AI Agents

For

Investing

Simply Boring AI

AI Agents for Investing

If you want to know more about me → quaintitative.com

Where I think out loud → simplyboring.ai

Where I post too much → linkedin.com/in/garyang/

And if you really really want to get in touch → gary@quaintitative.com

First edition, 2026

Contents

Preface **11**

Who This Is For . . . 11
What You Need . . . 12
What's Included . . . 12
What You'll Build . . . 15
What This Book Doesn't Cover . . . 16

1 The Trust Problem **18**

1.1 The Finance AI Problem . . . 19
1.2 The Fix: Give the LLM Tools . . . 20
1.3 The Hamburger Principle . . . 21
1.4 What This Changes . . . 22
1.5 The Four Patterns . . . 22

2 The Complexity Ladder **24**

2.1 The Three Levels . . . 24
2.2 Level 1: Single Tool Call . . . 24
2.3 Level 2: Workflow . . . 26
2.4 Level 3: Agent . . . 27
2.5 The Decision Framework . . . 27
2.6 Common Finance Tasks by Level . . . 28
2.7 Why This Matters: Cost and Reliability . . . 28
2.8 The Complexity Trap . . . 29
2.9 Climbing the Ladder . . . 29

3 Agent Architecture **31**

3.1 The Four Components . . . 31
3.2 Component 1: The Model . . . 32
3.3 Component 2: Tools . . . 32
3.4 Component 3: Instructions . . . 33
3.5 Component 4: Memory . . . 34
3.6 How the Components Work Together . . . 36
3.7 The smolagents Framework . . . 37
3.8 Your First Agent . . . 38
3.9 Seeing Inside the Agent . . . 39

3.10 Why This Architecture Matters . . . 40
3.11 The Architecture in Hamburger Terms . . . 40
3.12 What's Next . . . 41

4 The Three Types of Finance Tools 42
4.1 The Classification . . . 42
4.2 Type 1: Market Data Tools (Read-Only) . . . 43
4.3 Type 2: Analytical Tools (Computation) . . . 45
4.4 Type 3: Action Tools (Comes with Side Effects) . . . 47
4.5 What This Book Builds . . . 48
4.6 The Guardrail Framework . . . 48
4.7 Tool Design Principles . . . 49
4.7.1 Principle 1: Clear "When to Use" in Docstrings . . . 49
4.7.2 Principle 2: One Tool, One Job . . . 49
4.7.3 Principle 3: Structured Returns . . . 50
4.8 The CodeAct Hybrid . . . 50
4.9 Common Mistakes . . . 51

5 Your First Agent - Tool Calling 52
5.1 The Setup . . . 52
5.2 The @tool Decorator . . . 53
5.3 Creating the Agent . . . 54
5.4 The Magic Moment . . . 54
5.5 Seeing Inside the Agent . . . 55
5.6 Adding More Tools . . . 56
5.7 Multi-Tool Queries . . . 57
5.8 The Key Principles . . . 58
5.8.1 Principle 1: Clear Docstrings . . . 58
5.8.2 Principle 2: Type Hints Matter . . . 59
5.8.3 Principle 3: One Tool, One Job . . . 59
5.9 Memory: The Internal Tool . . . 59
5.9.1 Using Memory for Conversations . . . 60
5.9.2 Inspecting Memory . . . 60
5.10 Data Sources: Where the Numbers Come From . . . 60
5.11 Exercise: Build Your Own Tool . . . 61
5.12 Solution . . . 62
5.13 What You Built . . . 63

6 The ReAct Pattern — Multi-Step Reasoning 64
6.1 What is ReAct? . . . 65
6.2 Setting Up the Tools . . . 65
6.3 The Multi-Step Query . . . 66
6.4 Anatomy of the ReAct Trace . . . 68
6.5 Why ReAct Matters . . . 69

6.5.1 1. Complex Questions Become Possible . . . 69
6.5.2 2. Transparency and Auditability . . . 69
6.5.3 3. Graceful Handling of Edge Cases . . . 69
6.6 A More Complex Example . . . 70
6.7 ReAct vs. Simple Tool Calling . . . 71
6.8 ReAct Principles . . . 71
6.8.1 Principle 1: Ask Complete Questions . . . 71
6.8.2 Principle 2: Let the Agent Think . . . 72
6.8.3 Principle 3: Watch the Trace . . . 72
6.9 Memory Across ReAct Steps . . . 72
6.10 Finance-Specific ReAct Patterns . . . 72
6.11 Exercise: Build Your Own Multi-Step Query . . . 73
6.12 Tracking Steps Programmatically . . . 73
6.13 The Hamburger Check . . . 74

7 The CodeAct Pattern - Dynamic Computation 75
7.1 What is CodeAct? . . . 76
7.2 Setting Up CodeAct . . . 76
7.3 The Sharpe Ratio Example . . . 77
7.4 Why This is Powerful . . . 79
7.5 More Examples . . . 79
7.5.1 Maximum Drawdown . . . 79
7.5.2 Correlation Analysis . . . 80
7.6 Combining Tools AND Code . . . 81
7.7 Security Considerations . . . 83
7.7.1 The Whitelist Approach . . . 83
7.7.2 Production Safeguards . . . 84
7.8 CodeAct Principles . . . 84
7.8.1 Principle 1: Be Specific About What You Want . . . 84
7.8.2 Principle 2: Ask for Explanations . . . 84
7.8.3 Principle 3: Combine Tools and Code . . . 84
7.9 When to Use CodeAct vs. Pre-Built Tools . . . 85
7.10 Exercise: Dynamic Calculation . . . 85
7.11 The Hamburger Check . . . 85

8 Orchestration - Putting It All Together 87
8.1 The Portfolio Optimization Agent . . . 87
8.2 The Tools . . . 88
8.2.1 Tool 1: Fetch Historical Prices . . . 88
8.2.2 Tool 2: Optimize for Maximum Sharpe Ratio . . . 89
8.2.3 Tool 3: Optimize for Minimum Volatility . . . 90
8.2.4 Tool 4: Optimize for Target Return . . . 90
8.2.5 Tool 5: Calculate Discrete Allocation . . . 91

8.2.6 Tool 6: Compare Strategies . . . 92
8.3 Creating the Agent . . . 92
8.4 Orchestration in Action . . . 92
8.5 Multi-Pattern Orchestration . . . 94
8.6 Memory and Follow-Up Questions . . . 97
8.7 The Complete Architecture . . . 98
8.8 The Hamburger in Full . . . 98
8.9 What You've Built . . . 99
8.10 From Demo to Production . . . 100

9 What's Next - The Production Gap 102
9.1 The Gap: Demo vs. Production . . . 102
9.2 Preview: The Guardrail Framework . . . 103
9.3 Preview: Key Guardrail Layers . . . 103
9.4 What Production Requires (Preview) . . . 103
9.5 Coming Next . . . 104
9.6 For Now: What You Can Do . . . 104
9.7 The Honest Assessment . . . 105

10 Quick Reference - Pattern Cheat Sheets 107
10.1 Pattern 1: Tool Calling . . . 107
10.2 Pattern 2: ReAct . . . 108
10.3 Pattern 3: CodeAct . . . 109
10.4 Pattern 4: Orchestration . . . 110
10.5 Quick Reference: Finance Tools . . . 111
10.5.1 Market Data (Read-Only) . . . 112
10.5.2 Calculation Tools . . . 112
10.5.3 CodeAct Calculations . . . 112
10.6 Common Code Patterns . . . 112
10.6.1 Initialize Agent . . . 112
10.6.2 Basic Tool Template . . . 113
10.6.3 Follow-Up Questions . . . 113
10.6.4 Verbose Mode . . . 113
10.7 The Hamburger Mental Model . . . 114
10.8 Pattern Progression . . . 114

11 What I Hope You Take Away 115
11.1 The Pattern . . . 115
11.2 Four Patterns, One Principle . . . 116
11.3 What This Book Is (And Isn't) . . . 116
11.4 The Honest Assessment . . . 116
11.5 Where You Go From Here . . . 117
11.6 Thank You . . . 117
11.7 Post-Credits . . . 118

12 The n8n Path - Visual Agent Workflows 119
12.1 Why n8n? . . . 119
12.2 What You'll Need . . . 120
12.3 Pattern 1: Tool Calling . . . 120
12.3.1 Workflow Structure . . . 120
12.3.2 Key Nodes . . . 120
12.3.3 Test Query . . . 121
12.4 Pattern 2: ReAct . . . 121
12.4.1 Workflow Structure . . . 122
12.4.2 The ReAct System Prompt . . . 122
12.4.3 Tools . . . 122
12.4.4 Test Query . . . 122
12.5 Pattern 3: Agent + Code Tools . . . 123
12.5.1 Workflow Structure . . . 123
12.5.2 The Calculate Metric Code Tool . . . 123
12.5.3 Code Walkthrough: Calculate Metric . . . 123
12.5.4 How the Agent Uses It . . . 127
12.5.5 Test Queries . . . 127
12.6 Pattern 4: Full Orchestration . . . 128
12.6.1 Workflow Structure . . . 128
12.6.2 HTTP Tools . . . 128
12.6.3 Calculate Portfolio Metrics Code Tool . . . 128
12.6.4 Code Walkthrough: Portfolio Metrics . . . 129
12.6.5 Window Buffer Memory . . . 133
12.6.6 Test Queries . . . 133
12.7 Importing the Workflows . . . 133
12.7.1 Step 1: Download the JSON Files . . . 133
12.7.2 Step 2: Import into n8n . . . 133
12.7.3 Step 3: Configure Credentials . . . 133
12.7.4 Step 4: Test . . . 134
12.7.5 If Code in Code Tool Is Not Shown . . . 134
12.8 Comparing Python and n8n . . . 145
12.8.1 The Pattern Mapping . . . 145
12.9 n8n-Specific Considerations . . . 145
12.9.1 Benefits of Visual Workflows . . . 147
12.9.2 JavaScript vs Python . . . 147
12.10 Troubleshooting . . . 147
12.10.1 "Tool not being called" . . . 147
12.10.2 "FMP API errors" . . . 147
12.10.3 "Code Tool returns error" . . . 148

13 Financial Concepts - A Beginner's Guide 149
13.1 Three Layers of Financial Knowledge . . . 149

13.2 Stock Fundamentals . . . 150
13.2.1 Key Data Points . . . 150
13.2.2 Code: Stock Dashboard . . . 150
13.2.3 Market Cap Classification . . . 151
13.2.4 Volume as a Signal . . . 152
13.3 Position and Portfolio Value . . . 152
13.3.1 The Formulas . . . 152
13.3.2 Code: Portfolio Tracker . . . 152
13.3.3 Cost Basis and Performance . . . 153
13.4 Portfolio Weights . . . 154
13.4.1 The Formula . . . 154
13.4.2 Code: Weight Calculation . . . 154
13.4.3 Why Weights Matter More Than Dollar Values . . . 155
13.4.4 Weight Drift . . . 155
13.5 Returns and Performance . . . 155
13.5.1 Four Types of Returns . . . 156
13.5.2 Code: Daily Returns . . . 156
13.5.3 Volatility: The Standard Deviation of Returns . . . 157
13.5.4 Code: Volatility Calculation . . . 157
13.6 Sharpe Ratio: Return Per Unit of Risk . . . 158
13.6.1 The Formula . . . 158
13.6.2 Why Subtract the Risk-Free Rate? . . . 158
13.6.3 Code: Sharpe Ratio . . . 158
13.6.4 Interpretation . . . 159
13.6.5 Portfolio Sharpe . . . 160
13.7 Sortino Ratio: Only Downside Counts . . . 160
13.7.1 The Formula . . . 160
13.7.2 Downside Deviation . . . 160
13.7.3 Code: Downside Deviation . . . 160
13.7.4 Sharpe vs Sortino: What the Difference Tells You . . . 161
13.7.5 When to Use Each . . . 161
13.8 Maximum Drawdown: Your Worst Day . . . 162
13.8.1 The Formula . . . 162
13.8.2 Code: Drawdown Calculation . . . 162
13.8.3 The Asymmetry of Losses . . . 163
13.8.4 Calmar Ratio: Return per Drawdown . . . 163
13.8.5 Drawdown Duration . . . 164
13.9 Value at Risk (VaR) . . . 164
13.9.1 VaR Statement . . . 164
13.9.2 Three Methods . . . 164
13.9.3 Comparing the Methods . . . 166
13.9.4 Multi-Day VaR . . . 166
13.9.5 CVaR (Expected Shortfall) . . . 166

13.9.6 VaR Limitations . . . 167
13.10 Correlation and Diversification . . . 167
13.10.1 Correlation . . . 167
13.10.2 Code: Correlation Calculation . . . 167
13.10.3 The Math of Diversification . . . 168
13.10.4 Code: Diversification Benefit . . . 169
13.10.5 Correlation During Crises . . . 170
13.11 Sector Analysis . . . 170
13.11.1 The 11 GICS Sectors . . . 170
13.11.2 Code: Sector Breakdown . . . 171
13.11.3 Concentration Risk vs. Benchmark . . . 171
13.11.4 HHI: The Concentration Index . . . 172
13.11.5 Sector Attribution . . . 172
13.12 Portfolio Rebalancing . . . 173
13.12.1 Why Drift Happens . . . 173
13.12.2 Three Rebalancing Strategies . . . 173
13.12.3 Code: Rebalancing Trades . . . 173
13.12.4 The 5% Rule . . . 174
13.12.5 Tax-Efficient Rebalancing . . . 174
13.13 The Complete Risk Dashboard . . . 175
13.14 The Mental Model: Three Questions . . . 176

14 How This Book Was Built - Working With AI 177
14.1 AI or me? . . . 177
14.2 How I Actually Built It . . . 179
14.2.1 The Setup . . . 179
14.2.2 The Planning Phase . . . 179
14.2.3 The Writing Phase . . . 179
14.2.4 Conversion . . . 180
14.2.5 The Tools . . . 180
14.3 What AI Could Not Do . . . 181
14.4 What AI Made Possible . . . 181
14.5 The Part I Find Funny . . . 182

Preface

This book comes with code.

That might sound obvious, but I want to be upfront about it: you will get more out of this book if you run the notebooks alongside reading. The concepts stick differently when you see an agent reason through a problem in real time versus reading about it on a page.

Everything is designed to run in Google Colab - no local setup required, no environment headaches.

Download the files. Dump it into your Google Drive. Open the notebook (look for the Google Colab app), paste in the API keys, run the cells.

Who This Is For

This book sits at an intersection: AI x finance. It's my first attempt, so let me know if this makes it too confusing.

But here's my thought process on who this is for.

If you work in finance - you've probably already tried asking an LLM a financial question and gotten a confident answer you couldn't verify. This book shows you how to build tools where the data is real, the calculations can be observed, and the LLM handles communication rather than computation.

If you're a developer or technologist - you know how to code, but you want to apply AI from the investment angle. This book includes notebooks that get you up to speed on key investment concepts and math.

If you're somewhere in between - maybe you can read Python but don't write it daily, or you understand finance but want to explore what AI agents can actually do - this book is deliberately designed for that middle ground. Every pattern is explained conceptually *and* with code. The n8n bonus chapter covers the same patterns with visual workflows, no Python required.

What You Need

Knowledge

If you can read, you have enough to start.

Helpful but not required:

- Basic Python literacy (can read and modify code)
- Experience with APIs and data fetching
- Familiarity with pandas/numpy
- Previous exposure to LLMs or chatbots
- Understanding of fundamental finance concepts (prices, returns, ratios)

Not required at all:

- Machine learning expertise
- Deep learning or neural network knowledge
- Prior experience with AI agents

Tools

For the Python path:

- A Google account (for Colab) or Python installed locally
- An LLM API key (OpenAI, Anthropic, or HuggingFace - any works)
- Optional: FMP API key for real market data

For the n8n path:

- n8n account (cloud or self-hosted)
- An LLM API key
- Optional: FMP API key for real market data

Cost estimate:

- API costs for working through the book: $5–20 depending on usage. I include simulated versions if you don't want to get real financial data from APIs.

What's Included

Agent Notebooks

These are the core of the book. Each notebook maps to a pattern chapter and lets you build, run, and inspect a working agent.

I included two versions of each notebook - `simulated` folder has notebooks that use simple simulated data, no API to data needed; `api` folder uses APIs to fetch real data.

Notebook	Pattern	Chapter
`01_tool_calling.ipynb`	Tool Calling	Chapter 5
`02_react_pattern.ipynb`	ReAct	Chapter 6
`03_codeact_pattern.ipynb`	CodeAct	Chapter 7
`04_orchestration.ipynb`	Orchestration	Chapter 8
`05_portfolio_optimization.ipynb`	All four combined	Chapter 8

Start with these. The book explains the concepts; these notebooks let you experience them.

Finance Notebooks

You can build an agent that calculates a Sharpe ratio without understanding what a Sharpe ratio means. You just can't build a useful one.

These eleven notebooks cover the financial concepts your agents will use - from stock prices to portfolio rebalancing. Each one uses live market data via yfinance, so you're working with real numbers.

Notebook	Topic
`01_stock_basics.ipynb`	Prices, OHLC, 52-week ranges, market cap, volume
`02_position_and_portfolio_value.ipynb`	Calculating what your holdings are worth
`03_portfolio_weights.ipynb`	Allocation and weight calculations
`04_returns_and_performance.ipynb`	Daily, cumulative, and annualized returns
`05_sharpe_ratio.ipynb`	Risk-adjusted return measurement
`06_maximum_drawdown.ipynb`	Worst peak-to-trough decline
`07_correlation_and_diversification.ipynb`	How assets move together
`08_sortino_ratio.ipynb`	Downside risk measurement
`09_sector_analysis.ipynb`	Sector exposure and concentration
`10_value_at_risk.ipynb`	Estimating potential losses
`11_portfolio_rebalancing.ipynb`	Maintaining target allocations

These are companion material to the Finance Bonus Chapter. You don't need all of them

to follow the main book, but they'll make the agent examples more meaningful if you understand the metrics underneath.

Setup Notebooks

Getting started with LLMs and data APIs can be fiddly. These notebooks walk you through the setup step by step - install the package, enter your key, test the connection, done.

LLM Providers:

Notebook	Provider
`01_setup_openai.ipynb`	OpenAI
`02_setup_anthropic.ipynb`	Anthropic
`03_setup_gemini.ipynb`	Google Gemini
`04_setup_huggingface.ipynb`	Hugging Face
`05_setup_openrouter.ipynb`	OpenRouter

Data Providers:

Notebook	Provider
`06_setup_alpha_vantage.ipynb`	Alpha Vantage
`07_setup_yfinance.ipynb`	yfinance
`08_setup_massive.ipynb`	Massive
`09_setup_fmp.ipynb`	Financial Modeling Prep
`10_setup_twelve_data.ipynb`	Twelve Data
`11_setup_eodhd.ipynb`	EODHD

You only need one LLM provider and one data provider to get started. The book uses OpenAI and yfinance or FMP by default, but you can swap in whatever you prefer with the help of these notebooks.

n8n Workflows

Not everyone wants to write Python. The n8n bonus chapter shows you the same four patterns implemented as visual workflows - drag-and-drop, no code required.

These are importable JSON files. Set up an n8n account (free trial or local version works), import the file, add your API credentials, and you have a working agent workflow. The n8n bonus chapter walks through the details.

Workflow	Pattern
`01 - Tool Calling.json`	Tool Calling
`02 - REACT.json`	ReAct
`03 - Agent + Code Tools.json`	CodeAct
`04 - Portfolio Optimization_Full Orchestration Agent.json`	Full Orchestration

What You'll Build

By the end of this book, you'll have built a working portfolio assistant that:

Fetches real data

- Current prices from market APIs
- Fundamental data (P/E, market cap, dividends)
- Historical returns for risk calculations

Calculates real metrics

- Portfolio volatility
- Sharpe and Sortino ratios
- Value at Risk (VaR)
- Correlation matrices
- Sector concentration

Reasons transparently

- Shows its thinking at each step
- Explains why it made each decision
- Produces audit trails you can review

Handles conversations

- Remembers your portfolio across questions
- Answers follow-ups ("what if I sold some NVDA?")
- Runs what-if scenarios

You can use it as-is, extend it for your needs, or use it as a reference for building your own tools.

What This Book Doesn't Cover

Let me be really clear about scope (and add a health warning):

In this book:

- The four agent patterns for finance
- Working code you can run and modify
- Foundational understanding of when and how to use agents

Not in this book:

- Production deployment (scaling, monitoring, CI/CD)
- Multi-agent systems (teams of specialized agents)
- Fine-tuning or training custom models
- Regulatory compliance frameworks (we show audit trails, not legal advice)
- Real-time trading systems (we focus on analysis, not execution)

This book gets you from "LLMs hallucinate" to "I have a working portfolio assistant." Going from portfolio assistant to production system is the next step - and a bigger one.

About Me

I bridge AI, quantitative finance, and risk management (or at least I think I do). Most recently, I led AI risk supervision at the Monetary Authority of Singapore (MAS), where I developed Singapore's first AI risk management guidelines for the financial sector. Before that, I was division head for investment risk management, overseeing risk management of Singapore's foreign reserves. I also spent years deep in the weeds of Basel 2, 2.5, and 3 capital rules, model risk audits, and banking policy. Basically, I did work so technical that it clears a room at parties when people ask what I do.

I have a PhD in Computer Science, where I researched deep learning for networks, time series, and multimodal data, publishing 11 papers at venues like ACL, ACM Transactions on the Web, and IEEE Big Data, including an honourable mention at ACM IUI. I also have Masters degrees in Financial Engineering and Knowledge Engineering from NUS, and an Electrical Engineering degree from the University of Toronto. Yes, that's a lot of degrees. No, I don't know when to stop.

I've taught Python, machine learning, deep learning, and AI at NUS, SMU, SUSS, Nanyang Polytechnic, and MAS Academy, to audiences ranging from central bankers to healthcare workers to policemen. I'm currently developing AI risk management materials for the Cambridge Centre for Alternative Finance. I actually enjoy explaining complicated things to rooms full of people who didn't ask for it. Masochist, I know.

Before all of this, I spent four years at the Ministry of Information, Communications and the Arts, where I helped develop Singapore's creative industries strategy and created the Creative Youth eXchange, a regional design competition that somehow became a reality TV programme. I also helped (coloring and contributing a chapter) with a series of Mr Kiasu books from 2017-2019, which may be the most widely read publication I've had a hand in.

Beyond finance and technology, I'm a watercolor artist and illustrator whose work has been commissioned by Wallpaper, Tatler, Jetstar, and the Esplanade, hard proof that not everything I do involves code.

This book was written carefully, but with AI's help. There's a whole chapter 14 about how. I practise what I preach.

Follow me at linkedin.com/in/garyang/
Subscribe to my newsletter at simplyboring.ai/
Email me at gary@quaintitative.com

Let's get started.

1

The Trust Problem

You've heard all the hype about how you can just use ChatGPT (or Claude, or Gemini) to manage your finances. As long as you know the right prompts.

Try it yourself.

Open your Large Language Model (LLM) of choice, and paste in: *"I have $10,000. Help me invest in stocks. I'm risk-averse and want at least 2% annual return."*

Watch what happens.

You'll definitely get a response. Maybe it suggests 40% in dividend stocks, 30% in bonds, 30% in a broad market ETF. It mentions "low volatility" and "stable returns." It sounds reasonable. And confident.

But here's the question: **can you trust it?** Don't you still feel a slight niggling doubt in your mind? If not, I'll introduce the doubt here.

Did the LLM actually check whether those stocks have low volatility? Did it calculate whether that allocation can realistically deliver 2% given current market conditions? Did it verify that the correlation between those holdings provides real diversification?

There's a chance that it did. And there's also a significant chance it did not.

When it didn't, it may have done what LLMs do when they don't have the data: it hallucinated or bullshitted. It generated recommendations that *sound* like they fit your criteria - low volatility, stable returns, diversified - without actually checking if any of that is true.

And it delivered all of this with complete confidence. Just like a human bullshitting.

And even when it did do it properly, how can you be sure if it did it right?

Now imagine this powers your portfolio recommendations. Your risk assessments. Your personal portfolio.

Good luck trying to make sense of things.

1.1 The Finance AI Problem

This isn't a bug in ChatGPT or Gemini or Claude. It's a fundamental limitation of how LLMs work.

LLMs are good at two things: understanding language and generating fluent responses. Natively, they're *not* databases. They're *not* calculators. They can't access real-time data, and they can't perform complex computations. Unless you give them tools, which is what we will cover in this book.

When you ask a simple question - "What's Apple's current stock price?" - modern LLMs with web search (which is a tool) can handle it. They look it up. Problem solved.

But real finance questions aren't simple lookups. They're multi-step problems:

"Build me a low-volatility portfolio" requires:

- Fetching historical price data for candidate stocks
- Calculating each stock's volatility
- Checking correlations between them
- Optimizing the allocation to minimize overall risk

"Can this portfolio deliver 2% returns?" requires:

- Historical return data
- Expected return calculations
- Risk-adjusted projections

"Is my portfolio actually diversified?" requires:

- Correlation analysis between holdings
- Sector exposure breakdown
- Concentration risk assessment

When it can't, it hallucinates. It generates recommendations that sound right. It produces analysis that seems sophisticated. And it will never tells you it's guessing.

So here's what you should keep at the back of your mind.

You can't tell which responses are grounded and which are fiction. A portfolio that *sounds* low-volatility might be concentrated in correlated assets. A return projection that *seems* reasonable might ignore current market conditions entirely. Advice that *appears* personalized might just be generic patterns from training data, dressed up in confident language. Reasoning traces from a the reasoning versions of LLMs these days can still be hallucinations.

Increasingly, ChatGPT, Claude, Gemini, and many other interfaces are adding tools to do these other steps. But not always in a transparent way.

In this book, we will take a peek under the hood to see how this happens. I find that it helps when one has a mental model of what's actually happening.

1.2 The Fix: Give the LLM Tools

There is a fix that helps.

It's not perfect as you are still using an LLM with all its warts. I'll be honest about the caveats as we go on, but it's still a step change from *hoping* the LLM gets it right.

The problem isn't that LLMs are bad at reasoning. They're actually getting quite good with reasoning. The problem is that they can't *do* things natively - fetch data, run calculations, take actions.

So give them tools that can. The code below is a simple version of tools that can be created and given to tools to perform tasks. We will go deeper into them in Chapter 5.

```
from smolagents import CodeAgent, tool

@tool
def get_stock_volatility(ticker: str) -> float:
    """Calculate annualized volatility from historical returns."""
    returns = fetch_historical_data(ticker)
    return calculate_volatility(returns)

@tool
def get_correlation(ticker1: str, ticker2: str) -> float:
    """Calculate correlation between two stocks."""
    return calculate_correlation(ticker1, ticker2)

agent = CodeAgent(
    tools=[get_stock_volatility, get_correlation],
    model=model,
    additional_authorized_imports=["numpy"]
)

agent.run("Build a low-volatility portfolio for a risk-averse investor")
```

Instead of asking the LLM to guess, you give it tools - functions that fetch real data and run real calculations.

An AI agent is an LLM that can *act*, not just respond. It receives your request, decides which tools to call, interprets the results, and iterates until it has what it needs to answer you. Think of it as the difference between asking someone a question and asking someone to *go find out*. We'll break down the architecture in Chapter 3 - for now, the key idea is simple: it starts with letting an LLM use tools.

Now when you ask for a low-volatility portfolio, here's what happens:

1. The LLM receives your request
2. It calls `get_stock_volatility()` for candidate stocks
3. It calls `get_correlation()` to check how they move together
4. It writes code to optimize the allocation
5. The code executes with actual data
6. The LLM explains why the portfolio fits your risk profile

Same question. Real answer. Lower chance of hallucination.

The LLM still does the reasoning - understanding your goals, choosing which tools to use, interpreting results. But it's no longer guessing. It's working with facts.

1.3 The Hamburger Principle

Here's a mental model that will serve you throughout this book:

The buns are fluffy and essential - they're what you interact with. They parse your question ("build me a low-volatility portfolio"), understand your intent (risk-averse investor), and synthesize the output into something useful ("here's an allocation with 12% annualized volatility, and here's why it fits your profile").

But the buns aren't doing the real work. The meat is.

The meat is traditional computation: fetching historical returns from an API, calculating volatility with numpy, running correlation analysis, optimizing allocations with scipy. This is the consistent, explainable, verifiable part. You don't need an LLM to do these when you can write the actual tools to do it.

The vegetables are the boring but essential infrastructure: data validation, error handling, rate limiting, audit logs. Not glamorous, but they're what makes the system robust. That's out of the scope of this book, but we will cover it in the future.

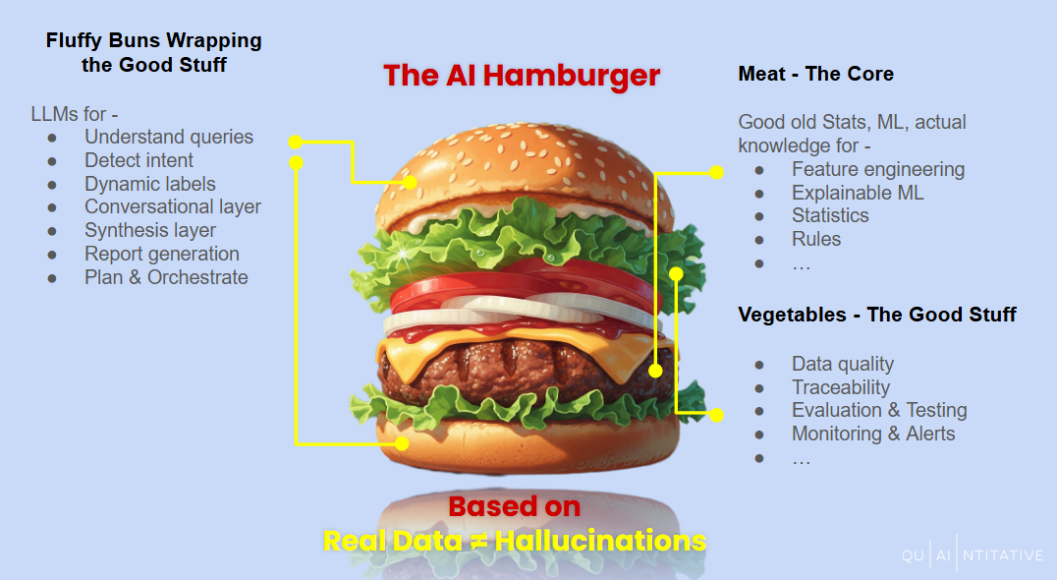

Figure 1.1: LLMs work best as buns, not meat.

Don't treat the LLM as the meat. Ask it to know facts, run calculations, make decisions. That's when it hallucinates - because you're asking the bun to do the meat's job.

The shift is straightforward. Let the LLM handle the interface - understanding what you want, choosing which tools to call, explaining results in plain language. Let traditional computation handle the core. Let the LLM be the bun that wraps around solid computational meat.

That's what AI agents can do if you let them.

1.4 What This Changes

Before agents: People try to use LLMs as meat. They ask ChatGPT to calculate volatility, and it invents numbers. The whole hamburger is just bun - fluffy and unsatisfying. You can't trust it.

After agents: The LLM stays in its lane as the bun. It handles understanding and communication. The tools handle computation. You get a complete hamburger - fluffy interface, solid core, and you can actually have a little bit more trust on what comes out.

This is the foundation of AI agents for finance. Not smarter models. Not bigger training data. Just the right architecture: LLMs as buns, computation as meat.

1.5 The Four Patterns

Tool calling is the foundation. But real finance work requires more.

Consider our opening example - building a portfolio for a risk-averse investor. A proper

answer needs:

- **Data fetching** for prices and historical returns
- **Multi-step iterative reasoning** to evaluate multiple candidates
- **Custom calculations** for volatility, correlations, expected returns
- **Memory** to handle follow-ups like "what if I increased my bond allocation?"

Over the next chapters, we'll cover four patterns that handle increasingly complex tasks:

Pattern	What It Does	Finance Example
Tool Calling	Fetch real data	"What's AAPL's price?"
ReAct	Multi-step reasoning	"Compare JPM and GS valuations"
CodeAct	Write and run code	"Calculate Sharpe ratio for these returns"
Orchestration	Combine all patterns	"Analyze my portfolio risk"

By the end of this book, you'll have built a portfolio assistant that can:

- Fetch real-time prices and fundamentals
- Compare multiple stocks with explicit, auditable reasoning
- Calculate Sharpe ratios, VaR, and custom metrics on the fly
- Remember your portfolio and run what-if scenarios

No hallucination. Real data. Auditable reasoning.

Let's build it.

Key Takeaways

LLMs can look up prices and basic facts, but they can't natively perform real financial analysis - checking volatility, calculating correlations, optimizing portfolios. When asked to do these things, they may hallucinate. The fix is **AI agents**: use LLMs as the "buns" (understanding input, explaining output) while traditional computation serves as the "meat" (real data, real calculations). This book teaches four simple patterns that make this work: Tool Calling, ReAct, CodeAct, and Orchestration.

2

The Complexity Ladder

Here's a question I get constantly: "Should I use an agent for this?"

Usually, the answer is NO. If you can keep it simple, why make life complicated and uncertain?

Not because agents aren't powerful - they are. But because most tasks don't need that power, and using an agent when a simpler solution works is like driving a Formula 1 car to the grocery store. Impressive, expensive, and completely unnecessary, and perhaps a little frivolous and showy.

This chapter gives you a framework for deciding when to use what.

2.1 The Three Levels

Think of capabilities as a ladder with three rungs.

Each level adds capability - and complexity. The goal is to use the lowest level that gets the job done.

2.2 Level 1: Single Tool Call

```
@tool
def get_stock_price(ticker: str) -> dict:
    """Get current price for a stock."""
    return fetch_from_api(ticker)
```

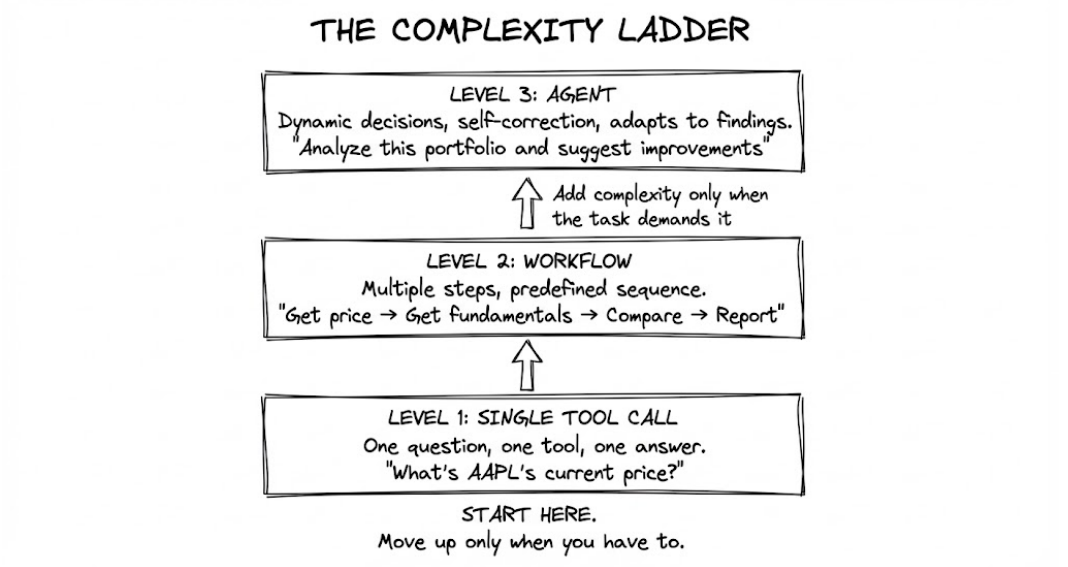

Figure 2.1: Keep it as simple as you can.

```
# User asks -> Tool fires -> Answer returns
result = get_stock_price("AAPL")
# {"price": 178.50, "change": 2.30, "change_pct": 1.31}
```

What it is: One question, one tool, one answer.

Example: "What's Apple's current stock price?"

When to use it:

- Simple lookups (price, P/E ratio, market cap)
- Single data points from one source
- Questions with one clear answer

The hamburger at Level 1: This is keto. Minimal bun, pure meat. The LLM barely needs to do anything - it just routes the question to the right tool and formats the response.

Most finance questions that seem simple actually are simple.

- "What's the price?"
- "What's the P/E?"
- "What sector is this company in?"

These don't need agents. They need a tool call.

2.3 Level 2: Workflow

What it is: Multiple steps in a predefined sequence. You know the steps in advance; you're just automating them.

Example: "Compare JPMorgan and Goldman Sachs on valuation metrics."

```
# Predefined workflow: Get data for both, then compare
def compare_banks(ticker1: str, ticker2: str) -> dict:
    # Step 1: Get fundamentals for first bank
    bank1 = get_fundamentals(ticker1)

    # Step 2: Get fundamentals for second bank
    bank2 = get_fundamentals(ticker2)

    # Step 3: Compare and format
    return {
        "comparison": {
            "pe_ratio": {ticker1: bank1["pe"], ticker2: bank2["pe"]},
            "pb_ratio": {ticker1: bank1["pb"], ticker2: bank2["pb"]},
            "dividend_yield": {ticker1: bank1["div_yield"], ticker2:
                bank2["div_yield"]}
        },
        "lower_pe": ticker1 if bank1["pe"] < bank2["pe"] else ticker2
    }

result = compare_banks("JPM", "GS")
```

Notice the LLM isn't doing much here - it might parse the user's request at the start and format the output at the end, but the sequence itself is hardcoded. That's the point. When you know the steps, you don't need an agent to figure them out.

When to use it:

- Multi-step tasks where you know the steps in advance
- Batch operations (run this for each stock in a list)
- Report generation with a fixed template
- Any task you could write as a flowchart

The hamburger at Level 2: The meat is a recipe - a series of computational steps you've defined. The bun handles input parsing and output formatting, but the sequence is fixed.

Workflows are underrated. Most "I need an agent" requests are actually "I need a workflow." in disguise.

2.4 Level 3: Agent

What it is: Dynamic decision-making. The agent figures out what steps to take based on what it finds along the way.

Example: "Analyze my portfolio and identify the biggest risks."

```
agent = CodeAgent(
    tools=[get_stock_price, get_fundamentals, get_historical_returns,
        get_correlations],
    model=model
)

result = agent.run("Analyze my portfolio: 40% AAPL, 35% NVDA, 25% MSFT.
    Identify the biggest risks.")
```

The agent might:

1. Fetch fundamentals for all three → Notice they're all tech
2. Decide to check correlations → Find AAPL-NVDA correlation is 0.72
3. Realize it should calculate concentration risk → Write code to compute it
4. Decide VaR would be useful → Calculate 95% VaR
5. Synthesize findings into a risk report

You didn't specify these steps. The agent figured them out based on the question, what it understood, and planned for it to happen.

When to use it:

- Open-ended analysis where you don't know the steps in advance
- Tasks that require adapting based on intermediate findings
- Complex reasoning across multiple data sources
- Conversational interfaces with follow-up questions

The hamburger at Level 3: The bun is doing more work - understanding intent, choosing tools, deciding what to do next. But the meat is still the computational core. The agent orchestrates the meat; it doesn't replace it.

2.5 The Decision Framework

One thing worth noting: even when building agents, you should write down the steps you *expect* it to take. The difference is that an agent can deviate from the plan based on

what it finds. But having a plan in your head - even a rough one - helps you choose the right tools, write better prompts, and catch unexpected behavior.

The key question: Do you know the exact steps in advance? Exact. Not approximate.

If yes → Workflow or Single Tool Call

If no → Agent

Most tasks fall into "yes." You know you need to fetch prices, calculate something, and format a report. That's a workflow, not an agent. Always remember that choosing an agent comes with risks.

2.6 Common Finance Tasks by Level

Task	Level	Why
"What's AAPL's price?"	1	Single lookup
"What's NVDA's P/E ratio?"	1	Single lookup
"Get prices for my watchlist"	2	Batch operation, known steps
"Compare JPM and GS valuations"	2	Known steps: fetch both, compare
"Generate weekly portfolio report"	2	Fixed template, known data sources
"Calculate Sharpe ratio for AAPL"	2	Known formula, known data needs
"Analyze my portfolio for risks"	3	Open-ended, adapts to findings
"Help me rebalance based on my goals"	3	Requires understanding goals, adapting
"Research whether TSLA is overvalued"	3	Multi-source, adapts to what it finds

Notice how few tasks actually need Level 3. The "impressive" agent demos often solve problems that a well-designed workflow handles better.

2.7 Why This Matters: Cost and Reliability

Each level up the ladder costs more:

An agent analyzing a portfolio might make 10–15 LLM calls as it reasons through the problem. That's 10–15x the cost of a single call, plus the added complexity of handling failures, loops, and unexpected behavior.

Reliability also decreases as you climb:

- **Level 1:** Almost always works. Tool either returns data or errors.
- **Level 2:** Highly reliable. Steps are predefined; failures are predictable.
- **Level 3:** Sometimes surprising. Agent might take unexpected paths, get stuck, or

produce inconsistent results.

This isn't a reason to avoid agents - it's a reason to use them only when you need them. Apply your judgment. Be judicious.

2.8 The Complexity Trap

Here's the trap I see constantly:

> "I want to build an AI portfolio assistant."
>
> *Immediately starts building an agent.*

But what does "portfolio assistant" actually mean? Break it down:

- "Show me my current positions" → Level 1 (fetch from database)
- "What's my portfolio worth today?" → Level 2 (fetch prices, multiply by shares, sum)
- "How has my portfolio performed this month?" → Level 2 (fetch historical prices, calculate returns)
- "Generate my monthly report" → Level 2 (known template, known data)
- "Should I rebalance?" → Level 3 (requires analyzing goals, current allocation, market conditions)

Four out of five features are Level 1 or 2. Only one needs an agent.

The right approach: Build the workflows first. Add agent capabilities only for the tasks that truly need dynamic reasoning. Your system will be faster, cheaper, and more reliable.

Always break things down. Never just dump it into an LLM and ask it to figure out what to do.

2.9 Climbing the Ladder

Throughout this book, we'll climb the ladder deliberately:

Chapter 5 (Tool Calling): Level 1. Single tools, single calls. The foundation. Level 2 when you add more tools and steps.

Chapter 6 (ReAct): Level 3 reasoning pattern. How agents think through multi-step problems.

Chapter 7 (CodeAct): Level 3 computation. How agents write and execute code.

Chapter 8 (Orchestration): Combining levels. When to use workflows vs. agents, and how to build systems that use both.

We start at the bottom and earn our way up. By the time you build your first agent, you'll understand exactly why you need one - and when you don't.

My wish is that you realise that the best way to build an agent is to know when not to build one. And if you boss tells you to build an agent when its not needed? It's ok, just build do Level 1 or 2 and tell him you built an agent. I'm fairly sure you will do fine.

Key Takeaways

Not every task needs an agent. The Complexity Ladder has three levels:

- **Level 1 (Single Tool Call):** One question, one tool, one answer. Use for simple lookups.
- **Level 2 (Workflow):** Multiple steps in a predefined sequence. Use when you know the steps in advance.
- **Level 3 (Agent):** Dynamic decisions based on intermediate findings. Use when the task requires adapting.

The key question: Can you write down the steps before running? If yes, you probably don't need an agent.

Start at Level 1. Move up only when the task demands it. Agents are powerful, but workflows are often better: faster, cheaper, more reliable.

3

Agent Architecture

Before we build anything, let's understand what we're building. An AI agent isn't magic. It's usually a specific architecture with four components that work together. Understanding these components - what they do and how they interact - will make everything else in this book click.

3.1 The Four Components

Every agent, regardless of complexity, usually has these basic components:

Model: The LLM that does the reasoning. GPT-X.X, Claude X.X - any model that can follow instructions and generate structured outputs.

Tools: Functions the model can call. These are your connection to the real world - fetching data, running calculations, taking actions.

Instructions: The system prompt that shapes behavior. This tells the model who it is, what it should do, and how it should respond.

Memory: The conversation history and context. This lets the agent remember what you've discussed and maintain coherent conversations.

That's it. Four components. Every agent pattern we'll cover - Tool Calling, ReAct, CodeAct, Orchestration - is just a different way of configuring and coordinating these four pieces. There are newer patterns for sure, but for the purpose of this book, let's keep it simple.

3.2 Component 1: The Model

The model is the LLM that does the reasoning. It reads your question, decides what to do, interprets results, and generates responses. It's the fluffy buns.

For finance applications, you want a model that:

- Follows instructions reliably
- Handles structured data (JSON, tables)
- Reasons through multi-step problems
- Knows when to use tools vs. answer directly

Which model should you use?

Start with the cheap ones. GPT-xyz-mini or Claude xyz Haiku. Cheap, fast, perhaps good enough. Then slowly go up the ladder. Also try the Chinese models - Qwen, Minimax - which are getting better and better for agents. The notebooks in this book work with any of these. Start cheap, upgrade when you need to. Experiment. The frontier keeps moving - both the cheap and expensive parts of it.

The model doesn't know anything about your portfolio, current stock prices, or market conditions. That's not its job. Its job is to reason about what tools to call and how to interpret the results.

3.3 Component 2: Tools

Tools are functions the model can call. They're how the agent *does things* - fetching data, running calculations, taking actions.

```
@tool
def get_stock_price(ticker: str) -> dict:
    """
    Get current price for a stock.

    Args:
        ticker: Stock symbol (e.g., 'AAPL', 'NVDA')

    Returns:
        Dict with price, change, and percent change
    """
    data = fetch_from_api(ticker)
    return {
        "ticker": ticker,
        "price": data["price"],
```

```
        "change": data["change"],
        "change_pct": data["change_pct"]
    }
```

A tool has three parts the model reads:

Name: What the model calls it (`get_stock_price`)

Signature: What inputs it takes and outputs it returns (`ticker: str -> dict`)

Docstring: Natural language description the model reads to understand when and how to use the tool

The docstring is surprisingly important. The model decides which tool to use based on your description. A vague docstring leads to wrong tool choices. A clear docstring leads to reliable behavior. Notice how much more detailed it is then when I first introduced it to you. In this case, talk is cheap, and really useful!

The docstring is for the AI and not just humans

Usually we write docstrings as documentation for humans. We live in a different era now. However, you can still write your docstrings as if you're explaining to a new colleague when to use this function. Be specific about what it does, what inputs it expects, and what outputs it returns.

The hamburger connection: Tools are the meat. They're the computational core - the part that actually touches real data and runs real calculations. No hallucination possible; it's just code executing.

We'll cover tools in detail in Chapters 4 and 5. For now, just understand: tools are functions with metadata that tells the model how to use them.

3.4 Component 3: Instructions

Instructions are the system prompt - the persistent context that shapes how the model behaves.

Instructions for a finance agent might look like this:

```
SYSTEM_PROMPT = """
You are a portfolio analysis assistant. Your role is to help users
understand their investments using real data and calculations.

RULES:
- Always use tools to fetch real data. Never make up prices or metrics.
- Show your reasoning step by step.
```

```
- When uncertain, say so. It's better to acknowledge limitations than guess.
- Format numbers clearly: prices to 2 decimals, percentages to 1 decimal.

STYLE:
- Be concise and precise
- Use financial terminology appropriately

You have access to tools for:
- Getting current stock prices
- Fetching company fundamentals
- Calculating portfolio metrics
- Analyzing correlations and risk

When analyzing a portfolio:
1. First, get current prices for all holdings
2. Calculate total value and weights
3. Check for concentration risk (any position > 25%)
4. If asked about risk, calculate volatility and correlations
"""
```

What instructions control:

- Persona (who the agent pretends to be)
- Constraints (what it should and shouldn't do)
- Style (how it communicates)
- Priorities (what to optimize for)

Notice the instructions above include a mini-workflow: "when analyzing a portfolio, first do X, then Y, then Z." This is a useful pattern - embedding common procedures directly in the system prompt.

Common mistake: Trying to put too much logic in instructions. Instructions shape behavior; they don't replace tools. "Calculate Sharpe ratio" shouldn't be an instruction - it should be a tool or code the agent writes.

And don't think that the model will always follow the instructions to a tee. It may sometimes do its own thing. But the more details your instructions, the more context you provide, the better.

3.5 Component 4: Memory

Memory is how the agent maintains context across interactions. Without memory, every question starts fresh. With memory, you can have conversations.

Without memory:

You: What's my portfolio worth?

Agent: Your portfolio (40% AAPL, 35% NVDA, 25% MSFT) is worth $147,230.

You: What if I sold half the NVDA?

Agent: I don't know what portfolio you're referring to.

With memory:

You: What's my portfolio worth?

Agent: Your portfolio (40% AAPL, 35% NVDA, 25% MSFT) is worth $147,230.

You: What if I sold half the NVDA?

Agent: If you sold half your NVDA position, your portfolio would be worth $138,450, with weights shifting to 43% AAPL, 19% NVDA, 27% MSFT, and 11% cash.

Memory is what makes an agent feel like a conversation rather than a series of disconnected queries.

In code, memory can be as simple as a list of messages:

```
memory = [
    {"role": "user", "content": "What's my portfolio worth?"},
    {"role": "assistant", "content": "Your portfolio is worth $147,230..."},
    {"role": "user", "content": "What if I sold half the NVDA?"},
]
```

Each new query appends to this list. The model sees the full history and can reference previous context.

In smolagents, you control memory with `reset=False`:

```
agent.run("What's AAPL's price?")                          # -> $178.50
agent.run("How does that compare to NVDA?", reset=False)    # -> AAPL is
    $178.50, NVDA is $875.30...
```

Memory types:

For this book: We use session memory. The agent remembers within a conversation but starts fresh each time you restart. Persistent memory is a production concern.

Type	Scope	Use Case
Session	Current conversation	Default - resets when you close
Persistent	Across sessions	Remember client portfolios over time
Vector	Semantic search	Find relevant past interactions

Memory has limits

Most models have a context window - a maximum amount of text they can process at once. For long conversations, you may need to summarize older messages or implement more sophisticated memory management. For the examples in this book, simple message history is sufficient.

3.6 How the Components Work Together

Here's what happens when you ask an agent a question:

1. **Input arrives:** "What's Apple's current P/E ratio?"
2. **Context is assembled:** Instructions + Memory + New input get combined into a single prompt
3. **Model reasons:** The LLM reads the context, sees available tools, decides what to do
4. **Tool calls execute:** Model calls `get_fundamentals("AAPL")`, gets real data back
5. **Model continues:** With tool results in hand, model may call more tools or generate a response
6. **Response returns:** Final answer goes back to user
7. **Memory updates:** The exchange gets added to conversation history

Follow-up: "Is that high compared to the sector?"

```
8. MEMORY provides context
     "We were just discussing AAPL's P/E of 28.5"

9. MODEL reasons with context
     Thinks: "I need sector comparison. Let me get other tech P/Es"
     Decides: Call get_fundamentals for MSFT, GOOGL

10. TOOLS execute
      MSFT pe=35.2, GOOGL pe=25.1

11. MODEL synthesizes
      "At 28.5, Apple's P/E is moderate for tech-higher than
       Alphabet (25.1) but lower than Microsoft (35.2)."

12. MEMORY updates
      Now includes the comparison context
```

This loop - reason, act, observe, repeat - is the heartbeat of every agent. The patterns we'll learn are variations on this theme.

3.7 The smolagents Framework

You've already seen smolagents code in earlier chapters - the `@tool` decorator and `CodeAgent` in the examples. Here's the formal introduction. We use **smolagents** from Hugging Face throughout this book. It's lightweight, well-documented, and designed for learning. Full documentation: huggingface.co/docs/smolagents

Why smolagents:

- Clean, readable code
- Works with multiple LLM providers
- Good defaults for beginners
- Transparent execution (you see what's happening)

Installation:

```
pip install smolagents
```

Other frameworks exist: LangChain, CrewAI, AutoGen. They're more feature-rich but also more complex. smolagents is ideal for learning the patterns. Once you understand the patterns, you can use any framework - or build your own. With LLMs helping you,

using any other framework once you understand the fundamentals is trivial. Trust me.

3.8 Your First Agent

Let's build a minimal agent to see the components in action. (The companion notebook `01_tool_calling.ipynb` walks through this hands-on.)

```
from smolagents import CodeAgent, tool, OpenAIServerModel

# Component 1: MODEL
model = OpenAIServerModel(model_id="gpt-4o-mini")

# Component 2: TOOLS
@tool
def get_stock_price(ticker: str) -> dict:
    """
    Get current price for a stock ticker.

    Use this when the user asks about stock prices or
    how a stock is doing today.

    Args:
        ticker: Stock symbol (e.g., 'AAPL', 'NVDA')

    Returns:
        Dict with price and daily change
    """
    # Simulated data for demo
    prices = {
        "AAPL": {"price": 178.50, "change": 2.30, "change_pct": 1.31},
        "NVDA": {"price": 875.30, "change": -12.40, "change_pct": -1.40},
    }
    return prices.get(ticker.upper(), {"error": f"Unknown ticker:
        {ticker}"})

# Component 3: INSTRUCTIONS
# smolagents uses sensible defaults; we'll customize later

# Create the agent
agent = CodeAgent(
    tools=[get_stock_price],
    model=model
)

# Component 4: MEMORY
# Handled automatically by CodeAgent
```

```
# Run it
result = agent.run("What's NVIDIA's stock price?")
print(result)
# Output: NVIDIA (NVDA) is currently trading at $875.30, down $12.40
    (-1.40%) today.
```

Four components, working together:

- **Model:** `OpenAIServerModel` wrapping GPT-4o-mini
- **Tools:** `get_stock_price` function with `@tool` decorator
- **Instructions:** `system_prompt` parameter (using defaults here)
- **Memory:** Handled automatically by `CodeAgent`

3.9 Seeing Inside the Agent

To understand what's happening, turn on verbose mode:

```
from smolagents.monitoring import LogLevel

agent = CodeAgent(
    tools=[get_stock_price],
    model=model,
    verbosity_level=LogLevel.INFO  # Show reasoning
)

result = agent.run("What's NVIDIA's stock price?")
```

Output: (*Note that outputs will almost certainly vary.*)

```
╭──────────────────────────────────────────────────╮
Thinking...
The user wants NVIDIA's stock price. I should use the
get_stock_price tool with ticker 'NVDA'.

Calling tool: get_stock_price
   Arguments: {"ticker": "NVDA"}

Tool result: {"price": 875.30, "change": -12.40, "change_pct": -1.40}

Thinking...
I have the price data. Let me format a clear response.
```

```
Final answer: NVIDIA (NVDA) is currently trading at $875.30,
   down $12.40 (-1.40%) today.
================================================
```

This transparency is crucial for finance. You can see exactly what the agent thought, what tools it called, and what data it used. No black box.

3.10 Why This Architecture Matters

Understanding the four components helps you debug, extend, and reason about agents.

Agent giving wrong answers? Check your tools. Are they returning correct data?

Agent using wrong tools? Check your docstrings. Are they clear about when to use each tool?

Agent forgetting context? Check your memory. Is conversation history being preserved?

Agent behaving erratically? Check your instructions. Are they specific enough?

Most agent problems trace back to one of these four components. When something goes wrong, systematically check each one.

The word 'agent' is loosely used. What's more important is understanding the role of these components.

3.11 The Architecture in Hamburger Terms

- **Top Bun:** Model (understanding) + Instructions (behavior)
  - Parses "What's NVIDIA's price?", decides to call tool
- **Vegetables (Pre):** Input guardrails, error handling, input validation, rate limiting
  - A topic for a future series
- **Meat:** Tools — the actual computation
  - `get_stock_price()` fetches real data
- **Vegetables (Post):** Output guardrails, logging, audit trails, caching
  - A topic for a future series
- **Bottom Bun:** Model (synthesis) + Memory (saving context)
  - Formats response, remembers for follow-ups

The model appears twice - as top bun (understanding input, choosing tools) and bottom bun (synthesizing output, using memory). That's because LLMs handle both ends of the sandwich. The meat in the middle is pure computation. Or more predictable models.

3.12 What's Next

You now understand the architecture: Model, Tools, Instructions, Memory.

In the next chapters, we'll go deep on each component:

- **Chapter 5 (Tool Calling):** How to build tools the model can use
- **Chapter 6 (ReAct):** How the model reasons through multi-step problems
- **Chapter 7 (CodeAct):** How the model writes and executes code
- **Chapter 8 (Orchestration):** How to combine everything with memory

Each pattern builds on this foundation. The architecture stays the same; we just use it in increasingly sophisticated ways.

Key Takeaways

An AI agent usually has four components:

- **Model:** The LLM that reasons and decides
- **Tools:** Functions the model can call to interact with the real world
- **Instructions:** System prompt that shapes behavior
- **Memory:** Conversation history for context

The agent loop: receive input → assemble context → reason → call tools → generate response → update memory.

Every pattern in this book is a variation on this architecture. Understand it once, and the rest follows. Simple, right?

4

The Three Types of Finance Tools

Not all tools are created equal. Some tools just read data - harmless. Others place trades or send alerts - consequential. Before you build anything, you need a framework for thinking about what your tools can do and what risks they carry.

This chapter gives you that framework. And we'll ground it in the actual tools you'll build in this book.

4.1 The Classification

One can think about finance tools (and maybe tools in many other domains) as being in one of three categories:

	Retrieving *Read-Only*	**Analyzing** *Computation*	**Acting** *Comes with Side Effects*
Examples	Prices Company info Historical returns	Sharpe ratio VaR Optimization	Place orders Send alerts Rebalance
Functions	`get_price` `get_financials`	`calc_sharpe` `calc_risk`	`place_order` `send_email`
Risk Profile	**LOW RISK**	**LOW-MEDIUM RISK**	**MEDIUM-HIGH RISK**
Focus	*Covered in this book*	*Covered in this book*	*Production topic*

Table 4.1: Three Categories

Let's examine each type - some with real examples from the notebooks you'll run.

4.2 Type 1: Market Data Tools (Read-Only)

Market data tools fetch information. They don't change anything - they just read.

We'll start with simulated data so you can see what's happening. Later, we will use APIs to get real financial data.

From the Tool Calling notebook (Module 1):

```
@tool
def get_stock_price(ticker: str) -> float:
    """
    Get the current price for a stock ticker.

    Args:
        ticker: The stock symbol (e.g., 'AAPL', 'NVDA')

    Returns:
        The current stock price as a float
    """
    # Simulated prices for demo
    prices = {
        "AAPL": 178.50,
        "NVDA": 875.30,
        "MSFT": 378.90
    }
```

```
    return prices.get(ticker.upper(), 0.0)
```

```
@tool
def get_company_info(ticker: str) -> str:
    """
    Get basic company information for a stock ticker.

    Args:
        ticker: The stock symbol (e.g., 'AAPL', 'NVDA')

    Returns:
        A string with company name, sector, and market cap
    """
    companies = {
        "AAPL": "Apple Inc. | Technology | $2.8T market cap",
        "NVDA": "NVIDIA Corporation | Semiconductors | $2.2T market cap"
    }
    return companies.get(ticker.upper(), "Company not found")
```

From the Portfolio Optimization notebook (Module 4):

```
@tool
def get_historical_prices(tickers: str, years: int = 2) -> str:
    """
    Fetch historical adjusted close prices for a list of stock tickers.

    Args:
        tickers: Comma-separated stock symbols (e.g., 'AAPL,MSFT,GOOGL')
        years: Number of years of historical data (default: 2)

    Returns:
        JSON string with price statistics and recent prices
    """
    # Uses yfinance to fetch real data
    prices = yf.download(ticker_list, start=start_date,
        end=end_date)["Close"]
    # Returns statistics: latest_price, annual_return, annual_volatility
```

Characteristics:

- **No side effects:** Calling them doesn't change anything in the world
- **Idempotent:** Call them 10 times, get the same result (aside from market movements)

- **Safe to automate:** Let the agent call these freely

Risk level: LOW-MEDIUM

You can let an agent call market data tools without human approval. The worst case is wasted API calls, not major financial damage. There is this slight risk that the LLM uses the tools wrong. But, usually it either fails or the answer just does not make sense.

The hamburger connection: Market data tools are pure meat. They fetch facts. No hallucination possible - it's just data retrieval from real sources.

4.3 Type 2: Analytical Tools (Computation)

Analytical tools take data and compute something from it. They don't change external state - they just crunch numbers.

From the CodeAct notebook (Module 3):

The agent can write code on the fly to calculate:

```
# Agent writes this code:
def calculate_sharpe_ratio(returns, risk_free_rate=0.005):
    excess_returns = [r - risk_free_rate for r in returns]
    return np.mean(excess_returns) / np.std(excess_returns, ddof=1)

returns = [0.05, 0.03, -0.02, 0.04, 0.01, -0.01, 0.03, 0.02]
sharpe = calculate_sharpe_ratio(returns)
# Result: 0.72
```

From the Portfolio Optimization notebook (Module 4):

```
@tool
def optimize_max_sharpe(tickers: str, risk_free_rate: float = 0.02) -> str:
    """
    Find the portfolio with the maximum Sharpe ratio (best risk-adjusted
        return).

    Args:
        tickers: Comma-separated stock symbols (e.g.,
            'AAPL,MSFT,GOOGL,AMZN')
        risk_free_rate: Annual risk-free rate as decimal (default: 0.02 =
            2%)

    Returns:
        JSON with optimal weights and expected performance
    """
```

```
    # Uses PyPortfolioOpt for mean-variance optimization
    ef = EfficientFrontier(mu, S)
    weights = ef.max_sharpe(risk_free_rate=risk_free_rate)
    # Returns: weights, expected_return, volatility, sharpe_ratio
```

```
@tool
def optimize_min_volatility(tickers: str) -> str:
    """
    Find the portfolio with minimum volatility (lowest risk).
    Best for conservative investors who prioritize capital preservation.
    """
    ef = EfficientFrontier(mu, S)
    weights = ef.min_volatility()
```

```
@tool
def compare_strategies(tickers: str) -> str:
    """
    Compare Max Sharpe vs Min Volatility strategies for the same assets.
    """
    # Runs both optimizations and returns side-by-side comparison
```

From the Finance Concepts notebooks:

The finance notebooks teach the underlying calculations:

- **Sharpe ratio:** $(R_p - R_f)/\sigma_p$ - return per unit of risk
- **Value at Risk:** Historical, Parametric, and Monte Carlo methods
- **Maximum drawdown:** Worst peak-to-trough decline
- **Correlation:** How assets move together

Characteristics:

- **Pure computation:** Input data, output results
- **Deterministic:** Same inputs produce same outputs
- **Safe to automate:** Let the agent compute freely

Risk level: LOW

Calculation tools don't touch external systems. They just do math. The agent can call them without human approval.

The hamburger connection: Calculation tools are also meat - the computational core. They transform data into insights. No side effects, no risk.

4.4 Type 3: Action Tools (Comes with Side Effects)

Action tools *do things*. They change state in the world - placing orders, sending messages, updating records.

Examples (NOT built in this book):

```
@tool
def place_order(ticker: str, quantity: int, order_type: str) -> dict:
    """
    Place a stock order. WARNING: EXECUTES REAL TRADES.

    This tool requires human confirmation before execution.
    """
    # This would connect to a brokerage API
    # and actually buy or sell shares
    pass

@tool
def send_alert(recipient: str, message: str) -> dict:
    """
    Send an alert to a client or colleague.

    WARNING: Sends real messages. Requires approval.
    """
    pass

@tool
def rebalance_portfolio(target_weights: dict) -> dict:
    """
    Execute trades to rebalance portfolio to target weights.

    WARNING: Executes multiple trades. Requires approval.
    """
    pass
```

Characteristics:

- **Has side effects:** Calling them changes something in the real world
- **Not idempotent:** Call twice, get different results (two orders, two alerts)
- **Needs oversight:** Human should approve, or at least review

Risk level: MEDIUM to HIGH

This is where things get serious. An agent that can place orders can lose money. An agent that can send alerts can spam clients. An agent that can update records can corrupt data.

The hamburger connection: Action tools are still meat - computation - but they're meat that can bite back. You need vegetables (guardrails) wrapped around them.

4.5 What This Book Builds

Let me be clear about scope:

This book: Market Data + Analytical Tools

We focus on tools that fetch data and compute metrics. Here's what you'll actually build:

Notebook	Tools
`01_tool_calling`	`get_stock_price`, `get_company_info`, `calculate_position_value`, `get_52_week_high`
`02_react_pattern`	Same tools, multi-step reasoning
`03_codeact_pattern`	`get_historical_returns` + dynamic code for Sharpe, VaR, correlation
`05_portfolio_optimization`	`get_historical_prices`, `optimize_max_sharpe`, `optimize_min_volatility`, `optimize_target_return`, `calculate_allocation`, `compare_strategies`

All read-only or pure computation. All safe to experiment with. The agent can't lose money or damage anything by calling them.

Mentioned but not built: Action Tools

We discuss action tools conceptually and show the guardrail framework. We don't build tools that place trades or send alerts. That's production territory requiring:

- Approval workflows
- Audit logging
- Rollback capabilities
- Compliance review

4.6 The Guardrail Framework

Given these three types, how do you decide what to allow?

Simple rule:

- **Read-only?** → Auto-approve
- **Pure computation?** → Auto-approve but good to add some checks.
- **Changes external state?** → Human reviews or robust verifications

But in all cases, if there is an easy way to verify the answers, do it!

4.7 Tool Design Principles

Here's what makes the difference between a tool that works and one that confuses the agent.

4.7.1 Principle 1: Clear "When to Use" in Docstrings

The docstring isn't just documentation - it's instructions for the AI. Look at this example from our optimization notebook:

```
@tool
def optimize_min_volatility(tickers: str) -> str:
    """
    Find the portfolio with minimum volatility (lowest risk).
    Best for conservative investors who prioritize capital preservation.

    Args:
        tickers: Comma-separated stock symbols (e.g.,
            'AAPL,MSFT,GOOGL,AMZN')

    Returns:
        JSON with optimal weights and expected performance
    """
```

Notice: "Best for conservative investors who prioritize capital preservation." That tells the agent *when* to choose this tool over `optimize_max_sharpe`.

4.7.2 Principle 2: One Tool, One Job

Don't make tools that do five things. Make five tools that each do one thing well.

Our portfolio optimization notebook has six focused tools:

- `get_historical_prices` - fetch data
- `optimize_max_sharpe` - one optimization strategy
- `optimize_min_volatility` - another strategy

- `optimize_target_return` - yet another strategy
- `calculate_allocation` - convert weights to shares
- `compare_strategies` - compare two strategies

The agent combines them as needed. You don't need a giant `do_everything` tool.

4.7.3 Principle 3: Structured Returns

Return data the agent can work with:

```
return json.dumps({
    "optimization": "Maximum Sharpe Ratio",
    "weights": {"AAPL": 63.7, "NVDA": 36.3},
    "performance": {
        "expected_annual_return": 20.3,
        "annual_volatility": 25.3,
        "sharpe_ratio": 0.80
    }
}, indent=2)
```

Not a wall of text. Not raw numbers. Structured data with clear field names.

4.8 The CodeAct Hybrid

There's a fourth pattern worth noting: **tools + code generation**.

In Module 3, the agent uses a tool to fetch data and then writes custom code to analyze it:

```
# Agent with BOTH a tool AND code execution
agent_combined = CodeAgent(
    tools=[get_historical_returns],  # Tool for fetching
    model=model,
    additional_authorized_imports=["numpy", "pandas"]  # Code for analysis
)

agent_combined.run("Compare the Sharpe ratios of AAPL, NVDA, and MSFT")
```

The agent:

1. Calls `get_historical_returns` for each stock (tool)
2. Writes Python code to calculate Sharpe ratios (CodeAct)

3. Compares and ranks them (reasoning)

This is the power pattern: Use tools for reliable data fetching, use CodeAct for flexible analysis.

4.9 Common Mistakes

Mistake 1: Vague tool descriptions

"Get data" tells the model nothing. Compare:

Bad: *Get stock data.*

Good: *Get PE ratio and EPS for valuation questions - do NOT use for current prices (use get_stock_price)*

Mistake 2: Too many tools

More tools = more confusion. Start with 3–5 focused tools. Add more only when needed.

Our portfolio optimization agent has exactly 6 tools - enough to be powerful, few enough to be clear.

Mistake 3: Returning strings when you need structure

If the agent needs to use the result in calculations, return structured data (dict, JSON), not formatted text.

Mistake 4: Mixing read and write in one tool

A tool that both fetches data AND updates a database has two jobs. Split it into two tools.

Key Takeaways

Finance tools fall into three categories:

- **Market Data (Read-Only):** Fetch prices, fundamentals, historical data. Safe to automate.
- **Analytics (Computation):** Sharpe ratios, VaR, optimization. Pure math, safe to automate.
- **Actions (Side Effects):** Place orders, send alerts, update records. Require human oversight.

The rule: Read-only and computation tools are lower risk. State-changing tools are high risk and need human review or robust verification.

5

Your First Agent - Tool Calling

Enough theory. Let's build something.

By the end of this chapter, you'll have a working agent that can answer questions about stock prices using real data (first, simulated real data for ease of understanding, then connected to live APIs). More importantly, you'll understand *why* it works.

5.1 The Setup

Everything you need to get started is right here. The companion notebooks include the same setup steps - so if you're following along in Colab, you can skip ahead to the code.

We're using **smolagents** from Hugging Face - a lightweight, well-documented framework designed for learning. It works with multiple LLM providers (OpenAI, Anthropic, Hugging Face) and makes the internals visible.

I like simple and boring. So, the lightweight **smolagents** is perfect. Starting simple is usually better for learning. In any case the patterns learned here can be applied to any other agent framework/library.

Installation:

```
pip install smolagents
```

Core imports:

```
from smolagents import CodeAgent, tool
from smolagents import OpenAIServerModel
from smolagents.monitoring import LogLevel
```

Initialize the model:

```
import getpass
API_KEY = getpass.getpass("Enter your OpenAI API key: ")

model = OpenAIServerModel("gpt-4o-mini", api_key=API_KEY)
print("Model initialized!")
```

We're using `gpt-4o-mini` - OpenAI's cost-effective model. It's capable enough for learning and cheap enough to experiment freely. You can swap in Claude or an open-source model later.

Use the 'LLM Providers' notebooks to set up and get your API key if you are new to this. Trust me, it's easier than it looks.

5.2 The @tool Decorator

This is where the magic happens. The `@tool` decorator transforms a regular Python function into something an LLM can call.

```
@tool
def get_stock_price(ticker: str) -> float:
    """
    Get the current price for a stock ticker.

    Args:
        ticker: The stock symbol (e.g., 'AAPL', 'NVDA')

    Returns:
        The current stock price as a float
    """
    # Simulated prices for demo (no API key needed!)
    prices = {
        "AAPL": 178.50,
        "NVDA": 875.30,
        "MSFT": 378.90,
        "GOOGL": 141.25,
        "AMZN": 178.75
    }
    return prices.get(ticker.upper(), 0.0)
```

What the decorator does:

1. **Registers the function** as something the LLM can call
2. **Extracts the signature** - the LLM knows it needs a `ticker` string and gets back a float
3. **Uses the docstring** as instructions for the LLM

Key insight: The docstring isn't just documentation for humans - it's instructions for the AI. The LLM reads it to decide *when* to use this tool.

Let's test it standalone:

```
print(f"AAPL price: ${get_stock_price('AAPL')}")
# Output: AAPL price: $178.50
```

The function works on its own. Now let's give it to an agent.

5.3 Creating the Agent

```
agent = CodeAgent(
    tools=[get_stock_price],
    model=model
)

print("Agent created with 1 tool: get_stock_price")
```

That's it. Three lines:

1. Import the components
2. Create the model
3. Create the agent with tools

Now let's run it.

5.4 The Magic Moment

```
result = agent.run("What is Apple's current stock price?")
print(result)
```

Output:

```
Apple's current stock price is $178.50.
```

What just happened:

1. We asked in natural language: "What is Apple's current stock price?"
2. The agent figured out it needed to call `get_stock_price` with `'AAPL'`
3. The tool returned the real number (well, our simulated number)
4. The agent formulated a human-readable response

No hallucination. Real data.

The LLM didn't guess the price. It didn't search its training data. It called a function that returned a specific value. That's the difference.

5.5 Seeing Inside the Agent

Let's turn on verbose mode to see the agent's reasoning:

```
agent_verbose = CodeAgent(
    tools=[get_stock_price],
    model=model,
    verbosity_level=LogLevel.INFO  # Shows the reasoning
)

result = agent_verbose.run("What is Apple's current stock price?")
```

Output:

```
╭──────────────────────────────────────────────────╮
Thinking...
The user wants Apple's stock price. I should use the
get_stock_price tool with ticker 'AAPL'.

Calling tool: get_stock_price
   Arguments: {"ticker": "AAPL"}
```

```
Tool result: 178.5

Thinking...
I have the price. Let me format a clear response.

Final answer: Apple's current stock price is $178.50.
==================================================
```

This transparency is powerful. You can see exactly:

- What the agent thought
- What tool it decided to call
- What arguments it passed
- What result came back
- How it synthesized the response

No black box. Every step visible. This matters for compliance, debugging, and building trust.

5.6 Adding More Tools

One tool is useful. Multiple tools are powerful. Let's add a few more:

```
@tool
def get_company_info(ticker: str) -> str:
    """
    Get basic company information for a stock ticker.

    Args:
        ticker: The stock symbol (e.g., 'AAPL', 'NVDA')

    Returns:
        A string with company name, sector, and market cap
    """
    companies = {
        "AAPL": "Apple Inc. | Technology | $2.8T market cap",
        "NVDA": "NVIDIA Corporation | Semiconductors | $2.2T market cap",
        "MSFT": "Microsoft Corporation | Technology | $3.1T market cap",
        "GOOGL": "Alphabet Inc. | Technology | $1.9T market cap",
        "AMZN": "Amazon.com Inc. | Consumer Cyclical | $1.9T market cap"
    }
    return companies.get(ticker.upper(), "Company not found")
```

```
@tool
def calculate_position_value(ticker: str, shares: int) -> float:
    """
    Calculate the total value of a stock position.

    Args:
        ticker: The stock symbol
        shares: Number of shares owned

    Returns:
        Total position value in dollars
    """
    price = get_stock_price(ticker)
    return price * shares
```

Notice the patterns:

- Clear docstrings with "when to use" context
- Type hints on all parameters
- Structured return values

Now create an agent with all three tools:

```
agent = CodeAgent(
    tools=[get_stock_price, get_company_info, calculate_position_value],
    model=model,
    verbosity_level=LogLevel.INFO
)

print("Agent now has 3 tools!")
```

5.7 Multi-Tool Queries

Here's where it gets interesting. Ask a question that requires multiple tools:

```
result = agent.run("Tell me about NVIDIA and what 100 shares would be
    worth")
```

Output:

```
==================================================
Thinking...
The user wants information about NVIDIA and a position calculation.
I'll need to use get_company_info for the company details
and calculate_position_value for the 100 shares.

Calling tool: get_company_info
   Arguments: {"ticker": "NVDA"}

Tool result: NVIDIA Corporation | Semiconductors | $2.2T market cap

Calling tool: calculate_position_value
   Arguments: {"ticker": "NVDA", "shares": 100}

Tool result: 87530.0

Final answer: NVIDIA Corporation is a semiconductor company with a
$2.2 trillion market cap. 100 shares would be worth $87,530.00 at
the current price.
==================================================
```

Two tools, one query! The agent:

1. Understood it needed company info AND position value
2. Called the right tools in sequence
3. Combined the results into a coherent response

This is the power of tool calling: the LLM handles **understanding and reasoning**, the tools handle **doing**.

5.8 The Key Principles

Three principles that make tools work well:

5.8.1 Principle 1: Clear Docstrings

The LLM reads your docstring to decide when to use the tool. Be specific about:

- **What** the tool does
- **When** to use it
- **What** it returns

Bad: *Get data.*

Good: *Get current price for a stock ticker. Use when the user asks about stock prices or valuations. Returns price as a float.*

5.8.2 Principle 2: Type Hints Matter

```
def get_stock_price(ticker: str) -> float:
```

The `ticker: str` tells the LLM to pass a string. The `-> float` tells it to expect a number back. Don't skip these.

5.8.3 Principle 3: One Tool, One Job

Don't make a tool that:

- Gets prices AND company info AND calculates positions

Make three tools that each do one thing well:

- `get_stock_price` - prices
- `get_company_info` - company details
- `calculate_position_value` - position math

The agent combines them as needed. You get flexibility without complexity.

5.9 Memory: The Internal Tool

So far we've talked about **external tools** - functions that fetch data from the world. But agents have an **internal tool** that's just as important: **memory**.

- **External Tools** (The `@tool` decorator)
  - What we have covered. Functions that fetch data from the outside world.
  - **Examples:** `get_stock_price`, `get_company_info`, `calculate_value`.
- **Internal Tools** (The `agent.memory`)
  - Systems that store context and enable recall.
  - **Role:** Ensures continuity across conversation turns.

Memory stores everything: what you asked, what tools were called, what results came back. It's what lets the agent have a **conversation** rather than just answering one-off questions.

5.9.1 Using Memory for Conversations

By default, each `agent.run()` call resets memory. To maintain context across queries, use `reset=False`:

```
# First query - starts fresh
result = agent.run("What's Apple's stock price?")
print(result)
# Output: Apple's current stock price is $178.50.

# Follow-up - preserves memory from previous run
result = agent.run("And what about NVIDIA?", reset=False)
print(result)
# Output: NVIDIA's current stock price is $875.30.
```

Without `reset=False`, the agent wouldn't know "what about" refers to stock prices. With it, the agent remembers the context and understands the follow-up.

5.9.2 Inspecting Memory

You can see what the agent remembers:

```
print("Agent Memory:")
for i, step in enumerate(agent.memory.steps):
    step_type = type(step).__name__
    print(f"\nStep {i+1}: {step_type}")

    if step_type == "TaskStep":
        print(f"   Task: {step.task}")

    elif step_type == "ActionStep":
        if hasattr(step, 'tool_calls') and step.tool_calls:
            for tc in step.tool_calls:
                print(f"   Tool: {tc.name}({tc.arguments})")
```

This shows every step: what was asked, what tools were called, what came back. Full audit trail.

5.10 Data Sources: Where the Numbers Come From

Our examples use simulated data - hardcoded dictionaries. That's intentional for learning (no API keys, no rate limits, predictable results). But in production, you'd connect to real sources.

Upgrading the tool:

```
@tool
def get_stock_price_live(ticker: str) -> float:
    """Get current price from Yahoo Finance."""
    import yfinance as yf
    stock = yf.Ticker(ticker)
    return stock.info.get('regularMarketPrice', 0.0)
```

Same interface, real data. The agent doesn't care where the data comes from - it just calls the tool.

We'll use yfinance in the Orchestration chapter (Chapter 8) to demonstrate real data workflows.

5.11 Exercise: Build Your Own Tool

Here's your assignment. Create a tool that returns the 52-week high for a stock:

```
@tool
def get_52_week_high(ticker: str) -> float:
    """
    # YOUR DOCSTRING HERE
    # Describe what the tool does, its args, and return value
    """
    # YOUR CODE HERE
    # Create a dictionary of 52-week highs and return the value
    pass
```

Hints:

- Follow the same pattern as `get_stock_price`
- Include clear docstring with when-to-use guidance
- Use type hints
- Return a float

Test it:

```
print(f"AAPL 52-week high: ${get_52_week_high('AAPL')}")
```

Then add it to an agent:

```
agent_with_high = CodeAgent(
    tools=[get_stock_price, get_company_info, get_52_week_high],
    model=model,
    verbosity_level=LogLevel.INFO
)

result = agent_with_high.run("How far is Apple from its 52-week high?")
```

The agent should call both `get_stock_price` and `get_52_week_high`, then calculate the difference.

5.12 Solution

Don't peek until you've tried it:

```
@tool
def get_52_week_high(ticker: str) -> float:
    """
    Get the 52-week high price for a stock ticker.

    Use this when the user asks about highs, peaks, or
    maximum prices over the past year.

    Args:
        ticker: The stock symbol (e.g., 'AAPL', 'NVDA')

    Returns:
        The 52-week high price as a float
    """
    import yfinance as yf
    stock = yf.Ticker(ticker)
    return stock.info.get('fiftyTwoWeekHigh', 0.0)
```

Key elements:

- Clear "when to use" in docstring ("highs, peaks, maximum prices")
- Type hints on input and output
- Simple implementation with simulated data

5.13 What You Built

In this chapter, you built:

1. **A working agent** with the `@tool` decorator and `CodeAgent`
2. **Three finance tools:** price, company info, position value
3. **Understanding of memory** for conversation continuity
4. **Your own tool** for 52-week highs

You've seen:

- How docstrings guide tool selection
- How the agent reasons through multi-tool queries
- How verbose mode exposes the decision process
- How memory enables follow-up questions

The hamburger check: The bun (LLM) understands the question and chooses tools. The meat (tools) fetches real data. Lower risk of hallucination because the LLM never has to make up numbers - it just asks. There is of course still a risk that the API call somehow fails, and the LLM makes things up. **Nothing in life is perfect.** But at least now you can log that.

Key Takeaways

The `@tool` decorator bridges LLMs and real data. Three things make tools work:

1. **Clear docstrings** - Tell the LLM when to use the tool
2. **Type hints** - Tell it what to pass and expect back
3. **One job per tool** - Keep them focused and composable

Memory is an internal tool that maintains conversation context. Use `reset=False` to enable follow-up questions.

This is Pattern 1: Tool Calling. The foundation everything else builds on.

6

The ReAct Pattern — Multi-Step Reasoning

In the last chapter, you built an agent that calls tools. Ask a question, call a tool, get an answer. Simple.

But real finance questions aren't simple:

> "Compare Apple and NVIDIA's stock performance. Which one is trading closer to its 52-week high?"

To answer this, you need to:

1. Get Apple's current price
2. Get Apple's 52-week high
3. Get NVIDIA's current price
4. Get NVIDIA's 52-week high
5. Calculate the percentage for each
6. Compare and explain

That's not one tool call. That's a **chain of reasoning with multiple steps**.

This is where the **ReAct pattern** comes in.

6.1 What is ReAct?

ReAct stands for **Reason + Act**. It's a pattern where the agent alternates between thinking and doing.

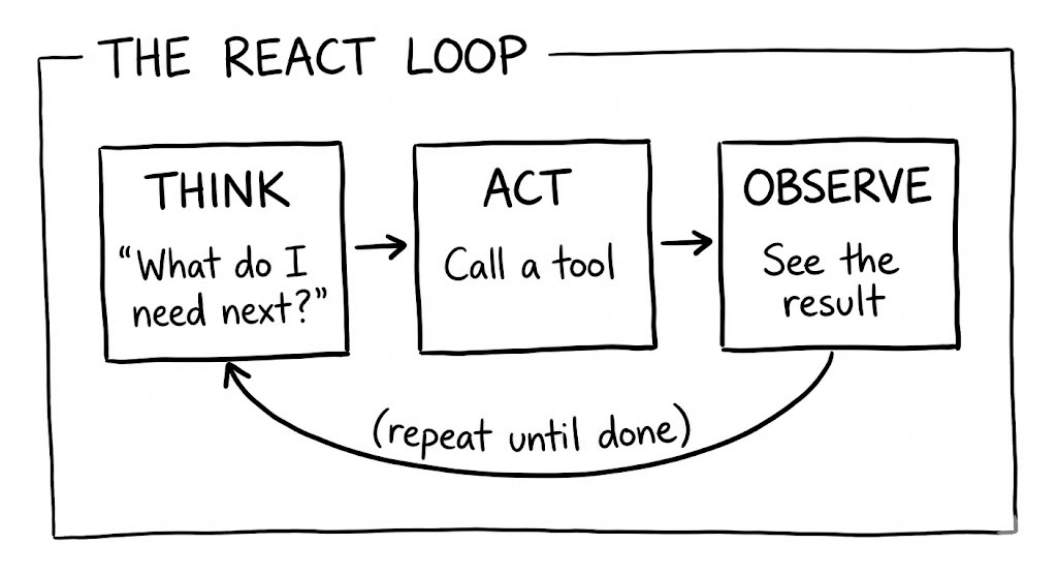

Figure 6.1: Round and round we go.

The loop:

- **Think:** The agent reasons about what to do next
- **Act:** It calls a tool to get information
- **Observe:** It sees the result

Then it loops back to **Think**: What do I do with this? Do I need more information? Is this enough to answer the question?

This continues until the agent has everything it needs.

6.2 Setting Up the Tools

Let's create three tools for stock analysis. Unlike Chapter 5's simulated data, these use yfinance to fetch live market data:

```
@tool
def get_stock_price(ticker: str) -> float:
    """Get the current price for a stock ticker.

    Args:
        ticker: The stock symbol (e.g., 'AAPL', 'NVDA', 'MSFT')
```

```
    Returns:
        The current stock price as a float
    """
    import yfinance as yf
    stock = yf.Ticker(ticker)
    return stock.info.get('regularMarketPrice', 0.0)

@tool
def get_52_week_high(ticker: str) -> float:
    """Get the 52-week high price for a stock ticker.

    Args:
        ticker: The stock symbol (e.g., 'AAPL', 'NVDA', 'MSFT')

    Returns:
        The 52-week high price as a float
    """
    import yfinance as yf
    stock = yf.Ticker(ticker)
    return stock.info.get('fiftyTwoWeekHigh', 0.0)

@tool
def get_52_week_low(ticker: str) -> float:
    """Get the 52-week low price for a stock ticker.

    Args:
        ticker: The stock symbol (e.g., 'AAPL', 'NVDA', 'MSFT')

    Returns:
        The 52-week low price as a float
    """
    import yfinance as yf
    stock = yf.Ticker(ticker)
    return stock.info.get('fiftyTwoWeekLow', 0.0)
```

Three focused tools. Each does one thing well. Same structure as before — same docstrings, same type hints — but now backed by real market data. The agent will combine them.

6.3 The Multi-Step Query

Now let's ask that complex question and watch the ReAct loop in action:

```
agent = CodeAgent(
    tools=[get_stock_price, get_52_week_high, get_52_week_low],
```

```
    model=model,
    verbosity_level=LogLevel.INFO  # So we can see the reasoning
)

result = agent.run("""
Compare Apple and NVIDIA:
- What's each stock's current price?
- What's each stock's 52-week high?
- Which stock is trading closer to its 52-week high (as a percentage)?
""")
```

What happens (with verbose output). Note: LLM outputs are non-deterministic — your results may differ in wording, step order, and actual values (these are live market prices). The pattern will be the same:

```
========================= Step 1 =========================

THINK: I need to get Apple's current price first.

ACT: get_stock_price(ticker="AAPL")

OBSERVE: 178.5

========================= Step 2 =========================

THINK: Got Apple's price. Now I need Apple's 52-week high.

ACT: get_52_week_high(ticker="AAPL")

OBSERVE: 199.62

========================= Step 3 =========================

THINK: I have Apple's data. Now I need NVIDIA's current price.

ACT: get_stock_price(ticker="NVDA")

OBSERVE: 875.3

========================= Step 4 =========================

THINK: Got NVIDIA's price. Now I need NVIDIA's 52-week high.

ACT: get_52_week_high(ticker="NVDA")

OBSERVE: 974.0
```

```
========================= Step 5 =========================

THINK: I have all the data. Let me calculate the percentages.

   Apple:  178.50 / 199.62 = 89.4% of 52-week high
   NVIDIA: 875.30 / 974.00 = 89.9% of 52-week high

   NVIDIA is trading closer to its 52-week high.

FINAL ANSWER:

Apple is currently at $178.50, which is 89.4% of its 52-week high ($199.62).
NVIDIA is at $875.30, which is 89.9% of its 52-week high ($974.00).

NVIDIA is trading closer to its 52-week high.
============================================================
```

Five steps. Four tool calls. One synthesized answer.

The agent isn't following a script. It's reasoning through the problem dynamically — deciding at each step what information it still needs.

6.4 Anatomy of the ReAct Trace

Let's break down what happened:

```
Step 1: THINK   -> "I need Apple's price"
        ACT     -> get_stock_price("AAPL")
        OBSERVE -> 178.5

Step 2: THINK   -> "Now I need Apple's 52-week high"
        ACT     -> get_52_week_high("AAPL")
        OBSERVE -> 199.62

Step 3: THINK   -> "Got Apple. Now NVIDIA's price"
        ACT     -> get_stock_price("NVDA")
        OBSERVE -> 875.3

Step 4: THINK   -> "Now NVIDIA's 52-week high"
        ACT     -> get_52_week_high("NVDA")
        OBSERVE -> 974.0

Step 5: THINK      -> "I have everything. Time to calculate and compare."
        SYNTHESIZE -> Calculate percentages, compare, explain
        FINAL      -> "NVIDIA is trading closer..."
```

Notice the pattern:

- Each THINK step decides what to do next based on what's already known
- The agent gathers data systematically (Apple first, then NVIDIA)
- Only when all data is collected does it move to synthesis
- The final answer addresses the actual question (which is closer?)

6.5 Why ReAct Matters

Three reasons this pattern is essential for finance:

6.5.1 1. Complex Questions Become Possible

Without ReAct, you'd have to hardcode the sequence: "first call this, then that, then combine them." With ReAct, the agent figures out the steps itself.

Ask a different question — "Compare AAPL, NVDA, and MSFT" — and the agent adapts. It adds more steps. It doesn't break.

6.5.2 2. Transparency and Auditability

You can see exactly why the agent did what it did. Every step is logged:

- What it thought
- What tool it called
- What result it got
- How it reasoned about the result

In finance, this matters. You can't use a black box for investment decisions. ReAct provides that. It's not explainability. But it helps.

6.5.3 3. Graceful Handling of Edge Cases

What happens when data is missing? The agent reasons about it:

```
result = agent.run("What's the 52-week range for PESLA? Not TSLA.")
```

Output:

```
Step 1: get_52_week_high("PESLA") -> 0.0
        get_52_week_low("PESLA") -> 0.0

Step 2: THINK: Both values are 0.0. This indicates the ticker
        wasn't found by yfinance.

FINAL ANSWER: No valid data available for PESLA 52-week range.
```

The agent didn't just return "0 to 0." It recognized that zero means missing data and said so explicitly. That's the reasoning component at work.

6.6 A More Complex Example

Let's ask a question that requires reasoning about an investment decision:

```
result = agent.run("""
I'm thinking about buying one of these stocks.
Which one is furthest from its 52-week high,
suggesting more potential upside?
Compare AAPL, NVDA, and MSFT.
""")
```

Output (yours may vary — the agent might order steps differently, phrase things differently, or return different prices since these are live. The reasoning pattern is what matters):

```
============================================================

[Agent gathers price and 52-week high for all three stocks...]

Step 7: THINK: Let me calculate how far each is from its high:

   AAPL: $178.50 vs $199.62 high -> 89.4% (10.6% below high)
   NVDA: $875.30 vs $974.00 high -> 89.9% (10.1% below high)
   MSFT: $378.90 vs $420.82 high -> 90.0% (10.0% below high)

   AAPL is furthest from its 52-week high at 10.6% below.

FINAL ANSWER:

Based on distance from 52-week highs:
- Apple (AAPL) is 10.6% below its high
- NVIDIA (NVDA) is 10.1% below its high
```

```
- Microsoft (MSFT) is 10.0% below its high

Apple is furthest from its 52-week high, which could suggest
more potential upside if it returns to previous highs.

However, note that this is a simplified analysis. Other factors
like fundamentals, growth prospects, and market conditions should
also be considered.
============================================================
```

The agent doesn't just return numbers. It answers the **actual question**: which has more potential upside? And it adds appropriate caveats — this is a simplified analysis.

That's the **reasoning** part of ReAct.

6.7 ReAct vs. Simple Tool Calling

Here's the difference:

Aspect	Tool Calling (Ch. 5)	ReAct (Ch. 6)
Steps	Single call	Multiple calls
Reasoning	Not Needed	Explicit, visible
Adaptation	Fixed	Dynamic
Questions	Simple lookups	Complex analysis

Tool calling answers: "What's AAPL's price?"

ReAct answers: "Compare three stocks and tell me which to buy."

Both are useful. The Complexity Ladder from Chapter 2 tells you when to use which.

6.8 ReAct Principles

6.8.1 Principle 1: Ask Complete Questions

Don't just say "AAPL price." Say "What is Apple's current price and how does it compare to its 52-week high?"

The more context you provide, the better the agent can reason about what tools to call and how to synthesize the results.

6.8.2 Principle 2: Let the Agent Think

Don't try to micromanage the steps. The agent often finds better approaches than you'd hardcode. Trust the reasoning loop.

If the reasoning looks wrong, you can adjust the prompt or add clarifying instructions. But let it try first.

6.8.3 Principle 3: Watch the Trace

The verbose output isn't just for debugging. It's how you **understand and trust** your agent.

If the reasoning looks wrong, the answer probably is too. Catching flawed reasoning early is easier than catching a wrong answer at the end.

6.9 Memory Across ReAct Steps

ReAct works naturally with memory. After the agent completes a multi-step analysis, you can ask follow-up questions:

```
# First query: full analysis
result = agent.run("""
Compare AAPL and NVDA: which is closer to its 52-week high?
""")

# Follow-up: build on previous analysis
result = agent.run("""
Now add MSFT to that comparison.
""", reset=False)  # Preserve memory!
```

The agent remembers the previous analysis and extends it rather than starting over.

6.10 Finance-Specific ReAct Patterns

Common multi-step patterns in finance:

Pattern	Steps
Stock Comparison	Price → Fundamentals → Compare → Recommend
Valuation Analysis	P/E → P/B → EPS → Sector average → Verdict
Risk Assessment	Price → Returns → Volatility → VaR → Report
Portfolio Review	Positions → Concentration → Correlation → Risks

Each of these requires multiple tool calls and synthesis. ReAct handles them naturally.

6.11 Exercise: Build Your Own Multi-Step Query

Write a query that requires:

1. Getting data for at least 2 stocks
2. Some kind of comparison or calculation
3. A recommendation or conclusion

Ideas:

- "Which stock has the widest 52-week range (high minus low)?"
- "If I had $10,000 to invest, how many shares of each could I buy?"
- "Which stock is most volatile based on its 52-week range as a percentage of its low?"

Run it with verbose mode and study the trace. Can you predict which tools the agent will call?

6.12 Tracking Steps Programmatically

You can also access the agent's memory to see all steps:

```
# Run a query
result = agent.run("Compare the 52-week ranges of AAPL and MSFT")

# Access memory to see all steps
print("AGENT MEMORY:")
for i, step in enumerate(agent.memory.steps):
    step_type = type(step).__name__
    print(f"\nStep {i+1}: {step_type}")

    if hasattr(step, 'tool_calls') and step.tool_calls:
        for tc in step.tool_calls:
            print(f"   Tool: {tc.name}({tc.arguments})")
```

This programmatic access is useful for:

- Building audit logs
- Debugging unexpected behavior
- Extracting tool call sequences for analysis

6.13 The Hamburger Check

Where does ReAct fit in the hamburger?

```
TOP BUN (LLM)
    Understands "compare AAPL and NVDA"
    Reasons about what data to fetch
    Decides the sequence of tool calls

MEAT (Tools)
    get_stock_price -> actual prices
    get_52_week_high -> actual highs
    Pure computation, no guessing

BOTTOM BUN (LLM)
    Synthesizes: "NVIDIA is closer to its high"
    Explains the reasoning
    Adds appropriate caveats
```

The reasoning happens in the bun. The data comes from the meat. ReAct is the pattern that connects them across multiple steps.

Key Takeaways

ReAct (Reason + Act) is the pattern for multi-step problems. The agent alternates:

- **Think:** What do I need next?
- **Act:** Call a tool
- **Observe:** See the result
- Repeat until done

This enables complex questions that require gathering data from multiple sources and synthesizing insights. Every step is visible — making the reasoning auditable and trustworthy.

Tool calling handles "What's AAPL's price?"

ReAct handles "Compare three stocks and tell me which to buy."

7

The CodeAct Pattern - Dynamic Computation

In Chapter 5, we gave the agent tools. In Chapter 6, we watched it reason through multi-step problems.

But here's a question those tools can't answer:

> "Calculate the Sharpe ratio for a portfolio with these monthly returns: 5%, 3%, -2%, 4%, 1%, -1%, 3%, 2%"

We don't have a `calculate_sharpe_ratio` tool. We could build one, but what about tomorrow's question?

What if someone asks for the **Sortino ratio**? Or **Value at Risk**? Or the **maximum drawdown**? Or a **custom metric** you've never heard of?

You can't pre-build tools for every possible calculation.

This is where **CodeAct** comes in. (I am not sure if CodeAct is an actual term, but I thought it makes sense.)

Now I am not saying that we use this to save us the step of building tools. We should always prefer to build tools. With pre-defined tools, we can check it's correct, add checks and verifications in the tool. With CodeAct, you can't.

7.1 What is CodeAct?

CodeAct is a pattern where the agent doesn't just call pre-built tools - it **writes and executes Python code on the fly**.

The agent:

1. Reasons about what code to write
2. Writes the Python code
3. The system executes it
4. The agent sees the result and can continue reasoning

The agent isn't limited to pre-built tools. It can compute anything Python can compute.

Why is executing code better than having the LLM reason in language? Because language-based math is unreliable - an LLM might get a Sharpe ratio calculation approximately right, or it might silently drop a step. Code either runs correctly or throws an error. There's no "close enough" in Python. `2 + 2` is always `4`, never "approximately 4." That's the difference between computation and generation.

7.2 Setting Up CodeAct

The key difference: we're not passing tools. We're enabling code execution:

```
from smolagents import CodeAgent
from smolagents import OpenAIServerModel
from smolagents.monitoring import LogLevel

model = OpenAIServerModel("gpt-4o-mini", api_key=API_KEY)

# CodeAgent with NO pre-built tools--only code generation
agent = CodeAgent(
    tools=[],  # No tools!
    model=model,
    verbosity_level=LogLevel.INFO,
    additional_authorized_imports=["numpy", "pandas"]  # Allow these imports
)

print("CodeAgent ready with NO tools -- only code generation!")
```

Notice `additional_authorized_imports`. This is how you control what the agent can import. More on security later.

7.3 The Sharpe Ratio Example

Let's ask that Sharpe ratio question and watch the agent work:

```
result = agent.run("""
Calculate the Sharpe ratio for a portfolio with these monthly returns:
5%, 3%, -2%, 4%, 1%, -1%, 3%, 2%

Assume a risk-free rate of 0.5% per month.
Show your calculation steps.
""")
```

What happens (LLM outputs are non-deterministic - your results may differ in wording or code structure, but the pattern will be the same):

```
========================= Step 1 =========================

THINK: I need to calculate the Sharpe ratio. The formula is:
   Sharpe = (Mean Return - Risk-Free Rate) / Standard Deviation

   Let me write Python code to calculate this.

WRITING CODE:

import numpy as np

# Monthly returns as decimals
returns = np.array([0.05, 0.03, -0.02, 0.04, 0.01, -0.01, 0.03, 0.02])
risk_free_rate = 0.005  # 0.5% per month

# Calculate excess returns
excess_returns = returns - risk_free_rate

# Mean excess return
mean_excess = np.mean(excess_returns)

# Standard deviation of returns
volatility = np.std(returns, ddof=1)  # ddof=1 for sample std

# Sharpe ratio (monthly)
sharpe_monthly = mean_excess / volatility

# Annualized Sharpe (multiply by sqrt(12) for monthly data)
sharpe_annualized = sharpe_monthly * np.sqrt(12)

print(f"Mean monthly return: {np.mean(returns)*100:.2f}%")
```

```
print(f"Mean excess return: {mean_excess*100:.2f}%")
print(f"Monthly volatility: {volatility*100:.2f}%")
print(f"Monthly Sharpe ratio: {sharpe_monthly:.3f}")
print(f"Annualized Sharpe ratio: {sharpe_annualized:.2f}")

EXECUTING...

OUTPUT:
Mean monthly return: 1.88%
Mean excess return: 1.38%
Monthly volatility: 2.36%
Monthly Sharpe ratio: 0.583
Annualized Sharpe ratio: 2.02

======================== Step 2 ========================

FINAL ANSWER:

The Sharpe ratio for this portfolio is:
- Monthly: 0.58
- Annualized: 2.02

This is an excellent risk-adjusted return. A Sharpe ratio above 1.0 is
generally considered good, and above 2.0 is exceptional.

Calculation breakdown:
- Average monthly return: 1.88%
- Risk-free rate: 0.50%
- Excess return: 1.38%
- Monthly volatility: 2.36%
- Sharpe = 1.38% / 2.36% = 0.58 (monthly)
============================================================
```

What just happened:

- The agent **knew the Sharpe ratio formula** (from its training)
- It **wrote Python code** to implement it
- The code **executed** and produced real numbers
- The agent **interpreted the results** and explained what they mean

We didn't build a Sharpe ratio tool. The agent knew the formula and implemented it on the spot.

7.4 Why This is Powerful

Think about what just happened:

- **The LLM brought knowledge** - it knew the Sharpe ratio formula
- **Python brought computation** - it did the actual math
- **CodeAct combined them** - knowledge + execution

This is why CodeAct is so useful for finance. Financial analysis often involves **custom calculations**. You can't pre-build everything. But the agent can figure it out.

7.5 More Examples

Let's try a few more to see the range of what's possible.

7.5.1 Maximum Drawdown

```
result = agent.run("""
Calculate the maximum drawdown for this sequence of portfolio values:
$100,000 -> $105,000 -> $98,000 -> $102,000 -> $95,000 -> $110,000

Explain what maximum drawdown means and show the calculation.
""")
```

Agent-generated code (yours may differ - the agent writes code dynamically each time):

```
import numpy as np

values = np.array([100000, 105000, 98000, 102000, 95000, 110000])

# Track the running maximum
running_max = np.maximum.accumulate(values)

# Drawdown at each point
drawdowns = (running_max - values) / running_max

# Maximum drawdown
max_drawdown = np.max(drawdowns)
max_dd_idx = np.argmax(drawdowns)

print(f"Maximum drawdown: {max_drawdown*100:.1f}%")
print(f"Occurred at value: ${values[max_dd_idx]:,}")
print(f"Peak before drawdown: ${running_max[max_dd_idx]:,}")
```

Output:

```
Maximum drawdown: 9.5%
Occurred at value: $95,000
Peak before drawdown: $105,000
```

The agent knew what maximum drawdown means, implemented the algorithm, and calculated it correctly.

7.5.2 Correlation Analysis

```
result = agent.run("""
Given these monthly returns for two stocks:

Stock A: 2%, -1%, 3%, 1%, -2%, 4%
Stock B: 1%, 0%, 2%, 2%, -1%, 3%

Calculate the correlation coefficient between them.
Explain whether they would provide good diversification.
""")
```

Agent-generated code:

```
import numpy as np

returns_a = np.array([0.02, -0.01, 0.03, 0.01, -0.02, 0.04])
returns_b = np.array([0.01, 0.00, 0.02, 0.02, -0.01, 0.03])

correlation = np.corrcoef(returns_a, returns_b)[0, 1]

print(f"Correlation coefficient: {correlation:.3f}")
```

Output and interpretation:

```
Correlation coefficient: 0.877

This is a high positive correlation (0.88). These two stocks tend to
move together--when one goes up, the other usually does too.

For diversification, you'd want stocks with LOW or NEGATIVE correlation.
A correlation of 0.88 means these stocks don't provide much diversification
benefit. Consider adding assets from different sectors or asset classes.
```

The agent didn't just calculate - it interpreted the result in context.

7.6 Combining Tools AND Code

Here's the really powerful thing: you can combine pre-built tools AND code generation.

```
@tool
def get_historical_returns(ticker: str) -> list:
    """Get historical monthly returns for a stock (last 12 months).

    Args:
        ticker: The stock symbol (e.g., 'AAPL', 'NVDA', 'MSFT')

    Returns:
        List of monthly returns as decimals
    """
    import yfinance as yf
    stock = yf.Ticker(ticker)
    hist = stock.history(period="1y", interval="1mo")
    returns = hist['Close'].pct_change().dropna().tolist()
    return returns
```

Now create an agent with **both** a tool and code execution:

```
agent_combined = CodeAgent(
    tools=[get_historical_returns],  # Tool for fetching data
    model=model,
    verbosity_level=LogLevel.INFO,
    additional_authorized_imports=["numpy", "pandas"]  # Code for analysis
)
```

```
result = agent_combined.run("""
Compare the Sharpe ratios of AAPL, NVDA, and MSFT.
Use a risk-free rate of 0.3% per month.
Rank them from best to worst risk-adjusted return.
""")
```

What happens (values reflect live market data - yours will differ):

```
========================= Step 1 =========================

TOOL CALL: get_historical_returns("AAPL")
RESULT: [0.02, 0.05, -0.03, 0.04, ...]

========================= Step 2 =========================

TOOL CALL: get_historical_returns("NVDA")
RESULT: [0.08, 0.12, -0.05, 0.15, ...]

========================= Step 3 =========================

TOOL CALL: get_historical_returns("MSFT")
RESULT: [0.03, 0.02, 0.01, 0.04, ...]

========================= Step 4 =========================

WRITING CODE:

import numpy as np

def calculate_sharpe(returns, risk_free_rate=0.003):
    returns = np.array(returns)
    excess = returns - risk_free_rate
    return np.mean(excess) / np.std(returns, ddof=1)

aapl_returns = [0.02, 0.05, -0.03, 0.04, 0.01, 0.03,
                -0.02, 0.06, 0.02, -0.01, 0.04, 0.03]
nvda_returns = [0.08, 0.12, -0.05, 0.15, -0.03, 0.10,
                0.07, -0.02, 0.09, 0.04, 0.11, 0.06]
msft_returns = [0.03, 0.02, 0.01, 0.04, -0.01, 0.02,
                0.03, 0.01, 0.02, 0.03, 0.01, 0.02]

sharpes = {
    'AAPL': calculate_sharpe(aapl_returns),
    'NVDA': calculate_sharpe(nvda_returns),
    'MSFT': calculate_sharpe(msft_returns)
}

# Rank by Sharpe
ranked = sorted(sharpes.items(), key=lambda x: x[1], reverse=True)

for ticker, sharpe in ranked:
    print(f"{ticker}: {sharpe:.3f}")

EXECUTING...

OUTPUT:
```

```
NVDA: 0.891
AAPL: 0.543
MSFT: 0.451

FINAL ANSWER:

Ranked by Sharpe ratio (best to worst):
1. NVDA: 0.89 -- Excellent risk-adjusted returns
2. AAPL: 0.54 -- Good risk-adjusted returns
3. MSFT: 0.45 -- Acceptable risk-adjusted returns

NVIDIA has the highest Sharpe ratio despite its higher volatility,
because its returns are substantially higher.
================================================================
```

Best of both worlds:

- The agent used the **tool** to get historical data
- It wrote **code** to calculate Sharpe ratios
- It **compared** and ranked them
- It **interpreted** what the numbers mean

This is a pattern I keep coming back to: **tools for data, code for analysis**. It's not the only way - you can pre-build tools for common calculations too - but it scales well when you're dealing with ad-hoc questions that you can't anticipate in advance.

7.7 Security Considerations

CodeAct is powerful. But let's be honest about something:

The agent is writing and executing code. That means it could potentially do things you don't expect.

7.7.1 The Whitelist Approach

Notice how we specified allowed imports:

```
agent = CodeAgent(
    tools=[],
    model=model,
    additional_authorized_imports=["numpy", "pandas"]
)
```

Only `numpy` and `pandas` can be imported. The agent can't import `os` to access the filesystem, `requests` to make network calls, or `subprocess` to run shell commands.

7.7.2 Production Safeguards

In a production environment, you'd want:

- **Sandboxed execution** - Code runs in an isolated environment
- **Resource limits** - Timeout after N seconds, limit memory usage
- **Restricted file access** - Control whether the agent can read or write files via the import whitelist
- **Restricted network access** - Control whether the agent can make HTTP requests by whitelisting (or not) libraries like `requests`
- **Logging** - Every code block is logged for audit
- **Review option** - Human approval before execution (for high-stakes scenarios)

We don't implement all of these in the learning notebooks - but you should in production.

7.8 CodeAct Principles

7.8.1 Principle 1: Be Specific About What You Want

The more context you give - formulas, constraints, output format - the better the generated code will be.

Bad: "Calculate Sharpe ratio"

Good: "Calculate the Sharpe ratio for these monthly returns: [5%, 3%, -2%]. Use a risk-free rate of 0.5% per month. Show the annualized Sharpe ratio."

7.8.2 Principle 2: Ask for Explanations

Add "show your work" or "explain the calculation" to your prompts. This makes the agent's reasoning visible and helps you verify correctness.

7.8.3 Principle 3: Combine Tools and Code

Use **tools for data fetching** (reliable, predictable) and **code for analysis** (flexible, custom).

This is the pattern that scales.

7.9 When to Use CodeAct vs. Pre-Built Tools

Scenario	Pre-Built Tool	CodeAct
Standard lookups (price, P/E)	Yes	
Common calculations (position value)	Yes	
Frequently used calculations	Yes	
Calculations requiring validation	Yes	
One-off calculations		Yes

Rule of thumb: If you'll use it often and want it tested and validated, build a tool. If it's custom or one-off, let CodeAct handle it.

7.10 Exercise: Dynamic Calculation

Ask the agent to perform a financial calculation that would require custom code.

Ideas:

- "Calculate the Sortino ratio (like Sharpe but only penalizes downside volatility)"
- "Calculate Value at Risk (VaR) at the 95% confidence level for these daily returns"
- "Calculate the Calmar ratio (annualized return / max drawdown)"
- "Calculate the information ratio relative to a benchmark"

Include the data in your prompt. Watch the agent write and execute the code.

7.11 The Hamburger Check

Where does CodeAct fit?

The LLM (the bun) brings what it's good at - knowing that the Sharpe ratio formula is $(R-R_f)/\sigma$, understanding that a ratio above 1.0 is generally considered good, explaining results in plain language. The generated Python code (the meat) brings what *it's* good at - executing that formula on real numbers without approximation or hallucination. CodeAct is the bridge between knowing and doing.

Key Takeaways

CodeAct lets agents write and execute Python code on the fly. This is extremely powerful for finance:

- **Sharpe ratio, Sortino, VaR** - calculated from formulas the LLM knows
- **Custom metrics** - anything you can express in Python

- **Flexible analysis** - no need to pre-build every tool

The power pattern: **tools fetch data, code analyzes it**.

Security matters: whitelist imports, sandbox execution, log everything. CodeAct is powerful precisely because it can do anything - which means you need guardrails in production.

8

Orchestration - Putting It All Together

You now have three patterns:

- **Tool Calling** - Fetch real data
- **ReAct** - Reason through multi-step problems
- **CodeAct** - Write and execute custom calculations

Real finance applications need all three. A portfolio optimization agent doesn't just call one tool - it fetches data, reasons about strategy, calculates metrics, and synthesizes recommendations.

This chapter shows you how they work together.

The 4th and 5th notebook are relevant here. Since `04_orchestration.ipynb` is just a gentle introduction to the full works, we will jump straight to `05_portfolio_optimization.ipynb` in this chapter.

8.1 The Portfolio Optimization Agent

We're building an agent that can:

- Fetch real historical price data
- Calculate efficient frontiers
- Find optimal portfolios (Max Sharpe, Min Volatility)

- Suggest actual share allocations
- Run what-if scenarios with memory

This brings together everything from the course:

Pattern	How We Use It
Tool Calling	Fetch prices, run optimizations
ReAct	Reason through optimization choices
CodeAct	Calculate custom metrics, generate reports
Memory	Remember context for follow-ups

8.2 The Tools

We'll build six focused tools, each doing one job well:

8.2.1 Tool 1: Fetch Historical Prices

```
@tool
def get_historical_prices(tickers: str, years: int = 2) -> str:
    """
    Fetch historical adjusted close prices for a list of stock tickers.

    Args:
        tickers: Comma-separated stock symbols (e.g., 'AAPL,MSFT,GOOGL')
        years: Number of years of historical data (default: 2)

    Returns:
        JSON string with price statistics and recent prices
    """
    import yfinance as yf
    import json
    from datetime import datetime, timedelta

    ticker_list = [t.strip().upper() for t in tickers.split(',')]
    end_date = datetime.now()
    start_date = end_date - timedelta(days=365*years)

    prices = yf.download(ticker_list, start=start_date,
        end=end_date)["Close"]

    # Calculate stats for each ticker
    stats = {}
    for ticker in prices.columns:
        returns = prices[ticker].pct_change().dropna()
```

```
        stats[ticker] = {
            "latest_price": round(prices[ticker].iloc[-1], 2),
            "annual_return": round(returns.mean() * 252 * 100, 2),
            "annual_volatility": round(returns.std() * np.sqrt(252) * 100,
                2)
        }

    return json.dumps({"tickers": ticker_list, "statistics": stats},
        indent=2)
```

This tool uses **yfinance** to fetch real market data. No simulated numbers - actual historical prices.

8.2.2 Tool 2: Optimize for Maximum Sharpe Ratio

```
@tool
def optimize_max_sharpe(tickers: str, risk_free_rate: float = 0.02) -> str:
    """
    Find the portfolio with the maximum Sharpe ratio (best risk-adjusted
        return).

    Args:
        tickers: Comma-separated stock symbols (e.g.,
            'AAPL,MSFT,GOOGL,AMZN')
        risk_free_rate: Annual risk-free rate as decimal (default: 0.02 =
            2%)

    Returns:
        JSON with optimal weights and expected performance
    """
    from pypfopt import EfficientFrontier, expected_returns, risk_models

    # Fetch prices
    prices = yf.download(ticker_list, start=start_date,
        end=end_date)["Close"]

    # Calculate expected returns and covariance
    mu = expected_returns.mean_historical_return(prices)
    S = risk_models.sample_cov(prices)

    # Optimize
    ef = EfficientFrontier(mu, S)
    weights = ef.max_sharpe(risk_free_rate=risk_free_rate)
    cleaned_weights = ef.clean_weights()

    # Get performance metrics
```

```
    expected_return, volatility, sharpe = ef.portfolio_performance()

    return json.dumps({
        "optimization": "Maximum Sharpe Ratio",
        "weights": {k: round(v*100, 2) for k, v in cleaned_weights.items()},
        "performance": {
            "expected_annual_return": round(expected_return*100, 2),
            "annual_volatility": round(volatility*100, 2),
            "sharpe_ratio": round(sharpe, 3)
        }
    }, indent=2)
```

This uses **PyPortfolioOpt** - a proper portfolio optimization library implementing Modern Portfolio Theory.

8.2.3 Tool 3: Optimize for Minimum Volatility

```
@tool
def optimize_min_volatility(tickers: str) -> str:
    """
    Find the portfolio with minimum volatility (lowest risk).
    Best for conservative investors who prioritize capital preservation.

    Args:
        tickers: Comma-separated stock symbols

    Returns:
        JSON with optimal weights and expected performance
    """
    # Similar to max_sharpe, but calls ef.min_volatility()
```

Notice the docstring: "Best for conservative investors who prioritize capital preservation." This tells the agent *when* to recommend this tool.

8.2.4 Tool 4: Optimize for Target Return

```
@tool
def optimize_target_return(tickers: str, target_return: float) -> str:
    """
    Find the minimum volatility portfolio that achieves a specific target
        return.

    Args:
```

```
        tickers: Comma-separated stock symbols
        target_return: Desired annual return as decimal (e.g., 0.15 = 15%)

    Returns:
        JSON with optimal weights and expected performance
    """
    # Uses ef.efficient_return(target_return=target_return)
```

8.2.5 Tool 5: Calculate Discrete Allocation

```
@tool
def calculate_allocation(tickers: str, total_investment: float,
                         optimization: str = "max_sharpe") -> str:
    """
    Calculate the exact number of shares to buy for a given investment
        amount.

    Args:
        tickers: Comma-separated stock symbols
        total_investment: Total dollar amount to invest
        optimization: Strategy - 'max_sharpe' or 'min_volatility'

    Returns:
        JSON with shares to buy for each stock and leftover cash
    """
    from pypfopt.discrete_allocation import DiscreteAllocation

    # ... optimization logic ...

    da = DiscreteAllocation(weights, latest_prices,
                            total_portfolio_value=total_investment)
    allocation, leftover = da.greedy_portfolio()

    return json.dumps({
        "allocation": {
            ticker: {"shares": shares, "price": price, "total": shares *
                price}
            for ticker, shares in allocation.items()
        },
        "leftover_cash": round(leftover, 2)
    }, indent=2)
```

This converts theoretical weights (60% AAPL, 40% NVDA) into actual share counts you can trade.

8.2.6 Tool 6: Compare Strategies

```
@tool
def compare_strategies(tickers: str) -> str:
    """
    Compare Max Sharpe vs Min Volatility strategies for the same assets.

    Args:
        tickers: Comma-separated stock symbols

    Returns:
        JSON comparing both strategies side-by-side
    """
    # Runs both optimizations, returns comparison
```

8.3 Creating the Agent

Now we combine all tools into one intelligent agent:

```
portfolio_optimizer = CodeAgent(
    tools=[
        get_historical_prices,
        optimize_max_sharpe,
        optimize_min_volatility,
        optimize_target_return,
        calculate_allocation,
        compare_strategies
    ],
    model=model,
    verbosity_level=LogLevel.INFO,
    max_steps=10,
    additional_authorized_imports=["numpy", "pandas", "json"]
)
```

Six tools. Code execution capability. Memory for conversations.

8.4 Orchestration in Action

Let's see the agent handle a complex request:

```
result = portfolio_optimizer.run("""
```

```
I want to build a tech-focused portfolio with these stocks:
AAPL, MSFT, GOOGL, NVDA, META

Find the optimal allocation using the Max Sharpe strategy.
Show me the expected performance metrics.
""")
```

What happens (values reflect live market data - yours will differ):

```
========================= Step 1 ========================

THINK: The user wants a tech portfolio optimized for Max Sharpe.
   I'll use the optimize_max_sharpe tool.

TOOL CALL: optimize_max_sharpe(tickers="AAPL,MSFT,GOOGL,NVDA,META")

RESULT:
{
  "optimization": "Maximum Sharpe Ratio",
  "weights": {
    "AAPL": 35.2,
    "MSFT": 0.0,
    "GOOGL": 28.4,
    "NVDA": 36.4,
    "META": 0.0
  },
  "performance": {
    "expected_annual_return": 42.3,
    "annual_volatility": 31.2,
    "sharpe_ratio": 1.29
  }
}

========================= Step 2 ========================

FINAL ANSWER:

Optimal Tech Portfolio (Max Sharpe Strategy)

Recommended Allocation:
- NVDA: 36.4%
- AAPL: 35.2%
- GOOGL: 28.4%
- MSFT: 0.0%
- META: 0.0%

Expected Performance:
```

```
- Annual Return: 42.3%
- Annual Volatility: 31.2%
- Sharpe Ratio: 1.29

Key Insights:
The optimizer excludes MSFT and META because their risk-adjusted
returns don't improve the portfolio. The high concentration in
NVDA reflects its strong recent performance, though this also
increases concentration risk.

A Sharpe ratio of 1.29 is good--you're getting meaningful return
per unit of risk.
============================================================
```

One tool call, full analysis. But watch what happens with a more complex query.

8.5 Multi-Pattern Orchestration

```
result = portfolio_optimizer.run("""
I have $50,000 to invest in AAPL, MSFT, GOOGL, NVDA, META.

1. Compare Max Sharpe vs Min Volatility strategies
2. Show me exactly how many shares to buy for each strategy
3. Which would you recommend for someone 5 years from retirement?
""")
```

What happens (values reflect live market data - yours will differ):

```
========================= Step 1 =========================

THINK: User wants to compare strategies. I'll use compare_strategies.

TOOL CALL: compare_strategies(tickers="AAPL,MSFT,GOOGL,NVDA,META")

RESULT:
{
  "comparison": {
    "max_sharpe": {
      "weights": {"AAPL": 35.2, "GOOGL": 28.4, "NVDA": 36.4, ...},
      "expected_return": 42.3,
      "volatility": 31.2,
      "sharpe_ratio": 1.29
    },
```

```
    "min_volatility": {
      "weights": {"AAPL": 22.1, "MSFT": 45.3, "GOOGL": 18.2, ...},
      "expected_return": 18.7,
      "volatility": 22.4,
      "sharpe_ratio": 0.74
    }
  }
}

========================= Step 2 =========================

THINK: Now I need discrete allocations for both strategies.

TOOL CALL: calculate_allocation(
    tickers="AAPL,MSFT,GOOGL,NVDA,META",
    total_investment=50000,
    optimization="max_sharpe"
)

RESULT:
{
  "allocation": {
    "AAPL": {"shares": 65, "price": 270.01, "total": 17550.65},
    "GOOGL": {"shares": 41, "price": 343.69, "total": 14091.29},
    "NVDA": {"shares": 15, "price": 1205.40, "total": 18081.00}
  },
  "leftover_cash": 277.06
}

========================= Step 3 =========================

TOOL CALL: calculate_allocation(
    tickers="AAPL,MSFT,GOOGL,NVDA,META",
    total_investment=50000,
    optimization="min_volatility"
)

RESULT:
{
  "allocation": {
    "AAPL": {"shares": 41, "price": 270.01, "total": 11070.41},
    "MSFT": {"shares": 53, "price": 423.37, "total": 22438.61},
    "GOOGL": {"shares": 26, "price": 343.69, "total": 8935.94},
    "NVDA": {"shares": 6, "price": 1205.40, "total": 7232.40}
  },
  "leftover_cash": 322.64
}
```

```
========================= Step 4 =========================

FINAL ANSWER:

Strategy Comparison for $50,000 Investment

Option 1: Max Sharpe (Higher Return, Higher Risk)
  Stock   Shares   Cost
  AAPL    65       $17,551
  GOOGL   41       $14,091
  NVDA    15       $18,081
  Leftover         $277

Expected Return: 42.3% | Volatility: 31.2% | Sharpe: 1.29

Option 2: Min Volatility (Lower Return, Lower Risk)
  Stock   Shares   Cost
  MSFT    53       $22,439
  AAPL    41       $11,070
  GOOGL   26       $8,936
  NVDA    6        $7,232
  Leftover         $323

Expected Return: 18.7% | Volatility: 22.4% | Sharpe: 0.74

Recommendation for 5 Years to Retirement:

I recommend the Min Volatility strategy. Here's why:

1. Capital preservation matters more -- With 5 years to retirement,
   you can't afford a 30%+ drawdown that might occur with the
   high-volatility portfolio.

2. 18.7% expected return is still strong -- This would nearly double
   your money in 5 years if it holds.

3. Lower volatility = better sleep -- 22% vs 31% volatility means
   smaller swings and less stress as you approach retirement.

The Max Sharpe strategy is better for someone with 20+ years to invest.
============================================================
```

Look at what happened:

- **Tool Calling**: Three distinct tool calls for comparison and allocation
- **ReAct**: Agent reasoned through what tools to call in what order

- **Synthesis**: Combined results into a coherent recommendation
- **Context awareness**: Considered the "5 years to retirement" factor

This is orchestration. The agent dynamically combined patterns to answer a complex question.

8.6 Memory and Follow-Up Questions

The agent remembers context. After the analysis above:

```
result = portfolio_optimizer.run("""
What if I removed NVDA and added more to the safer stocks?
""", reset=False)  # Preserve memory!
```

Output:

```
I understand you want to reconsider the allocation without NVDA.

Let me run a Min Volatility optimization with just AAPL, MSFT, GOOGL,
    META...

[runs optimization]

Without NVDA, the Min Volatility portfolio becomes:
- MSFT: 52%
- AAPL: 28%
- GOOGL: 20%

Expected return drops to 15.2%, but volatility also drops to 19.8%.

For your timeline, this might actually be more appropriate--less
concentration risk and lower overall volatility.
```

The agent remembered:

- The stocks under consideration
- The $50,000 investment
- The 5-year retirement timeline
- The recommendation context

No need to repeat everything. That's the power of memory.

8.7 The Complete Architecture

Here's how it all fits together:

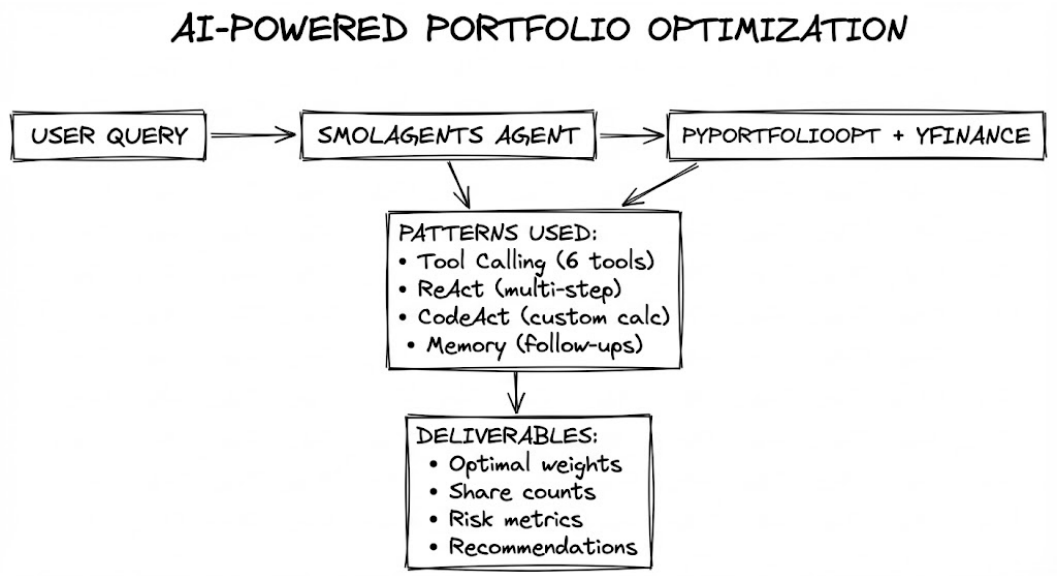

Figure 8.1: Not that complex, right?

8.8 The Hamburger in Full

Now we can see the complete hamburger:

```
TOP BUN (LLM)
    |-- Understands "optimize for retirement in 5 years"
    |-- Decides which tools to call
    |-- Reasons about investor context
    +-- Chooses appropriate strategy

VEGETABLES (*Some for a future book)
    |-- Input validation
    |-- Error handling
    |-- Rate limiting for APIs
    +-- Logging for audit

MEAT (Computation)
    |-- PyPortfolioOpt: Mean-variance optimization
    |-- yfinance: Real market data
    |-- DiscreteAllocation: Share calculations
    +-- Custom metrics via CodeAct

VEGETABLES (*Some for a future book)
    |-- JSON formatting
    |-- Memory management
    +-- Response caching

BOTTOM BUN (LLM)
    |-- Synthesizes results
    |-- Explains implications
    |-- Makes recommendations
    +-- Handles follow-ups
```

The LLM provides understanding and synthesis. The tools provide computation. Infrastructure holds it together. That's a complete hamburger.

8.9 What You've Built

By completing this chapter, you've built a portfolio optimization assistant that:

Fetches real data

- Historical prices from Yahoo Finance
- Current market data

Runs real optimizations

- Mean-variance optimization

- Efficient frontier calculations
- Sharpe ratio maximization
- Volatility minimization

Provides actionable outputs

- Exact share counts to buy
- Dollar allocations
- Leftover cash calculations

Reasons about context

- Investor timeline
- Risk tolerance
- Strategy trade-offs

Handles conversations

- Remembers previous analysis
- Supports what-if scenarios
- Builds on earlier context

8.10 From Demo to Production

This is a working demo. Production requires more:

Demo	Production
Trust tool outputs	Validate and sanity-check
Single user	Authentication and sessions
No persistence	Database for portfolios
Verbose output	Clean UI with optional detail
No guardrails	Approval for trades, limits on recommendations

We preview the production gap - guardrails, compliance, human-in-the-loop - in Chapter 9.

Key Takeaways

Orchestration is how the patterns work together:

- **Tool Calling** fetches data and runs optimizations
- **ReAct** reasons through which tools to call and in what order

- **CodeAct** handles custom calculations
- **Memory** enables follow-up questions

The portfolio optimization agent demonstrates all four patterns in a real application: taking natural language requests, gathering data, running optimizations, and providing actionable recommendations.

This is the capstone pattern. Once you understand orchestration, you can build any finance AI application by combining the right tools with the right patterns.

9

What's Next - The Production Gap

You've learned the four patterns. You've built a working portfolio optimization agent. You understand how Tool Calling, ReAct, CodeAct, and Orchestration fit together.

But to be honest, we've been working in a sandbox.

There's a gap between what we've built and what you'd deploy in production. This chapter previews that gap - what you'll need to think about when you're ready to build real systems.

Important: This chapter is a roadmap, not an implementation guide. We're showing you the terrain ahead, not building the roads. A future series will cover production deployment in depth.

9.1 The Gap: Demo vs. Production

What We Built	What Production Needs
You, in a notebook	Clients, advisors, traders
Learning environment	Compliance requirements
"Interesting error"	Financial liability
No consequences	Real money at stake

The patterns don't change. But everything around them does.

9.2 Preview: The Guardrail Framework

In production, you should classify every tool by risk level.

This book focused on LOW risk tools. Read-only data access and pure computation. Safe to experiment with.

Production systems need the full spectrum - including high-risk actions with human oversight.

9.3 Preview: Key Guardrail Layers

Production finance AI needs protection at every layer:

```
Layer 1: INPUT VALIDATION
         Block dangerous requests before they reach the agent

Layer 2: TOOL CLASSIFICATION
         Risk level determines approval requirements

Layer 3: EXECUTION CONTROLS
         Position limits, rate limits, circuit breakers

Layer 4: OUTPUT VALIDATION
         Disclaimers, PII scrubbing, compliance checks

Layer 5: AUDIT LOGGING
         Record everything for compliance and debugging
```

We haven't implemented any of these. In production, you need all of them. And evaluation and testing. And cybersecurity in place. And human oversight. And more. All depending on the risks.

9.4 What Production Requires (Preview)

Here's an illustrative (not exhaustive) look at the kinds of things that show up in a production checklist. Your specific needs will depend on your use case, regulatory environment, and risk appetite:

Infrastructure

- Authentication and authorization
- Session management and persistence

- Database integration for portfolios
- Secure API key management

Safety

- Input validation for dangerous requests
- Tool classification system
- Human approval workflows
- Position and trading limits
- Rate limiting

Compliance

- Comprehensive audit logging
- Required disclaimers
- PII handling
- Regulatory requirements (SEC, FINRA, etc.)

Operations

- Monitoring and alerting
- Error handling and recovery
- Incident response procedures
- Rollback capabilities

This is a significant engineering effort, depending on your needs.

9.5 Coming Next

We're planning a follow-up series that covers some of these. If you're interested, the best way to hear about it is to join the waitlist by signing up to my Substack at: simplyboring.ai.

9.6 For Now: What You Can Do

With what you've learned, you can build:

Safe internal tools

- Portfolio analysis dashboards

- Research assistants
- Risk calculation tools
- Comparison and screening tools

All of these use read-only data and computation - the LOW risk category we've mastered.

Proof of concepts

- Demonstrate the patterns to stakeholders
- Show what's possible with AI agents
- Build the case for production investment

Learning environments

- Train advisors on portfolio concepts
- Explain optimization strategies
- Interactive education tools

You don't need production guardrails for these use cases. The patterns you've learned are sufficient.

9.7 The Honest Assessment

Let me be clear about what we've covered and what we haven't:

What you now understand:

- How AI agents work (architecture, patterns)
- How to build reliable finance tools (less hallucination)
- How the patterns combine (orchestration)
- Why guardrails matter (conceptual framework)

What you're NOT ready for:

- Deploying agents that place real trades
- Client-facing systems with real money
- Meeting regulatory requirements
- Operating at production scale

That's okay. Most people jumping into "AI for finance" don't even understand the patterns. You do. The production layer is an engineering challenge, not a conceptual one.

Key Takeaways

There's a gap between demo and production. This book taught you the patterns - Tool Calling, ReAct, CodeAct, Orchestration. Production requires additional layers: input validation, tool classification, execution controls, output validation, and audit logging.

This book: Patterns and LOW-risk tools (read-only, computation)

Future series: HIGH-risk tools, guardrails, human-in-the-loop, compliance

For now, you can build safe internal tools, proof of concepts, and learning environments. Production deployment of trading agents requires the infrastructure we've previewed but not built.

The patterns are the foundation. The guardrails are next. And many of them actually have little to do with AI itself.

10

Quick Reference - Pattern Cheat Sheets

This chapter is designed to be printed, bookmarked, or kept open while you work. No explanations - just the essentials.

10.1 Pattern 1: Tool Calling

Note: All patterns use smolagents' `CodeAgent` - it's the main agent class that handles tool calling, ReAct reasoning, and code generation. The pattern name refers to *how* the agent solves the problem, not which class you use.

What it does: Gives the LLM access to functions that fetch real data or perform computations.

When to use: Simple lookups, single data points, one-step operations.

The pattern:

```
from smolagents import CodeAgent, tool

@tool
def get_stock_price(ticker: str) -> float:
    """
    Get current price for a stock ticker.

    Args:
```

```
        ticker: Stock symbol (e.g., 'AAPL')

    Returns:
        Current price as float
    """
    return fetch_from_api(ticker)

agent = CodeAgent(tools=[get_stock_price], model=model)
result = agent.run("What's Apple's stock price?")
```

Key elements:

- `@tool` decorator registers the function
- Type hints tell LLM what to pass/expect
- Docstring tells LLM when to use the tool

Common mistakes:

- Bad: Vague docstrings ("Get data")
- Bad: Missing type hints
- Bad: One tool doing multiple jobs
- Good: Clear "when to use" in docstring
- Good: One tool, one job

10.2 Pattern 2: ReAct

Origin: The ReAct pattern comes from Yao et al. (2022) - "ReAct: Synergizing Reasoning and Acting in Language Models." The core insight: interleaving reasoning traces with actions outperforms either reasoning or acting alone.

What it does: Agent reasons through multi-step problems with Think → Act → Observe loops.

When to use: Comparisons, multi-source analysis, questions requiring multiple data points.

The pattern:

```
agent = CodeAgent(
    tools=[get_stock_price, get_52_week_high, get_fundamentals],
    model=model,
    verbosity_level=LogLevel.INFO  # See the reasoning
```

```
)

result = agent.run("""
Compare AAPL and NVDA: which is trading closer to its 52-week high?
""")
```

The loop:

```
THINK   -> "I need AAPL's price"
ACT     -> get_stock_price("AAPL")
OBSERVE -> 178.50

THINK   -> "Now I need AAPL's 52-week high"
ACT     -> get_52_week_high("AAPL")
OBSERVE -> 199.62

... repeat until enough data ...

THINK   -> "Now I can compare and answer"
FINAL   -> "AAPL is at 89.4% of its high..."
```

Key elements:

- Multiple tools available
- Agent decides the sequence
- Each step is logged and visible

Common mistakes:

- Bad: Not using verbose mode to verify reasoning
- Good: Ask complete questions with context
- Good: Let the agent figure out the sequence

10.3 Pattern 3: CodeAct

What it does: Agent writes and executes Python code for custom calculations.

When to use: Custom metrics, calculations you can't pre-build, one-off analysis.

The pattern:

```
agent = CodeAgent(
```

```
    tools=[],  # No pre-built tools needed
    model=model,
    additional_authorized_imports=["numpy", "pandas"]
)

result = agent.run("""
Calculate the Sharpe ratio for these monthly returns:
5%, 3%, -2%, 4%, 1%, -1%, 3%, 2%
Risk-free rate: 0.5% per month.
""")
```

What happens:

```
# Agent writes this:
import numpy as np
returns = np.array([0.05, 0.03, -0.02, 0.04, 0.01, -0.01, 0.03, 0.02])
risk_free = 0.005
excess = returns - risk_free
sharpe = np.mean(excess) / np.std(returns, ddof=1)
# Result: 0.58
```

Key elements:

- `additional_authorized_imports` whitelists libraries
- Agent generates code on the fly
- Code executes and returns results

Common mistakes:

- Bad: Allowing unrestricted imports
- Bad: Vague calculation requests
- Good: Whitelist only needed libraries
- Good: Be specific about formulas and format

10.4 Pattern 4: Orchestration

What it does: Combines all patterns for complex, multi-stage tasks.

When to use: Full analysis workflows, tasks requiring data + reasoning + calculation.

The pattern:

```
agent = CodeAgent(
    tools=[
        get_historical_prices,
        optimize_max_sharpe,
        optimize_min_volatility,
        calculate_allocation,
        compare_strategies
    ],
    model=model,
    max_steps=10,
    additional_authorized_imports=["numpy", "pandas", "json"]
)

result = agent.run("""
I have $50,000 to invest in AAPL, MSFT, GOOGL, NVDA.
Compare Max Sharpe vs Min Volatility strategies.
Show me exactly how many shares to buy for each.
Which is better for someone 5 years from retirement?
""")
```

What happens:

1. **Tool Calling:** Fetches data, runs optimizations
2. **ReAct:** Reasons through which tools to call
3. **CodeAct:** Custom calculations if needed
4. **Memory:** Enables follow-up questions

Key elements:

- Multiple specialized tools
- Agent orchestrates the sequence
- Memory preserves context (`reset=False` for follow-ups)

Common mistakes:

- Bad: One mega-tool that does everything
- Bad: Too many tools (confusion)
- Good: Focused tools that compose well
- Good: 5–8 tools is usually the sweet spot

10.5 Quick Reference: Finance Tools

10.5.1 Market Data (Read-Only)

Tool	Purpose	Example
`get_stock_price`	Current price	"What's AAPL trading at?"
`get_company_info`	Sector, market cap	"Tell me about NVIDIA"
`get_historical_prices`	Price history	"Get 2 years of AAPL data"
`get_52_week_high`	Year high	"What's the 52-week high?"
`get_52_week_low`	Year low	"What's the 52-week low?"
`get_fundamentals`	P/E, P/B, EPS	"What's AAPL's P/E ratio?"

10.5.2 Calculation Tools

Tool	Purpose	Example
`optimize_max_sharpe`	Best risk-adjusted	"Optimize for Sharpe"
`optimize_min_volatility`	Lowest risk	"Minimize volatility"
`optimize_target_return`	Hit specific return	"Target 15% return"
`calculate_allocation`	Shares to buy	"How many shares for $50k?"
`compare_strategies`	Side-by-side	"Compare strategies"

10.5.3 CodeAct Calculations

Metric	Formula	Use Case
Sharpe Ratio	$(R - R_f)/\sigma$	Risk-adjusted return
Sortino Ratio	$(R - R_f)/\sigma_{\text{down}}$	Downside-only risk
Max Drawdown	(Peak – Trough) / Peak	Worst decline
VaR (95%)	5th percentile of returns	Potential loss
Correlation	`np.corrcoef(a, b)`	Diversification

10.6 Common Code Patterns

10.6.1 Initialize Agent

```
from smolagents import CodeAgent, tool
from smolagents import OpenAIServerModel
from smolagents.monitoring import LogLevel

model = OpenAIServerModel("gpt-4o-mini", api_key=API_KEY)
```

```
agent = CodeAgent(
    tools=[tool1, tool2, tool3],
    model=model,
    verbosity_level=LogLevel.INFO,
    additional_authorized_imports=["numpy", "pandas"]
)
```

10.6.2 Basic Tool Template

```
@tool
def tool_name(param: str) -> ReturnType:
    """
    One-line description.

    When to use: [specific trigger conditions]

    Args:
        param: Description of parameter

    Returns:
        Description of return value
    """
    # Implementation
    return result
```

10.6.3 Follow-Up Questions

```
# First query
result = agent.run("Analyze AAPL")

# Follow-up with memory
result = agent.run("Now compare to NVDA", reset=False)
```

10.6.4 Verbose Mode

```
agent = CodeAgent(
    tools=[...],
    model=model,
    verbosity_level=LogLevel.INFO  # See Think/Act/Observe
```

```
)
```

10.7 The Hamburger Mental Model

```
TOP BUN: LLM understands, chooses tools, reasons
VEGETABLES: Validation, error handling, logging
MEAT: Tools, calculations, real data
VEGETABLES: Formatting, caching, audit
BOTTOM BUN: LLM synthesizes, explains, responds
```

Remember: LLMs are buns, not meat. They are best used for understanding and communication. Wherever possible, tools handle computation and data.

10.8 Pattern Progression

```
PATTERN 1: TOOL CALLING
           "What's AAPL's price?"
           One tool, one answer

PATTERN 2: REACT
           "Compare AAPL and NVDA valuations"
           Multiple tools, reasoning chain

PATTERN 3: CODEACT
           "Calculate Sharpe ratio for this portfolio"
           Custom computation

PATTERN 4: ORCHESTRATION
           "Analyze my portfolio and recommend changes"
           All patterns combined
```

Start with Tool Calling. Add patterns only when the task demands it. (See Chapter 2's Complexity Ladder for when you need an agent at all vs. a simpler workflow.)

11

What I Hope You Take Away

I wrote this book to make sense of something that had been bothering me.

Is there a way to at least make this less fragile?

11.1 The Pattern

The answer wasn't smarter models. Which won't hurt, of course. It wasn't bigger training datasets. Again won't hurt. But do you have a lot of spare cash lying around to train models with your own data?

Don't ask the LLM to know things. Just ask it to do things. And equip it with the tools to do the things. Tools you know work.

When you give an LLM a tool that fetches real prices, it stops making them up. When you give it a function that calculates Sharpe ratios, it stops approximating them. When you let it write code and execute it, you can verify the math.

The hamburger metaphor came from trying to explain this to someone who kept dumping everything into LLMs. "They're not the meat," I said. "They're the buns. Fluffy, essential, but use them for what they are good at."

It stuck. And it turned out to be useful beyond that conversation.

LLMs are buns. They handle understanding and communication beautifully. They parse "build me a low-volatility portfolio for retirement" and figure out what that means. They take raw data and synthesize it into explanations you can actually read.

But the meat, the actual data, the real calculations, the verifiable computation, my advice

is that if you have a tool that is less uncertain than an LLM, use it.

11.2 Four Patterns, One Principle

The four patterns in this book are really just variations on that one principle.

Tool Calling says: give the LLM access to functions that return real data.

ReAct says: let the LLM reason through multi-step problems by calling those tools in sequence.

CodeAct says: when you can't pre-build every tool, let the LLM write the computation as a tool itself.

Orchestration says: combine all of the above for complex, real-world tasks.

Different names. Same fundamentals. The LLM handles the understanding. Something else handles the truth.

11.3 What This Book Is (And Isn't)

Let me be honest about the scope of this book.

This book teaches patterns. It gives you working code. It explains why things work the way they do.

It does not give you a production system. The gap between "working notebook" and "deployed application" is a mile wide. Authentication, persistence, guardrails, compliance, monitoring - we previewed these in Chapter 9, but we didn't build them. That's a different book. Maybe a future one.

What I hope you have now is an understanding of how it works under the hood.

You know why agents work when they work. You know why they fail when they fail. You know the difference between asking an LLM to guess and asking it to orchestrate.

That understanding is portable. smolagents might not be the framework you use in production. The specific tools will change. But the patterns - Tool Calling, ReAct, CodeAct, Orchestration - those are transferable.

11.4 The Honest Assessment

If you've followed along and run the notebooks, here's where you are:

You understand:

- How AI agents actually work (not magic, just architecture)

- Why LLMs hallucinate and how tools fix it
- Four patterns that combine in different ways for different problems
- Why guardrails matter, even if you haven't built them yet

You can build:

- Internal tools that fetch real data and calculate real metrics
- Proof of concepts that demonstrate the patterns
- Prototypes that show what's possible

You're not ready to:

- Deploy agents that trade real money
- Build client-facing systems without additional infrastructure
- Meet regulatory requirements on your own

That's all. But that's fine. Most people jumping into "AI agents for finance" don't even understand the patterns. You do. And it's a good start.

11.5 Where You Go From Here

I don't know what you'll build. That's yours to figure out.

But I know what I hope you'll avoid: the temptation to treat LLMs as do-it-alls.

They're not. They're powerful pattern matchers with a gift for language. When you give them tools, they become something more useful - orchestrators that can coordinate real work. When you don't, they're just very confident guessers if you ask them something they don't know.

The difference is entirely in how you use them.

11.6 Thank You

If you made it this far, you've done the work.

You didn't just read about agents. You ran the notebooks. You saw the traces. You built tools and watched the LLM use them.

That's the only way this stuff actually makes sense. Reading about it is one thing. Seeing it work - and seeing it fail, and understanding why - that's something else.

So thank you. For your attention. For running the code. For caring enough about getting this right that you worked through a book about it.

Now go build something.

11.7 Post-Credits

The book does not end here. I included two more chapters after this for readers who are interested in the core financial concepts we used as tools earlier. As well as how to translate these concepts to n8n, the no-code automation platform with AI agent nodes.

Read on if interested.

Key Takeaways

LLMs hallucinate when you ask them to know things. They work when you ask them to do things.
Four patterns give you the architecture: Tool Calling, ReAct, CodeAct, Orchestration.
One mental model keeps it straight: LLMs are buns, not meat.
The rest is engineering. Important engineering. But engineering you can learn, now that you understand what you're building.

End of Book

12

The n8n Path - Visual Agent Workflows

Everything you learned in this book works in code. But not everyone wants to write code.

n8n is a workflow automation platform with built-in AI agent capabilities. If you prefer visual tools, drag-and-drop interfaces, or need to build agents for teams that don't code - this chapter is for you.

The four patterns are identical. Only the interface changes.

12.1 Why n8n?

Three reasons to choose n8n over Python:

1. Visual Debugging

When something goes wrong, you see exactly which node failed. The execution trace shows every input, every output, every decision. For compliance teams reviewing agent behavior, this visibility is invaluable.

2. Team Accessibility

Your colleague can review the workflow. Your PM can suggest changes. Your analyst can modify tools. Everyone sees the same visual representation - no code reading required.

3. Easy Deployment

n8n workflows deploy to n8n Cloud or self-hosted infrastructure. No DevOps required for basic deployments. The same workflow that works in development works in production.

12.2 What You'll Need

Before importing the workflows, set up:

n8n Account

- n8n Cloud (free tier available): https://n8n.io
- Or self-hosted (Docker, npm)

API Keys

- OpenAI API key (for the AI Agent node)
- Financial Modeling Prep (FMP) API key (free tier: 250 calls/day): https://financialmodelingprep.com

The Workflow Files

- `01 - Tool Calling.json`
- `02 - REACT.json`
- `03 - Agent + Code Tools.json`
- `04 - Portfolio Optimization: Full Orchestration Agent.json`

These are included in the course materials.

12.3 Pattern 1: Tool Calling

Workflow: `01 - Tool Calling.json`

What it does: Basic agent with a single HTTP tool to fetch company profiles.

12.3.1 Workflow Structure

12.3.2 Key Nodes

Chat Trigger

- Receives user messages
- Provides `chatInput` to downstream nodes

Edit Fields

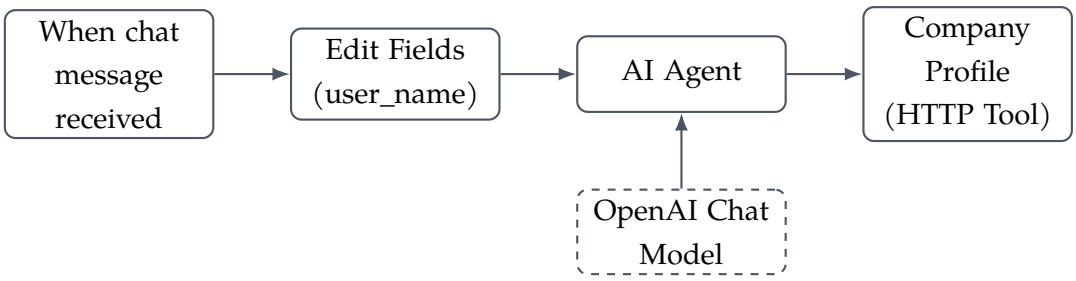

Figure 12.1: Workflow Structure

- Adds `user_name` variable (set to "Gary" by default)
- Demonstrates passing context to the agent

AI Agent

- Prompt: `"I am {{ $json.user_name }}. Help me with this query {{ $json.chatInput }}"`
- System message: "Address user by name when responding."

Company Profile (HTTP Tool)

- URL: `https://financialmodelingprep.com/api/v3/profile/{{ $fromAI('ticker') }}`
- Tool description: "FMP"
- The `$fromAI('ticker')` syntax lets the agent pass the ticker dynamically

12.3.3 Test Query

"Tell me about Apple"

The agent will call the Company Profile tool with ticker "AAPL" and return company information.

12.4 Pattern 2: ReAct

Workflow: `02 - REACT.json`

What it does: Multi-step reasoning with explicit Think → Act → Observe loop.

12.4.1 Workflow Structure

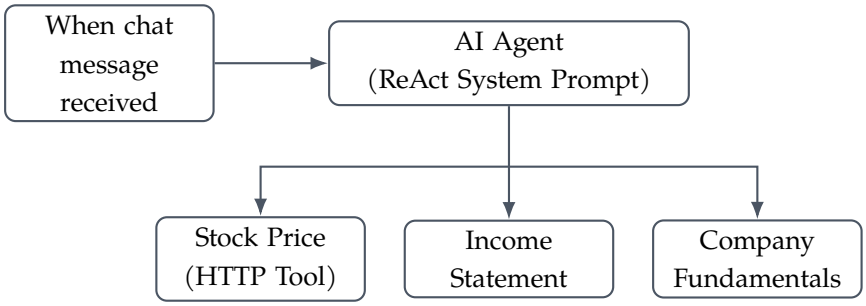

Figure 12.2: ReAct Agent Workflow

12.4.2 The ReAct System Prompt

This is what makes it ReAct - the system prompt explicitly teaches the pattern:

```
You are a financial analyst using the ReAct pattern.
For every question, you must:

THINK: What do I need to know?
ACT: Call a tool.
OBSERVE: Analyze the result.
LOOP: Do I need more info? If yes, repeat.
CONCLUDE: Answer the user.

Always gather data for ALL companies mentioned before answering.
```

12.4.3 Tools

Tool	API Endpoint	Purpose
Stock Price	`/api/v3/quote/{ticker}`	Current price, change, volume
Income Statement	`/api/v3/income-statement/{ticker}`	Revenue, earnings, EPS
Company Fundamentals	`/api/v3/key-metrics-ttm/{ticker}`	P/E, ROE, debt ratios

12.4.4 Test Query

"Compare JPMorgan and Goldman Sachs on revenue and P/E ratio"

Watch the execution log - you'll see the agent:

1. THINK: "I need data for both companies"

2. ACT: Call Stock Price for JPM
3. OBSERVE: Got JPM price
4. ACT: Call Company Fundamentals for JPM
5. OBSERVE: Got JPM P/E
6. ACT: Call Income Statement for JPM
7. OBSERVE: Got JPM revenue
8. (Repeat for GS)
9. CONCLUDE: Compare and respond

12.5 Pattern 3: Agent + Code Tools

Workflow: `03 - Agent + Code Tools.json`

What it does: Combines HTTP tools for data fetching with a JavaScript Code Tool for calculations. Unfortunately, there is no node in n8n that allows the agent to generate code and run them automatically, so we just illustrate the use of code in a Javascript Code Tool.

12.5.1 Workflow Structure

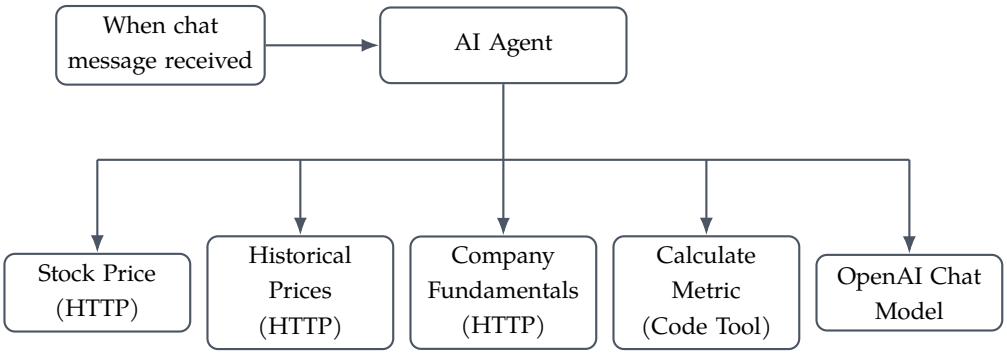

Figure 12.3: Agent Tool Selection Workflow

12.5.2 The Calculate Metric Code Tool

This is the CodeAct pattern - pre-built JavaScript that performs financial calculations.

Supported Calculation Types:

12.5.3 Code Walkthrough: Calculate Metric

The JavaScript code follows a clear structure. Here's what each part does:

Type	Description	Output
`sharpe`	Sharpe ratio	Annualized return, volatility, Sharpe
`sortino`	Sortino ratio	Uses downside deviation only
`var95`	Value at Risk 95%	Daily VaR at 95% confidence
`var99`	Value at Risk 99%	Daily VaR at 99% confidence
`volatility`	Annualized volatility	Daily and annualized vol
`returns`	Return metrics	Total, daily avg, annualized
`max_drawdown`	Maximum drawdown	Largest peak-to-trough decline

Input Parsing

```
let params;
try {
  if (typeof $input.item.json.query === 'string') {
    params = JSON.parse($input.item.json.query);
  } else if ($input.item.json.query) {
    params = $input.item.json.query;
  } else if (typeof $input.item.json.input === 'string') {
    params = JSON.parse($input.item.json.input);
  } else {
    params = $input.item.json;
  }
} catch (e) {
  return 'Error parsing input: ' + e.message;
}
```

Why this matters: n8n's AI Agent can pass tool parameters in different formats. This code checks multiple possible locations (`query` , `input` , or direct on `json`) and handles both string and object formats. Defensive parsing like this prevents runtime errors.

Helper Functions

```
function calcReturns(priceArray) {
  return priceArray.map((p, i) =>
    i > 0 ? (p - priceArray[i-1]) / priceArray[i-1] : 0
  ).slice(1);
}
```

What it does: Converts price series to daily returns. For prices `[100, 102, 101]` , it calculates:

- Day 1→2: $(102-100)/100 = 0.02$ (2%)
- Day 2→3: $(101-102)/102 = -0.0098$ (−0.98%)

The `.slice(1)` removes the first element (which would be 0).

```
function mean(arr) {
  return arr.reduce((a, b) => a + b, 0) / arr.length;
}

function std(arr) {
  const m = mean(arr);
  return Math.sqrt(
    arr.map(x => Math.pow(x - m, 2))
       .reduce((a, b) => a + b, 0) / arr.length
  );
}
```

What it does: Standard statistical functions. `std` calculates population standard deviation - the average distance from the mean.

```
function downsideStd(arr) {
  const negatives = arr.filter(r => r < 0);
  if (negatives.length === 0) return 0;
  return Math.sqrt(
    negatives.map(r => Math.pow(r, 2))
             .reduce((a, b) => a + b, 0) / negatives.length
  );
}
```

What it does: Used for Sortino ratio. Only considers negative returns (losses), ignoring positive days. This captures "bad volatility" rather than total volatility.

Sharpe Ratio Calculation

```
sharpe: () => {
  const returns = calcReturns(prices);
  const avgReturn = mean(returns);
  const stdDev = std(returns);
  const annualizedReturn = avgReturn * 252;       // 252 trading days/year
  const annualizedVol = stdDev * Math.sqrt(252);  // Vol scales with
      sqrt(time)
  const sharpe = (annualizedReturn - riskFreeRate) / annualizedVol;
  return { sharpeRatio: sharpe.toFixed(4), ... };
}
```

The formula: Sharpe = $(R - R_f)/\sigma$

Why 252? There are ~252 trading days per year. Daily returns are multiplied by 252 to annualize. Volatility is multiplied by $\sqrt{252}$ because variance (not standard deviation) scales linearly with time.

Interpretation: A Sharpe > 1 is good, > 2 is excellent. It measures return per unit of risk.

Sortino Ratio Calculation

```
sortino: () => {
  const returns = calcReturns(prices);
  const avgReturn = mean(returns);
  const downside = downsideStd(returns);              // Only negative
      returns
  const annualizedReturn = avgReturn * 252;
  const annualizedDownside = downside * Math.sqrt(252);
  const sortino = annualizedDownside > 0
    ? (annualizedReturn - riskFreeRate) / annualizedDownside
    : null;
  return {
    sortinoRatio: sortino ? sortino.toFixed(4) : 'N/A', ...
  };
}
```

Difference from Sharpe: Uses only downside deviation. A stock that gains 5% one day and loses 1% the next has low downside deviation but higher total volatility. Sortino rewards this asymmetry.

Value at Risk (VaR) Calculation

```
var95: () => {
  const returns = calcReturns(prices);
  const sorted = [...returns].sort((a, b) => a - b);  // Sort ascending
  const index = Math.floor(0.05 * sorted.length);     // 5th percentile
  const var95 = sorted[index];
  return {
    var95Daily: (var95 * 100).toFixed(2) + '%',
    interpretation: 'With 95% confidence, daily loss will not exceed ...'
  };
}
```

What it does: Sorts returns from worst to best, then picks the 5th percentile. This is the "historical VaR" method - simple and interpretable.

Example: If 50 days of returns are sorted and the 5th percentile (position 2–3) is −2.5%, then "with 95% confidence, you won't lose more than 2.5% in a day."

Maximum Drawdown Calculation

```
max_drawdown: () => {
  let peak = prices[0];
  let maxDrawdown = 0;

  for (let i = 0; i < prices.length; i++) {
    if (prices[i] > peak) peak = prices[i];       // New high-water mark
    const currentDrawdown = (peak - prices[i]) / peak;
    if (currentDrawdown > maxDrawdown) maxDrawdown = currentDrawdown;
  }

  return {
    maxDrawdown: (maxDrawdown * 100).toFixed(2) + '%',
    interpretation: 'The largest peak-to-trough decline was ...'
  };
}
```

What it does: Tracks the "high-water mark" (highest price seen so far). At each point, calculates how far below the peak the current price is. Reports the worst drop.

Why it matters: A stock could have great returns but terrifying drawdowns. If a stock dropped 50% before recovering, you'd need 100% gains just to break even.

12.5.4 How the Agent Uses It

The system prompt guides the agent:

```
When users ask for calculations like Sharpe ratio, VaR, Sortino ratio:
1. First fetch historical prices using the Historical Prices tool
2. Then call Calculate Metric with the calculationType and prices array
3. Explain the result in plain English
```

12.5.5 Test Queries

> *"Calculate the Sharpe ratio for AAPL"*

The agent will:

1. Call Historical Prices for AAPL (gets 50 days of data)
2. Extract the closing prices
3. Call Calculate Metric with `calculationType: 'sharpe'` and the prices array
4. Explain the result

   "What's the maximum drawdown for NVDA over the last 50 trading days?"

   "Compare the volatility of MSFT and GOOGL"

12.6 Pattern 4: Full Orchestration

Workflow: `04 - Portfolio Optimization: Full Orchestration Agent.json`

What it does: Complete portfolio analysis with 6 HTTP tools, 1 Code Tool, and Memory for follow-up questions. Unfortunately, there is no access to Python libraries in n8n so we do portfolio analysis (instead of portfolio optimization with the PyPortfolioOpt library).

12.6.1 Workflow Structure

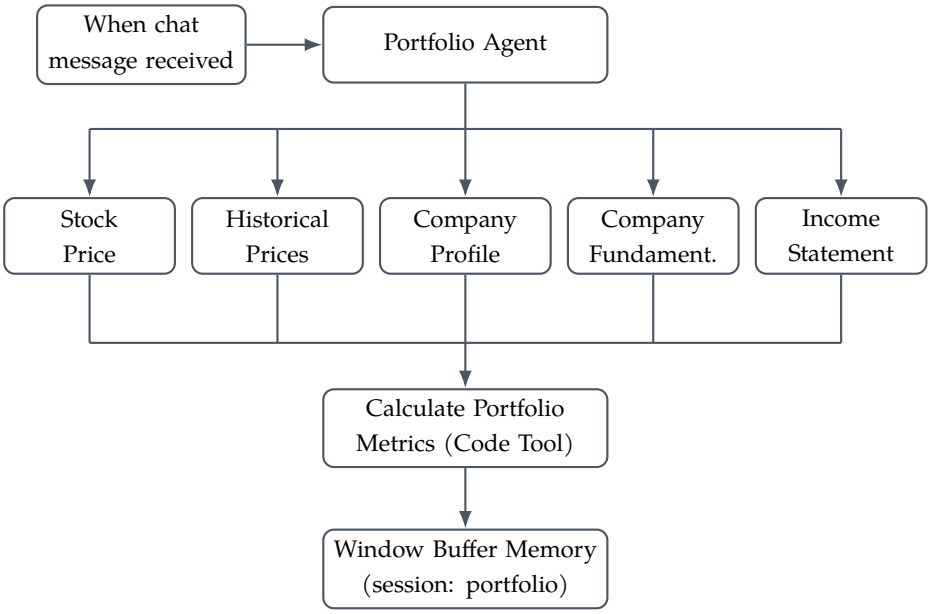

Figure 12.4: Portfolio Agent Data Flow

12.6.2 HTTP Tools

12.6.3 Calculate Portfolio Metrics Code Tool

This is the most sophisticated Code Tool - it handles portfolio-level calculations.

Supported Calculation Types:

Input Structure:

Tool	Purpose
Stock Price	Current price, change, volume
Historical Prices	Daily prices for return calculations
Company Profile	Sector, industry, description
Company Fundamentals	P/E, ROE, margins
Income Statement	Revenue, earnings, EPS

Type	Description	Required Input
`weights`	Portfolio weights and total value	holdings
`concentration`	HHI index, top holdings risk	holdings
`sector_allocation`	Breakdown by sector	holdings
`portfolio_return`	Weighted portfolio return	holdings + holdingsReturns
`portfolio_volatility`	Portfolio volatility	holdings + holdingsReturns
`portfolio_sharpe`	Portfolio Sharpe ratio	holdings + holdingsReturns
`rebalance`	Trades to hit target weights	holdings + targetWeights

```
{
  "calculationType": "weights",
  "holdings": [
    {"ticker": "AAPL", "shares": 100, "price": 178.50,
     "sector": "Technology"},
    {"ticker": "MSFT", "shares": 50, "price": 378.90,
     "sector": "Technology"},
    {"ticker": "JPM", "shares": 75, "price": 198.20,
     "sector": "Financial Services"}
  ]
}
```

12.6.4 Code Walkthrough: Portfolio Metrics

This Code Tool is more complex because it handles relationships *between* holdings. Here's what each calculation does:

Core Helper: getWeights()

```
function getWeights(holdingsArray) {
  const totalValue = holdingsArray.reduce(
    (sum, h) => sum + (h.shares * h.price), 0
  );
```

```
  if (totalValue === 0) return [];
  return holdingsArray.map(h => ({
    ticker: h.ticker,
    value: h.shares * h.price,
    weight: (h.shares * h.price) / totalValue,
    shares: h.shares,
    price: h.price,
    sector: h.sector || 'Unknown'
  }));
}
```

What it does: Calculates each holding's dollar value and portfolio weight. For holdings worth $17,850 (AAPL), $18,945 (MSFT), and $14,865 (JPM), total = $51,660. AAPL's weight = 17,850 / 51,660 = 34.5%.

Concentration Analysis (HHI)

```
concentration: () => {
  const weighted = getWeights(holdings);
  const sorted = [...weighted].sort((a, b) => b.weight - a.weight);

  // Herfindahl-Hirschman Index: sum of squared weights
  const hhi = weighted.reduce(
    (sum, w) => sum + Math.pow(w.weight, 2), 0
  );
  const effectiveN = hhi > 0 ? 1 / hhi : 0;

  const top1 = sorted[0] ? sorted[0].weight : 0;
  const top3 = sorted.slice(0, 3).reduce(
    (sum, w) => sum + w.weight, 0
  );

  return {
    herfindahlIndex: hhi.toFixed(4),
    effectiveNumberOfHoldings: effectiveN.toFixed(2),
    concentrationRisk:
      hhi > 0.25 ? 'HIGH' : hhi > 0.15 ? 'MEDIUM' : 'LOW'
  };
}
```

What HHI measures: The Herfindahl-Hirschman Index sums squared weights. If you have one stock at 100%, HHI = 1.0 (maximum concentration). If you have 10 equal stocks at 10% each, HHI = 0.10 (diversified).

Effective N: The inverse of HHI tells you "how many equal-weight holdings this port-

folio acts like." An HHI of 0.25 = effective N of 4 holdings.

Risk thresholds:

- HHI > 0.25 = HIGH (acts like fewer than 4 holdings)
- HHI > 0.15 = MEDIUM (acts like 4–7 holdings)
- HHI < 0.15 = LOW (well diversified)

Covariance for Portfolio Volatility

```
function covariance(arr1, arr2) {
  const m1 = mean(arr1);
  const m2 = mean(arr2);
  return arr1.map((x, i) => (x - m1) * (arr2[i] - m2))
             .reduce((a, b) => a + b, 0) / arr1.length;
}
```

What it does: Measures how two assets move together. Positive covariance = they move in the same direction. Negative = they move opposite. Zero = no relationship.

Why it matters for portfolios: Diversification benefit comes from holding assets with low or negative covariance. The portfolio volatility calculation uses the covariance matrix.

Portfolio Volatility (Not Just Weighted Average)

```
portfolio_volatility: () => {
  // Calculate variance using covariance matrix
  let portfolioVariance = 0;
  for (let i = 0; i < weighted.length; i++) {
    for (let j = 0; j < weighted.length; j++) {
      const wi = weighted[i].weight;
      const wj = weighted[j].weight;
      const cov = covariance(
        holdingsReturns[ti], holdingsReturns[tj]
      ) * 252;
      portfolioVariance += wi * wj * cov;
    }
  }
  const portfolioVol = Math.sqrt(Math.abs(portfolioVariance));

  return {
    portfolioVolatility: (portfolioVol * 100).toFixed(2) + '%',
    simpleWeightedVolatility: (simpleVol * 100).toFixed(2) + '%',
    diversificationBenefit:
```

```
      ((simpleVol - portfolioVol) * 100).toFixed(2) + '%'
  };
}
```

The key insight: Portfolio volatility ≠ weighted average of individual volatilities. If two assets are negatively correlated, combined volatility is *lower* than the weighted average.

Diversification benefit: The difference between "simple weighted vol" and "actual portfolio vol" shows how much risk reduction you get from holding multiple assets.

Rebalancing Calculator

```
rebalance: () => {
  const totalValue = weighted.reduce(
    (sum, w) => sum + w.value, 0
  );
  const trades = [];

  for (const w of weighted) {
    const target = targetWeights[w.ticker] || 0;
    const currentValue = w.value;
    const targetValue = target * totalValue;
    const difference = targetValue - currentValue;
    const sharesDiff = w.price > 0 ? difference / w.price : 0;

    trades.push({
      ticker: w.ticker,
      action: difference > 10 ? 'BUY'
            : difference < -10 ? 'SELL' : 'HOLD',
      shares: Math.abs(sharesDiff).toFixed(2),
      value: '$' + Math.abs(difference).toFixed(2)
    });
  }

  return {
    rebalancingTrades: trades.filter(t => t.action !== 'HOLD')
  };
}
```

What it does: Given target weights (e.g., equal weight = 33.3% each), calculates exactly how many shares to buy/sell.

Example: If AAPL is 40% and target is 33.3%, with total portfolio $100,000:

- Current AAPL value: $40,000

- Target AAPL value: $33,300
- Sell $6,700 worth of AAPL
- At $178.50/share = sell 37.5 shares

The $10 threshold prevents micro-trades that aren't worth executing.

12.6.5 Window Buffer Memory

The Memory node enables follow-up questions:

```
Session Key: portfolio_session
```

When a user asks about their portfolio, then follows up with "What if I sold some AAPL?" - the agent remembers the portfolio from the first question.

12.6.6 Test Queries

> *"I have 100 shares of AAPL, 50 shares of MSFT, and 75 shares of JPM. What's my portfolio worth and how concentrated is it?"*
>
> *"What sectors am I exposed to?"*
>
> *"What if I wanted to rebalance to equal weights?"*

12.7 Importing the Workflows

12.7.1 Step 1: Download the JSON Files

The course includes four workflow files in the `n8n_workflows` folder.

12.7.2 Step 2: Import into n8n

1. Open n8n (Cloud or self-hosted)
2. Click **Workflows → Import from File**
3. Select the JSON file
4. The workflow appears on your canvas

12.7.3 Step 3: Configure Credentials

OpenAI Credentials:

1. Go to **Credentials** in the left menu

2. Click **Add Credential → OpenAI**
3. Enter your API key
4. Save

FMP API Key:

The workflows have the FMP API key embedded in the HTTP Request nodes. To use your own key:

1. Click on each HTTP Request Tool node
2. Find the `apikey` parameter
3. Replace with your FMP API key

12.7.4 Step 4: Test

1. Open the workflow
2. Click **Execute Workflow** (or use the chat interface)
3. Send a test message
4. Watch the execution trace in the right panel

12.7.5 If Code in Code Tool Is Not Shown

If the JSON import doesn't preserve the Code Tool contents, paste the code below into the appropriate Code Tool node.

For 03 - Agent + Code Tools.json:

```
// Get input from the 'query' property
// (how n8n AI tools receive data)
let params;

try {
  if (typeof $input.item.json.query === 'string') {
    params = JSON.parse($input.item.json.query);
  } else if ($input.item.json.query) {
    params = $input.item.json.query;
  } else if (typeof $input.item.json.input === 'string') {
    params = JSON.parse($input.item.json.input);
  } else {
    params = $input.item.json;
  }
} catch (e) {
  return 'Error parsing input: ' + e.message;
}
```

```
const calculationType = params.calculationType;
const prices = params.prices;
const riskFreeRate = params.riskFreeRate || 0.05;

// Validate inputs
if (!prices || !Array.isArray(prices) || prices.length < 2) {
  return 'Error: prices must be an array with at least 2 values';
}

// Helper functions
function calcReturns(priceArray) {
  return priceArray.map((p, i) =>
    i > 0 ? (p - priceArray[i-1]) / priceArray[i-1] : 0
  ).slice(1);
}

function mean(arr) {
  return arr.reduce((a, b) => a + b, 0) / arr.length;
}

function std(arr) {
  const m = mean(arr);
  return Math.sqrt(
    arr.map(x => Math.pow(x - m, 2))
       .reduce((a, b) => a + b, 0) / arr.length
  );
}

function downsideStd(arr) {
  const negatives = arr.filter(r => r < 0);
  if (negatives.length === 0) return 0;
  return Math.sqrt(
    negatives.map(r => Math.pow(r, 2))
             .reduce((a, b) => a + b, 0) / negatives.length
  );
}

// Calculations
const calculations = {
  sharpe: () => {
    const returns = calcReturns(prices);
    const avgReturn = mean(returns);
    const stdDev = std(returns);
    const annualizedReturn = avgReturn * 252;
    const annualizedVol = stdDev * Math.sqrt(252);
    const sharpe =
      (annualizedReturn - riskFreeRate) / annualizedVol;
```

```
    return {
      sharpeRatio: sharpe.toFixed(4),
      annualizedReturn:
        (annualizedReturn * 100).toFixed(2) + '%',
      annualizedVolatility:
        (annualizedVol * 100).toFixed(2) + '%',
      riskFreeRateUsed:
        (riskFreeRate * 100).toFixed(2) + '%',
      tradingDays: returns.length
    };
  },

  sortino: () => {
    const returns = calcReturns(prices);
    const avgReturn = mean(returns);
    const downside = downsideStd(returns);
    const annualizedReturn = avgReturn * 252;
    const annualizedDownside = downside * Math.sqrt(252);
    const sortino = annualizedDownside > 0
      ? (annualizedReturn - riskFreeRate) / annualizedDownside
      : null;
    return {
      sortinoRatio: sortino
        ? sortino.toFixed(4) : 'N/A (no downside)',
      annualizedReturn:
        (annualizedReturn * 100).toFixed(2) + '%',
      annualizedDownsideDeviation:
        (annualizedDownside * 100).toFixed(2) + '%',
      riskFreeRateUsed:
        (riskFreeRate * 100).toFixed(2) + '%',
      tradingDays: returns.length
    };
  },

  var95: () => {
    const returns = calcReturns(prices);
    const sorted = [...returns].sort((a, b) => a - b);
    const index = Math.floor(0.05 * sorted.length);
    const var95 = sorted[index];
    return {
      var95Daily: (var95 * 100).toFixed(2) + '%',
      interpretation:
        'With 95% confidence, daily loss will not exceed '
        + (Math.abs(var95) * 100).toFixed(2) + '%',
      tradingDays: returns.length
    };
  },
```

```
  var99: () => {
    const returns = calcReturns(prices);
    const sorted = [...returns].sort((a, b) => a - b);
    const index = Math.floor(0.01 * sorted.length);
    const var99 = sorted[index];
    return {
      var99Daily: (var99 * 100).toFixed(2) + '%',
      interpretation:
        'With 99% confidence, daily loss will not exceed '
        + (Math.abs(var99) * 100).toFixed(2) + '%',
      tradingDays: returns.length
    };
  },

  volatility: () => {
    const returns = calcReturns(prices);
    const dailyVol = std(returns);
    const annualizedVol = dailyVol * Math.sqrt(252);
    return {
      dailyVolatility: (dailyVol * 100).toFixed(2) + '%',
      annualizedVolatility:
        (annualizedVol * 100).toFixed(2) + '%',
      tradingDays: returns.length
    };
  },

  returns: () => {
    const returns = calcReturns(prices);
    const totalReturn =
      (prices[prices.length - 1] - prices[0]) / prices[0];
    const avgDaily = mean(returns);
    const annualized = avgDaily * 252;
    return {
      totalReturn: (totalReturn * 100).toFixed(2) + '%',
      averageDailyReturn:
        (avgDaily * 100).toFixed(4) + '%',
      annualizedReturn: (annualized * 100).toFixed(2) + '%',
      tradingDays: returns.length
    };
  },

  max_drawdown: () => {
    let peak = prices[0];
    let maxDrawdown = 0;

    for (let i = 0; i < prices.length; i++) {
      if (prices[i] > peak) peak = prices[i];
      const currentDrawdown = (peak - prices[i]) / peak;
```

```
      if (currentDrawdown > maxDrawdown)
        maxDrawdown = currentDrawdown;
    }

    return {
      maxDrawdown: (maxDrawdown * 100).toFixed(2) + '%',
      interpretation:
        'The largest peak-to-trough decline was '
        + (maxDrawdown * 100).toFixed(2) + '%',
      tradingDays: prices.length
    };
  }
};

// Execute
if (!calculations[calculationType]) {
  return 'Error: Unknown calculation type "'
    + calculationType + '". Available: '
    + Object.keys(calculations).join(', ');
}

// Return as JSON string (required by n8n Code Tool)
return JSON.stringify(calculations[calculationType]());
```

For 04 - Portfolio Optimization: Full Orchestration Agent.json:

```
// Get input - handle ALL possible formats
let params;

try {
  const json = $input.item.json;

  // Check all possible input locations
  if (typeof json.query === 'string') {
    params = JSON.parse(json.query);
  } else if (typeof json.input === 'string') {
    params = JSON.parse(json.input);
  } else if (json.input && typeof json.input === 'object') {
    params = json.input;
  } else if (json.calculationType) {
    params = json;
  } else {
    return 'Error: Could not find valid input. Keys received: '
      + Object.keys(json).join(', ');
  }
} catch (e) {
  return 'Error parsing input: ' + e.message
```

```
    + '. Raw input: '
    + JSON.stringify($input.item.json).substring(0, 200);
}

const calculationType = params.calculationType;
const holdings = params.holdings || [];
const holdingsReturns = params.holdingsReturns || {};
const targetWeights = params.targetWeights || {};
const riskFreeRate = params.riskFreeRate || 0.05;

// Validate
if (!calculationType) {
  return 'Error: calculationType is required. Received params: '
    + JSON.stringify(params).substring(0, 200);
}

if (!holdings || !Array.isArray(holdings)
    || holdings.length === 0) {
  return 'Error: holdings array required with '
    + '{ticker, shares, price, sector}';
}

// Helper functions
function mean(arr) {
  if (!arr || arr.length === 0) return 0;
  return arr.reduce((a, b) => a + b, 0) / arr.length;
}

function std(arr) {
  if (!arr || arr.length === 0) return 0;
  const m = mean(arr);
  return Math.sqrt(
    arr.map(x => Math.pow(x - m, 2))
       .reduce((a, b) => a + b, 0) / arr.length
  );
}

function covariance(arr1, arr2) {
  if (!arr1 || !arr2 || arr1.length === 0) return 0;
  const m1 = mean(arr1);
  const m2 = mean(arr2);
  return arr1.map((x, i) => (x - m1) * (arr2[i] - m2))
             .reduce((a, b) => a + b, 0) / arr1.length;
}

// Calculate weights
function getWeights(holdingsArray) {
  const totalValue = holdingsArray.reduce(
```

```
    (sum, h) => sum + (h.shares * h.price), 0
  );
  if (totalValue === 0) return [];
  return holdingsArray.map(h => ({
    ticker: h.ticker,
    value: h.shares * h.price,
    weight: (h.shares * h.price) / totalValue,
    shares: h.shares,
    price: h.price,
    sector: h.sector || 'Unknown'
  }));
}

// Calculations
const calculations = {
  weights: () => {
    const weighted = getWeights(holdings);
    const totalValue = weighted.reduce(
      (sum, w) => sum + w.value, 0
    );
    return {
      holdings: weighted.map(w => ({
        ticker: w.ticker,
        value: '$' + w.value.toFixed(2),
        weight: (w.weight * 100).toFixed(2) + '%'
      })),
      totalPortfolioValue: '$' + totalValue.toFixed(2)
    };
  },

  concentration: () => {
    const weighted = getWeights(holdings);
    const sorted = [...weighted].sort(
      (a, b) => b.weight - a.weight
    );

    const hhi = weighted.reduce(
      (sum, w) => sum + Math.pow(w.weight, 2), 0
    );
    const effectiveN = hhi > 0 ? 1 / hhi : 0;

    const top1 = sorted[0] ? sorted[0].weight : 0;
    const top3 = sorted.slice(0, 3).reduce(
      (sum, w) => sum + w.weight, 0
    );
    const top5 = sorted.slice(0, 5).reduce(
      (sum, w) => sum + w.weight, 0
    );
```

```
    return {
      herfindahlIndex: hhi.toFixed(4),
      effectiveNumberOfHoldings: effectiveN.toFixed(2),
      top1Concentration: (top1 * 100).toFixed(2) + '%',
      top3Concentration: (top3 * 100).toFixed(2) + '%',
      top5Concentration: (top5 * 100).toFixed(2) + '%',
      topHoldings: sorted.slice(0, 5).map(w => ({
        ticker: w.ticker,
        weight: (w.weight * 100).toFixed(2) + '%'
      })),
      concentrationRisk: hhi > 0.25 ? 'HIGH'
        : hhi > 0.15 ? 'MEDIUM' : 'LOW'
    };
  },

  sector_allocation: () => {
    const weighted = getWeights(holdings);
    const sectorWeights = {};

    for (const w of weighted) {
      const sector = w.sector || 'Unknown';
      sectorWeights[sector] =
        (sectorWeights[sector] || 0) + w.weight;
    }

    return {
      sectorAllocation: Object.entries(sectorWeights)
        .sort((a, b) => b[1] - a[1])
        .map(([sector, weight]) => ({
          sector,
          weight: (weight * 100).toFixed(2) + '%'
        })),
      numberOfSectors: Object.keys(sectorWeights).length
    };
  },

  portfolio_return: () => {
    const weighted = getWeights(holdings);
    const tickers = Object.keys(holdingsReturns);

    if (tickers.length === 0) {
      return { error:
        'holdingsReturns required with ticker keys '
        + 'and returns arrays' };
    }

    let portfolioReturn = 0;
```

```
    const details = [];

    for (const w of weighted) {
      if (holdingsReturns[w.ticker]) {
        const returns = holdingsReturns[w.ticker];
        const annualizedReturn = mean(returns) * 252;
        portfolioReturn += w.weight * annualizedReturn;
        details.push({
          ticker: w.ticker,
          weight: (w.weight * 100).toFixed(2) + '%',
          annualizedReturn:
            (annualizedReturn * 100).toFixed(2) + '%',
          contribution:
            (w.weight * annualizedReturn * 100).toFixed(2)
            + '%'
        });
      }
    }

    return {
      portfolioAnnualizedReturn:
        (portfolioReturn * 100).toFixed(2) + '%',
      holdingDetails: details
    };
  },

  portfolio_volatility: () => {
    const weighted = getWeights(holdings);
    const tickers = Object.keys(holdingsReturns);

    if (tickers.length === 0) {
      return { error: 'holdingsReturns required' };
    }

    const vols = {};
    for (const ticker of tickers) {
      vols[ticker] =
        std(holdingsReturns[ticker]) * Math.sqrt(252);
    }

    let simpleVol = 0;
    for (const w of weighted) {
      if (vols[w.ticker]) {
        simpleVol += w.weight * vols[w.ticker];
      }
    }

    let portfolioVariance = 0;
```

```
    for (let i = 0; i < weighted.length; i++) {
      for (let j = 0; j < weighted.length; j++) {
        const ti = weighted[i].ticker;
        const tj = weighted[j].ticker;
        if (holdingsReturns[ti] && holdingsReturns[tj]) {
          const wi = weighted[i].weight;
          const wj = weighted[j].weight;
          const cov = covariance(
            holdingsReturns[ti], holdingsReturns[tj]
          ) * 252;
          portfolioVariance += wi * wj * cov;
        }
      }
    }

    const portfolioVol =
      Math.sqrt(Math.abs(portfolioVariance));

    return {
      portfolioVolatility:
        (portfolioVol * 100).toFixed(2) + '%',
      simpleWeightedVolatility:
        (simpleVol * 100).toFixed(2) + '%',
      diversificationBenefit:
        ((simpleVol - portfolioVol) * 100).toFixed(2) + '%'
    };
  },

  portfolio_sharpe: () => {
    const weighted = getWeights(holdings);
    const tickers = Object.keys(holdingsReturns);

    if (tickers.length === 0) {
      return { error: 'holdingsReturns required' };
    }

    let portfolioReturn = 0;
    for (const w of weighted) {
      if (holdingsReturns[w.ticker]) {
        portfolioReturn +=
          w.weight * mean(holdingsReturns[w.ticker]) * 252;
      }
    }

    let portfolioVariance = 0;
    for (let i = 0; i < weighted.length; i++) {
      for (let j = 0; j < weighted.length; j++) {
        const ti = weighted[i].ticker;
```

```
        const tj = weighted[j].ticker;
        if (holdingsReturns[ti] && holdingsReturns[tj]) {
          const wi = weighted[i].weight;
          const wj = weighted[j].weight;
          const cov = covariance(
            holdingsReturns[ti], holdingsReturns[tj]
          ) * 252;
          portfolioVariance += wi * wj * cov;
        }
      }
    }

    const portfolioVol =
      Math.sqrt(Math.abs(portfolioVariance));
    const sharpe = portfolioVol > 0
      ? (portfolioReturn - riskFreeRate) / portfolioVol
      : 0;

    return {
      portfolioSharpeRatio: sharpe.toFixed(4),
      portfolioAnnualizedReturn:
        (portfolioReturn * 100).toFixed(2) + '%',
      portfolioVolatility:
        (portfolioVol * 100).toFixed(2) + '%',
      riskFreeRateUsed:
        (riskFreeRate * 100).toFixed(2) + '%'
    };
  },

  rebalance: () => {
    const weighted = getWeights(holdings);
    const totalValue = weighted.reduce(
      (sum, w) => sum + w.value, 0
    );

    if (Object.keys(targetWeights).length === 0) {
      return { error:
        'targetWeights required as object '
        + 'with ticker keys and weights 0-1' };
    }

    const trades = [];
    for (const w of weighted) {
      const target = targetWeights[w.ticker] || 0;
      const currentValue = w.value;
      const targetValue = target * totalValue;
      const difference = targetValue - currentValue;
      const sharesDiff = w.price > 0
```

```
        ? difference / w.price : 0;

      trades.push({
        ticker: w.ticker,
        currentWeight:
          (w.weight * 100).toFixed(2) + '%',
        targetWeight: (target * 100).toFixed(2) + '%',
        action: difference > 10 ? 'BUY'
              : difference < -10 ? 'SELL' : 'HOLD',
        shares: Math.abs(sharesDiff).toFixed(2),
        value: '$' + Math.abs(difference).toFixed(2)
      });
    }

    return {
      rebalancingTrades:
        trades.filter(t => t.action !== 'HOLD'),
      noActionNeeded:
        trades.filter(t => t.action === 'HOLD')
              .map(t => t.ticker),
      portfolioValue: '$' + totalValue.toFixed(2)
    };
  }
};

// Execute
if (!calculations[calculationType]) {
  return 'Error: Unknown calculation type "'
    + calculationType + '". Available: '
    + Object.keys(calculations).join(', ');
}

try {
  const result = calculations[calculationType]();
  return JSON.stringify(result);
} catch (e) {
  return 'Error executing calculation: ' + e.message;
}
```

12.8 Comparing Python and n8n

12.8.1 The Pattern Mapping

12.9 n8n-Specific Considerations

Aspect	Python (smolagents)	n8n
Tool Definition	`@tool` decorator + docstring	HTTP Tool + description field
Code Execution	`additional_authorized_imports`	Code Tool node with JavaScript
Memory	`reset=False` parameter	Window Buffer Memory node
ReAct	Built into CodeAgent	System prompt + multiple tools
API Calls	Python requests/yfinance	HTTP Request Tool node

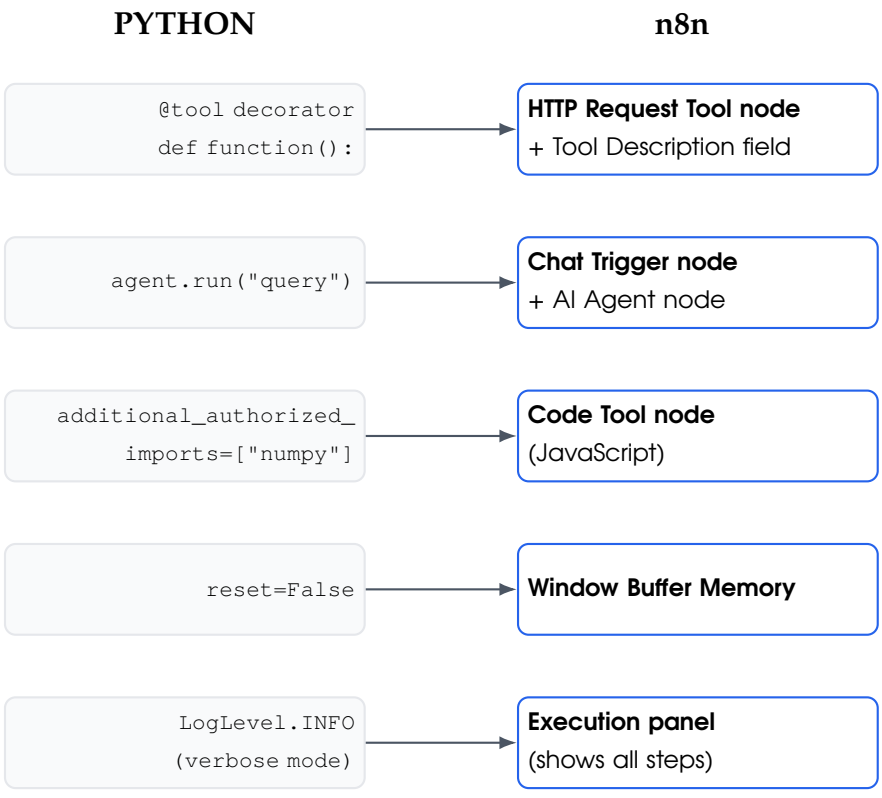

Figure 12.5: Concept Mapping: Python vs n8n

12.9.1 Benefits of Visual Workflows

1. **Visual Audit Trails:** Every execution shows which tools were called, in what order, with what inputs - making debugging and review straightforward
2. **Workflow Versioning:** Export JSON files for version control, sharing, and backup
3. **Access Control:** n8n Cloud supports role-based access for team collaboration
4. **Execution History:** Review any past run at any time to understand what happened

12.9.2 JavaScript vs Python

The Code Tools use JavaScript, not Python. Key differences:

Python	JavaScript
`numpy.mean(arr)`	`arr.reduce((a,b) => a+b) / arr.length`
`numpy.std(arr)`	Custom function (see code in workflows)
`pandas`	Not available - use array methods
`scipy.optimize`	Not available - use external API

For complex calculations (like mean-variance optimization), you have options:

1. Implement in Python and expose via API
2. Use a simpler approximation in JavaScript
3. Call an external optimization service

12.10 Troubleshooting

12.10.1 "Tool not being called"

- Check that the Tool Description clearly explains when to use it
- Verify the tool is connected to the AI Agent node
- Check the system prompt guides the agent to use tools

12.10.2 "FMP API errors"

- Verify your API key is correct
- Check rate limits
- Some endpoints require paid tier

12.10.3 "Code Tool returns error"

- Check the JavaScript syntax
- Use `return JSON.stringify(result)` for objects
- Check input parsing - the tool receives input in `$input.item.json`

13

Financial Concepts - A Beginner's Guide

This is for folks who are new to investment. If you are experienced, feel free to skip this.

You can build an AI agent that calculates a Sharpe ratio without understanding what a Sharpe ratio means. But, you just can't understand what you built.

This chapter covers the financial concepts your agents will use - from stock prices to portfolio risk metrics. Each section explains the concept, walks through the code, and connects it to the agent patterns you learned in the main book.

The companion notebooks (included in the course materials) let you run every calculation yourself with live market data.

13.1 Three Layers of Financial Knowledge

Think of financial analysis as three layers:

```
LAYER 3: PORTFOLIO INTELLIGENCE OR ANALYTICS
How assets work together
Correlation, diversification, sector analysis, rebalancing

LAYER 2: RISK METRICS
How to measure what could go wrong
Returns, volatility, Sharpe, Sortino, VaR, drawdown

LAYER 1: THE BASICS
What you own and what it's worth
Stocks, positions, portfolio value, weights
```

Layer 1: The Basics

13.2 Stock Fundamentals

Notebook: `01_stock_basics.ipynb`

Before your agent can analyze anything, it needs to understand the data it's working with. Every stock has a **ticker symbol** - the short code that identifies it. When your agent fetches Apple's price, it asks for `"AAPL"`, not `"Apple Inc."`.

13.2.1 Key Data Points

Term	What It Is	Why It Matters
Ticker	Stock identifier (AAPL, NVDA)	How agents address companies in API calls
Open	First trade price of the day	Shows overnight sentiment
High/Low	Highest/lowest price during the day	Shows intraday range
Close	Last trade price of the day	Used for most analysis
Volume	Shares traded in a period	High volume = high liquidity and interest
Market Cap	Price × Shares Outstanding	Total company value
52-Week High/Low	Year's price range	Shows where current price sits

13.2.2 Code: Stock Dashboard

The notebook builds a `stock_dashboard()` function that pulls all these metrics together:

```
import yfinance as yf
```

```
def stock_dashboard(ticker_symbol):
    """Create a comprehensive stock dashboard."""
    ticker = yf.Ticker(ticker_symbol)
    info = ticker.info

    # Price data
    current = info.get('currentPrice') or info.get('regularMarketPrice')
    open_price = info.get('regularMarketOpen', 0)
    high = info.get('regularMarketDayHigh', 0)
    low = info.get('regularMarketDayLow', 0)
    prev_close = info.get('previousClose', 0)

    # Derived metrics
    daily_change = current - prev_close
    daily_change_pct = (daily_change / prev_close * 100)

    # 52-week position
    high_52 = info.get('fiftyTwoWeekHigh', 0)
    low_52 = info.get('fiftyTwoWeekLow', 0)
    from_52_high = ((high_52 - current) / high_52 * 100)

    return {
        'current': current,
        'change': daily_change_pct,
        'from_52_high': from_52_high,
        'market_cap': info.get('marketCap', 0),
        'volume': info.get('regularMarketVolume', 0)
    }
```

Why this matters for agents: This is exactly what the `get_stock_price` and `get_company_profile` tools return. When your agent says "AAPL is trading at $249.78, down 13.5% from its 52-week high" - these are the calculations behind that statement.

13.2.3 Market Cap Classification

Market cap tells you the *size* of the company.

Category	Market Cap	Example
Mega-cap	$200B+	AAPL, MSFT, NVDA
Large-cap	$10B - $200B	Most S&P 500 stocks
Mid-cap	$2B - $10B	Growing companies
Small-cap	$300M - $2B	Emerging companies
Micro-cap	< $300M	Smallest public companies

Market cap, not share price, tells you how big a company is. A $500 stock isn't necessarily "more expensive" than a $50 stock - it depends on how many shares exist.

13.2.4 Volume as a Signal

Volume tells you about *interest* and *liquidity*:

```
volume_ratio = today_volume / avg_volume

if volume_ratio > 1.5:
    interpretation = "Unusual interest, something is happening"
elif volume_ratio > 1.0:
    interpretation = "Above average, active day"
elif volume_ratio > 0.7:
    interpretation = "Normal trading"
else:
    interpretation = "Quiet day, low interest"
```

13.3 Position and Portfolio Value

Notebook: `02_position_and_portfolio_value.ipynb`

A **position** is your holding in a single stock. A **portfolio** is all your positions combined.

13.3.1 The Formulas

Position Value:

$$\text{Position Value} = \text{Shares} \times \text{Current Price}$$

Portfolio Value:

$$\text{Portfolio Value} = \sum_{i=1}^{n} (\text{Shares}_i \times \text{Price}_i)$$

Cost Basis (what you paid):

$$\text{Cost Basis} = \text{Shares} \times \text{Purchase Price}$$

Unrealized Gain/Loss:

$$\text{Gain/Loss} = \text{Current Value} - \text{Cost Basis}$$

13.3.2 Code: Portfolio Tracker

```
def calculate_portfolio_value(holdings):
    """Calculate total portfolio value from holdings.

    Args:
        holdings: Dictionary of {ticker: shares}

    Returns:
        Dictionary with positions and total value
    """
    positions = []

    for ticker, shares in holdings.items():
        price = get_price(ticker)
        value = shares * price
        positions.append({
            'ticker': ticker,
            'shares': shares,
            'price': price,
            'value': value
        })

    total_value = sum(p['value'] for p in positions)

    return {
        'positions': positions,
        'total_value': total_value
    }
```

What it does: Loops through each holding, fetches the current price, multiplies by shares, and sums everything up. This is the foundation of every portfolio analysis - your agent needs to know the *state* of the portfolio before it can analyze anything.

13.3.3 Cost Basis and Performance

The notebook adds `purchase_price` tracking to calculate unrealized gains:

```
def calculate_position_with_cost(ticker, shares, purchase_price):
    current_price = get_price(ticker)

    cost_basis = shares * purchase_price
    current_value = shares * current_price
    gain_loss = current_value - cost_basis
    return_pct = (gain_loss / cost_basis) * 100

    return {
        'cost_basis': cost_basis,
```

```
        'current_value': current_value,
        'gain_loss': gain_loss,
        'return_pct': return_pct
    }
```

Example output:

```
AAPL: 100 shares @ $150.00 -> $249.79
      Cost Basis: $15,000 | Current: $24,979 | Gain: +$9,979 (+66.5%)
```

Why this matters for agents: When a user asks "How is my portfolio doing?", the agent needs to report both current value and performance against what the user originally paid.

13.4 Portfolio Weights

Notebook: `03_portfolio_weights.ipynb`

Dollar values tell you how much each position is worth. **Weights** tell you something more important: how *concentrated* or *diversified* your portfolio is.

13.4.1 The Formula

$$\text{Weight}_i = \frac{\text{Position Value}_i}{\text{Total Portfolio Value}} \times 100\%$$

13.4.2 Code: Weight Calculation

```
def calculate_portfolio_weights(holdings):
    """Calculate the weight of each position in a portfolio."""
    positions = []

    for ticker, shares in holdings.items():
        price = get_price(ticker)
        value = shares * price
        positions.append({
            'ticker': ticker,
            'shares': shares,
            'price': price,
            'value': value
        })

    df = pd.DataFrame(positions)
```

```
    total_value = df['value'].sum()
    df['weight'] = (df['value'] / total_value) * 100

    return df, total_value
```

13.4.3 Why Weights Matter More Than Dollar Values

The notebook introduces a crucial concept: **portfolio impact**.

$$\text{Portfolio Impact} = \text{Position Weight} \times \text{Position Return}$$

If MSFT is 40% of your portfolio and drops 20%, your portfolio loses 8%. If AMZN is 7% of your portfolio and rallies 30%, your portfolio gains only 2.1%.

The lesson: Big positions dominate your returns. A small position can double and barely move the needle.

13.4.4 Weight Drift

Over time, winners grow and losers shrink - changing your weights:

```
# Start: 20% each in 5 stocks
# NVDA doubles (+100%), others have modest returns

# Result:
# NVDA: 20% -> 32% (now dominates)
# Others: shrunk proportionally
```

Why this matters for agents: The Portfolio Agent (Pattern 4) needs to calculate weights before it can assess concentration risk, sector exposure, or suggest rebalancing.

Layer 2: Risk Metrics

13.5 Returns and Performance

Notebook: `04_returns_and_performance.ipynb`

"I made $1,000!" sounds great. But if you invested $100,000, that's only 1%. If you invested $5,000, that's 20%.

Returns standardize performance so you can compare apples to apples.

13.5.1 Four Types of Returns

1. Simple Return - The basic calculation

$$R = \frac{P_{\text{end}} - P_{\text{start}}}{P_{\text{start}}} \times 100\%$$

2. Daily Return - Day-to-day changes

$$R_t = \frac{P_t - P_{t-1}}{P_{t-1}}$$

This is the building block for everything else. Every risk metric - Sharpe, Sortino, VaR - starts with daily returns.

3. Cumulative Return - The full journey

$$\text{Cumulative Return} = \prod_{i=1}^{n}(1 + r_i) - 1$$

Why multiply, not add? Because returns compound. A 10% gain followed by a 10% loss doesn't return you to break-even:

- $\$100 \times 1.10 = \$110$
- $\$110 \times 0.90 = \$99$ (not \$100)

4. Annualized Return - Standardized to one year

$$R_{\text{annual}} = (1 + R_{\text{total}})^{\frac{252}{\text{trading days}}} - 1$$

Why 252? There are approximately 252 trading days per year. This lets you compare a 3-month investment to a 2-year investment on equal footing.

13.5.2 Code: Daily Returns

```
import yfinance as yf
import numpy as np

def get_daily_returns(ticker, period="6mo"):
    """Calculate daily returns for a stock."""
    stock = yf.Ticker(ticker)
    hist = stock.history(period=period)

    # pct_change() computes: (current - previous) / previous
    hist['daily_return'] = hist['Close'].pct_change() * 100
```

```
    return hist[['Close', 'daily_return']].dropna()
```

What `pct_change()` does: For each day, it calculates `(today - yesterday) / yesterday`. This single line converts a price series into a return series - the foundation of all risk analysis.

13.5.3 Volatility: The Standard Deviation of Returns

Volatility measures how much returns fluctuate. Higher volatility = more uncertainty = more risk.

$$\sigma_{\text{daily}} = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(r_i - \bar{r})^2}$$

Annualized Volatility:

$$\sigma_{\text{annual}} = \sigma_{\text{daily}} \times \sqrt{252}$$

Why square root of 252? Variance (not standard deviation) scales linearly with time. Since standard deviation is the square root of variance, it scales with the square root of time.

13.5.4 Code: Volatility Calculation

```
def calculate_volatility(ticker, period="1y"):
    """Calculate daily and annualized volatility."""
    stock = yf.Ticker(ticker)
    hist = stock.history(period=period)
    daily_returns = hist['Close'].pct_change().dropna()

    daily_vol = daily_returns.std()
    annual_vol = daily_vol * np.sqrt(252)

    return {
        'daily_vol': daily_vol * 100,
        'annual_vol': annual_vol * 100
    }
```

Interpretation guide:

Annual Volatility	Risk Level	Example
< 15%	Low	Bond ETFs, utilities
15% - 25%	Moderate	S&P 500, large-cap stocks
25% - 40%	High	Growth stocks, tech
40% - 60%	Very High	TSLA, meme stocks
> 60%	Extreme	Crypto, penny stocks

13.6 Sharpe Ratio: Return Per Unit of Risk

Notebook: `05_sharpe_ratio.ipynb`

A 50% return sounds amazing. But what if you had to endure 80% volatility to get it? The **Sharpe Ratio** answers: "Was the return worth the risk?"

13.6.1 The Formula

$$\text{Sharpe Ratio} = \frac{R_p - R_f}{\sigma_p}$$

Where:

- R_p = Portfolio/asset return (annualized)
- R_f = Risk-free rate (what you'd earn with zero risk - Treasury bills)
- σ_p = Volatility (annualized standard deviation)

In plain English: How much *excess* return did you earn for each unit of risk you took?

13.6.2 Why Subtract the Risk-Free Rate?

If you can earn 5% risk-free in Treasury bills, you only deserve credit for returns *above* that. Taking risk to earn 4% when you could earn 5% risk-free would be foolish. The risk-free rate is your baseline - the "free lunch" rate.

13.6.3 Code: Sharpe Ratio

```
def calculate_sharpe_ratio(returns, risk_free_rate=0.05):
    """Calculate the Sharpe Ratio from daily returns.

    Args:
        returns: Array of daily returns (as decimals)
        risk_free_rate: Annual risk-free rate (default 5%)

    Returns:
```

```
        Annualized Sharpe Ratio
    """
    # Annualize daily return
    annual_return = np.mean(returns) * 252

    # Annualize volatility
    annual_volatility = np.std(returns, ddof=1) * np.sqrt(252)

    # Sharpe = excess return / volatility
    sharpe = (annual_return - risk_free_rate) / annual_volatility

    return {
        'sharpe_ratio': sharpe,
        'annual_return': annual_return * 100,
        'annual_volatility': annual_volatility * 100
    }
```

Key details in the code:

- `ddof=1` uses sample standard deviation (divides by $n-1$, not n) - more accurate for estimating population volatility from a sample
- `* 252` annualizes daily returns (252 trading days)
- `* np.sqrt(252)` annualizes daily volatility (because variance, not std, scales linearly)

13.6.4 Interpretation

Sharpe Ratio	Rating	What It Means
< 0	Negative	Losing to the risk-free rate
0 - 0.5	Poor	Very little return for the risk taken
0.5 - 1.0	Acceptable	Decent risk-adjusted performance
1.0 - 2.0	Good	Strong performance per unit of risk
> 2.0	Excellent	Very high (or possibly suspicious)

Example from the notebook:

```
GOOGL: Sharpe 2.11 (Excellent - 72.2% return, 31.8% vol)
MSFT:  Sharpe -0.05 (Negative - 3.8% return, below risk-free rate)
```

GOOGL earned much more return per unit of risk. Even though both are tech stocks, their risk-adjusted performance was dramatically different.

13.6.5 Portfolio Sharpe

The notebook also calculates Sharpe for an entire portfolio - not just individual stocks. The portfolio version uses *portfolio returns* (weighted sum of individual returns), which accounts for diversification effects.

```
# Calculate portfolio returns
portfolio_returns = sum(df[ticker] * weight for ticker, weight in
    holdings.items())

# Then calculate Sharpe on the portfolio return series
```

Why this matters for agents: When the agent calculates portfolio Sharpe using the Code Tool, it follows exactly this logic - annualize the returns, annualize the volatility, divide.

13.7 Sortino Ratio: Only Downside Counts

Notebook: `08_sortino_ratio.ipynb`

The Sharpe Ratio treats all volatility as equally bad. But is upside volatility really a problem? If a stock jumps 25% one day, Sharpe penalizes that as "risky." Sortino says: "Wait - only *downside* moves should count."

13.7.1 The Formula

$$\text{Sortino Ratio} = \frac{R_p - R_{\text{target}}}{\sigma_{\text{downside}}}$$

The only difference from Sharpe: The denominator uses **downside deviation** instead of total standard deviation.

13.7.2 Downside Deviation

$$\sigma_{\text{downside}} = \sqrt{\frac{1}{n_{\text{down}}} \sum \min(R_i - R_{\text{target}}, 0)^2}$$

In plain English: Only look at returns *below* your target (usually 0). Square them, average them, take the square root.

13.7.3 Code: Downside Deviation

```
def calculate_downside_deviation(returns, target_return=0):
    """Calculate downside deviation - only negative returns count."""
    # Filter to returns below target
```

```
    downside_returns = returns - target_return
    downside_returns = downside_returns[downside_returns < 0]

    if len(downside_returns) == 0:
        return 0

    # RMS of negative returns
    downside_squared = downside_returns ** 2
    downside_dev = np.sqrt(downside_squared.mean())

    return downside_dev
```

What it does: Filters out all positive returns, then calculates the standard deviation of only the negative ones. A stock that gains 5% one day and loses 1% the next has low downside deviation but higher total volatility. Sortino rewards this asymmetry.

13.7.4 Sharpe vs Sortino: What the Difference Tells You

If Sortino > Sharpe	If Sortino < Sharpe	If roughly equal
More upside volatility than downside	More downside volatility than upside	Symmetric returns
Favorable asymmetry	Unfavorable asymmetry	Sharpe is fine alone
Sharpe *understates* risk-adjusted return	Sharpe *overstates* risk-adjusted return	Both tell the same story

Example from the notebook:

```
GOOGL: Sharpe 1.64, Sortino 1.68  (+0.04) -> Favorable
JPM:   Sharpe 1.38, Sortino 1.29  (-0.09) -> Unfavorable
```

GOOGL's upside surprises are bigger than its downside ones. JPM has more downside volatility - its losses tend to be larger relative to its gains.

13.7.5 When to Use Each

Use Sharpe	Use Sortino
Diversified portfolios	Individual stocks
Symmetric return distributions	Skewed returns
Industry standard required	More nuanced analysis
Comparing to benchmarks	Evaluating asymmetric strategies

Best practice: Calculate both. The difference itself is informative.

13.8 Maximum Drawdown: Your Worst Day

Notebook: `06_maximum_drawdown.ipynb`

Sharpe and Sortino tell you about *average* risk. But investors often care more about the worst case. Maximum Drawdown answers: "What's the most I could have lost?"

13.8.1 The Formula

$$\text{Drawdown}_t = \frac{V_t - \text{Peak}_t}{\text{Peak}_t}$$

$$\text{Maximum Drawdown} = \min_t(\text{Drawdown}_t)$$

The mental model: Imagine climbing a mountain. Every time you reach a new high point, that becomes your peak. If you descend, the drawdown is how far you've fallen from that peak. Maximum drawdown is the deepest valley.

13.8.2 Code: Drawdown Calculation

```
def calculate_drawdown(prices):
    """Calculate drawdown series from price data."""
    # Running maximum - the "peak" at each point in time
    running_max = prices.cummax()

    # How far below the peak at each point
    drawdown = (prices - running_max) / running_max

    # The worst drawdown
    max_drawdown = drawdown.min()

    # When did it happen?
    max_dd_end = drawdown.idxmin()                # Trough date
    max_dd_start = prices[:max_dd_end].idxmax()   # Peak date before trough

    return {
        'max_drawdown': max_drawdown,
        'peak_date': max_dd_start,
        'trough_date': max_dd_end,
        'drawdown_series': drawdown
    }
```

Key line: `prices.cummax()` computes the running maximum - the highest value seen *so far* at each point. This is the "high-water mark." The drawdown at any point is simply how far the current price is below that mark.

13.8.3 The Asymmetry of Losses

This is the most important insight in the notebook:

Loss	Gain Needed to Recover
−10%	+11.1%
−20%	+25.0%
−30%	+42.9%
−50%	+100.0%
−75%	+300.0%

A 50% loss requires a 100% gain just to break even. This is why drawdowns matter so much - the deeper the hole, the harder it is to climb out.

13.8.4 Calmar Ratio: Return per Drawdown

The notebook introduces the **Calmar Ratio**, which combines return and drawdown:

$$\text{Calmar Ratio} = \frac{\text{Annualized Return}}{|\text{Maximum Drawdown}|}$$

Calmar	Meaning
< 0.5	Poor - too much drawdown for the return
0.5 - 1.0	Acceptable
1.0 - 2.0	Good - solid return relative to worst case
> 2.0	Excellent

Example from the notebook:

```
NVDA:  Calmar 2.91 (107.5% return, -36.9% max drawdown)
MSFT:  Calmar 0.80 (19.0% return, -23.7% max drawdown)
```

NVDA's drawdown was worse than MSFT's, but the returns more than compensated. MSFT had a mild drawdown but mediocre returns.

13.8.5 Drawdown Duration

The notebook also tracks how long you're "underwater" - from peak to recovery:

```
AAPL worst drawdown: -33.4%
  Decline: 101 days (peak to trough)
  Recovery: 195 days (trough to new peak)
  Total underwater: 296 days
```

Why this matters for agents: When the Code Tool calculates `max_drawdown`, it walks through prices tracking the high-water mark - exactly this algorithm. The agent can then say "your portfolio's worst peak-to-trough decline was 28.7%" and explain what that means.

13.9 Value at Risk (VaR)

Notebook: `10_value_at_risk.ipynb`

Risk managers need to answer a specific question: "With X% confidence, what's the most we could lose over the next N days?"

VaR provides a statistical answer.

13.9.1 VaR Statement

> "With 95% confidence, we will not lose more than $2,706 in a single day."

Three components:

1. **Confidence level** (95%, 99%) - How sure are we?
2. **Time horizon** (1 day, 10 days) - Over what period?
3. **Loss amount** ($2,706) - The threshold

13.9.2 Three Methods

1. Historical VaR - Use actual returns, find the percentile.

```python
def calculate_historical_var(returns, confidence_level=0.95,
    portfolio_value=100000):
    """Historical VaR: simply find the percentile of actual returns."""
    var_percentile = 1 - confidence_level   # 0.05 for 95% confidence
    var_return = np.percentile(returns, var_percentile * 100)
    var_dollar = portfolio_value * abs(var_return)
```

```
    return {
        'var_return_pct': var_return * 100,
        'var_dollar': var_dollar
    }
```

What it does: Sorts all historical daily returns from worst to best. The 5th percentile (for 95% VaR) is your threshold. If you have 500 days of history, the 25th worst day is your VaR.

2. Parametric VaR - Assume normal distribution, use the formula.

$$\text{VaR} = \mu + z \times \sigma$$

Where $z = -1.645$ for 95% confidence, -2.326 for 99%.

```
from scipy import stats

def calculate_parametric_var(returns, confidence_level=0.95,
    portfolio_value=100000):
    """Parametric VaR: assumes normal distribution."""
    mu = returns.mean()
    sigma = returns.std()
    z_score = stats.norm.ppf(1 - confidence_level)  # -1.645 for 95%

    var_return = mu + z_score * sigma
    var_dollar = portfolio_value * abs(var_return)

    return {
        'var_return_pct': var_return * 100,
        'var_dollar': var_dollar
    }
```

3. Monte Carlo VaR - Simulate thousands of scenarios, find the percentile.

```
def calculate_monte_carlo_var(returns, confidence_level=0.95,
    portfolio_value=100000,
                              n_simulations=10000):
    """Monte Carlo VaR: simulate random scenarios."""
    mu = returns.mean()
    sigma = returns.std()

    simulated_returns = np.random.normal(mu, sigma, n_simulations)
    var_return = np.percentile(simulated_returns, (1 - confidence_level) *
        100)
```

```
    var_dollar = portfolio_value * abs(var_return)

    return {
        'var_return_pct': var_return * 100,
        'var_dollar': var_dollar
    }
```

13.9.3 Comparing the Methods

From the notebook (AAPL, $100,000 portfolio, 95% confidence):

```
Historical:  $2,706
Parametric:  $2,816
Monte Carlo: $2,834
Average:     $2,785
```

All three methods give similar results for well-behaved distributions. The differences matter more for assets with non-normal returns (fat tails, skew).

13.9.4 Multi-Day VaR

To scale from 1-day to N-day VaR:

$$\text{VaR}_{n\text{-day}} = \text{VaR}_{1\text{-day}} \times \sqrt{n}$$

Horizon	Scaling	Example ($2,706 base)
1 day	×1.00	$2,706
1 week (5 days)	×2.24	$6,050
1 month (21 days)	×4.58	$12,399
1 quarter (63 days)	×7.94	$21,476

13.9.5 CVaR (Expected Shortfall)

VaR tells you the threshold. But what if you *exceed* it? **CVaR** (Conditional VaR) answers: "When we do exceed VaR, how bad is it on average?"

$$\text{CVaR} = E[\text{Loss} \mid \text{Loss} > \text{VaR}]$$

```
def calculate_cvar(returns, confidence_level=0.95, portfolio_value=100000):
```

```
    """CVaR: average loss when losses exceed VaR."""
    var_return = np.percentile(returns, (1 - confidence_level) * 100)

    # All returns worse than VaR
    tail_returns = returns[returns <= var_return]

    # Average of the tail
    cvar_return = tail_returns.mean()
    cvar_dollar = portfolio_value * abs(cvar_return)

    return {
        'var_dollar': portfolio_value * abs(var_return),
        'cvar_dollar': cvar_dollar
    }
```

From the notebook: VaR = $2,706, CVaR = $3,800+. When you *do* have a bad day, it's typically ~40% worse than VaR suggests.

13.9.6 VaR Limitations

VaR is a regulatory standard, but it has well-known weaknesses:

1. **It's a threshold, not a worst case.** The 5% of days worse than VaR could be *much* worse.
2. **It assumes "normal" markets.** During crises (2008, COVID), correlations spike and VaR models break down.
3. **Different methods give different answers.** Which one is "right"?

Layer 3: Portfolio Intelligence

13.10 Correlation and Diversification

Notebook: `07_correlation_and_diversification.ipynb`

You've heard "don't put all your eggs in one basket." Correlation explains *why* this works mathematically.

13.10.1 Correlation

Correlation measures how two assets move together:

13.10.2 Code: Correlation Calculation

Value	Meaning	Diversification Benefit
+1.0	Perfect positive - always move together	None
+0.7	Strong positive - usually move same direction	Limited
+0.3	Weak positive - slight tendency	Moderate
0.0	No relationship - independent	Strong
−0.3	Weak negative - slight opposite tendency	Very strong
−1.0	Perfect negative - always opposite	Maximum

```python
def calculate_correlation(ticker1, ticker2, period="2y"):
    """Calculate correlation between two stocks."""
    returns1 = get_returns(ticker1, period)
    returns2 = get_returns(ticker2, period)

    # Align to matching dates
    combined = pd.DataFrame({
        ticker1: returns1,
        ticker2: returns2
    }).dropna()

    correlation = combined[ticker1].corr(combined[ticker2])

    return correlation
```

Example from the notebook:

```
Correlation Matrix:
       AAPL  MSFT  GOOGL  NVDA   JPM   XOM   GLD   TLT
AAPL   1.00  0.43   0.44  0.34  0.31  0.23  0.01  0.09
JPM    0.31  0.30   0.29  0.27  1.00  0.31  0.01 -0.12
GLD    0.01  0.04   0.09  0.05  0.01  0.10  1.00  0.10
TLT    0.09 -0.01  -0.01 -0.04 -0.12 -0.02  0.10  1.00
```

What to notice: Tech stocks (AAPL, MSFT, NVDA) correlate with each other (0.34-0.51). Gold (GLD) and bonds (TLT) have near-zero correlation with everything - they provide real diversification.

13.10.3 The Math of Diversification

For two assets, portfolio variance is:

$$\sigma_p^2 = w_1^2\sigma_1^2 + w_2^2\sigma_2^2 + 2w_1w_2\sigma_1\sigma_2\rho_{1,2}$$

The key insight: When correlation (ρ) is less than 1, the portfolio volatility is *less* than the weighted average of individual volatilities. The lower the correlation, the bigger the diversification benefit.

13.10.4 Code: Diversification Benefit

```
def analyze_portfolio_diversification(holdings, period="2y"):
    """Analyze diversification quality of a portfolio."""
    # ... get returns, calculate correlation matrix ...

    # Weighted average individual volatility
    weighted_avg_vol = sum(holdings[t] * individual_vols[t] for t in
        tickers)

    # Actual portfolio volatility (using covariance matrix)
    port_var = np.dot(weights.T, np.dot(cov_matrix, weights))
    port_vol = np.sqrt(port_var) * 100

    # Diversification ratio (> 1 means diversification is working)
    div_ratio = weighted_avg_vol / port_vol

    return {
        'weighted_avg_vol': weighted_avg_vol,
        'portfolio_vol': port_vol,
        'diversification_ratio': div_ratio
    }
```

Example from the notebook:

```
PORTFOLIO 1: All Tech (AAPL, MSFT, GOOGL, NVDA)
  Weighted Avg Vol: 33.1%
  Actual Portfolio Vol: 25.2%
  Diversification Ratio: 1.31x
  Average Correlation: 0.426

PORTFOLIO 2: Multi-Asset (AAPL, JPM, XOM, GLD, TLT)
  Weighted Avg Vol: 21.4%
  Actual Portfolio Vol: 12.0%
  Diversification Ratio: 1.78x
  Average Correlation: 0.102
```

Portfolio 2 has a much higher diversification ratio - its actual volatility is 44% lower than the weighted average of its holdings. Portfolio 1 gets some benefit from diversification (1.31x), but the tech stocks are too correlated to reduce risk substantially.

13.10.5 Correlation During Crises

The notebook includes a critical warning: **correlations increase during market stress.**

```
Normal Days:   Average correlation 0.170
Stressed Days: Average correlation 0.245 (+0.074)

Correlations INCREASE during stress.
Diversification benefit is REDUCED when you need it most.
```

This is called "correlation breakdown" or "contagion." Assets that seem uncorrelated in calm markets often start moving together during crashes - precisely when you need diversification most.

Why this matters for agents: The Portfolio Agent uses covariance matrices to calculate portfolio volatility. The `portfolio_volatility` Code Tool computes the double sum $\sum_i \sum_j w_i w_j \text{Cov}(i, j)$ - the full portfolio variance formula. This is why portfolio volatility differs from a simple weighted average.

13.11 Sector Analysis

Notebook: `09_sector_analysis.ipynb`

You might own 20 different stocks and think you're diversified. But if they're all tech companies, you have **sector concentration risk**.

13.11.1 The 11 GICS Sectors

Sector	Description	Examples
Technology	Software, Hardware, Semiconductors	AAPL, MSFT, NVDA
Healthcare	Pharma, Biotech, Medical Devices	JNJ, PFE, UNH
Financials	Banks, Insurance, Asset Managers	JPM, BAC, GS
Consumer Disc.	Retail, Auto, Entertainment	AMZN, TSLA, NKE
Consumer Staples	Food, Beverages, Household	PG, KO, WMT
Energy	Oil, Gas, Renewables	XOM, CVX, COP
Industrials	Manufacturing, Aerospace	BA, CAT, UPS
Materials	Mining, Chemicals	LIN, SHW, FCX
Utilities	Electric, Gas, Water	NEE, DUK, SO
Real Estate	REITs, Property	AMT, PLD, SPG
Comm. Services	Telecom, Media, Internet	GOOGL, META, DIS

The mental model: Think of the economy as a pie. Each sector is a slice. Different slices grow at different times - tech booms in growth periods, utilities hold up in recessions, energy moves with commodity prices.

13.11.2 Code: Sector Breakdown

```
def analyze_portfolio_sectors(holdings):
    """Analyze sector breakdown of a portfolio."""
    positions = []

    for ticker, shares in holdings.items():
        stock = yf.Ticker(ticker)
        info = stock.info
        price = info.get('currentPrice') or info.get('regularMarketPrice')

        positions.append({
            'ticker': ticker,
            'sector': info.get('sector', 'Unknown'),
            'industry': info.get('industry', 'Unknown'),
            'value': shares * price
        })

    df = pd.DataFrame(positions)
    df['weight'] = df['value'] / df['value'].sum() * 100

    return df
```

13.11.3 Concentration Risk vs. Benchmark

The notebook compares your portfolio's sector weights to the S&P 500:

```
Sector          Portfolio    S&P 500     Diff        Risk
Technology          51.9%      30.0%   +21.9%    HIGH
Financial Services  13.7%       0.0%   +13.7%    MODERATE
Energy              10.2%       4.0%    +6.2%    MODERATE
Industrials          0.0%       8.0%    -8.0%    MODERATE
```

Risk thresholds:

- Within ±5% of benchmark → LOW risk
- 5-15% over/underweight → MODERATE risk
- >15% over/underweight → HIGH risk

13.11.4 HHI: The Concentration Index

The notebook and the Portfolio Agent both use the **Herfindahl-Hirschman Index (HHI)** to measure concentration:

$$\text{HHI} = \sum_{i=1}^{n} w_i^2$$

```
hhi = sum(weight ** 2 for weight in portfolio_weights)
effective_n = 1 / hhi  # "acts like" this many equal-weight holdings
```

HHI	Effective N	Concentration
1.0	1	One stock (maximum)
0.25	4	High
0.15	6-7	Medium
0.10	10	Low (diversified)

Example: If you have three stocks at 50%, 30%, 20%:

- $\text{HHI} = 0.50^2 + 0.30^2 + 0.20^2 = 0.25 + 0.09 + 0.04 = 0.38$
- Effective N $= 1/0.38 = 2.6$ holdings
- Even though you own 3 stocks, your portfolio "acts like" you own 2.6 due to the imbalanced weights.

13.11.5 Sector Attribution

The notebook calculates how much of your portfolio's return came from each sector:

$$\text{Sector Contribution} = \text{Sector Weight} \times \text{Sector Return}$$

Example:

```
Technology:      51.9% weight x 17% return = +8.80% contribution
Communication:   10.2% weight x 71% return = +7.27% contribution
```

Communication Services (GOOGL) contributed almost as much as Technology despite being one-fifth the weight - because its return was much higher.

Why this matters for agents: The `sector_allocation` and `concentration` calculations in the Portfolio Agent's Code Tool use these exact concepts - sector grouping, HHI, and weight analysis.

13.12 Portfolio Rebalancing

Notebook: `11_portfolio_rebalancing.ipynb`

As different assets perform differently, your portfolio weights **drift** from your targets. Rebalancing brings them back.

13.12.1 Why Drift Happens

Start with 60% stocks / 40% bonds. If stocks rise 20% and bonds rise 5%:

- Stocks: $60,000 → $72,000
- Bonds: $40,000 → $42,000
- New total: $114,000
- New weights: **63.2% stocks / 36.8% bonds**

Your portfolio has drifted. Without rebalancing, the winning asset keeps growing as a share of your portfolio, increasing concentration risk.

13.12.2 Three Rebalancing Strategies

1. Calendar Rebalancing - Rebalance at fixed intervals (monthly, quarterly, annually).

- Pros: Simple, predictable
- Cons: May trade when unnecessary, may miss big moves

2. Threshold Rebalancing - Rebalance when any asset drifts beyond a set threshold (e.g., ±5%).

- Pros: Only trades when needed, responsive to market moves
- Cons: More monitoring, may trade frequently in volatile markets

3. Hybrid - Check at fixed intervals, but only rebalance if threshold is breached.

This is usually the best approach.

13.12.3 Code: Rebalancing Trades

```
def calculate_rebalancing_trades(current_weights, target_weights,
    portfolio_value):
```

```
    """Calculate the trades needed to rebalance."""
    trades = {}

    for ticker in target_weights:
        current_value = current_weights[ticker] * portfolio_value
        target_value = target_weights[ticker] * portfolio_value
        difference = target_value - current_value

        trades[ticker] = {
            'action': 'BUY' if difference > 10 else 'SELL' if difference <
                -10 else 'HOLD',
            'amount': abs(difference)
        }

    return trades
```

The $10 threshold: Prevents micro-trades that aren't worth executing. In real life, transaction costs and tax implications make tiny adjustments counterproductive.

13.12.4 The 5% Rule

The notebook compares strategies and finds a consistent result:

Strategy	# Trades	Total Traded	Final Value
No Rebalancing	0	$0	$132,252
Monthly	34	$45,566	$130,937
Quarterly	13	$31,089	$131,358
Threshold 5%	4	$22,844	$131,809
Threshold 10%	1	$12,610	$131,839

The insight: A 5% threshold provides good drift control with minimal trading. Monthly rebalancing generates 34 trades for no better results.

13.12.5 Tax-Efficient Rebalancing

The notebook introduces a smarter approach - using new contributions to rebalance:

```
# Portfolio drifted: 70% stocks, 30% bonds (target: 60/40)
# New contribution: $10,000

# Without new money: SELL $4,000 stocks, BUY $14,000 bonds (taxable event)
# With new money: Direct $10,000 to bonds, only SELL $4,000 stocks

# Tax savings: Reduced selling by using new money to buy underweight assets
```

Why this matters for agents: The `rebalance` calculation in the Portfolio Agent's Code Tool computes target values, current values, and the difference - exactly this algorithm. The agent can say "sell 37.5 shares of AAPL and buy 55.2 shares of BND to reach equal weight."

Putting It All Together

13.13 The Complete Risk Dashboard

Here's what your agent produces when all these concepts work together:

```
PORTFOLIO RISK REPORT
==============================================

Portfolio Value: $100,000

POSITION VALUES:
  AAPL: $25,000 (25.0%)
  NVDA: $25,000 (25.0%)
  JPM:  $25,000 (25.0%)
  XOM:  $25,000 (25.0%)

CONCENTRATION:
  HHI: 0.25 -> Effective N: 4.0 holdings
  Concentration Risk: MEDIUM

SECTOR ALLOCATION:
  Technology:         50.0% -> HIGH (vs 30% benchmark)
  Financial Services: 25.0%
  Energy:             25.0%

RISK METRICS:
  Portfolio Volatility: 16.3%
  Weighted Avg Vol:     25.2%
  Diversification Benefit: 8.9% reduction

  Sharpe Ratio:  1.65
  Sortino Ratio: 1.60

  95% VaR (1-day): $1,420
  Max Drawdown:    -28.7%

REBALANCING:
```

```
All positions within 5% threshold - no action needed.
```

Every number in that report maps to a concept from this chapter:

Report Section	Concept	Notebook	Agent Tool
Position Values	Portfolio value, weights	02, 03	`weights`
Concentration	HHI, effective N	09	`concentration`
Sector Allocation	GICS sectors, benchmark	09	`sector_allocation`
Portfolio Vol.	Covariance, diversification	07	`portfolio_volatility`
Sharpe / Sortino	Risk-adjusted returns	05, 08	`portfolio_sharpe`
95% VaR	Historical percentile	10	Calculate Metric tool
Max Drawdown	Peak-to-trough decline	06	Calculate Metric tool
Rebalancing	Threshold, trade calc.	11	`rebalance`

13.14 The Mental Model: Three Questions

When your agent analyzes a portfolio, it's answering three questions:

1. **What do I own?** (Layer 1)
   - Positions, values, weights, sectors
2. **How risky is it?** (Layer 2)
   - Volatility, Sharpe, Sortino, VaR, drawdown
3. **Is it well-constructed?** (Layer 3)
   - Correlation, diversification benefit, concentration, rebalancing needs

These three questions map directly to the three tool types in your agent:

- **HTTP Tools** answer question 1 (fetch current data)
- **Code Tools** answer question 2 (calculate risk metrics)
- **Both together** answer question 3 (portfolio-level analysis)

14

How This Book Was Built - Working With AI

When I shared an early draft with a friend, his first question was:

> *How much of this was AI?*

For a while, it made me a little sensitive. Even made me question if I should share this book.

Not because the question was unfair. It's a reasonable thing to ask. It's almost automatic to question the origins of any work these day.

But because it implied something I wasn't sure how to answer cleanly. If I said "a lot," it sounds like the book isn't mine. And perhaps just slop. If I said "not much," that's really not honest either.

So let me just show you. Full transparency.

14.1 AI or me?

This book is part me, part AI. And I want to show you exactly where the line is.

What's mine:

- The architecture. The four-pattern framework. The decision to organize the book around Tool Calling, ReAct, CodeAct, and Orchestration - that came from reading papers, building prototypes. No model suggested that structure.

- The frameworks. The **Hamburger Principle** mental model came from a LinkedIn post I did to explain how I use LLMs. The **Complexity Ladder** came from watching people skip straight to agents when a simple API call would suffice. These are my patterns that I got from observation and learning, not generation from a prompt.
- The judgment calls. What to include. What to leave out. When to go deep on code and when to step back and explain why it matters. The decision to start with the trust problem - not with "what is an LLM?" The decision to end with an assessment of what the reader can and can't build.
- The weird voice. The tone. The "I like simple and boring." That's not a style a model learned. That's the thing I have to tussle with every generation of LLMs. LLMs think they know too much.

What's AI:

- Drafting speed. First drafts of chapters, generated from detailed outlines I wrote. I'd specify the concept, the framework, the examples, the level - and Claude would produce a draft I could shape.
- Code scaffolding. The notebook code, the tool definitions, the API integrations. I described what each tool should do. AI wrote the implementation. I tested it, caught the errors, fixed the edge cases.
- Production work. Converting eleven chapters from Markdown to LaTeX is no joke. The grunt work that would have made me give up. AI also did the documentation of the n8n workflows I strung together, and of the financial concepts from my notebooks.
- Research synthesis. Pulling together documentation, API references, library specifications. Summarizing what I needed so I could decide what mattered.

If you've read this far in the book, you recognize the pattern. It's the one I've been talking about, incessantly, for the last eleven chapters.

The Hamburger Principle - applied to writing the book itself.

Claude was still the bun. It parsed my instructions, understood what I wanted, generated prose, and communicated ideas in readable English. That's what LLMs do - the language layer. Same role as in every agent pattern in this book.

The meat was my domain knowledge and the tools - LaTeX compilation, yfinance APIs, MCP servers. The real things that the language wrapped around.

And the vegetables? The infrastructure in between. The project files that kept everything organized. The version tracking. The convention lists. The consistency checks.

I was the chef. Not a layer of the hamburger - the one directing the whole thing. Choosing the ingredients, deciding what goes in, what comes out, and whether the result is any good. And the one getting frustrated at the bun.

The same approach I taught you in Chapter 3. Applied to its own creation. Quite meta right? I am quite pleased about this weird recursion.

14.2 How I Actually Built It

Here's the workflow. Not the idealized version. The real one.

14.2.1 The Setup

I work with two views of the same files. **Obsidian** - a free note-taking app that reads plain Markdown files from a folder - gives me a clean writing environment. **VS Code** with Claude Code in the terminal gives me access to a development environment with LLMs. Same files, two lenses. I write and edit in Obsidian. I research, draft, and build with Claude Code. You don't to use Claude Code. There's also Gemini CLI, Codex. You can even replace the Claude in Claude Code with other models like Kimi, Qwen or Minimax.

Every key project folder has three files that serve as persistent memory:

File	Purpose
CLAUDE.md	Instructions for Claude. How I want it to behave for this project, what context matters, what to ignore.
PIN.md	Project state. What's been decided, where we are, what's next.
LESSONS.md	What I've learned. Patterns to reuse, mistakes to avoid.

These files are the difference between productive sessions and wasted ones. Without them, every conversation with AI starts from scratch. With them, Claude picks up where we left off. It knows the project. It knows my preferences. It knows what's been decided.

14.2.2 The Planning Phase

Before AI wrote a single word of this book, I wrote a detailed plan. Co-created with the chat interface version of Claude.

That plan specified every chapter. The opening hook for each. The frameworks to introduce. The code examples to include. The level of detail. The voice. The audience. The page targets. The cross-references between chapters.

AI didn't write that plan. AI couldn't write that plan. It can't read my mind. Always remember that when you work with AI.

14.2.3 The Writing Phase

For each chapter, the workflow looked like this:

1. **I wrote the outline.** Not "write a chapter about tool calling." More like: here's the concept, here's the framework I want to use, here's the opening hook, here are the specific code examples, here's the level I'm writing at, here's what I don't want.

2. **Claude drafted.** Based on that detailed brief, it produced a first draft. Usually 70–80% of the way there.

3. **I shaped it.** Cut what didn't sound like me. Added the personal touches - the "I like simple and boring," the specific anecdotes, the real moments. Restructured paragraphs that were technically correct but didn't flow. Removed the over-explaining that AI tends to do. And toned down overconfident lines.

4. **We iterated.** "This section is too long." "The code example needs error handling." "Switch the order of these two concepts." "The tone is too formal here - make it more conversational."

14.2.4 Conversion

The Markdown chapters needed to become LaTeX for the printed PDF. Multiple files. Hundreds of pages. Dozens of tables, code blocks, ASCII diagrams, and math formulas.

I am proficient in Latex. As I used it during PhD studies before ChatGPT. But still, why waste the precious moments we have on earth tussling with that.

Here's what directing looked like:

I'd say: "Convert chapter07.tex from CHAPTER7.md."

Claude would read the Markdown source, apply the established LaTeX conventions and produce the `.tex` file.

Then I'd review. "Now change the examples so it doesn't use simulated data." Claude would identify the hardcoded return dictionaries, replace them with live yfinance API calls, and update the verbose output notes to reflect that values would differ with real market data.

This happened many times. Each conversion built on the conventions established in the previous one. Each review caught something. Each iteration improved the system.

That's not "AI wrote my book." That's directing an AI to do grunt work while I focused on what actually matters - whether the content makes sense and teaches what it needs to teach.

14.2.5 The Tools

I use MCP servers - plugins that give Claude Code access to external tools. Web search for current documentation. ArXiv for research papers. A persistent memory graph for facts that need to survive across sessions.

These are the same tool-calling patterns I taught you in Chapter 5. The AI agent (Claude Code) calls tools (search, fetch, memory) to get real information instead of guessing. The principle scales from stock price lookups to book production.

Note: This is not foolproof. An LLM agent can still hallucinate after using a tool, especially if the wrong information, or no information is fetched. So always try to check, if its something that matters.

14.3 What AI Could Not Do

The Hamburger Principle. AI didn't invent that. I did. In a LinkedIn post.

The Complexity Ladder. AI didn't see that. I built it from reading and watching.

"When NOT to use agents." This had to be in, AI or not.

The personal notes. "I like simple and boring." "confident nonsense meets real money."

Knowing what to leave out. The early drafts were longer. AI is generous with words. It explains things three ways when once is enough. The skill of cutting fluff becomes more and more important.

The book itself is the proof. Read it. Find the hamburger joke. Find the keto line. Find "I like simple and boring." Find the personal note about leaving regulation. That's not a model talking. That's me.

And finally. The checks by a human. All me. I suspect there may still be some errors, so let me know if you spot any.

And if you find any part of it that feels like slop, let me know too.

14.4 What AI Made Possible

What AI gave me was time.

Without it, this book would have taken six months. With it, weeks.

That matters. Not because speed is everything. But because I'm one person. Living a life. AI didn't replace the thinking. It compressed the production. It let me spend my time on the parts that actually require me - the architecture, the frameworks, the voice, the judgment - instead of the parts that don't.

The numbers: Close to 200 pages. Eleven chapters plus a preface and three bonus chapters. Working code in more than twenty Jupyter notebooks. Four n8n workflow templates. A LaTeX-typeset PDF with consistent formatting across multiple source files. A financial concepts chapter.

One person. 1-2 weeks.

14.5 The Part I Find Funny

You might have noticed the recursion.

This is a book about directing AI agents to do useful work. It was built by directing an AI agent to do useful work.

The four patterns from the book? I used all of them:

- **Tool Calling** - Claude fetched documentation, API references, and library specifications. Real data, not hallucinated guesses.
- **ReAct** - Multi-step reasoning through chapter dependencies. "Read this source file, apply these conventions, update the tracker."
- **CodeAct** - Generating LaTeX, writing math formulas, building tables, checking code.
- **Orchestration** - Combining all patterns across more than fifty files over multiple sessions with persistent memory.

The hamburger principle, applied to its own creation story.

I didn't plan it that way. But in retrospect, it couldn't have been any other way. The book teaches what I practice. And I practice what the book teaches.

Kind of funny. In a geeky, meta way.

About Me

I bridge AI, quantitative finance, and risk management (or at least I think I do). Most recently, I led AI risk supervision at the Monetary Authority of Singapore (MAS), where I developed Singapore's first AI risk management guidelines for the financial sector. Before that, I was division head for investment risk management, overseeing risk management of Singapore's foreign reserves. I also spent years deep in the weeds of Basel 2, 2.5, and 3 capital rules, model risk audits, and banking policy. Basically, I did work so technical that it clears a room at parties when people ask what I do.

I have a PhD in Computer Science, where I researched deep learning for networks, time series, and multimodal data, publishing 11 papers at venues like ACL, ACM Transactions on the Web, and IEEE Big Data, including an honourable mention at ACM IUI. I also have Masters degrees in Financial Engineering and Knowledge Engineering from NUS, and an Electrical Engineering degree from the University of Toronto. Yes, that's a lot of degrees. No, I don't know when to stop.

I've taught Python, machine learning, deep learning, and AI at NUS, SMU, SUSS, Nanyang Polytechnic, and MAS Academy, to audiences ranging from central bankers to healthcare workers to policemen. I'm currently developing AI risk management materials for

the Cambridge Centre for Alternative Finance. I actually enjoy explaining complicated things to rooms full of people who didn't ask for it. Masochist, I know.

Before all of this, I spent four years at the Ministry of Information, Communications and the Arts, where I helped develop Singapore's creative industries strategy and created the Creative Youth eXchange, a regional design competition that somehow became a reality TV programme. I also helped with a series of Mr Kiasu books from 2017-2019, which may be the most widely read publication I've had a hand in.

Beyond finance and technology, I'm a watercolor artist and illustrator whose work has been commissioned by Wallpaper, Tatler, Jetstar, and the Esplanade, hard proof that not everything I do involves code.

This book was written carefully, but with AI's help. There's a whole chapter 14 about how. I practise what I preach.

Follow me at linkedin.com/in/garyang/
Subscribe to my newsletter at simplyboring.ai/
Email me at gary@quaintitative.com

FIN

Simply Boring AI

by quaintitative