Mastering AWS Strands & Bedrock AgentCore

From Fresh Graduate to Production Agent Engineer

A practical, project-driven guide covering:

- The Strands Agents SDK (model-driven agent loop)
- The nine Bedrock AgentCore services
- Tools, MCP, memory, identity, and observability
- Multi-agent patterns (Agent-as-Tool, Swarm, Graph, Workflow)
- Production deployment, security, and cost
- Six hands-on projects and capstone exercises

Current to May 2026 — based on Strands Agents SDK v1.38+ and Amazon Bedrock AgentCore GA.

Table of Contents

How to Use This Book — 5
Chapter 1: The Agent Revolution — Why Strands and AgentCore Matter — 6
Chapter 2: Foundations — LLMs, Tools, and the Agent Loop — 10
Chapter 3: Your First Strands Agent — Hello, Agent World — 16
Chapter 4: Tools Deep Dive — Functions, MCP, and the Tool Ecosystem — 22
Chapter 5: Memory, State, and Context Management — 29
Chapter 6: Multi-Agent Patterns — Agent-as-Tool, Swarm, Graph, Workflow — 35
Chapter 7: Streaming, Async, and Bidirectional Agents — 42
Chapter 8: Observability with OpenTelemetry & CloudWatch — 47
Chapter 9: Introducing Amazon Bedrock AgentCore — 52
Chapter 10: AgentCore Runtime — Serverless Microsoft VMs for Agents — 57
Chapter 11: AgentCore Memory — Short- and Long-Term — 63
Chapter 12: AgentCore Gateway — APIs as MCP Tools — 68
Chapter 13: AgentCore Identity — Auth for Non-Human Workloads — 73
Chapter 14: AgentCore Built-in Tools — Browser & Code Interpreter — 78
Chapter 15: AgentCore Observability, Evaluations, and Policy — 83
Chapter 16: End-to-End — Deploying a Strands Agent on AgentCore — 88
Chapter 17: Production Hardening — Security, Cost, and Reliability — 94
Chapter 18: Six Capstone Projects to Build Your Portfolio — 99
Chapter 19: Interview Prep — Concepts, Questions, and System Design — 104
Appendix A: Cheat Sheets and Command Reference — 109
Appendix B: Resources, Communities, and Where to Go Next — 113

How to Use This Book

This book is designed as a single-track curriculum. If you read it cover to cover and do the exercises, you will move from "I have heard of AI agents" to "I have built and deployed production agents on AWS." That is the goal. No prerequisite beyond comfort with Python and basic AWS concepts (an IAM role, what an S3 bucket is).

The structure follows a deliberate progression:

Part I (Chapters 1–2): Concepts. What an agent actually is, why the field exists, and the mental model you need before writing code.

Part II (Chapters 3–8): Strands Agents SDK. Open-source, model-driven, framework-agnostic. This is the layer where you write your agent's logic.

Part III (Chapters 9–15): Bedrock AgentCore. AWS's managed platform for running agents in production. Each chapter covers one of the nine AgentCore services.

Part IV (Chapters 16–17): Putting it together. Deploying Strands code on AgentCore, then hardening for production.

Part V (Chapters 18–19): Career. Portfolio projects you can show employers, and interview preparation focused on what hiring managers actually ask.

Conventions used throughout:

Concept: Yellow boxes flag essential ideas worth re-reading.

Tip: Teal boxes are practical tips that save you hours.

Warning: Red boxes flag gotchas that will bite you in production.

Code blocks are runnable Python 3.10+ unless noted. Replace placeholder ARNs and account IDs with your own. Every chapter ends with exercises — do them. Reading code is not the same as writing it.

Tip: Set up your environment before Chapter 3: an AWS account with Bedrock model access enabled in **us-west-2** for Claude Sonnet 4, Python 3.10+, and AWS credentials configured via *aws configure*.

Chapter 1: The Agent Revolution

Why Strands and AgentCore Matter

In 2023, building an AI feature meant calling a chat API and rendering the response. By 2025, the conversation had shifted. Models had grown capable enough to plan multi-step tasks, call external functions, recover from errors, and operate over hours of work rather than seconds. The unit of software was no longer "a prompt." It was "an agent."

An agent, in the modern sense, is a program where a language model sits in a loop: it reads context, decides on an action, executes that action (often by calling a tool), observes the result, and decides again. The loop continues until the model judges the task complete. This sounds simple. Building it reliably, securely, and at scale is not.

What changed in 2025-2026

Three things shifted simultaneously, and together they created the demand for the tools this book covers.

First, the models got good at tool use. Claude Sonnet 4 and Opus 4, GPT-5, Gemini 2.5, Llama 4 — all reached a quality threshold where giving the model a list of tools and a goal actually works most of the time. Before that, you needed elaborate scaffolding (LangChain chains, hand-rolled state machines) to make the model behave. Now you just hand it tools and a system prompt.

Second, the Model Context Protocol (MCP) emerged as a real standard. Anthropic open-sourced MCP in late 2024 and within a year, hundreds of MCP servers existed for everything from GitHub to Slack to Salesforce. Suddenly, "give the agent access to your company's tools" stopped being a months-long integration project and started being a config file.

Third, production-grade infrastructure for agents became necessary. Once teams started deploying agents to real users, the same problems showed up everywhere: where does the agent run? How does it keep memory across sessions? How do you give it credentials without leaking them? How do you observe what it actually did when something goes wrong? How do you stop it from running for eight hours and bankrupting you? These are not problems the model solves. They are infrastructure problems.

Where Strands and AgentCore fit

Strands Agents is an open-source SDK that AWS released in mid-2025 and has rapidly iterated on since. It is what you use to write the agent. Three lines of Python and you have a working agent. It is model-agnostic (Bedrock, Anthropic, OpenAI, Ollama, anything), framework-flexible, and small enough that you can read the entire source if you want. Inside AWS, Strands powers Amazon Q Developer, AWS Glue, Kiro, VPC Reachability Analyzer, and more.

Amazon Bedrock AgentCore, announced at AWS re:Invent 2025 and GA in early 2026, is the managed platform for running agents. It is not a framework — you bring whatever framework you like (Strands, LangGraph, CrewAI, LlamaIndex, custom). What AgentCore gives you is nine services that solve the infrastructure problems above: Runtime (where the agent executes, in isolated microVMs), Memory

(managed conversation memory), Gateway (turn any API into an MCP tool), Identity (OAuth and IAM for agents), Browser (a managed browser for the agent to drive), Code Interpreter (sandboxed code execution), Observability (traces and metrics), Evaluations (quality monitoring), and Policy (guardrails).

> **Concept:** Strands is "what your agent *is*." AgentCore is "where your agent *lives*." You can use Strands without AgentCore (run locally, deploy to Lambda, etc.). You can use AgentCore without Strands (deploy a LangGraph agent there). But the combination is the path AWS is investing most heavily in, and the one we focus on in this book.

Who needs this stack

If you are building anything that fits one of these descriptions, you are in the right place: a customer support agent that needs memory across conversations; an internal developer assistant that runs commands across cloud accounts; a workflow agent that orchestrates ten APIs to onboard a new employee; a data analysis agent that writes and runs code on user-uploaded files; a research agent that drives a browser to gather information. All of these have moved from "research project" to "shipped product" in the last 18 months, and the teams who built them used tools like the ones this book covers.

The path through this book

A reasonable schedule is two chapters per week for ten weeks, with weekends for exercises. By the end of week three you will have a working agent on your laptop. By week six, a multi-agent system with memory. By week eight, the same system deployed to AgentCore. By week ten, a portfolio project you can show in interviews.

Let us begin with what an agent actually is.

Chapter 1 Exercises

1. Sign up for an AWS account if you do not have one. Enable model access for Anthropic Claude Sonnet 4 in **us-west-2** via the Bedrock console (Model access → Manage model access).

2. Install Python 3.10+ and create a fresh virtual environment named *strands-book*.

3. Run *aws sts get-caller-identity* and confirm you see your account ID.

4. Read the AWS Open Source Blog post announcing Strands, and the AWS What's New page on AgentCore GA. Write a one-paragraph summary of each in your own words.

5. List three problems you have personally seen (in projects, internships, or hobby code) where an agent might be a better solution than a fixed-pipeline approach. We will revisit this list in Chapter 18.

Chapter 2: Foundations

LLMs, Tools, and the Agent Loop

Before you write Strands code, you need to understand four things deeply: what a language model actually does, what a tool is in this context, what the agent loop looks like, and what makes one agent better than another. This chapter is conceptual. There is no code yet, by design. The single most common reason people build broken agents is that they skipped this chapter.

2.1 The language model: a function from text to text

A modern LLM is, mechanically, a function. You hand it a sequence of tokens (think "words and word-pieces") and it produces a probability distribution over the next token. You sample from that distribution, append the result, and repeat. That is all it does. Everything else — "reasoning," "tool use," "planning" — is the same operation applied to specially-formatted text.

This is important because it tells you what the model "can" and "cannot" do. The model cannot execute code. It cannot fetch a URL. It cannot read a database. What it can do is produce text that says "I want to execute this code" or "I want to fetch this URL," and your code — the agent runtime — reads that text and actually does the thing.

Concept: The model never touches your filesystem, your database, or the internet. The agent runtime does. The model only produces structured text saying which tool to invoke and with what arguments. The runtime parses that text, calls the tool, and feeds the result back to the model as more text. The illusion of agency comes from this loop.

2.2 What is a tool, really?

In the agent world, a tool is any function the model can ask to be called. It has a name, a description, a schema for its inputs, and an implementation. The first three are sent to the model. The fourth lives in your code.

When you register a tool, the model sees something like:

```
Tool: get_weather
Description: Get current weather for a city.
Input schema:
  city (string, required): the city name
  units (string, optional): "celsius" or "fahrenheit"
Returns: a JSON object with temperature, conditions, and humidity.
```

The model decides, based on the conversation, when to invoke *get_weather*. It produces a tool-call message specifying the name and arguments. Your runtime intercepts that message, runs the actual Python function, captures the return value, and feeds it back to the model as a tool-result message. The model sees the result and continues the conversation.

Tip: The description and parameter names matter **a lot**. The model uses these to decide when and how to call your tool. A tool named *fetch_data()* with description "fetches data" will be called erratically. A tool

named *get_employee_record_by_id(employee_id: str)* with description "Retrieve an employee's full record from the HRIS system, including name, role, manager, and start date" will be called precisely.

2.3 The agent loop

Here is the loop, in pseudocode. Every agent framework — Strands, LangGraph, CrewAI, your hand-rolled one — implements some variant of this:

```
def agent_loop(model, tools, system_prompt, user_input):
    messages = [
        {"role": "system",  "content": system_prompt},
        {"role": "user",    "content": user_input},
    ]

    while True:
        response = model.generate(messages, tools=tools)

        if response.stop_reason == "end_turn":
            return response.text

        if response.stop_reason == "tool_use":
            messages.append({"role": "assistant", "content": response.content})
            for tool_call in response.tool_calls:
                tool_fn = lookup_tool(tools, tool_call.name)
                result = tool_fn(**tool_call.arguments)
                messages.append({
                    "role": "tool",
                    "tool_call_id": tool_call.id,
                    "content": str(result),
                })
            continue

        raise UnexpectedStopReason(response.stop_reason)
```

That is essentially what Strands does for you, plus error handling, streaming, observability, multi-modal support, multi-agent orchestration, and configurable context-window management. But the core is fifteen lines.

2.4 What separates a good agent from a bad one

Three things, almost always:

The system prompt. This is the most under-appreciated lever in agent engineering. A good system prompt tells the model who it is, what its goal is, what tools it has, when to ask for clarification, when to stop, and what to do when it is stuck. A vague system prompt produces vague agents. We will write many system prompts in this book.

The tool design. Tools should do one thing well, have descriptive names, take unambiguous arguments, and return useful information (not just "OK"). A common anti-pattern: ten tools that all wrap the same API with different parameter combinations. Better: one tool with clear arguments.

The context management. Models have finite context windows. A long conversation will overflow. A bad agent crashes or starts forgetting things. A good agent summarizes, prunes, or offloads to persistent memory. Strands handles this; AgentCore Memory handles it at a higher level. We cover both.

2.5 A taxonomy of agent shapes

Most agents fall into one of four shapes. Recognizing which one you are building helps you pick the right pattern early.

Shape	When to use	Strands pattern
Single-agent loop	Single domain, clear goal, <20 tools	Plain Agent
Agent-as-tool	Specialist sub-agents called by a lead agent	Nested agents
Swarm	Peer agents debate or collaborate freely	Swarm primitive
Graph / Workflow	Deterministic sequence with branches	Graph / Workflow primitive

2.6 Failure modes you will see

Tool-call loops. The model calls the same tool over and over with slight variations, never reaching an answer. Usually caused by tool descriptions that overlap or a system prompt that does not say "stop when you have the answer."

Hallucinated tool calls. The model invents a tool that does not exist. Strands handles this gracefully, but it signals a system-prompt or tool-design problem.

Context overflow. Conversation grows past the model's window. Tokens get cut. The agent forgets. Solutions: summarize, persist to memory, or use AgentCore Memory.

Runaway costs. The model loops for thousands of turns. Always set a max iteration limit.

> **Warning:** Set max iterations on every agent, every time. The default in Strands is sensible, but double-check. An agent without a max-iteration cap is a cost incident waiting to happen.

Chapter 2 Exercises

1. Without looking at the chapter, write the agent loop from memory. Check yours against the version above. What did you miss?

2. Pick a tool you have used (a CLI command, a REST endpoint, anything). Write what its tool description, parameter names, and parameter descriptions should be if an LLM were to call it. Make it precise.

3. List three real-world tasks where context overflow would be the dominant problem. We will solve all three in later chapters.

4. Read one open-source agent's source code. Suggestion: the Strands quickstart sample on GitHub. Identify where the agent loop lives in the code.

Chapter 3: Your First Strands Agent

Hello, Agent World

This chapter is where you stop reading and start running code. By the end of it, you will have written and run three agents on your laptop, with increasing sophistication.

3.1 Installation

From your virtual environment:

```
pip install strands-agents strands-agents-tools

# Verify
python -c "from strands import Agent; print('Strands ready')"
```

Configure AWS credentials and ensure Bedrock model access for Claude Sonnet 4 in *us-west-2* is enabled.

```
aws configure        # if not done already
aws bedrock list-foundation-models --region us-west-2 \
    --query 'modelSummaries[?contains(modelId, `claude`)].modelId'
```

3.2 The three-line agent

Save this as *agent_v1.py*:

```
from strands import Agent

agent = Agent()
response = agent("Explain the agent loop in two sentences.")
print(response)
```

Run it. You should see the model respond. Three things just happened: Strands picked the default model (Claude Sonnet 4 on Bedrock), constructed a one-turn conversation, ran the agent loop (which terminated immediately because there were no tools to call), and returned the text.

3.3 Adding a tool

The model cannot do math reliably. Let us fix that.

```
from strands import Agent
from strands_tools import calculator

agent = Agent(tools=[calculator])
print(agent("What is the square root of 1764, then divided by 3?"))
```

Run it. You will see Strands log the tool call ("calculator was invoked with expression='sqrt(1764)/3'") and then return the answer (14). This is the agent loop in action: the model decided to call the calculator, Strands executed it, the result went back to the model, the model produced the final answer.

Tip: Enable verbose logging while learning. Set the environment variable *STRANDS_LOG_LEVEL=DEBUG* to see every step of the loop, including the model's reasoning before

each tool call.

3.4 Defining your own tool

The *@tool* decorator turns any Python function into a tool the agent can use. The docstring becomes the tool description, which the model reads.

```
from strands import Agent, tool

@tool
def word_count(text: str) -> int:
    """Count the number of words in a piece of text."""
    return len(text.split())

@tool
def reverse_string(s: str) -> str:
    """Reverse the characters in a string."""
    return s[::-1]

agent = Agent(
    tools=[word_count, reverse_string],
    system_prompt="You are a text-analysis assistant. Use your tools precisely.",
)

print(agent("How many words are in 'the quick brown fox jumps over the lazy dog' and what is it reversed?
```

The model will call *word_count* once and *reverse_string* once, then synthesize. Notice the system prompt — even a brief one shapes behavior. Try running this without the system prompt; the agent will still work but its style will differ.

3.5 Tool return types and rich results

Tools can return strings, numbers, dicts, lists, or anything JSON-serializable. The return value is converted to text and fed back to the model.

```
@tool
def get_user_profile(user_id: str) -> dict:
    """Retrieve a user profile by user ID.

    Args:
        user_id: The unique identifier of the user (UUID format).

    Returns:
        A dictionary with keys: name, email, role, created_at, status.
    """
    # In real code, you'd hit a database here.
    return {
        "name": "Asha Patel",
        "email": "asha@example.com",
        "role": "Senior Engineer",
        "created_at": "2023-04-12",
        "status": "active",
    }
```

Notice the detailed docstring. The model uses it to construct correct calls. Loose docstrings produce loose behavior.

3.6 System prompts that work

A system prompt should answer five questions: who is this agent, what is its goal, what tools does it have access to (briefly — the model already sees the tool list), what should it do when uncertain, and when should it stop.

```
SYSTEM_PROMPT = """You are an internal support agent for ACME Corp engineering team.

Your goal is to help engineers find information about teammates, projects, and
on-call schedules. You have tools to query the employee directory, project tracker,
and PagerDuty.

When a user asks something ambiguous (for example, two people with the same first
name), ask one clarifying question before acting. Never guess.

When you have answered the user's question fully, end the conversation cleanly.
Do not invent information. If a tool returns no results, say so plainly.
"""

agent = Agent(tools=[...], system_prompt=SYSTEM_PROMPT)
```

3.7 Choosing a different model

Strands is model-agnostic. To use a non-default model, pass a *model* argument.

```
from strands import Agent
from strands.models import BedrockModel

# Use Claude Opus 4 on Bedrock for harder tasks
model = BedrockModel(
    model_id="anthropic.claude-opus-4-20250514-v1:0",
    region="us-west-2",
    temperature=0.2,
)
agent = Agent(model=model)
```

Other providers work similarly — *AnthropicModel*, *OpenAIModel*, *OllamaModel*, and so on. Each has its own configuration knobs.

3.8 Streaming responses

For chat-like UIs you want incremental output. Strands supports streaming natively.

```
import asyncio
from strands import Agent

agent = Agent()

async def main():
    async for event in agent.stream_async("Write a haiku about kubernetes."):
        if event.get("data"):
            print(event["data"], end="", flush=True)
    print()

asyncio.run(main())
```

Events arrive as the model generates them. We cover streaming, including bidirectional audio streaming, in detail in Chapter 7.

3.9 Putting it together: a mini research assistant

Let us build something slightly real. An agent that takes a topic and produces a short briefing, using a fake "search" tool we will replace with a real one in Chapter 4.

```
from strands import Agent, tool
from datetime import datetime

@tool
def search_web(query: str, max_results: int = 5) -> list[dict]:
    """Search the web. Returns a list of {title, url, snippet} dicts."""
    # Stub. Returns canned data. Replace with real search in Chapter 4.
    return [
        {"title": f"Result {i+1} for {query}",
         "url": f"https://example.com/{i+1}",
         "snippet": f"This is a snippet about {query} (result {i+1})."}
        for i in range(max_results)
    ]

@tool
def current_date() -> str:
    """Return today's date in ISO 8601 format."""
    return datetime.utcnow().date().isoformat()

SYSTEM = """You are a research briefing agent. Given a topic, produce a concise
one-page briefing with: a 3-sentence summary, 3 key facts (each with source URL),
and 2 open questions. Use the search_web tool to gather information and current_date
to timestamp your work."""

agent = Agent(
    tools=[search_web, current_date],
    system_prompt=SYSTEM,
)

print(agent("Brief me on retrieval-augmented generation in 2026."))
```

Run it. Read the output critically. Where does the agent succeed? Where does the fake search data trip it up? This is the kind of qualitative evaluation you will do constantly as an agent engineer.

Chapter 3 Exercises

1. Build an agent with three tools: a fake weather API, a city-to-coordinates lookup, and a clothing recommendation function. Ask the agent "what should I wear in Lisbon today?" and trace which tools it calls in what order.

2. Take the research briefing agent above and change the system prompt to enforce a different output format (a JSON object instead of prose). Note what changes.

3. Intentionally write a bad tool: vague docstring, ambiguous parameter names. Compare its behavior to a well-described version. Document the difference.

4. Set the agent's max iterations to 2 explicitly using *Agent(max_iterations=2)*. Construct a query that would normally take more turns. Observe what happens.

Chapter 4: Tools Deep Dive

Functions, MCP, and the Tool Ecosystem

Tools are the bridge between the model's reasoning and the outside world. Get tool design right and your agent feels intelligent. Get it wrong and your agent feels broken. This chapter covers four levels of tool sophistication: simple decorated functions, complex stateful tools, MCP servers, and the strands-agents-tools library.

4.1 Anatomy of a well-designed tool

A good tool has:

A precise, verb-led name. *get_invoice* not *invoice*. *create_calendar_event* not *calendar*. *list_active_users* not *users*.

A type-hinted signature. Strands reads your type hints to construct the input schema. *def get_invoice(invoice_id: str, include_line_items: bool = False)* tells the model exactly what to pass.

A docstring that names every parameter and explains the return value.

Idempotency where possible. If the model calls your tool twice with the same arguments, the second call should be safe.

Useful error messages. If the tool fails, return a message the model can act on, not an opaque stack trace.

```
from strands import tool

@tool
def get_invoice(invoice_id: str, include_line_items: bool = False) -> dict:
    """Retrieve an invoice from the billing system.

    Args:
        invoice_id: The invoice ID in the format INV-YYYY-NNNNN.
        include_line_items: If True, includes per-line-item details (slower).

    Returns:
        A dictionary with keys: id, customer_id, amount_cents, currency,
        status, issued_at, due_at, and optionally line_items.
        Returns {"error": "not_found"} if the invoice does not exist.
    """
    if not invoice_id.startswith("INV-"):
        return {"error": "invalid_format",
                "message": "invoice_id must start with INV-"}
    # ... real lookup ...
    return {"id": invoice_id, "customer_id": "C-001",
            "amount_cents": 49900, "currency": "USD",
            "status": "paid", "issued_at": "2026-04-01",
            "due_at": "2026-04-30"}
```

4.2 Tools that need state or configuration

Sometimes a tool needs a database connection, an API key, or session state. Use a closure or a class.

```
from strands import tool

def make_db_tool(db_connection):
    @tool
    def query_employees(department: str) -> list[dict]:
        """List employees in a given department."""
        rows = db_connection.execute(
            "SELECT id, name, role FROM employees WHERE dept = %s",
            (department,)
        ).fetchall()
        return [dict(r) for r in rows]
    return query_employees

# Usage
import psycopg2
conn = psycopg2.connect(DATABASE_URL)
query_employees = make_db_tool(conn)

agent = Agent(tools=[query_employees], ...)
```

For more complex tools, subclass *strands.tools.Tool* directly. This gives you hooks for setup, teardown, and rich error handling. See the Strands docs for the full API.

4.3 Tools with side effects: principles

A tool that reads the world is safe to call. A tool that changes the world — sends an email, charges a card, deletes a file — is dangerous if the model invokes it incorrectly. Several mitigations:

Dry-run modes. Add a *dry_run: bool = False* parameter. The model can preview the action; you require *dry_run=False* for the real call.

Human approval. Have the tool emit an approval-required result. The runtime surfaces it to a human; the human approves; the tool is invoked.

Constrained inputs. A tool that can only send email to *@yourcompany.com* addresses cannot exfiltrate data.

Cedar policies (AgentCore Policy). At the platform level, enforce rules the model cannot violate. We cover this in Chapter 15.

> **Warning:** Never give an agent unconstrained access to mutate production data. The model will eventually do something unexpected. Constrain the tools, not just the prompt.

4.4 The strands-agents-tools library

Strands ships with 20+ pre-built tools. They are useful out of the box and useful as examples of well-designed tools. Selected highlights:

Category	Tools	Notes
File I/O	file_read, file_write, editor	Local filesystem access
Web	http_request, scrape_url	Network calls

AWS	aws_cli, s3_read, s3_write	Wraps boto3 and aws CLI
Compute	calculator, python_repl	Math and code execution
System	shell, current_time	Shell commands, dates
LLM	image_reader, generate_image	Multimodal helpers
Multi-agent	agent_as_tool, swarm, workflow	Composition primitives

Warning: Tools like *shell*, *python_repl*, and *http_request* are powerful and dangerous. Restrict them to development environments, or wrap them with allowlists in production.

4.5 MCP — Model Context Protocol

MCP is an open standard for connecting AI applications to tools and data. Think of it as "USB for AI agents." Anthropic released the spec in late 2024 and the ecosystem has exploded since. There are MCP servers for GitHub, Slack, Postgres, Linear, Notion, Salesforce, Jira, AWS, and hundreds more.

Strands has native MCP support. You can plug any MCP server into your agent.

```
from strands import Agent
from strands.tools.mcp import MCPClient
from mcp import StdioServerParameters

# Connect to an MCP server (here: a Postgres MCP server installed via uvx)
pg_server = MCPClient(
    StdioServerParameters(
        command="uvx",
        args=["mcp-server-postgres", "--connection-string",
              "postgresql://user:pass@localhost/mydb"],
    )
)

with pg_server:
    tools = pg_server.list_tools_sync()
    agent = Agent(tools=tools, system_prompt="You are a database analyst.")
    print(agent("How many orders were placed last week, by region?"))
```

The agent now has access to every tool the Postgres MCP server exposes — typically things like *list_tables*, *describe_table*, *run_query*. You did not write a single line of database glue code.

Tip: When picking MCP servers, prefer ones from official sources (the company that owns the service, or AWS Labs, or Anthropic). Third-party MCP servers run code on your machine and can have wide access. Read before installing.

4.6 Hosting your own MCP server

Sometimes you want to expose your internal APIs as MCP tools so any agent (yours or your colleagues') can use them. Strands provides a tiny MCP-server framework:

```
# my_company_mcp.py
from strands.tools.mcp import MCPServer, tool
```

```
@tool
def get_customer(customer_id: str) -> dict:
    """Look up a customer by ID."""
    # ... your lookup ...
    return {"id": customer_id, "name": "Acme Co.", "tier": "enterprise"}

@tool
def list_open_tickets(customer_id: str) -> list[dict]:
    """List open support tickets for a customer."""
    # ...
    return [{"ticket_id": "T-1", "subject": "Login broken"}]

server = MCPServer(name="acme-internal", tools=[get_customer, list_open_tickets])

if __name__ == "__main__":
    server.run_stdio()
```

Now any MCP-compatible client — Claude Desktop, Cursor, another Strands agent, an AgentCore Gateway — can pull these tools in. We will revisit this pattern in Chapter 12, where AgentCore Gateway gives you a managed version of this.

4.7 Tool selection at scale

When you have 50+ tools, models start getting confused. Strategies:

Group tools by domain into separate agents (Agent-as-Tool pattern, Chapter 6).

Use semantic search over your tool catalog and only attach the top-K relevant tools per turn. AgentCore Gateway has this built in.

Hierarchical tools. A single *aws_api* tool that takes *service* and *operation* parameters rather than 200 separate tools.

Chapter 4 Exercises

1. Write a tool wrapping the GitHub REST API: *get_repo(owner, repo) -> dict*. Make the docstring precise. Give your agent the tool and ask it about a public repo.

2. Install an official MCP server (the GitHub MCP server is a good one). Connect it to a Strands agent. Build an agent that summarizes pull requests in a repo.

3. Write a tool with a dry-run mode for sending Slack messages. Verify the model invokes it with *dry_run=True* first when you ask it to "draft a message."

4. Take a tool from *strands-agents-tools* on GitHub. Read its source. Note what makes it good or bad. Write down three lessons.

Chapter 5: Memory, State, and Context Management

An agent without memory is a chatbot that forgets the user every turn. An agent with poor memory is worse: it remembers irrelevant details and forgets important ones. This chapter covers the three layers of agent memory: in-context conversation history, session memory across conversations, and long-term semantic memory.

5.1 The three memory layers

Layer	Lifetime	Where it lives	Used for
Working / context	Single agent invocation	Model context window	Current task reasoning
Session	Across turns in one conversation	Conversation manager	Multi-turn chat
Long-term	Across conversations forever	External store (DB / vector index)	Personalization, recall

5.2 In-context conversation memory

By default, a Strands *Agent* object preserves message history across calls. The second call to the same agent sees the first call's messages and the model's responses.

```
from strands import Agent

agent = Agent(system_prompt="You are a friendly assistant. Remember user details.")

print(agent("Hi, I'm Alex and I work on agentic AI at a fintech."))
print(agent("What did I just tell you about my role?"))
# The second response will reference fintech and agentic AI.
```

When the conversation grows past the model's context window, Strands' default conversation manager truncates the oldest non-system messages. You can configure a summarizing conversation manager that compresses old turns instead.

```
from strands import Agent
from strands.agent.conversation_manager import SummarizingConversationManager

agent = Agent(
    conversation_manager=SummarizingConversationManager(
        keep_recent_turns=10,           # always keep last 10 turns verbatim
        summarize_threshold_tokens=8000  # summarize older turns past this point
    )
)
```

5.3 Session persistence

A web app that creates a new *Agent* object per HTTP request loses history. You need to persist sessions externally. Two approaches.

Approach A: Serialize and rehydrate. Store the messages list in your database, load it on each request, attach it to a fresh agent.

```
# After each turn
session_store[user_id] = agent.messages    # list of dicts

# At the start of the next turn
agent = Agent(...)
agent.messages = session_store.get(user_id, [])
```

Approach B: Use Strands' SessionManager. A built-in abstraction with backends for Redis, DynamoDB, and local files.

```
from strands import Agent
from strands.session import S3SessionManager

session_manager = S3SessionManager(
    bucket="my-agent-sessions",
    prefix="prod/",
)

agent = Agent(
    session_id=f"user-{user_id}",
    session_manager=session_manager,
)
# History is automatically loaded and saved.
```

5.4 Long-term semantic memory

Beyond a single conversation, you may want the agent to remember user preferences, prior decisions, or domain facts. This is long-term memory and it usually involves two operations: **extract and store** when something important happens, and **retrieve** relevant facts at the start of each turn.

You can build this yourself with a vector database, but in production you almost always want a managed service. AgentCore Memory (Chapter 11) is purpose-built for this. Until we get there, here is a minimal homemade version:

```
from strands import Agent, tool

memory_store = {}  # in real code: a vector DB or AgentCore Memory

@tool
def remember(fact: str, user_id: str) -> str:
    """Save an important fact about the user for future conversations."""
    memory_store.setdefault(user_id, []).append(fact)
    return "Saved."

@tool
def recall(user_id: str) -> list[str]:
    """Retrieve everything you have remembered about the user."""
    return memory_store.get(user_id, [])

SYSTEM = """You are a personal assistant. At the start of each conversation, call
recall to see what you know about the user. When the user shares preferences,
biographical facts, or constraints, call remember to save them."""

agent = Agent(tools=[remember, recall], system_prompt=SYSTEM)
```

This works, but at scale you want semantic search (the user mentions "my dog" and the agent retrieves the fact "user's dog is named Mochi"), automatic fact extraction (don't rely on the model deciding to call *remember*), and lifecycle management. That's what AgentCore Memory does.

5.5 Context-window strategy: a decision tree

When your agent will exceed context, choose:

Situation	Strategy
Short interactions (<50 turns), single session	Default truncation is fine
Long single sessions, recall matters	SummarizingConversationManager
Multi-session, same user	SessionManager + long-term store
Many users, personalization required	AgentCore Memory
Large document corpus to consult	RAG over a vector store, as a tool

5.6 RAG as a tool, not a framework

Retrieval-Augmented Generation is often presented as its own thing. In the agent world, RAG is just a tool: *search_corpus(query) -> list[chunk]*. The model decides when to call it. You can build sophisticated retrieval by composing tools: a high-level "answer from docs" tool that internally does hybrid search, reranking, and citation formatting.

Example with a simple chunk store:

```
from strands import tool

@tool
def search_docs(query: str, k: int = 5) -> list[dict]:
    """Search the company knowledge base. Returns up to k relevant passages
    with their source URLs. Use this whenever the user asks about company
    policies, products, or processes."""
    # Hit your vector DB / OpenSearch / Kendra here
    chunks = my_vector_search(query, k=k)
    return [{"text": c.text, "source": c.metadata["url"]} for c in chunks]
```

The system prompt should instruct the agent to cite sources from the returned chunks. Models are very good at this if you ask.

Chapter 5 Exercises

1. Build a chat agent with file-backed session persistence. Save sessions to *./sessions/{user_id}.json*. Resume them across restarts.

2. Implement a *SummarizingConversationManager*. Have a 50-turn conversation and observe how older turns get compressed.

3. Add a simple long-term memory using a SQLite table. Have the agent remember user preferences across three separate conversations.

4. Wrap a small text corpus (say 20 markdown files) with a RAG tool. Ask questions and verify the agent cites the right source.

Chapter 6: Multi-Agent Patterns

Agent-as-Tool, Swarm, Graph, Workflow

A single agent with too many tools, too many responsibilities, or too long a workflow becomes confused and slow. The fix is decomposition: split into specialist agents that collaborate. Strands offers four canonical patterns for this. Choose the one that matches your control needs.

6.1 The decision: which pattern?

Pattern	Control flow	When to use
Agent-as-Tool	Lead agent decides which specialist to call	Clear hierarchy, sub-agents are domain experts
Swarm	Peers collaborate, model-driven	Brainstorming, debate, exploration
Graph	Static DAG of agents	Deterministic stages with branching
Workflow	Linear pipeline with conditionals	Approval flows, ETL-like sequences

6.2 Agent-as-Tool: the workhorse

In this pattern, you have one lead agent and several specialist agents. Each specialist is exposed to the lead agent as a tool. The lead decides when to call each specialist based on the user's request.

```
from strands import Agent, tool

# Specialist 1: SQL analyst
sql_agent = Agent(
    system_prompt="You are a SQL expert. Given a question, write and run the query.",
    tools=[run_sql_query],
)

# Specialist 2: Chart generator
chart_agent = Agent(
    system_prompt="You generate matplotlib charts from structured data.",
    tools=[render_chart],
)

# Specialist 3: Report writer
writer_agent = Agent(
    system_prompt="You write polished executive summaries.",
)

# Expose specialists as tools to the lead agent
@tool
def ask_sql(question: str) -> str:
    """Delegate a data-retrieval question to the SQL specialist agent."""
    return str(sql_agent(question))

@tool
def make_chart(data_description: str) -> str:
    """Delegate chart generation to the chart specialist."""
```

```
    return str(chart_agent(data_description))

@tool
def write_summary(content: str) -> str:
    """Delegate summary writing to the writing specialist."""
    return str(writer_agent(content))

lead = Agent(
    system_prompt="""You are a business analyst lead. For data questions,
    use ask_sql. For visualizations, use make_chart. For final reports, use
    write_summary. Coordinate the work.""",
    tools=[ask_sql, make_chart, write_summary],
)

print(lead("Give me a Q1 sales summary with a chart of top 5 products."))
```

This is the pattern AWS uses internally for many of its Strands-powered services. It scales well because each specialist has a focused tool set and a focused system prompt.

6.3 Swarm: peer collaboration

Swarms are useful when you want agents to debate, critique, or refine each other's output without a fixed hierarchy.

```
from strands import Agent
from strands.multiagent import Swarm

optimist = Agent(system_prompt="You are an optimistic strategist. Argue for the proposal.")
skeptic  = Agent(system_prompt="You are a skeptical risk analyst. Find weaknesses.")
synth    = Agent(system_prompt="You synthesize debates into balanced conclusions.")

swarm = Swarm(
    agents=[optimist, skeptic, synth],
    max_rounds=3,
)

result = swarm("Should ACME Corp expand into Brazil in Q3?")
print(result)
```

6.4 Graph: deterministic DAG

When you need to enforce that agent A runs before agent B, and C only runs if B returns "needs review," use a Graph.

```
from strands import Agent
from strands.multiagent import Graph, Node, Edge

intake = Agent(system_prompt="Classify the incoming ticket. Output JSON.")
billing = Agent(system_prompt="Resolve billing issues.")
technical = Agent(system_prompt="Resolve technical issues.")
escalate = Agent(system_prompt="Draft an escalation to a human.")

graph = Graph(
    nodes=[
        Node("intake", intake),
        Node("billing", billing),
        Node("technical", technical),
```

```
        Node("escalate", escalate),
    ],
    edges=[
        Edge("intake", "billing", condition="category == 'billing'"),
        Edge("intake", "technical", condition="category == 'technical'"),
        Edge("intake", "escalate", condition="urgency == 'high'"),
    ],
)

graph("My payment failed but my account also shows the wrong plan.")
```

6.5 Workflow: pipelines with checkpoints

Workflows are linear sequences with optional human-approval steps. Good for compliance- heavy flows.

```
from strands import Agent
from strands.multiagent import Workflow, Step, ApprovalGate

draft = Agent(system_prompt="Draft an outbound marketing email.")
legal = Agent(system_prompt="Review for legal issues. Approve or revise.")
send  = Agent(system_prompt="Send via the email tool.", tools=[send_email])

wf = Workflow(steps=[
    Step("draft", draft),
    Step("legal_review", legal),
    ApprovalGate("human_sign_off", required_role="marketing-director"),
    Step("send", send),
])

wf.start(input="Promote our new product launch on Friday.")
```

6.6 Sharing context between agents

Each agent has its own conversation. To share state, you typically pass structured data between them — not raw prompts. Treat agent inputs and outputs as typed interfaces.

> **Tip:** Make sub-agents return JSON, not prose, when their output will feed another agent. Models reliably produce structured output when asked. Use Strands' *output_format* parameter or instruct it in the system prompt.

Chapter 6 Exercises

1. Convert the research briefing agent from Chapter 3 into an Agent-as-Tool system: a lead, a search specialist, and a fact-checking specialist.

2. Build a Swarm of three agents that debate a product pricing decision.

3. Build a Graph that triages incoming support emails into three categories.

4. Build a Workflow with a human-approval gate. Implement the approval as a function that pauses and prints a prompt to the console.

Chapter 7: Streaming, Async, and Bidirectional Agents

Real-time UX demands streaming. Real-time voice demands bidirectional streaming. This chapter covers both, briefly, because the API is small but the patterns matter.

7.1 Streaming with stream_async

```
import asyncio
from strands import Agent

agent = Agent()

async def main():
    async for event in agent.stream_async("Explain agents in one paragraph."):
        if "data" in event:
            print(event["data"], end="", flush=True)
        if event.get("event_type") == "tool_use":
            print(f"\n[tool: {event['tool_name']}]\n")

asyncio.run(main())
```

7.2 Event types you will encounter

data: incremental text tokens to display.
tool_use: model decided to call a tool.
tool_result: tool returned a value.
end_turn: agent finished.
error: something failed.

Use these to drive UIs: show a typing indicator on tool_use, show citations from tool_result, end the spinner on end_turn.

7.3 Bidirectional streaming with BidiAgent

For voice applications — a Whisper-style live transcript that the agent can interrupt — Strands has an experimental *BidiAgent* backed by Amazon Nova Sonic. It maintains a persistent connection so the user and agent can talk over each other naturally.

```
import asyncio
from strands.experimental.bidi import BidiAgent
from strands.experimental.bidi.models import BidiNovaSonicModel
from strands.experimental.bidi.io import BidiAudioIO, BidiTextIO
from strands.experimental.bidi.tools import stop_conversation
from strands_tools import calculator

async def main():
    agent = BidiAgent(
        model=BidiNovaSonicModel(),
        tools=[calculator, stop_conversation],
    )
```

```
    audio_io = BidiAudioIO()
    text_io  = BidiTextIO()
    await agent.run(
        inputs=[audio_io.input()],
        outputs=[audio_io.output(), text_io.output()],
    )

asyncio.run(main())
```

Install with *pip install strands-agents[bidi,bidi-io]*. Note this API is experimental and may shift.

7.4 Async tools

Tools can be async natively:

```
@tool
async def fetch_url(url: str) -> str:
    """Fetch a URL and return its body."""
    async with httpx.AsyncClient() as client:
        r = await client.get(url)
        return r.text
```

Use async tools for I/O-bound operations to keep your agent responsive.

Chapter 7 Exercises

1. Build a CLI chat that streams responses with proper line-flushing.

2. Add a spinner that shows during tool calls and stops when text resumes.

3. (Stretch) Try BidiAgent end-to-end if you have a microphone-equipped machine. Build a voice "what time is it in X" agent.

Chapter 8: Observability

OpenTelemetry and CloudWatch

You cannot fix what you cannot see. Agents are non-deterministic; debugging them without traces is brutal. Strands emits OpenTelemetry spans natively. Wire it up once and you get token usage, latencies, tool calls, and errors for free.

8.1 What gets traced

Out of the box, Strands traces: each model invocation (with prompt size and completion size in tokens), each tool call (with arguments and return values), the overall agent loop, and any sub-agent calls. Each trace has timing, status, and parent-child relationships.

8.2 Enabling OTel

```
pip install strands-agents[otel] opentelemetry-exporter-otlp

# environment
export OTEL_EXPORTER_OTLP_ENDPOINT="https://your-collector:4317"
export OTEL_SERVICE_NAME="my-agent"

from strands import Agent
from strands.observability import enable_telemetry

enable_telemetry()  # picks up env vars

agent = Agent(...)
agent("Hello.")    # spans automatically exported
```

8.3 Sending traces to CloudWatch

For AWS-native setups, use the AWS Distro for OpenTelemetry (ADOT). Set the endpoint to your ADOT collector; CloudWatch Application Signals will display the trace tree, service map, and metrics. We dive into this in Chapter 15 for AgentCore-hosted agents.

8.4 Custom spans

```
from opentelemetry import trace
tracer = trace.get_tracer(__name__)

@tool
def expensive_lookup(term: str) -> dict:
    with tracer.start_as_current_span("expensive_lookup.preprocess"):
        clean = preprocess(term)
    with tracer.start_as_current_span("expensive_lookup.db_call") as span:
        span.set_attribute("term", clean)
        result = db.lookup(clean)
        span.set_attribute("rows_returned", len(result))
    return result
```

Tip: Tag spans with business attributes (customer_id, tenant, agent_version). Three months from now when an agent misbehaves for one customer, you will be glad you can filter.

8.5 Metrics that matter

Tokens per turn: detect prompt bloat early.
Tool-call count per session: spikes mean the agent is looping.
Tool error rate: by tool, by tenant.
Time-to-first-token: user-facing latency.
Resolution rate: did the agent actually finish the task? Hard to measure automatically; usually requires a downstream signal (was the ticket marked resolved?) or a periodic eval set.

Chapter 8 Exercises

1. Enable OTel locally with a console exporter (*OTEL_TRACES_EXPORTER=console*). Run the research briefing agent. Read the trace.

2. Add three custom spans inside one of your tools. Verify they show up nested correctly.

3. Pick five metrics from section 8.5 and write SQL (or CloudWatch Metrics Insights) queries that would compute them from raw trace data.

Chapter 9: Introducing Amazon Bedrock AgentCore

You have spent eight chapters building agents in Strands. Now we shift to where they live in production. Amazon Bedrock AgentCore is AWS's managed platform for deploying and operating AI agents at scale. It is framework-agnostic — Strands works great with it, but so does LangGraph, CrewAI, LlamaIndex, or anything you write yourself.

9.1 What AgentCore is, in plain words

AgentCore is nine services that solve the production problems every agent team eventually hits. You can use them independently or together.

Service	What it does	Mental model
Runtime	Hosts your agent code in isolated microVMs	ECS Fargate, but for agents
Memory	Managed short- and long-term agent memory	Redis + vector DB + lifecycle
Gateway	Turns APIs into MCP tools	API Gateway, but for agents
Identity	Auth for agents (IAM, OAuth, keys)	IAM + Secrets Manager
Browser	Managed cloud browser for agents to drive	Headless Chrome as a service
Code Interpreter	Sandboxed code execution	Lambda, but agent-friendly
Observability	Traces, metrics, logs for agents	CloudWatch X-Ray for agents
Evaluations	Quality monitoring with eval sets	Like CI for prompts
Policy	Cedar-policy guardrails on actions	IAM-style policies for agent actions

9.2 How AgentCore relates to Bedrock Agents

Bedrock Agents (the original product) is a managed, opinionated agent service. You define an agent through a console wizard with action groups and knowledge bases. Good for simple internal tools, limited customization.

Bedrock AgentCore is the production platform for teams that want full code-level control. You write the agent in any framework. AgentCore provides infrastructure. Both can coexist in the same AWS account.

> **Concept:** If Bedrock Agents is "managed WordPress for AI agents," AgentCore is "managed Kubernetes for AI agents." Different audiences, both valid. Most production teams I see in 2026 are using AgentCore.

9.3 The AgentCore mental model

Imagine the lifecycle of a request to your agent:

1. A user sends a message via your app.

2. Your app calls AgentCore Runtime, identifying the user via Identity.

3. Runtime spins up (or routes to) an isolated microVM running your agent code.

4. The agent loads conversation history from Memory.

5. The agent reasons, deciding to call a tool. The tool lives behind Gateway.

6. Gateway authenticates the call (via Identity), enforces a Cedar Policy, invokes the backing API.

7. Tool result returns. Agent continues. Maybe writes new memory.

8. Final response streams back to user. Trace lands in Observability.

9. If this was a "show me a website" task, the agent might have used Browser. If "compute this," Code Interpreter.

10. After the session ends, the microVM is terminated and memory wiped.

You write the agent code (steps 4–7's "agent reasons" parts). AgentCore handles everything else.

9.4 Pricing model (May 2026)

AgentCore prices each service independently. For a moderate-traffic agent (10k conversations/month, 5 turns each), expect roughly $50–$200/month in AgentCore infrastructure, plus $200–$800/month in Bedrock model inference depending on which model you use. Total under $1,000/month for a real production agent serving thousands of users is typical.

> **Warning:** Always check current pricing on the AWS pricing page. Numbers above are illustrative, not contractual. AgentCore pricing has shifted twice since GA.

9.5 Reading roadmap for Part III

The next six chapters cover the AgentCore services in roughly the order you will use them. Runtime first (because your code has to live somewhere). Then Memory and Gateway (the two services you will reach for almost daily). Identity, Browser, Code Interpreter, and finally Observability/Evaluations/Policy together.

Chapter 9 Exercises

1. Enable AgentCore in your AWS account's us-west-2 region.

2. Install the AgentCore Python SDK: *pip install bedrock-agentcore*.

3. Read the AgentCore "What is AgentCore" page in the AWS docs. Note any of the nine services that surprise you in capability.

4. Sketch — on paper or in a diagram tool — how a customer support agent at a fictional company would map onto the nine services. We will refine this in Chapter 16.

Chapter 10: AgentCore Runtime

Serverless microVMs for agents

Runtime is where your agent code executes. It is the foundation that every other AgentCore service plugs into. Each user session runs in a dedicated Firecracker microVM — the same isolation primitive that powers Lambda — with isolated CPU, memory, and filesystem.

10.1 Why microVMs

Agents are stateful within a session. They accumulate context, generate files, install tools. Containers were not built for this kind of multi-tenant statefulness. MicroVMs give you strong isolation (a prompt injection in one session cannot leak into another) plus the ability to keep filesystem state across stops and resumes within the same session.

Key Runtime properties:

Session isolation. Each user session is its own microVM.

Memory sanitization. When a session ends, the microVM is destroyed and memory wiped.

Long-running. Sessions can last up to 8 hours, supporting multi-step research and async work.

Large payloads. Up to 100 MB per request, enabling multimodal inputs.

Resumable filesystem. Stop and resume a session; installed packages and files persist for the session lifetime.

10.2 The deployment surface

You package your agent as a container image and point AgentCore at it. The image must implement a small HTTP interface (the "agent protocol") — one endpoint for invoking the agent, optionally a streaming endpoint.

The Strands + AgentCore SDK pair makes this trivial:

```
# agent_app.py
from strands import Agent
from bedrock_agentcore.runtime import AgentCoreApp

agent = Agent(
    system_prompt="You are an internal company assistant.",
    tools=[...],
)

app = AgentCoreApp(agent=agent)

if __name__ == "__main__":
    app.run()    # serves the agent protocol on the expected port
```

Dockerfile:

```
FROM public.ecr.aws/bedrock-agentcore/python:3.12-slim
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
CMD ["python", "agent_app.py"]
```

10.3 Deploying

```
# Build and push to ECR
aws ecr create-repository --repository-name my-agent --region us-west-2
docker build -t my-agent .
docker tag my-agent:latest <account>.dkr.ecr.us-west-2.amazonaws.com/my-agent:latest
docker push <account>.dkr.ecr.us-west-2.amazonaws.com/my-agent:latest

# Create the agent in AgentCore
aws bedrock-agentcore-control create-agent-runtime \
    --agent-runtime-name my-agent \
    --container-uri <account>.dkr.ecr.us-west-2.amazonaws.com/my-agent:latest \
    --role-arn arn:aws:iam::<account>:role/AgentCoreRuntimeRole \
    --region us-west-2
```

Tip: For the smoothest experience, use the *agentcore* CLI: *pip install bedrock-agentcore-cli* then *agentcore deploy* from your project directory. It handles ECR, IAM, and create- agent-runtime in one command. We use this in Chapter 16.

10.4 Invoking from your app

```
import boto3

client = boto3.client("bedrock-agentcore", region_name="us-west-2")

response = client.invoke_agent_runtime(
    agentRuntimeArn="arn:aws:bedrock-agentcore:us-west-2:<acct>:agent-runtime/my-agent/v1",
    sessionId=f"user-{user_id}-session-{session_id}",
    payload={"input": user_message},
    enableTrace=True,
)
print(response["output"])
```

sessionId is critical. Reuse it across requests from the same user-session and Runtime routes you back to the same microVM (state preserved). Use a new *sessionId* for a fresh session.

10.5 Streaming

```
response = client.invoke_agent_runtime_stream(
    agentRuntimeArn=...,
    sessionId=...,
    payload={"input": user_message},
)
for event in response["eventStream"]:
    if "chunk" in event:
        print(event["chunk"]["data"].decode(), end="", flush=True)
```

10.6 Session lifecycle

Sessions auto-terminate after 8 hours or after configurable idle timeout. You can explicitly stop a session with *stop_session*, and resume an explicitly-stopped session with *resume_session* — the microVM is paused, not destroyed, preserving filesystem state. Useful for long-running research agents that need to wait on a human.

10.7 IAM role for Runtime

Your agent runs as a specific IAM role. Give it minimum permissions: Bedrock model invocation, the specific AWS APIs your tools call, CloudWatch logs. Avoid attaching broad managed policies. We cover this thoroughly in Chapter 17.

Chapter 10 Exercises

1. Deploy the three-line "hello agent" from Chapter 3 to AgentCore Runtime. Invoke it from your local machine.

2. Deploy a version that uses the calculator tool. Verify it works end-to-end.

3. Build a session manager: make two invocations with the same sessionId, verify the agent remembers context. Then a third with a new sessionId, verify it does not.

4. Trigger a stop_session + resume_session cycle. Confirm a file you wrote in the first session is still there after resume.

Chapter 11: AgentCore Memory

Short- and long-term memory, managed

Building memory yourself means a vector DB, embedding pipelines, extraction logic, and lifecycle management. AgentCore Memory is all of that as one service. You write to it and read from it; AWS handles the rest.

11.1 The two flavors of memory

Short-term memory: raw conversation events from the active session. Useful when you want to inspect or replay turns.

Long-term memory: extracted facts that persist across sessions. AgentCore automatically processes short-term events through extractors (semantic facts, user preferences, summaries) and stores them as searchable records.

11.2 Creating a memory store

```
import boto3

mem = boto3.client("bedrock-agentcore-control", region_name="us-west-2")

response = mem.create_memory(
    name="customer-support-agent-memory",
    description="Memory for our customer support agent",
    eventExpiryDuration=90,        # short-term events kept for 90 days
    memoryStrategies=[
        {"type": "SEMANTIC_FACT", "namespaces": ["facts/{userId}"]},
        {"type": "USER_PREFERENCE", "namespaces": ["prefs/{userId}"]},
        {"type": "SESSION_SUMMARY", "namespaces": ["summaries/{userId}/{sessionId}"]},
    ],
)
memory_id = response["memoryId"]
```

11.3 Writing events

```
data_plane = boto3.client("bedrock-agentcore", region_name="us-west-2")

data_plane.create_event(
    memoryId=memory_id,
    actorId="user-alex-123",
    sessionId="sess-2026-05-10-a",
    eventTimestamp=datetime.utcnow(),
    payload=[{
        "conversational": {
            "role": "user",
            "content": "I prefer responses in markdown, and I work in Pacific time."
        }
    }]
)
```

AgentCore's extractors detect that "I prefer markdown" and "I work in Pacific time" are user preferences and store them as long-term records with appropriate namespaces. The next time you query, those facts are retrievable.

11.4 Retrieving memories

```
results = data_plane.retrieve_memories(
    memoryId=memory_id,
    namespace="prefs/user-alex-123",
    query="response formatting preferences",
    maxResults=5,
)
for item in results["memoryRecordSummaries"]:
    print(item["content"]["text"])
# -> "User prefers markdown-formatted responses."
```

11.5 Integrating with Strands

Wire memory into your Strands agent as two tools:

```
from strands import tool

@tool
def remember_about_user(fact: str) -> str:
    """Save an important fact about the current user for future conversations."""
    data_plane.create_event(
        memoryId=MEMORY_ID,
        actorId=current_user_id(),
        sessionId=current_session_id(),
        eventTimestamp=datetime.utcnow(),
        payload=[{"conversational": {"role": "user", "content": fact}}]
    )
    return "Saved."

@tool
def recall_about_user(query: str) -> list[str]:
    """Search memory for facts about the current user relevant to the query."""
    r = data_plane.retrieve_memories(
        memoryId=MEMORY_ID,
        namespace=f"facts/{current_user_id()}",
        query=query,
        maxResults=5,
    )
    return [item["content"]["text"] for item in r["memoryRecordSummaries"]]
```

Or, even simpler, let AgentCore do the extraction automatically by writing every user turn as an event — you do not need to teach the model to call *remember*.

11.6 Lifecycle and privacy

Memories have configurable TTLs. Delete a user's memories on account deletion via *delete_memory_records*. For regulated data, you can scope a memory store to a VPC and use customer-managed KMS keys. AgentCore Memory supports cross-region replication for DR.

Warning: Memory is sensitive. A buggy agent that writes the wrong facts to long-term memory can poison the user's experience for months. Always include a UI surface that lets users see and delete what the agent remembers about them. It is good UX, and in some jurisdictions, it is legally required.

Chapter 11 Exercises

1. Create an AgentCore memory store. Write five user turns and inspect what long-term records get extracted.

2. Build a Strands agent that uses *remember_about_user* and *recall_about_user*. Have a two-session conversation that demonstrates cross-session recall.

3. Add a "what do you remember about me?" command that lists all extracted facts.

4. Implement a "forget X" command that calls *delete_memory_records* on matching records.

Chapter 12: AgentCore Gateway

Turn your APIs into agent-ready MCP tools

Every company has internal APIs that an agent would benefit from. Writing per-tool wrappers, handling auth, and managing the OAuth dances gets old fast. Gateway is a managed MCP server: you point it at an OpenAPI spec, a Smithy model, or a Lambda function, and Gateway exposes those operations as MCP tools to any agent.

12.1 What Gateway gives you

Protocol translation: REST/Lambda/OpenAPI in, MCP out.

Auth handling: inbound (who can call the gateway) and outbound (how the gateway authenticates to the backing API). IAM SigV4, OAuth 2.1, API keys.

Semantic search over tools: when you have hundreds of tools, the agent gets the top-K relevant ones per turn.

Cedar policies: enforce who can call what at the gateway layer, independent of agent logic.

Observability: every tool invocation logged and traced.

12.2 Creating a Gateway from an OpenAPI spec

```
acc = boto3.client("bedrock-agentcore-control", region_name="us-west-2")

gateway = acc.create_gateway(
    name="acme-internal-api-gateway",
    protocol="MCP",
    roleArn="arn:aws:iam::<acct>:role/AgentCoreGatewayRole",
)

target = acc.create_gateway_target(
    gatewayId=gateway["gatewayId"],
    name="acme-customer-api",
    targetConfiguration={
        "openApi": {
            "specUri": "s3://my-specs/customer-api.yaml",
            "credentialProvider": {
                "oauth2": {
                    "tokenUrl": "https://auth.acme.com/oauth/token",
                    "clientIdSecretArn": "arn:aws:secretsmanager:...:client_id",
                    "clientSecretSecretArn": "arn:aws:secretsmanager:...:client_secret",
                }
            }
        }
    }
)
```

AgentCore reads the OpenAPI spec, generates one MCP tool per operation, and handles the OAuth dance for outbound calls. Your agent now has tools like *customer_api.get_customer*, *customer_api.list_orders*, etc.

12.3 Lambda targets

When you have business logic that does not map to a clean REST API, wrap it in Lambda and point Gateway at the Lambda. Gateway becomes the agent-facing facade.

```
target = acc.create_gateway_target(
    gatewayId=gateway["gatewayId"],
    name="custom-logic",
    targetConfiguration={
        "lambda": {
            "lambdaArn": "arn:aws:lambda:us-west-2:<acct>:function:my-tools",
            "toolSchema": [...]    # describe what tools the Lambda exposes
        }
    }
)
```

12.4 Connecting a Strands agent to Gateway

```
from strands import Agent
from strands.tools.mcp import MCPClient

gw_client = MCPClient(
    url="https://<gateway-id>.gateway.bedrock-agentcore.us-west-2.amazonaws.com/mcp",
    auth="sigv4",
)

with gw_client:
    tools = gw_client.list_tools_sync()
    agent = Agent(
        tools=tools,
        system_prompt="You are an account manager assistant.",
    )
    print(agent("Get me the latest order for customer C-001."))
```

12.5 Semantic tool search

When you have 200 tools, sending all of them to the model per turn is wasteful and confusing. Gateway will pre-filter to the most relevant tools given the user's message.

```
# At MCPClient init, request semantic filtering
gw_client = MCPClient(
    url="...",
    semantic_filtering={
        "max_tools_per_request": 12,
        "embedding_model": "amazon.titan-embed-text-v2",
    }
)
```

12.6 Cedar policies at the Gateway

Cedar is AWS's policy language. At Gateway, you can enforce rules like "the support-agent identity can call get_customer but not delete_customer," independent of what the model or your agent code wants.

```
# Cedar policy attached to the gateway
permit (
    principal in Identity::"support-agent",
```

```
    action == Action::"customer_api.get_customer",
    resource
);
forbid (
    principal,
    action == Action::"customer_api.delete_customer",
    resource
);
```

Concept: The agent prompt-engineering layer and the Gateway policy layer are independent defenses. A prompt-injection attack might convince the agent to *try* to call delete_customer. The Cedar policy at Gateway blocks the call. Defense in depth.

Chapter 12 Exercises

1. Wrap a public OpenAPI spec (e.g. petstore.swagger.io) in a Gateway. Connect a Strands agent. Have it list pets.

2. Write a Lambda with two tools (*get_weather*, *get_forecast*). Expose via Gateway.

3. Write a Cedar policy that forbids a specific tool. Confirm the agent gets blocked when it tries.

4. Enable semantic filtering. Add 30 fake tools, verify the agent receives only the relevant subset.

Chapter 13: AgentCore Identity

Auth for non-human workloads

When humans use software, identity is well-understood: SSO, MFA, sessions. When agents act on a human's behalf, identity is messier. Who is the agent? What does it have access to? How does it authenticate to a third-party SaaS — with its own credentials or the user's? AgentCore Identity exists to answer these questions cleanly.

13.1 The two directions of auth

Inbound auth: who is allowed to invoke the agent? Typically your end users, authenticated through Okta, Entra ID, or Cognito. AgentCore Identity integrates with all major IdPs.

Outbound auth: when the agent calls Slack, GitHub, or your internal API, what credentials does it use? Options: the user's own (delegated OAuth), the agent's service account, or short-lived API keys retrieved from Secrets Manager.

13.2 Workload identity

Every agent in AgentCore has a workload identity. Think of it as the agent's IAM role. It owns OAuth tokens, API keys, and AWS credentials. Tools the agent calls use this identity transparently.

```
identity = boto3.client("bedrock-agentcore-control", region_name="us-west-2")

wi = identity.create_workload_identity(
    name="support-agent-prod",
    description="Production identity for the customer support agent",
    iamRoleArn="arn:aws:iam::<acct>:role/SupportAgentRole",
)
```

13.3 OAuth flows for outbound

For "agent acts on behalf of user," you need delegated OAuth. AgentCore Identity implements the full flow.

```
# 1. Register an outbound OAuth provider
identity.create_oauth2_credential_provider(
    name="slack-oauth",
    providerConfig={
        "clientId": "...",
        "clientSecret": "...",
        "authorizationEndpoint": "https://slack.com/oauth/v2/authorize",
        "tokenEndpoint": "https://slack.com/api/oauth.v2.access",
        "scopes": ["chat:write", "channels:read"],
    },
)

# 2. When the user first opts in, AgentCore redirects them to Slack to grant access.
# 3. AgentCore stores the user-scoped tokens linked to that user's identity.

# 4. In your tool, retrieve the user's token transparently:
@tool
```

```
def post_to_slack(channel: str, message: str) -> str:
    """Post a message to a Slack channel on behalf of the current user."""
    token = identity.get_outbound_token(
        userId=current_user_id(),
        provider="slack-oauth",
    )
    # use token to call Slack API
    ...
```

13.4 API key vaulting

For service-to-service tools, store keys in Identity rather than env vars.

```
identity.create_api_key_credential_provider(
    name="stripe-prod",
    apiKeySecretArn="arn:aws:secretsmanager:...:stripe-prod-key",
)
```

Inside tools, fetch the key via Identity. Keys are rotated by Secrets Manager; the agent always sees the current one.

13.5 Audit logging

Every credential issuance and tool invocation under an identity is logged. When an agent does something unexpected, you can answer "which identity, which tool, which user, which time" within seconds.

Warning: Never bake API keys into agent code or env vars in container images. The agent will log it somewhere eventually (in a trace, in an error message). Always retrieve from Identity or Secrets Manager at call time.

Chapter 13 Exercises

1. Create a workload identity for one of your earlier agents.

2. Register an OAuth provider for a service you use (GitHub is good for practice). Walk through the user-consent flow.

3. Modify a tool to fetch its API key from Identity rather than an env var.

4. Review the Identity audit logs for an agent invocation. Identify the trace.

Chapter 14: AgentCore Built-in Tools

Browser and Code Interpreter

Two of AgentCore's nine services are built-in tools rather than infrastructure primitives: Browser and Code Interpreter. They are common enough that AWS provides them as managed capabilities.

14.1 Browser: when an agent needs to drive a website

A managed cloud browser the agent can control. Use it for: scraping sites without an API, automating SaaS that lacks an MCP server, filling out web forms, taking screenshots of customer-facing pages.

```
from bedrock_agentcore.tools import BrowserClient

browser = BrowserClient(region="us-west-2")
session = browser.create_session()

browser.navigate(session, "https://news.ycombinator.com")
content = browser.extract_text(session)

# Or with the high-level Strands integration:
from strands_tools.agentcore import browser_tool

agent = Agent(
    tools=[browser_tool],
    system_prompt="You are a web research assistant.",
)
print(agent("Summarize today's top story on Hacker News."))
```

Browser sessions run in Firecracker microVMs. Each session has its own profile, cookies, and browsing history — isolated from other users. Browser supports custom root CAs (for accessing internal sites), Chrome enterprise policies, and OS-level interactions for tasks that require keyboard input outside the browser viewport.

14.2 Code Interpreter: when an agent needs to compute

A sandboxed Python (and increasingly other-language) execution environment. Use it for: data analysis on user-uploaded CSVs, generating charts, running unit tests on generated code, calculations beyond what the model can do reliably.

```
from bedrock_agentcore.tools import CodeInterpreterClient

ci = CodeInterpreterClient(region="us-west-2")
session = ci.create_session()

result = ci.execute(
    session,
    code="""
import pandas as pd
df = pd.read_csv('/mnt/user-uploads/sales.csv')
print(df.groupby('region')['revenue'].sum())
"""
)
```

```
print(result["stdout"])
```

Files uploaded by the user appear at */mnt/user-uploads/*. Files the code produces can be returned to the user (an agent that generates a chart can save it as PNG and return the path).

14.3 Choosing between Browser and Code Interpreter

Browser: the action lives on a website. The agent clicks, types, scrolls.

Code Interpreter: the action is a computation. The agent runs code.

Many agents use both. A data analysis agent might use Code Interpreter to crunch numbers and Browser to fetch an external dashboard's screenshot for comparison.

Chapter 14 Exercises

1. Build an agent that uses Browser to monitor a public stock ticker page.

2. Build a data-analysis agent that takes a CSV upload and returns summary statistics + a chart.

3. Combine: build an agent that scrapes the latest GDP data from a public source (Browser) and computes a year-over-year change (Code Interpreter).

Chapter 15: Observability, Evaluations, Policy

Three services that share a theme: keeping agents safe and high-quality in production.

15.1 AgentCore Observability

Every AgentCore-hosted agent emits CloudWatch metrics by default: request count, latency, errors. To get trace-level detail, enable CloudWatch Transaction Search (one-time setup) and instrument your agent code with the ADOT SDK. Strands already emits OTel; you just route it to AgentCore's collector.

```
# In your agent_app.py
import os
os.environ["OTEL_EXPORTER_OTLP_ENDPOINT"] = "http://localhost:4318"
os.environ["OTEL_SERVICE_NAME"] = "support-agent"

from strands.observability import enable_telemetry
enable_telemetry()
```

Once enabled, you get end-to-end traces in the AgentCore console showing: model calls, tool invocations, memory reads/writes, Gateway calls, nested sub-agent calls, and timing for each. Trace IDs propagate across services so a multi-agent flow shows up as a single tree.

15.2 AgentCore Evaluations

Quality monitoring with eval sets. You define test inputs and graders; AgentCore runs your agent against them on a schedule.

```
eval_set = {
    "name": "support-agent-regression-v3",
    "cases": [
        {
            "input": "I forgot my password",
            "graders": [
                {"type": "contains", "value": "reset"},
                {"type": "llm_judge",
                 "criteria": "Did the agent offer a password reset link?"}
            ]
        },
        # ... 100 more cases
    ]
}

# Run on demand or schedule
acc.start_evaluation_run(
    agentRuntimeArn=...,
    evalSetArn=...,
    schedule="rate(1 day)",
)
```

Use this as CI for prompts: every prompt change must pass the eval set before deployment. Drift detection: if scores drop over time on a stable prompt (because the underlying model updated), you find out before users do.

15.3 AgentCore Policy

Cedar-policy guardrails for agent behavior. Goes beyond Gateway-level Cedar by covering the agent's whole action surface: memory reads, tool calls, model invocations, file writes.

```
# Example: forbid the agent from accessing PII in customer records during
# a non-business-hours session
forbid (
    principal in Agent::"support-agent",
    action == Action::"customer_api.get_customer_pii",
    resource
) when {
    context.session.hour_of_day < 6 ||
    context.session.hour_of_day >= 22
};
```

Policies are evaluated at decision points by AgentCore. If a policy denies, the action fails before reaching the backing system. The agent sees the denial as a tool error and can adapt its plan.

15.4 The full observability story

In production you typically wire up: structured logs (CloudWatch Logs), traces (X-Ray / Application Signals via OTel), metrics (CloudWatch), and eval scores (AgentCore Evaluations). Build dashboards that combine them: a graph of "resolution rate (from evals) vs. p95 latency (from traces) over time."

Chapter 15 Exercises

1. Enable CloudWatch Transaction Search. Trace one invocation of a deployed agent. Identify the model call, tool call, and any memory access.

2. Create an eval set with 10 cases. Run it. Inspect which cases pass and fail.

3. Write one Cedar policy that forbids a specific action under a specific condition. Verify it triggers.

4. Build a CloudWatch dashboard with 4 panels: invocations, p95 latency, error rate, eval pass rate.

Chapter 16: End-to-End

Deploying a Strands agent on AgentCore

Now we put everything together. We will build a customer support agent for a fictional SaaS company, deploy it on AgentCore with Memory + Gateway + Identity + Observability, and walk through every step. The full code is in the chapter exercise repo, but the key fragments appear here.

16.1 The system

ACME SaaS has customers who file tickets. The agent should: identify the customer (from session metadata), look up their account in the billing system (via Gateway), look up their open tickets (via Gateway), help resolve common issues (refund a charge, update an email address, escalate to human), and remember customer preferences across conversations (via Memory).

16.2 Project structure

```
acme-support-agent/
  agent_app.py             # the Strands agent entry point
  tools/
    __init__.py
    memory_tools.py        # remember/recall via AgentCore Memory
    gateway_tools.py       # MCP client connection to AgentCore Gateway
  prompts/
    system_prompt.md       # the system prompt
  Dockerfile
  requirements.txt
  agentcore.yaml           # AgentCore CLI config
```

16.3 The system prompt (excerpt)

```
You are ACME SaaS's customer support assistant. You help customers with billing,
account, and product questions.

Capabilities:
- Look up account details and tickets via the gateway tools.
- Issue refunds up to $200 without human approval; escalate higher.
- Update account email addresses after verifying the customer.
- Remember customer preferences (notification frequency, preferred contact channel)
  using remember_about_user. Recall them at the start of each conversation using
  recall_about_user.

Rules:
- Always confirm a customer's identity by their email + last 4 of card before
  performing sensitive actions.
- Never share account details with anyone who fails identity verification.
- When you cannot resolve an issue, escalate by calling create_human_ticket
  with a clear summary.
- Always cite the source of any factual claim using the data the gateway tools
  return.
```

16.4 The agent code

```
# agent_app.py
import os
from strands import Agent
from strands.tools.mcp import MCPClient
from bedrock_agentcore.runtime import AgentCoreApp

from tools.memory_tools import remember_about_user, recall_about_user

# Connect to the AgentCore Gateway exposing our billing/tickets APIs
gw = MCPClient(
    url=os.environ["AGENTCORE_GATEWAY_URL"],
    auth="sigv4",
)
gw.connect()
gateway_tools = gw.list_tools_sync()

with open("prompts/system_prompt.md") as f:
    SYSTEM = f.read()

agent = Agent(
    system_prompt=SYSTEM,
    tools=[*gateway_tools, remember_about_user, recall_about_user],
    max_iterations=12,
)

app = AgentCoreApp(agent=agent)

if __name__ == "__main__":
    app.run()
```

16.5 Deploying with the AgentCore CLI

```
# agentcore.yaml
agent:
  name: acme-support-agent
  framework: strands
  runtime:
    timeout_seconds: 600
    memory_mb: 1024
  identity:
    workload_identity: support-agent-prod
  memory:
    memory_id: !env ACME_MEMORY_ID
  observability:
    tracing: enabled

# Deploy
agentcore deploy

# Output:
# Building Docker image... OK
# Pushing to ECR... OK
# Creating agent runtime... OK
# Wiring memory... OK
# Wiring identity... OK
# Wiring observability... OK
# Agent runtime ARN: arn:aws:bedrock-agentcore:us-west-2:.../agent-runtime/acme-support-agent/v1
```

16.6 Calling from a frontend

```
// Node.js example
import { BedrockAgentCoreClient, InvokeAgentRuntimeCommand }
  from "@aws-sdk/client-bedrock-agentcore";

const client = new BedrockAgentCoreClient({ region: "us-west-2" });

async function chat(userId, sessionId, message) {
  const cmd = new InvokeAgentRuntimeCommand({
    agentRuntimeArn: process.env.AGENT_RUNTIME_ARN,
    sessionId: `${userId}::${sessionId}`,
    payload: JSON.stringify({ input: message, userId }),
    enableTrace: true,
  });
  const resp = await client.send(cmd);
  return resp.output;
}
```

16.7 What we just did

We deployed an agent that runs in isolated microVMs, has persistent memory across sessions, can call internal APIs through a managed gateway, identifies users through Cognito, and emits OpenTelemetry traces to CloudWatch. The agent code itself is <50 lines of Python. The rest is AgentCore.

This is the production pattern. Every project after this is variations on this foundation.

Chapter 16 Exercises

1. Clone the example repo (or build from scratch) and deploy the agent.

2. Modify the system prompt to also handle product feature questions, sourcing answers from a RAG tool.

3. Add a Cedar policy that prevents refunds > $500.

4. Set up an eval set with 20 representative tickets. Run it weekly.

Chapter 17: Production Hardening

Security, cost, reliability

Shipping the demo is one job. Operating it is another. This chapter is what experienced agent engineers learn the hard way.

17.1 The security mental model

Agents are subject to prompt injection: a user (or a piece of content the agent reads) can convince the model to do something its operator did not intend. You cannot prompt- engineer your way out of this. You design infrastructure assuming the agent will sometimes do the wrong thing, and limit what "wrong" means.

Defense layers, in order of priority:

1. **Tool design.** Tools should do narrow things. *send_email_to_user* is safer than *execute_arbitrary_action*.

2. **Cedar policies at Gateway and AgentCore Policy.** Block dangerous actions regardless of agent reasoning.

3. **IAM least privilege.** The agent's workload identity should have only the permissions it actually needs.

4. **Identity scoping for outbound auth.** Per-user OAuth tokens, not shared service accounts where possible.

5. **Output filtering.** Block PII in responses; redact sensitive data the model might echo back.

6. **Human approval for irreversible actions.** Always.

17.2 Cost management

Three cost drivers, in order: model tokens (usually 70-90% of total), AgentCore infrastructure, and downstream tool costs (API calls, data transfer).

Reduce tokens:

• Pick the cheapest model that meets quality. Haiku for simple tasks, Sonnet for most, Opus only for hardest.

• Cap iterations. Set *max_iterations* conservatively.

• Use SummarizingConversationManager to compress old turns.

• Cache stable prompts (system prompts, tool definitions) using model providers' prompt-caching features.

Reduce infra cost:

• Use AgentCore Runtime's idle timeout to terminate idle sessions quickly.

• Right-size memory/CPU per agent. Most agents do not need more than 1 GB.

• For batch/async work, schedule rather than maintain live sessions.

Budget alarms: set CloudWatch billing alarms by tag for every agent in production. The first sign of a runaway is usually a token bill.

17.3 Reliability patterns

Retries: agents should retry transient tool failures (network, throttling) but not business errors (invalid input). Strands has built-in retry policies; configure them per tool.

Idempotency: tools that mutate should accept an idempotency key. A retry should not create two records.

Fallbacks: if the primary model is down, fall back to a secondary (often a smaller model). Strands supports model fallback chains.

Health checks: Runtime supports a */health* endpoint; implement it to do a real loop-back test, not just "200 OK."

Canary deploys: AgentCore Runtime supports traffic shifting between agent versions. Send 1% of traffic to a new version, watch eval scores and error rates, then promote.

17.4 Operational runbooks you need from day one

1. How do we revert a prompt change? (Answer: agent versions in Runtime are immutable; you point traffic at the previous version. Have the rollback CLI in your runbook.)

2. How do we kill all sessions for a specific user? (Useful for support cases or abuse.)

3. How do we purge memory for a user? (GDPR right-to-be-forgotten.)

4. What do we do when an eval set regresses on a new model version?

5. How do we trace a specific user's problem from a support ticket back to a specific agent invocation?

17.5 Testing

Unit tests for tools: pure functions, easy to test.

Mocked-model tests for agent logic: replace the model with a deterministic stub that returns scripted tool calls.

Replay tests: capture real traces, replay them against a new agent version, diff the outcomes.

Eval sets: as in Chapter 15, the production substitute for "did my prompt change break anything?"

Chaos tests: kill the Gateway mid-session, inject a 500 from a tool, verify the agent recovers or gracefully escalates.

Chapter 17 Exercises

1. Add CloudWatch budget alarms for your AgentCore-hosted agent.

2. Implement a mocked-model test for one of your earlier agents.

3. Write three operational runbooks from section 17.4. Save them in your repo.

4. Conduct a chaos test: deliberately break one tool and observe how your agent reacts. Adjust the system prompt accordingly.

Chapter 18: Six Capstone Projects

Build your portfolio

These are the projects to actually build. Each one demonstrates a different combination of skills and is something you can show in an interview or put on your resume. Pick at least two: one "individual" and one "ambitious." Solo dev pace: one or two weekends per project.

Project 1: Personal CLI Assistant

Difficulty: ★

A terminal-based agent that helps with your daily dev work. Tools: file_read/write, shell (with allowlist), git, the GitHub MCP server, calendar. Memory: AgentCore Memory remembers your project context across sessions. Deployment: local (or AgentCore for cloud-anywhere access). **Skills demonstrated:** tool design, MCP integration, memory.

Project 2: Customer Support Triage Agent

Difficulty: ★★

Like the Chapter 16 walkthrough, but for a domain you know (e.g. a forum or GitHub repo you frequent). Use AgentCore: Runtime + Memory + Gateway (to wrap an OpenAPI spec) + Observability + Evaluations. **Skills demonstrated:** end-to-end AgentCore deployment, observability, eval-driven dev.

Project 3: Multi-Agent Research System

Difficulty: ★★

A Strands graph: a planner agent decomposes the question, three parallel research agents pursue sub-questions (each with its own tools), a synthesizer writes the final brief with citations. Use Code Interpreter for any quantitative claims. **Skills demonstrated:** multi-agent patterns, async, multi-modal output.

Project 4: Browser-driving SaaS Automator

Difficulty: ★★★

An agent that automates a specific SaaS workflow you find tedious. Use AgentCore Browser. Handle login (via Identity), navigation, form-filling, screenshot capture, error recovery. **Skills demonstrated:** Browser tool, robust error handling, real-world flake.

Project 5: Code Reviewer with Sandboxed Execution

Difficulty: ★★★

An agent that reviews pull requests: pulls the diff via GitHub MCP, identifies high-risk changes, runs the test suite in Code Interpreter, generates review comments. Integrate with GitHub via webhook + Lambda + AgentCore Runtime. **Skills demonstrated:** Code Interpreter, async/long-running workflows, integration.

Project 6: Voice-Driven Operations Agent

Difficulty: ★★★★

BidiAgent + Nova Sonic for voice interaction. Drives a real workflow (deploy a service, query metrics, file a ticket). The agent must handle interruption, confirmation, and graceful refusal of dangerous commands. **Skills demonstrated:** bidirectional streaming, high-stakes tool design, Cedar policies for guardrails.

How to present these in interviews

For each project, prepare a 90-second pitch with: the problem, the architecture diagram (1 slide), the hardest thing you had to solve, what you would do differently. Have the live repo open. Be ready to do a 20-minute deep-dive on any one of the nine AgentCore services that touches your project.

Chapter 18 Exercises

1. Pick two projects. Sketch the architecture for each before writing code.

2. Build them. Push to GitHub with a thorough README.

3. Record a 5-minute Loom demo of one of them. Practice the pitch.

Chapter 19: Interview Prep

Concepts, questions, system design

Roles that ask about Strands and AgentCore in 2026: AWS Solutions Architect, AI/ML Engineer at AWS-heavy shops, "Agent Engineer" (a new title that did not exist two years ago), Forward Deployed Engineer at AI startups, and increasingly Senior Backend with "agentic AI" listed in the JD. Below is what they ask.

19.1 Conceptual questions you should be able to answer cold

1. Walk me through the agent loop. Where can it fail?

2. What is MCP? Why does it exist? What problem did it solve that existed before?

3. Strands vs. LangGraph vs. CrewAI — what are the meaningful trade-offs?

4. What does "model-driven" mean? Contrast with "workflow-driven" frameworks.

5. List the nine AgentCore services and describe each in one sentence.

6. Where would you NOT use AgentCore? (Hint: simple single-shot LLM calls, on-prem requirements with no AWS option, framework lock-in concerns.)

7. Why microVMs for Runtime sessions and not containers?

8. What is the difference between Bedrock Agents and Bedrock AgentCore?

9. How does Cedar Policy at the Gateway layer defend against prompt injection?

10. AgentCore Memory: what does it do that you would otherwise have to build yourself?

19.2 System design exercises

Practice these out loud. Time yourself: aim for 35 minutes per design.

Design 1: An internal IT helpdesk agent for a 10,000-person company.

Cover: which AgentCore services, what tools, identity strategy (employees auth via Entra ID), memory namespaces (per-employee), eval set design, cost projection, escalation to human, multi-language support.

Design 2: A research agent that produces hour-long deep-dive reports.

Cover: long-running sessions (8h cap), Browser + Code Interpreter combo, multi-agent orchestration (planner, researcher fleet, synthesizer), citations and source verification, cost (research agents are expensive), human approval before "publishing."

Design 3: A real-time voice agent for a call center.

Cover: BidiAgent, latency budget, interruption handling, fallback to human after N failures, PII redaction in logs, compliance (call recording).

Design 4: A code-review agent in a regulated industry (e.g. finance).

Cover: read-only access to repos, sandboxed test execution, Cedar policies for what the agent can and cannot comment on, audit log requirements, eval sets that test for false positives/negatives.

19.3 Behavioral / experience questions

"Tell me about a time the agent did something unexpected. How did you debug it?"

Have a story ready. Real or from your capstone projects. Structure: situation, specific symptom, hypothesis, what you actually checked (traces, logs, eval scores), root cause, fix, prevention.

"How do you write a system prompt?"

Walk them through your method: define role, goals, tools, constraints, output format, escalation rules. Show iteration: how you refined a prompt based on eval failures.

"How do you decide when to add a new tool vs. extend an existing one?"

Discuss: single responsibility, model confusion when tools overlap, the cost of prompt bloat from too many tools.

19.4 Coding rounds

Expect either: "Build a small agent that does X" with a laptop, or "Walk me through this Strands code and identify the bugs." Practice both. Common bug patterns to know:

• Missing max_iterations on a loopy agent

• Tool docstrings that are too vague

• Forgetting to attach memory or sharing memory across users

• Untyped tool returns causing parse errors

• Synchronous tools blocking an async agent

• No retry/idempotency on mutating tools

19.5 Questions to ask them

Always ask. It signals seriousness. Good ones for agent roles:

• "How do you measure agent quality in production today?"

• "What's your prompt-change rollout process?"

• "How do you handle prompt injection and data exfiltration risk?"

• "What's the largest agent workflow you've deployed end-to-end?"

• "Where does the team think agents are heading in the next 18 months?"

Appendix A: Cheat Sheets and Command Reference

A.1 Strands API at a glance

```
# Install
pip install strands-agents strands-agents-tools

# Minimal agent
from strands import Agent
agent = Agent()
agent("Hello")

# With tools
from strands import tool
@tool
def my_tool(x: int) -> int:
    """Doc string the model reads."""
    return x * 2

agent = Agent(tools=[my_tool], system_prompt="...")

# Different model
from strands.models import BedrockModel, AnthropicModel, OpenAIModel
agent = Agent(model=BedrockModel(model_id="...", region="us-west-2"))

# Streaming
async for ev in agent.stream_async("..."):
    if "data" in ev:
        print(ev["data"], end="")

# MCP
from strands.tools.mcp import MCPClient
from mcp import StdioServerParameters
client = MCPClient(StdioServerParameters(command="...", args=[...]))
with client:
    agent = Agent(tools=client.list_tools_sync())

# Session persistence
from strands.session import S3SessionManager
agent = Agent(session_id="...", session_manager=S3SessionManager(bucket="..."))

# Conversation manager
from strands.agent.conversation_manager import SummarizingConversationManager
agent = Agent(conversation_manager=SummarizingConversationManager(...))

# Multi-agent
from strands.multiagent import Swarm, Graph, Workflow
swarm = Swarm(agents=[a1, a2, a3])
```

A.2 AgentCore CLI commands

```
# Install
pip install bedrock-agentcore bedrock-agentcore-cli

# Deploy from a project directory with agentcore.yaml
```

```
agentcore deploy

# List agents
agentcore list

# Invoke locally
agentcore invoke <agent-name> --session-id test-1 --input "Hello"

# Stream logs
agentcore logs <agent-name> --follow

# Roll back
agentcore deploy --version <previous-version-id>

# Run evals
agentcore eval run --eval-set <name>

# Memory
agentcore memory list
agentcore memory inspect <memory-id> --user <user-id>

# Gateway
agentcore gateway create --spec ./openapi.yaml
```

A.3 The nine AgentCore services in one line each

```
Runtime          - Where the agent code runs. Isolated Firecracker microVMs.
Memory           - Managed short and long-term memory with extractors.
Gateway          - Turns APIs/Lambda/MCP servers into agent-ready MCP tools.
Identity         - Auth for non-human workloads. Inbound + outbound.
Browser          - Managed cloud browser the agent can drive.
Code Interpreter - Sandboxed code execution for the agent.
Observability    - OTel traces, CloudWatch metrics, structured logs.
Evaluations      - Eval sets that run on a schedule against your agents.
Policy           - Cedar-policy guardrails at decision points.
```

A.4 Common error messages and what they mean

Error	Cause	Fix
`AccessDeniedException: bedrock:InvokeModel`	Model access not enabled or wrong region	Enable in Bedrock console, us-west-2
`ToolValidationError: schema mismatch`	Tool args don't match type hints	Tighten docstring + signatures
`MaxIterationsExceeded`	Agent looping	Lower iterations or fix prompt
`ContextWindowExceeded`	Conversation too long	Use SummarizingConversationManager
`GatewayTimeout (5xx from Gateway)`	Backing API slow or down	Add retry; investigate target
`MemoryValidationError: invalid namespace`	Namespace template not filled	Pass actorId/sessionId when writing

Appendix B: Resources and Where to Go Next

B.1 Official documentation

Strands Agents

• Website: *strandsagents.com*

• GitHub: *github.com/strands-agents/sdk-python*

• Tools repo: *github.com/strands-agents/tools*

• AWS announcement post: search "AWS Open Source Blog Strands Agents"

• Deep-dive technical post: search "Strands Agents SDK technical deep dive AWS Machine Learning Blog"

• AWS Prescriptive Guidance for Strands

Amazon Bedrock AgentCore

• Product page: *aws.amazon.com/bedrock/agentcore*

• Docs: *docs.aws.amazon.com/bedrock-agentcore*

• FAQs page (covers pricing, comparison to Bedrock Agents): see product page

• AgentCore samples repo on GitHub: *awslabs/amazon-bedrock-agentcore-samples*

• AgentCore MCP server (for use inside IDEs): *awslabs/mcp* repo

B.2 Related standards and frameworks worth knowing

• **MCP (Model Context Protocol)**: *modelcontextprotocol.io*

• **A2A (Agent-to-Agent protocol)**: an emerging standard for inter-agent communication

• **OpenTelemetry GenAI semantic conventions**: the spec for how agent telemetry should be structured

• **Cedar policy language**: *cedarpolicy.com*

B.3 Communities

• Strands Agents Discord (linked from strandsagents.com)

• AWS re:Post for AgentCore questions

• r/LocalLLaMA and r/MachineLearning for broader agent discussion

• AWS Community Builders program if you publish content

B.4 Further reading

• "Building Effective Agents" by Anthropic — the seminal post that established how to think about agent design patterns

• The AWS What's New feed, filtered for Bedrock AgentCore — service is evolving fast; check monthly

• The OpenAI / Anthropic / Google blogs on model agentic capabilities — the model layer matters

• AWS re:Invent talks on agentic AI — the 2025 and 2026 sessions go deep on production patterns

B.5 What to build next

You have read about agents and you have built some. The next step is one of three paths, depending on what you want:

Path 1: Get hired. Polish two capstone projects, write three thoughtful blog posts on what you learned, apply to companies whose tech blogs mention agents. Refer back to Chapter 19 weekly.

Path 2: Ship something real. Pick a workflow you or someone you know does manually. Build an agent for it. Use it daily. Iterate based on what breaks. This is the fastest way to internalize what production agents need.

Path 3: Contribute. Strands is open-source. Find a "good first issue." Or write a new MCP server for a tool you use. Your name in a CHANGELOG is worth more than another certification.

Closing

The field is young and the tools are still settling. By the time you read this, some API in this book will have changed. The concepts — the agent loop, tool design, memory layers, the production-platform problems AgentCore solves — will not. Master those, and you will be able to learn the next framework, the next platform, and the next standard quickly. That is the goal of this book.

Now stop reading and go build.

End of book.