the agent re-read the target file and retry.

```
// Concurrency / Miss Case:
// File was modified by human developer after model read it:
"oldstring": "if (attempts > 5) LockAccount();"

// Tool Execution in Host:
if (!fileContent.Contains(oldstring))
{
    return "Error: oldstring not found. File may have changed.";
}
if (CountMatches(fileContent, oldstring) > 1)
{
    return "Error: oldstring matches multiple locations. Provide more context.";
}
```

**The Failure Is the Feature: Refusal to apply mismatched edits protects codebase integrity against stale reads.**

# The Shell: 50 Years of Unix

- **One Interface, Infinite Utility** : A single bash(command) tool provides search, build, and test.
- **Rapid Discovery** : grep and find locate relevant files in milliseconds across large repos.
- **Automated Verification** : Running unit tests gives the agent objective, executable feedback.

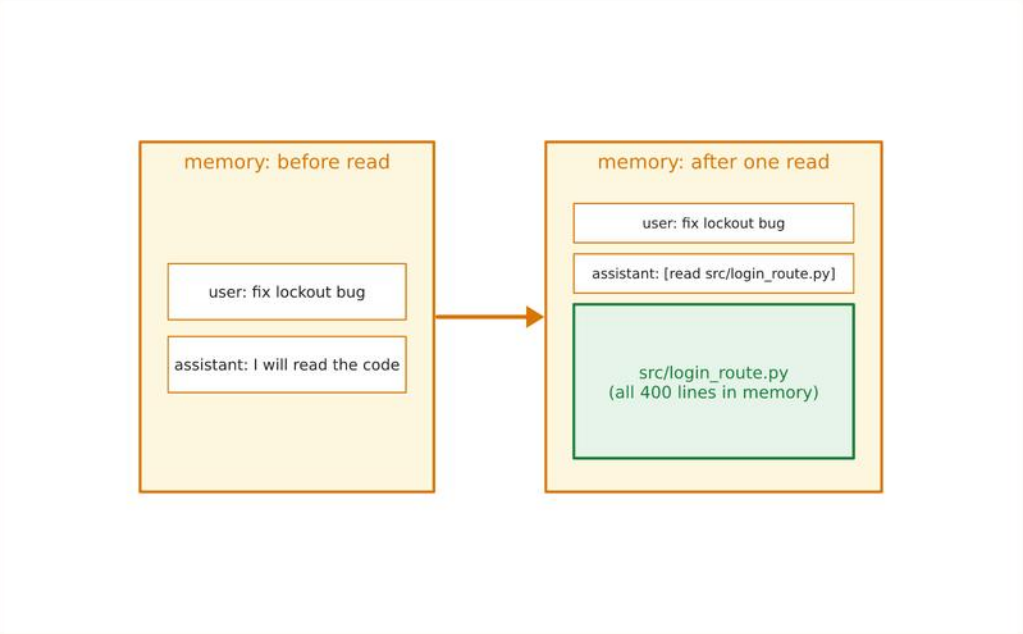
```
// 1. Search across repository in one turn:
$ grep -rn "LockAccount" src/
src/auth/login.py:42:  if (attempts > 5) LockAccount();

// 2. Perform surgical edit...

// 3. Verify fix by executing test runner:
$ dotnet test --filter FullyQualifiedName~LoginTests
Passed! - Failed: 0, Passed: 14, Total: 14
```

# The River: Flood & Permissions

- **The River of Tool Outputs** : Every file read, grep output, and test log accumulates in the array.
- **Security & Approvals** : Shell access requires strict sandboxing and human-in-the-loop approvals.
- **The Impending Limit** : Complex tasks quickly generate tens of thousands of tokens of history.



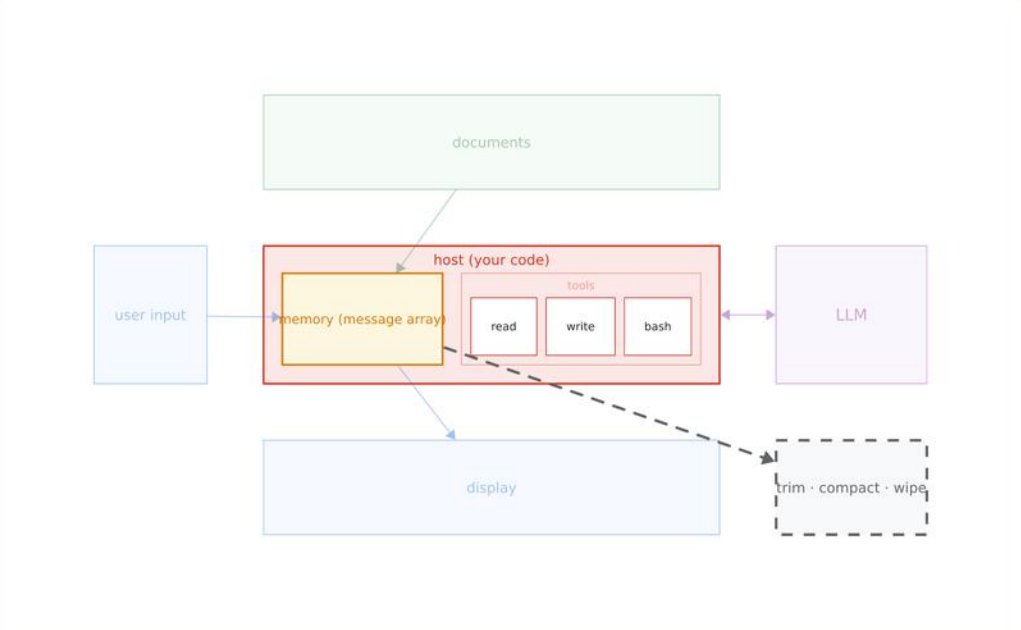
SECTION 6

# Memory Management

Compaction, Sessions, and the Machine Assembled

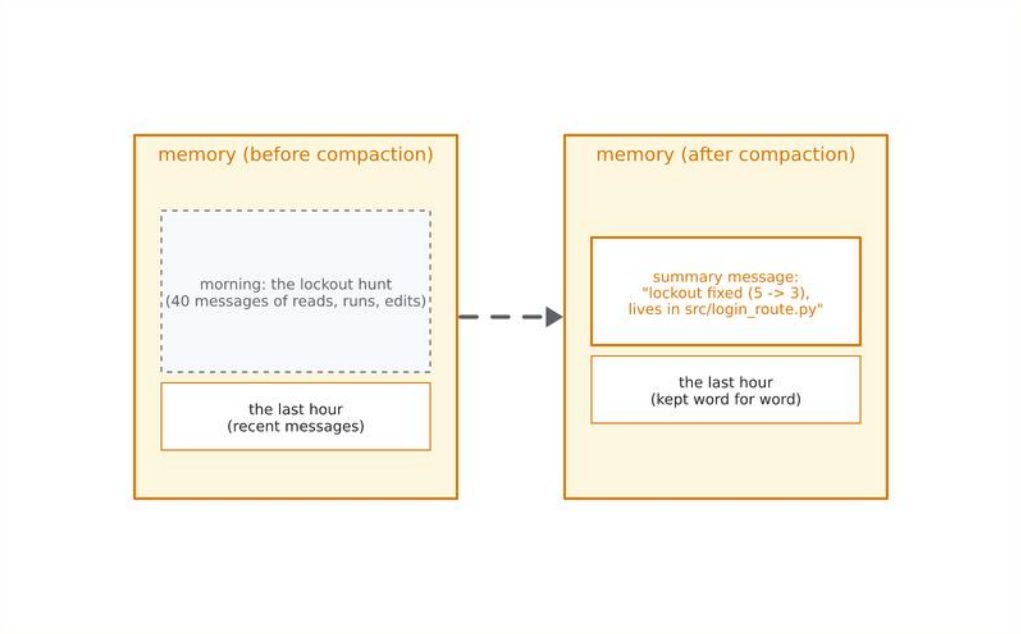
# Active Memory Management

- **The Host Must Subtract** : The loop naturally appends; the host must actively trim and clean.
- **Three Primary Operations** :
  - **Trim** : Drop oldest non-system turns from the head of the array.
  - **Compact** : Summarize earlier dialogue into a concise handoff block.
  - **Wipe** : Discard the current array and start a fresh session.



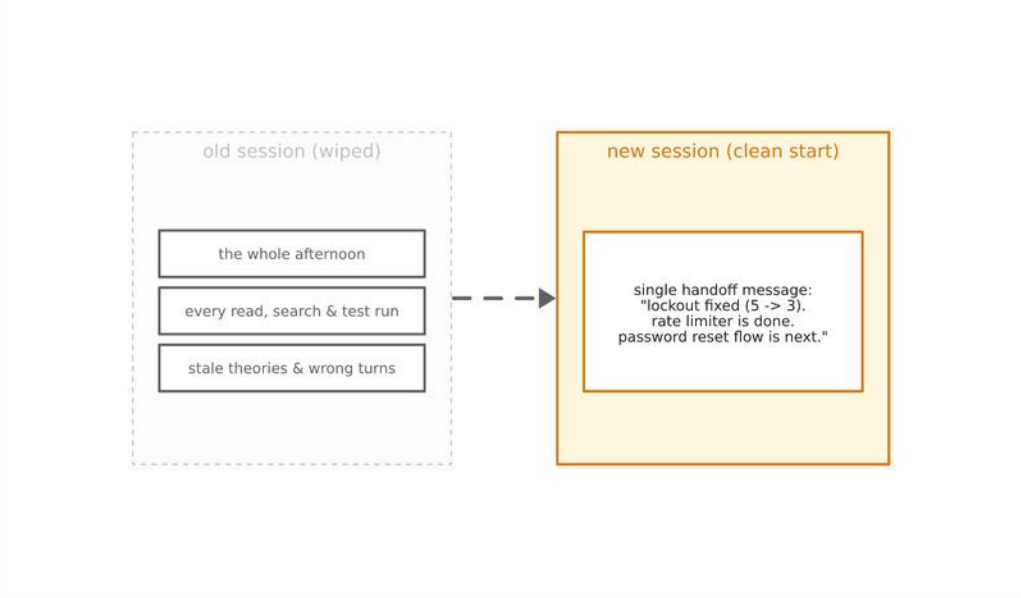
# Compaction in Practice

- **Summarize & Replace** : 40 turns of exploratory tool calls compress into a single summary block.
- **Focus on Handoff State** : Record decisions made, modified file paths, and pending tasks.
- **Preserve Recent Turns** : Retain the latest user and assistant turns for immediate continuity.
- **Controlled Information Loss** : Compaction drops noise while preserving critical working state.



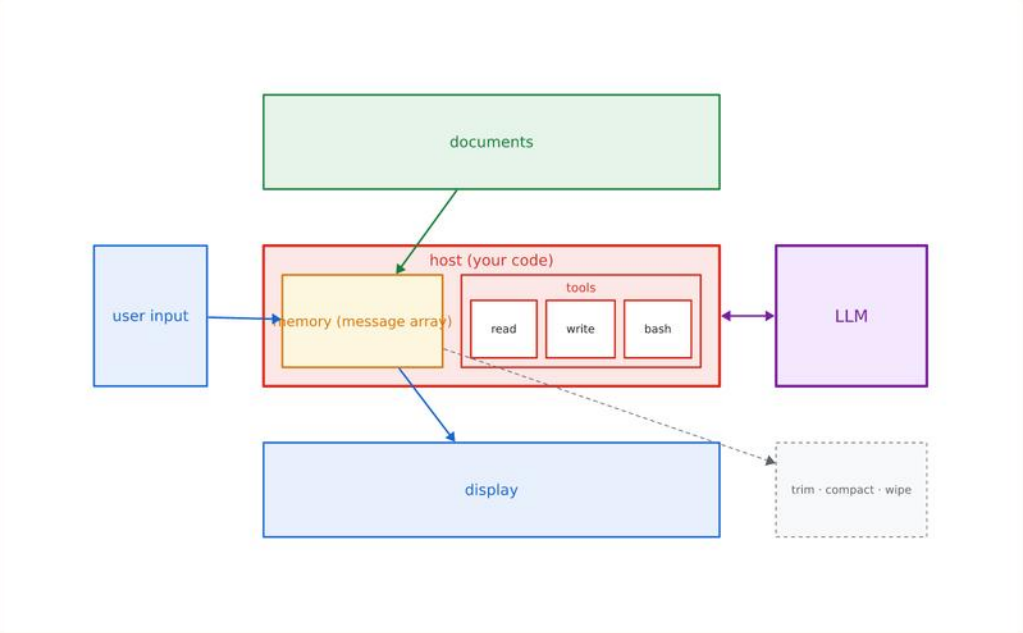
# Sessions: The Power of the Wipe

- **Wipe as a Feature** : Clearing memory prevents toxic context or mistaken theories from persisting.
- **Clean Handoffs** : Start new sessions with an explicit summary prompt describing the current baseline.
- **Reproducible Testing** : Isolated sessions allow deterministic testing and eval replay.



# The Machine, Assembled

- **Deconstructing Modern Agents** : Tools like Cursor, Claude Code, and Copilot are this exact architecture.
- **Every Box Is Code** : Loop (Ch 2), Knowledge (Ch 3), Tools (Ch 4), Toolbox (Ch 5), Memory (Ch 6).
- **No Magic** : An agent is a stateless HTTP call, an array, JSON schemas, and a loop.



SECTION 7

# Recap & Knowledge Check

Summary, Review Questions, and Next Steps

# Key Takeaways & What Is Next

- **Stateless Core** : The model has no persistent memory; the host manages all context.
- **Structured Tool Contract** : Never scrape text; declare schemas and handle tool blocks.
- **Memory as a Budget** : Proactively trim and compact history before hitting token limits.
- **Session 3 Preview** : Extending the agent harness into autonomous coding and test repair.

```
// The Complete Agent Harness Architecture
while (taskPending)
{
    // 1. Compact memory if token budget > 80%
    if (GetTokenCount(memory) > Threshold)
        memory = await CompactHistory(memory);

    // 2. Call model with tools & memory
    var response = await CallLLM(memory, tools);

    // 3. Dispatch tool calls or display final reply
    if (response.HasToolUse)
        await DispatchToolsAndAppend(memory, response);
    else
        Display(response.Text);
}
```

# Check Your Understanding

1. What component in an agent architecture owns and stores the message array?
2. How do multi-modal inputs (images, attachments) and tool results enter a message?
3. Why should you treat streamed LLM outputs as unverified until completion?
4. What is the fundamental problem with the naive 'append-only' loop?
5. What is the operational difference between RAG and whole-document attachment?
6. Why is the 'oldstring / newstring' surgical write pattern superior to whole-file replacement?
7. How does the 'oldstring' parameter serve as a safety lock during file edits?
8. What are the three primary operations used by the host to manage memory growth?

# Check Your Understanding: Answer Key (1–4)

**1. What component in an agent architecture owns and stores the message array?**
➜ The Host (your application code). The LLM is a stateless HTTP engine that retains no memory.

**2. How do multi-modal inputs (images, attachments) and tool results enter a message?**
➜ As Content Blocks inside the message's content array (typed as text, image, or tool_result).

**3. Why should you treat streamed LLM outputs as unverified until completion?**
➜ Partial output is incomplete. Acting on half-streamed JSON causes parsing and runtime failures.

**4. What is the fundamental problem with the naive 'append-only' loop?**
➜ Token Explosion. Re-sending growing arrays multiplies costs, latency, and exhausts context windows.

# Check Your Understanding: Answer Key (5–8)

**5. What is the operational difference between RAG and whole-document attachment?**
➜ RAG retrieves only relevant candidate passages, whereas attachments inject the entire file whole.

**6. Why is the 'oldstring / newstring' surgical write pattern superior to whole-file replacement?**
➜ Token efficiency (costs proportional to diff), clear reviewable diffs, and zero risk to untouched code.

**7. How does the 'oldstring' parameter serve as a safety lock during file edits?**
➜ It verifies target file contents before replacing; mismatched or ambiguous targets fail safely.

**8. What are the three primary operations used by the host to manage memory growth?**
➜ Trim (drop oldest turns), Compact (summarize past turns into a handoff block), and Wipe (reset session).

# Thank You

Questions? Comments? Concerns?

Monarch Wadia · monarchwadia.com