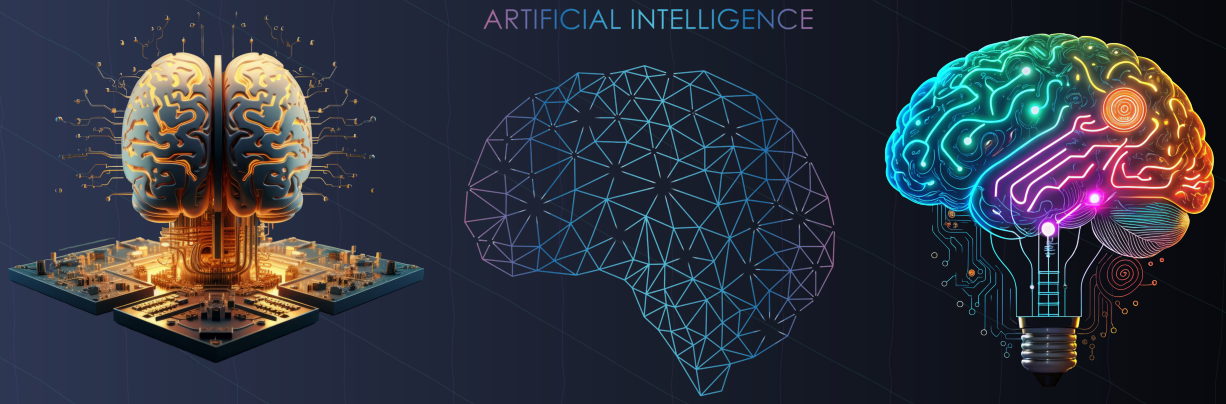


Multi-Agent Deal Discovery System

A Comprehensive Python Programming Guide

From Architecture to Implementation:

Machine Learning, Web Scraping, and System Design

SIJAN PAUDEL

Generated Technical Documentation

Contents

List of Figures

List of Tables

Listings

Preface

This comprehensive guide provides an in-depth analysis of a sophisticated multi-agent deal discovery system implemented in Python. The system demonstrates advanced software engineering principles, modern Python programming techniques, and practical machine learning integration.

What You'll Learn

This documentation covers:

- **Advanced Python Programming**: Object-oriented design, inheritance patterns, composition, and modern Python idioms
- **System Architecture**: Multi-agent design patterns, component interaction, and scalable system design
- **Machine Learning Integration**: Ensemble methods, LLM APIs, vector databases, and traditional ML models
- **Web Technologies**: RSS parsing, HTML scraping, HTTP clients, and API integration
- **Production Practices**: Error handling, configuration management, logging, and performance optimization

How to Use This Guide

The guide is structured in seven comprehensive parts:

1. **System Overview**: High-level architecture and component relationships
2. **File Analysis**: Detailed examination of each Python file
3. **Class Hierarchy**: Object-oriented design patterns and inheritance
4. **Function Breakdown**: Method-by-method analysis with execution details
5. **Python Concepts**: Advanced language features with practical examples
6. **Agent Interactions**: Workflow analysis and sequence diagrams
7. **Summary & Knowledge**: Complete system recap and learning outcomes

Each part builds upon the previous sections while remaining accessible for focused study of specific topics.

Prerequisites

Readers should have:

- Basic Python programming knowledge
- Understanding of object-oriented programming concepts
- Familiarity with common Python libraries (requests, json, etc.)
- Basic knowledge of machine learning concepts (optional but helpful)

Code Conventions

Throughout this guide:

- Code snippets are syntax-highlighted for readability

  Important concepts are marked with ratings (★☆☆☆☆ to ★★★★★)
- Cross-references link related concepts across chapters
- Diagrams illustrate complex system interactions

Chapter 1

LLM Cheat Sheet

1.1 Handling Imports from Parent Folders in Python and Jupyter

When a Python script or Jupyter Notebook is located in a subfolder, modules in a parent folder may not be automatically discoverable by Python. The following methods illustrate how to properly import them.

1.1.1 Using ___file___ in Python Scripts

For scripts executed directly, ___file___ contains the path of the script. Use it to construct the parent folder path:

Python Script: Using ___file___

```
import sys
import os

# Add parent folder to Python path
sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), '..')))

# Import module from parent folder
from LLMengineering import key_utils
```

Explanation:

- `os.path.dirname(__file__)`: folder of the current script.
- `os.path.join(..., '..')`: go one level up.
- `os.path.abspath(...)`: get absolute path.
- `sys.path.append(...)`: add this path to Python's module search paths.

Using `os.getcwd()` in Jupyter Notebooks

In Jupyter, ___file___ is not defined. Use the current working directory instead:

Jupyter Notebook: Using os.getcwd()

```
import sys
import os

# Add parent folder of current working directory
sys.path.append(os.path.abspath(os.path.join(os.getcwd(), '..')))

# Import module from parent folder
from key_utils import some_function
```

Notes:

- `os.getcwd()`: returns the current working directory of the notebook.
- `..` moves one level up; `../..` moves two levels up.
- This approach is portable for notebooks without relying on __file__.

Using Absolute Paths (Less Portable)

Absolute Path Approach

```
import sys
sys.path.append("/run/media/sijanpaudel/New Volume/New folder/LLMengineering")

from key_utils import some_function
```

Caution: This works immediately but is hard-coded. If the project moves, the path must be updated.

Recommended Long-Term Approach

1. Add an empty __init__.py file in the parent folder (e.g., `LLMengineering/`) to make it a package. 2. Use relative or absolute imports within scripts:

Long-Term Package Approach

```
    from LLMengineering import key_utils
```

This makes imports clean, portable, and avoids repeated `sys.path.append` hacks.

1.2 Comparing LLMs – The Basics (1)

Feature	Description / Notes
Open-source or Closed	Whether the model's weights, architecture, and training code are publicly available. Examples: GPT-4 (closed), LLaMA 2 (open).
Release Date & Knowledge Cut-off	The date the model was released and the latest point in time the training data covers. Important for up-to-date responses.
Parameters	Number of trainable weights in the model (e.g., billions of parameters). Determines model capacity.
Training Tokens	Amount of text data used during training, usually in billions of tokens. More tokens generally improve performance.
Context Length	Maximum input length (in tokens) the model can handle in a single prompt. Longer context allows better understanding of large inputs.

1.3 Comparing LLMs – The Basics (2)

Feature	Description / Notes
Inference Cost	Computational resources required to generate output (GPU, CPU usage).
API Charge / Subscription / Runtime Compute	How much it costs to access the model through a cloud service.
Training Cost	Cost to pretrain the model, including compute, electricity, and infrastructure.
Build Cost	Cost to fine-tune, deploy, or customize the model.
Time to Market	How quickly you can deploy and use the model in production.
Rate Limits	Restrictions on API usage, e.g., calls per minute or per day.
Speed & Latency	How fast the model responds, depends on model size, hardware, and context length.
License	Terms for use, redistribution, and commercial deployment. Some open-source licenses restrict commercial use.

1.4 Chinchilla Scaling Law

Key Principle: Number of model parameters should scale roughly proportional to the number of training tokens.

Implications:

- Increasing model size without enough data leads to plateaued or degraded performance.
- Having lots of training data with a small model underperforms; parameters must scale up.

Rule of Thumb: Let N = model parameters, D = training tokens. For optimal training:

$$N \propto D$$

Example:

- Doubling model parameters $\rightarrow$ need roughly double the training tokens.
- Insufficient tokens $\rightarrow$ diminishing returns.

Additional Notes:

- Smaller, well-trained models can outperform larger, undertrained ones.
- Helps determine compute-efficient configurations for new LLMs.

1.5 Common LLM Benchmarks

Benchmark	Focus / Task	Description
MMLU	Knowledge across multiple subjects	Assess general knowledge and reasoning; used to compare models like GPT-3, Chinchilla, and Gopher.
BIG-bench	Broad suite of diverse reasoning tasks	Tests reasoning, factual knowledge, ethics, math, code; hundreds of tasks evaluating beyond narrow QA.
HellaSwag	Commonsense reasoning	Multiple-choice questions for everyday situations; measures ability to predict plausible outcomes.
TruthfulQA	Factual accuracy / truthfulness	QA tasks designed to detect hallucinations; evaluates honesty of LLM answers.
WinoGrande / Winograd Schema Challenge	Pronoun resolution / coreference	Tests commonsense reasoning and context understanding; resolves ambiguous references.
ARC	Science and reasoning	Multiple-choice science questions; evaluates problem-solving and reasoning in STEM.
HumanEval	Coding and code generation	Tests Python programming ability; measures functional correctness of generated code.

1.6 Specific Benchmarks

Benchmark	What's Being Evaluated	Description
ELO	Model ranking / performance consistency	Evaluates models via pairwise comparisons; creates a relative ranking of LLMs across multiple tasks.
HumanEval	Code generation / functional correctness	Tests an LLM's ability to write Python functions that pass unit tests; measures coding logic and correctness.
Multipl-E	Multimodal reasoning	Evaluates LLMs on tasks combining text and images (or multiple modalities); measures reasoning and comprehension across modalities.

1.7 Limitations of Benchmarks

Limitation	**Explanation**
Narrow focus	Many benchmarks test only specific skills (e.g., coding, factual QA, commonsense), not overall intelligence or adaptability.
Static datasets	Benchmarks are fixed in time, so models trained after the cut-off may have an unfair advantage or miss newer knowledge.
Lack of real-world context	Benchmarks often use idealized tasks, not messy, ambiguous, or multi-step real-world scenarios.
Gaming / overfitting	Models can be fine-tuned or prompted to specifically excel on benchmark tasks without improving general capabilities.
Limited multimodality	Most benchmarks focus on text-only tasks; few measure image, audio, or multimodal reasoning.
Subjectivity	Some benchmarks (e.g., ethics, creativity, hallucination detection) are hard to score objectively.
Compute bias	Larger models may perform better mainly due to size, not reasoning ability, skewing benchmark results.
Hard to measure nuanced reasoning	Benchmarks mostly measure surface correctness; they often cannot capture multi-step reasoning, context-dependent judgment, creativity, or reasoning accuracy vs. fluency.

1.8 Advanced Benchmarks for Large Language Models

Benchmark	Focus / Task	Description / Meaning
GPQA	Graduate-level question answering	Evaluates performance on graduate-level tests with 448 expert questions. Non-PhD humans score only 34% even with web access. Measures LLM ability to handle highly specialized knowledge.
BBHard	Future capabilities	Includes 204 tasks previously thought beyond LLM capabilities. Designed to test reasoning, logic, and generalization at a next-level difficulty.
Math Lv 5	High-school math competition problems	Measures model's ability to solve advanced math problems requiring multi-step reasoning and problem-solving skills. Useful for chain-of-thought evaluation.
IFEval	Instruction following	Tests the model's ability to follow complex instructions, e.g., "write more than 400 words" and "mention AI at least 3 times." Evaluates comprehension and compliance with nuanced prompts.
MuSR	Multistep soft reasoning	Assesses logical deduction and multi-step reasoning. Example: analyzing a 1,000-word story and identifying "who has means, motive, and opportunity." Tests reasoning beyond surface facts.
MMLU-PRO	Harder MMLU version	Advanced, cleaned-up version of MMLU with questions having 10 possible answers instead of 4. Evaluates deeper knowledge, multi-choice reasoning, and generalization.

Notes:

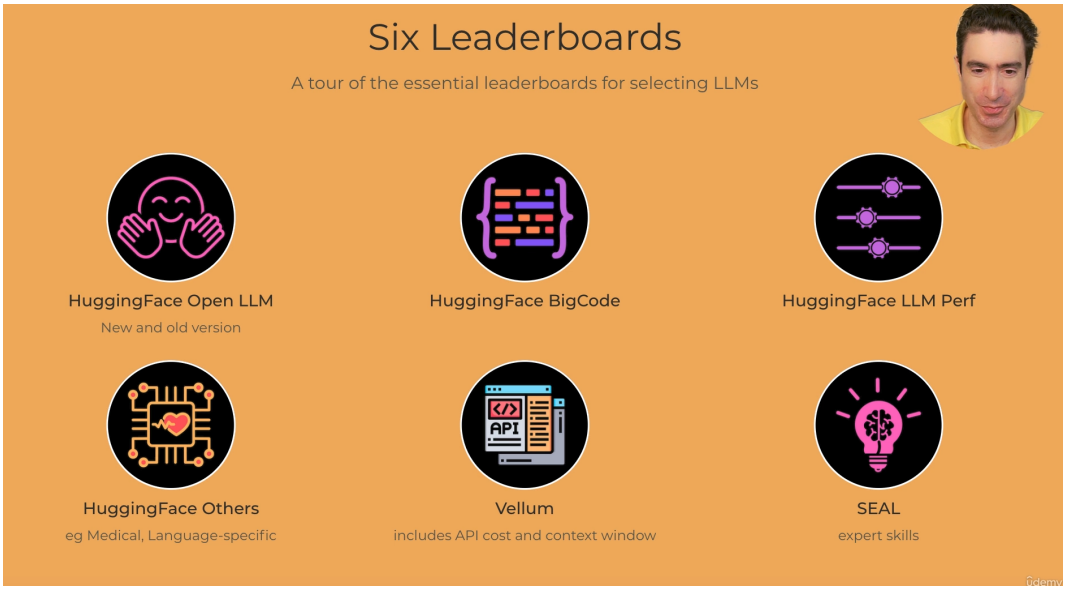
- These benchmarks are considered "next-level" because they test advanced reasoning, multi-step logic, and high-level knowledge.
- Useful for evaluating models that go beyond standard QA, commonsense reasoning, or basic coding tasks.
- Many of these benchmarks also measure compliance with complex instructions and reasoning under uncertainty.

Benchmark	**Purpose / Examples**
GPQA	Specialized knowledge, expert-level Q&A, academic reasoning, physics, chemistry, biology expertise. **Examples:** Explain the biochemical steps of glycolysis; Describe Newton's laws with practical applications; Explain photosynthesis in detail.
BBHard	Advanced reasoning, logic, generalization in challenging scenarios. **Examples:** Predict how a hypothetical AI agent could optimize a supply chain in a novel scenario; Design a strategy to minimize energy consumption in an unfamiliar industrial process; Solve a logic puzzle with multiple constraints.
Math Lv 5	Complex problem solving, chain-of-thought evaluation, mathematical reasoning. **Examples:** Solve: If $x^2 - 5x + 6 = 0$, find all integer solutions; Evaluate combinatorial problems like "How many ways can 5 books be arranged on a shelf?"; Solve calculus problems requiring multiple steps.
IFEval	Precise instruction compliance, comprehension, content generation with constraints. **Examples:** Write a 450-word essay on renewable energy mentioning AI at least 3 times; Summarize a research paper in 300 words including key terms; Rewrite a paragraph to follow a formal tone while keeping original meaning.
MuSR	Multi-step reasoning, deduction, understanding narratives beyond surface facts, solving prime puzzles and reasoning tasks. **Examples:** Analyze a 1,000-word mystery story to determine "who has means, motive, and opportunity"; Identify the next prime number in a complex sequence; Solve multi-step logic puzzles.
MMLU-PRO	Deep knowledge, advanced multi-choice understanding, broader knowledge evaluation, general language understanding. **Examples:** Choose the best explanation for why the sky is blue from 10 options; Identify the correct historical fact among 10 alternatives; Evaluate grammar and style in multiple-choice questions.

Tips to distinguish benchmarks:

- **GPQA** – focus on subject expertise.
- **BBHard** – focus on novel reasoning and logic.
- **IFEval** – focus on following instructions accurately.
- **MuSR** – focus on multi-step deduction and problem solving (e.g., prime puzzles, mysteries).
- **MMLU-PRO** – focus on broad knowledge and multiple-choice reasoning.

1.9 Leaderboard Image



Leaderboard	**Purpose / Use**
HuggingFace Open LLM	Tracks performance of open-source LLMs (new and old versions). Useful for selecting models for research or deployment with full weight access.
HuggingFace BigCode	Focused on code generation LLMs. Helps compare models for coding tasks, programming language coverage, and code quality.
HuggingFace LLM Perf	Benchmarking general LLM performance across various NLP tasks. Useful for measuring speed, accuracy, RAM memory usage and general reasoning ability.
HuggingFace Others	Specialized leaderboards, e.g., medical, domain-specific, or language-specific models. Guides selection for niche applications.
Vellum	Includes context window, API cost, and usage constraints. Useful for developers to choose models based on practical deployment factors.
SEAL	Focused on expert skills and high-level reasoning. Useful for selecting LLMs that excel in professional or complex knowledge domains.

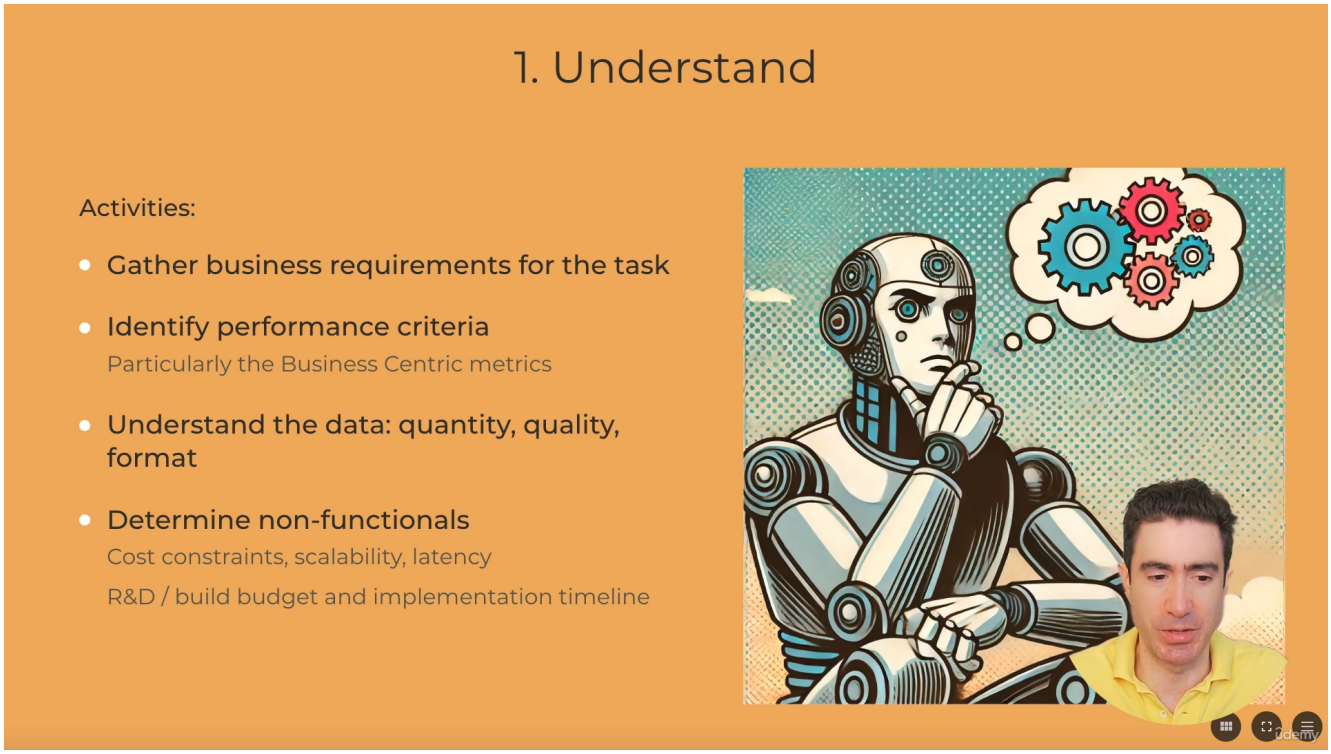
1.10 A 5-Step Strategy for Using AI Models

This is a simple guide to choosing, training, and using a Large Language Model (LLM) to solve a real-world business problem.

1.10.1 Understand (Figure Out What You Need)

Before you start, you need a clear plan. Think about what you're trying to achieve.

- **Gather Business Needs:** What is the specific problem you want the AI to solve? (e.g., "Answer customer support questions automatically.")
- **Define Success:** How will you know if the AI is doing a good job? Focus on what matters to the business, not just technical scores. (e.g., "Reduce customer wait times by 50
- **Look at Your Data:** What information do you have? Is it a lot? Is it messy or clean? What format is it in?
- **Consider the Limits:** Think about the practical stuff.
  - How much money can you spend?
  - Does it need to be super fast?
  - How many people will be using it?
  - What is the deadline to get this working?



1.10.2 Prepare (Do Your Homework)

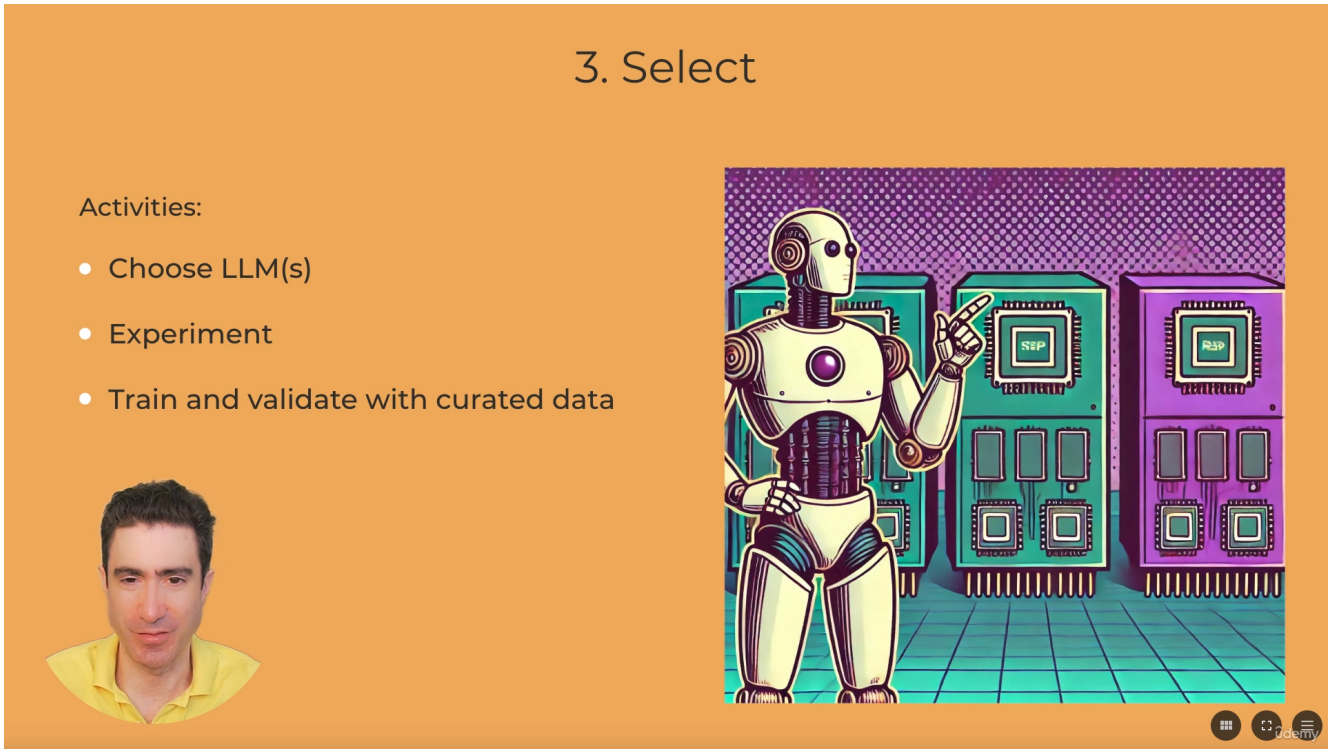
Once you have a plan, it's time to get your resources ready.

- **Research Solutions:** See how this problem is already being solved. Look at solutions that don't use a big AI model to set a "baseline" for performance.
- **Compare AI Models:** Look at different LLMs.
  - **The Basics:** How much do they cost? How much text can they handle at once (context length)? Can you legally use them for your business?
  - **Performance:** Check online tests (Benchmarks, Leaderboards) to see which models are best at certain tasks.
- **Clean Your Data:** This is very important! You need to organize, clean, and prepare your data so the AI can learn from it effectively.

1.10.3 Select (Pick Your Tools and Test Them)

Now you can choose your model and see how it performs with your data.

- **Choose Your LLM(s):** Based on your research, pick one or two models that seem like the best fit.
- **Experiment:** Play around with the models. See what they're good at and where they struggle.
- **Train and Check:** Use your clean data to test (validate) the models to get real performance numbers.



1.10.4 Customize (Make the AI Better at Your Task)

Out-of-the-box models are good, but you can make them great by using special techniques to improve their performance on your specific problem. There are three main ways to do this.

Three Key Techniques

Prompting

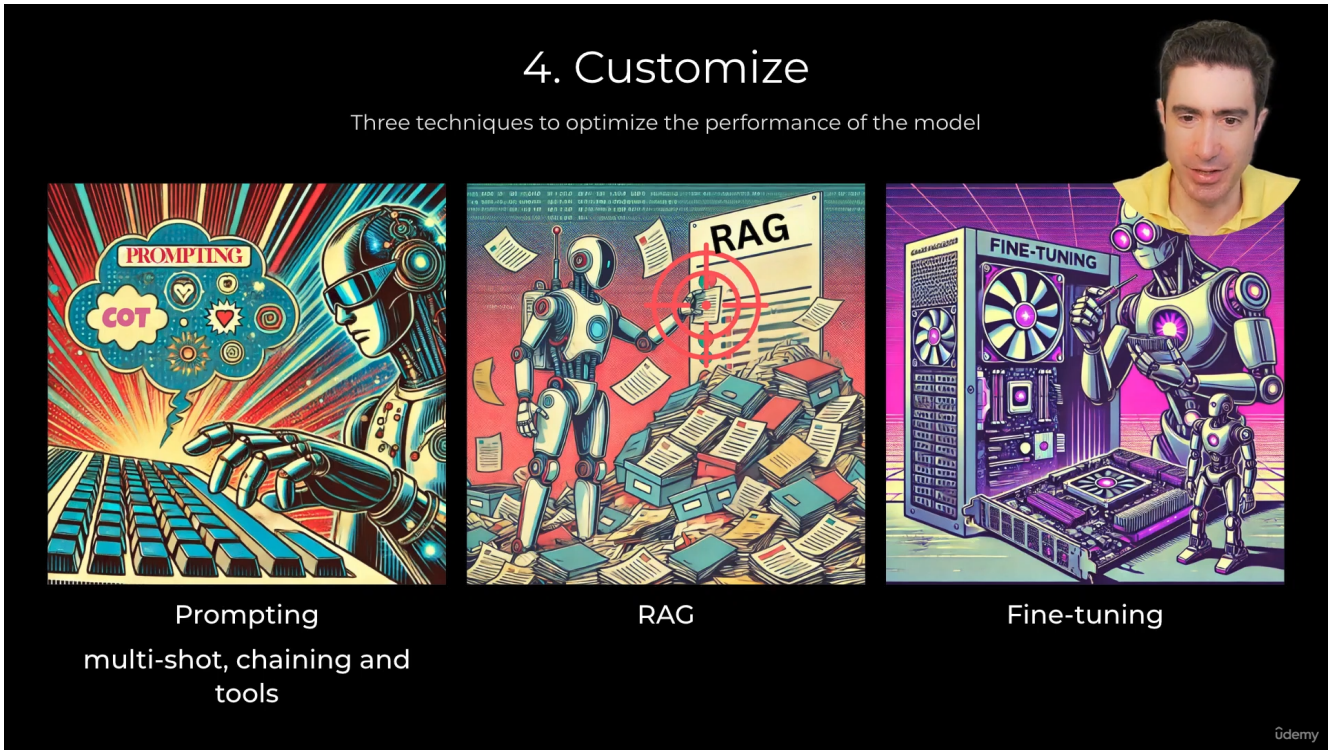
This means giving the AI better instructions. It's the fastest and cheapest way to improve performance. You can give it examples, tell it to "think step-by-step," or connect it to other software tools.

RAG (Retrieval-Augmented Generation)

This is like giving the AI an "open book" for a test. You connect the model to a knowledge base (like your company's internal documents). When a question is asked, the AI first finds the relevant information from your documents and then uses that information to create the answer. This ensures the answers are accurate and up-to-date.

Fine-Tuning

This is the most advanced technique. You actually re-train a part of the AI model on your own data. This teaches the model a new skill or a specific style, making it an expert in your particular area. It's expensive and requires a lot of data but can lead to the best performance.



Comparing the Three Techniques

Pros: The Good Stuff

Prompting	RAG	Fine-tuning
1. Quick to do	1. More accurate answers	1. Creates a true expert
2. Cheap	2. Works with lots of data	2. Understands subtle details
3. See improvements right away	3. Efficient	3. Can learn a specific tone or style
		4. Faster and cheaper answers

Cons: The Downsides

Prompting	RAG	Fine-tuning
1. Limited by how much text the AI can read at once	1. More complex to set up	1. Takes a lot of effort
2. Improvements eventually stop	2. Needs accurate, up-to-date info	2. Needs a lot of clean data
3. Can be slow and expensive for each answer	3. Can miss subtle details	3. Costs money to train
		4. Risk of the AI forgetting its original general knowledge

Three Techniques: **Pros**

Prompting

1. Fast to implement
2. Low cost
3. Often immediate improvement

RAG

1. Accuracy improvement with low data needs
2. Scalable
3. Efficient

Fine-tuning

1. Deep expertise & specialist knowledge
2. Nuance
3. Learn a different tone / style
4. Faster and cheaper inference

Three Techniques: **Cons**

Prompting

1. Limited by context length
2. Diminishing returns
3. Slower, more expensive inference

RAG

1. Harder to implement
2. Requires up-to-date, accurate data
3. Lacks nuance

Fine-tuning

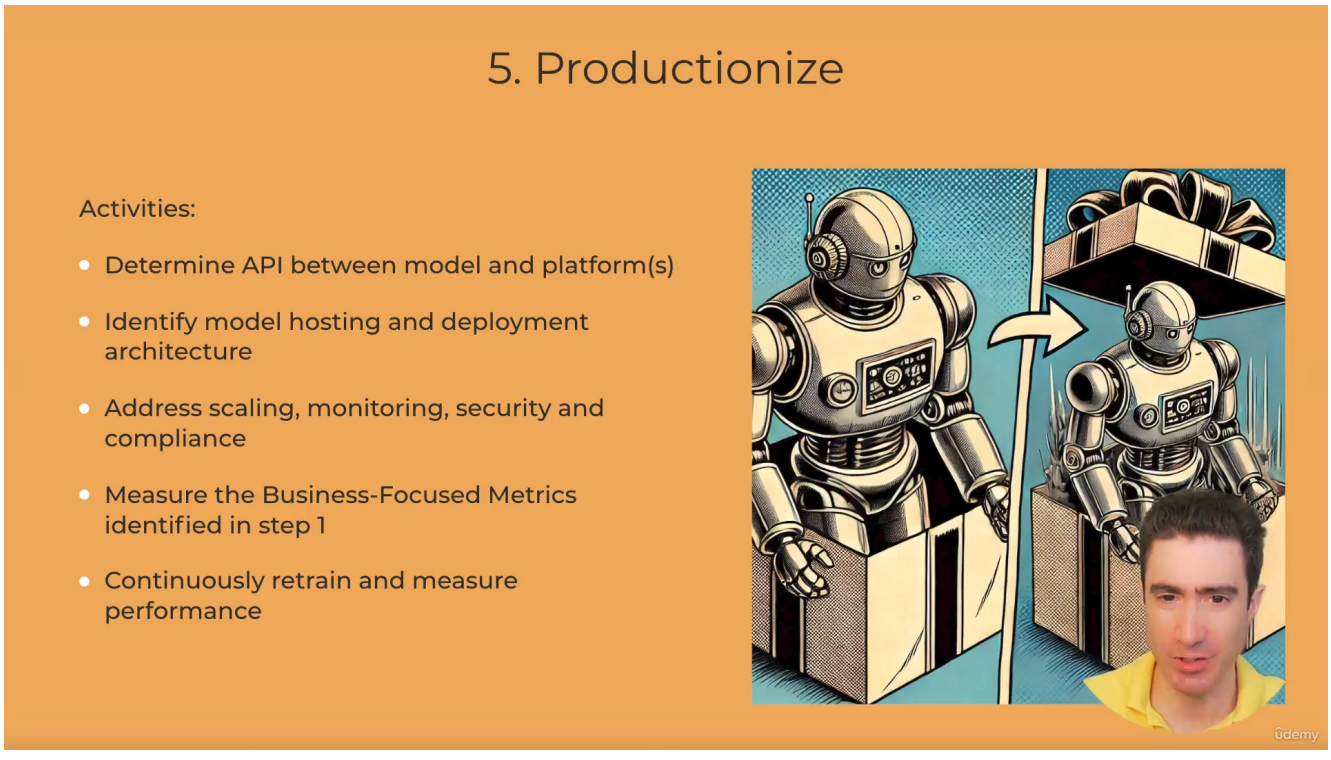
1. Significant effort to implement
2. High data needs
3. Training cost
4. Risk of "catastrophic forgetting"

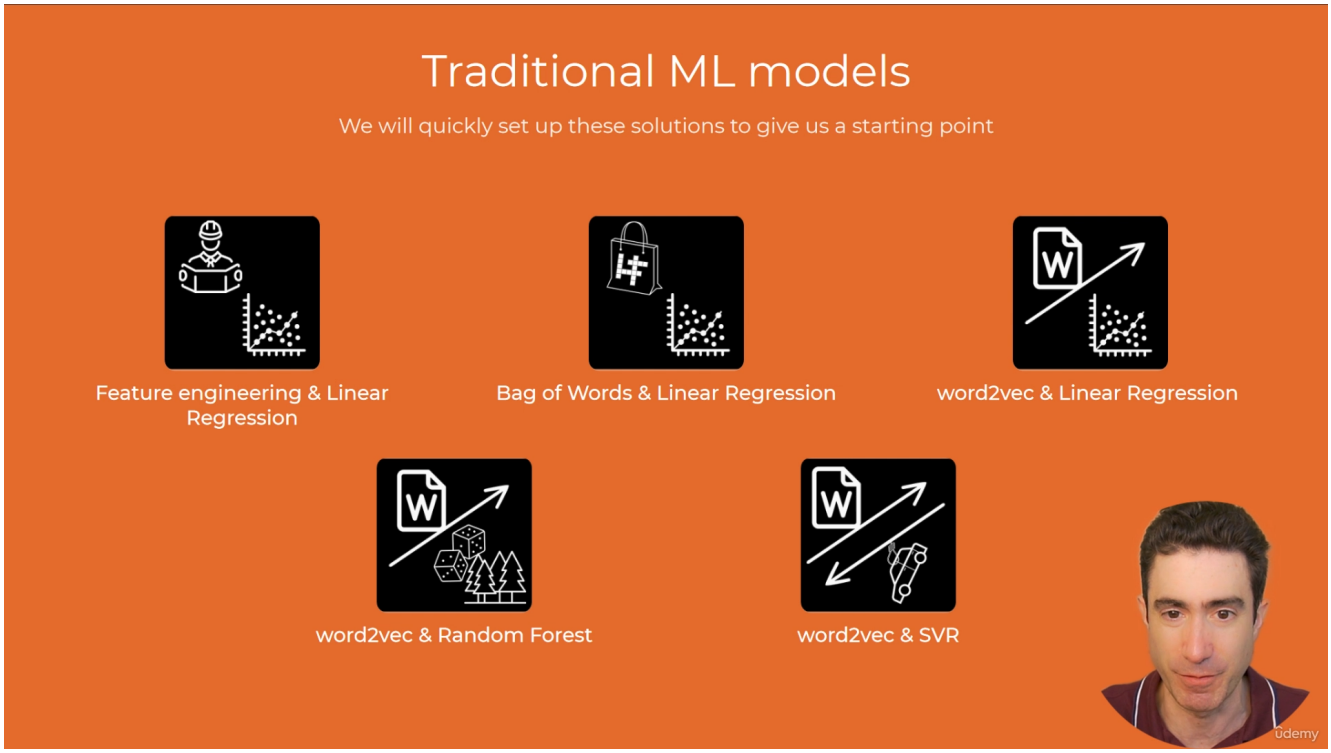

Three Techniques: Use Cases
PROMPTING
COT
RAG
FINE-TUNING
Prompting
Often the starting point for optimizing a project,with a Frontier LLM
RAG
You need high accuracy without the cost of fine-tuning; you have a Knowledge Base
Fine-tuning
You have a specialized task with a high volume of data, and you need top performance
ûdemy

1.10.5 Productionize (Get It Ready for the Real World)

Once the model is working well, you need to make it a reliable part of your business.

- **Connect it:** How will your website or other software "talk" to the AI model?
- **Deploy it:** Where will the model run? On your own computers or in the cloud?
- **Plan for Growth:** What happens if you get thousands of users? Make sure it's secure, and you're monitoring it for problems.
- **Measure What Matters:** Go back to Step 1 and measure the business goals you set. Is it actually working?
- **Keep it Updated:** Continuously check the AI's performance and re-train it with new data to keep it from becoming outdated.




Traditional ML models
We will quickly set up these solutions to give us a starting point
Feature engineering & Linear Regression
Bag of Words & Linear Regression
word2vec & Linear Regression
word2vec & Random Forest
word2vec & SVR
Udemy

1.11 LoRA (Low-Rank Adaptation)

1.11.1 Introduction

LoRA (Low-Rank Adaptation) is a technique designed to **fine-tune large pre-trained models efficiently**. Large language models (LLMs) like GPT, LLaMA, etc., have **billions of parameters**, making traditional fine-tuning:

- Extremely expensive in computation.
- Memory-intensive.
- Hard to maintain multiple fine-tuned versions.

LoRA solves this by training **small, additional modules** instead of updating the entire model.

How LoRA Works (Detailed)

Consider a weight matrix in a neural network layer:

$$W \in \mathbb{R}^{d \times k}$$

Full fine-tuning adjusts every element in W, which is costly.

LoRA introduces **two low-rank matrices** A and B and approximates the weight update:

$$\Delta W \approx AB$$

where:

$$A \in \mathbb{R}^{d \times r}, \quad B \in \mathbb{R}^{r \times k}, \quad r \ll \min(d, k)$$

Key points:

- r is small, so A and B are **tiny compared to** W.
- Only A and B are trained; the original W is **frozen**.
- Memory and computation are drastically reduced.

Step-by-Step Intuition

1. Start with a pre-trained model whose parameters W are **frozen**.
2. Add small trainable matrices A and B to selected layers (often attention layers).
3. Forward pass: compute original output using W and add low-rank contribution:

$$y = Wx + \alpha(ABx)$$

   where α is a scaling factor.

4. Backpropagation only updates A and B, leaving W untouched.
5. After training, save only A and B as the **LoRA adapter**.

Simple Analogy

Analogy

Think of a professional camera with many settings:

- Full fine-tuning = manually adjusting every knob and lens component each time you want a new photo style.
- LoRA = keep the camera fixed, but attach a **small filter**.

The filter is light, cheap, and can be swapped out, while the main camera stays unchanged.

Why LoRA is Effective

- Neural network updates often lie in a **low-rank subspace**, so approximating with small matrices captures most of the needed changes.
- Reduces training cost dramatically.
- Reduces storage: instead of storing full fine-tuned model (10s of GB), store just adapters (10s of MB).
- Enables **multi-tasking**: use same base model with different adapters for different tasks.

Benefits vs Limitations

Pros

- **Memory-efficient** and fast training.
- Base model remains **unchanged**, enabling multiple adapters.
- Easy integration with pre-trained models.

Cons

- Only works well when **low-rank approximation is sufficient**.
- Might not capture highly complex changes needed for very different tasks.

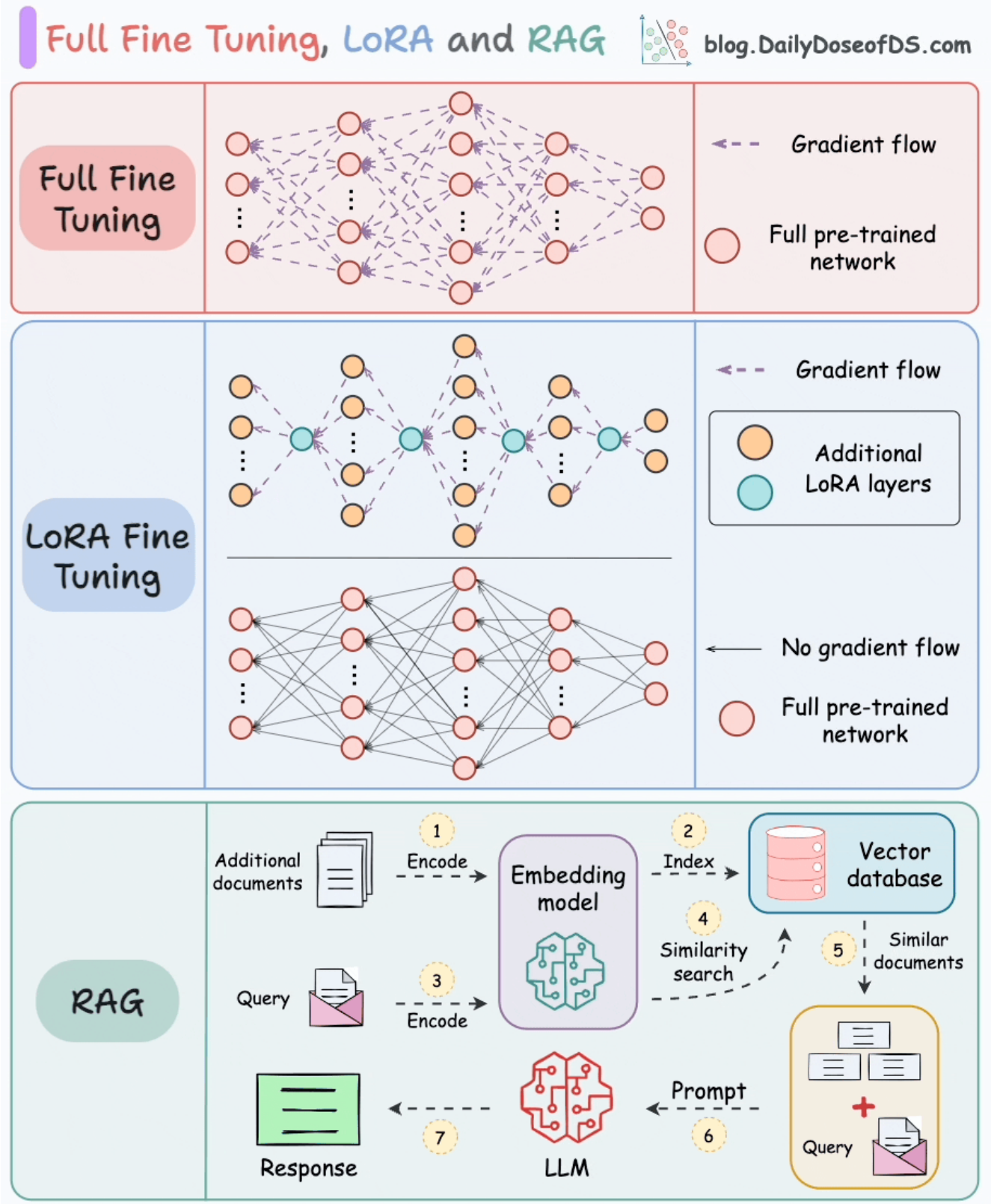
Summary in Simple Words

LoRA Simplified

You can think of LoRA as adding **tiny adapters on top of a frozen large model**. Imagine a giant model with billions of knobs. Fine-tuning everything is slow and expensive. LoRA adds a few small sliders (matrices) on top of the frozen model. During training, only these sliders move, but they are enough to guide the model to a new task.

Visualization Idea

You can include a visual showing LoRA adapters on top of a frozen model: Link



1.12 LoRA Hyperparameters: r, α, and Target Modules

LoRA (Low-Rank Adaptation) introduces a few crucial hyperparameters that control the learning capacity, scaling, and location of the adaptation. Understanding them is key to effective fine-tuning.

1.12.1 Rank r (Low-Rank Dimension)

The hyperparameter r determines the **rank of the low-rank matrices** $A \in \mathbb{R}^{d \times r}$ and $B \in \mathbb{R}^{r \times k}$.

- Controls the number of parameters introduced in the LoRA adapter.
- Small $r \Rightarrow$ fewer parameters, less memory, faster training, but less expressive.
- Large $r \Rightarrow$ more parameters, higher capacity, but increased memory and computation.

Rule of Thumb:

Start with $r \in [4, 64]$ for LLMs. Increase r if the task is complex or performance is insufficient.

Scaling Factor α

The hyperparameter α scales the output of the LoRA adapter:

$$y = Wx + \frac{\alpha}{r}(ABx)$$

- Controls the **strength of the low-rank update** relative to the original weights.
- Small $\alpha \Rightarrow$ subtle influence, safer but slower adaptation.
- Large $\alpha \Rightarrow$ strong influence, faster adaptation, but may destabilize training.

Rule of Thumb:

Use $\alpha \in [8, 32]$ for LLMs. Adjust based on task difficulty and stability.

Target Modules

Target modules define **which layers of the base model receive LoRA adapters**.

- Common targets: attention layers (query/key/value), feed-forward layers.
- Choosing too many modules $\Rightarrow$ more parameters, higher memory.
- Choosing too few modules $\Rightarrow$ may underfit the task.

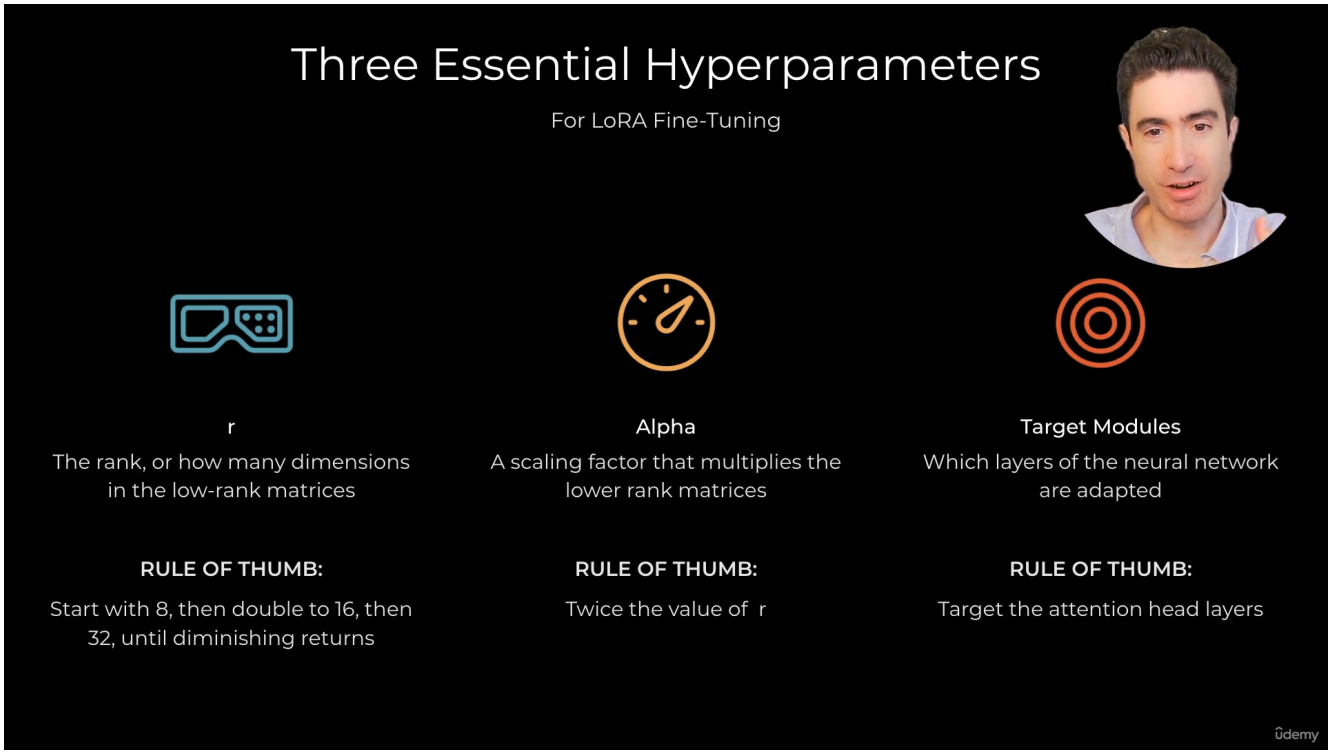
Rule of Thumb:

Start by adding LoRA to attention projection matrices (query and value) for LLMs. Expand to feed-forward layers only if necessary.

1.12.2 Summary Table

LoRA Hyperparameters Summary

Hyperparameter	Role	Rule of Thumb
r	Low-rank dimension (number of adapter parameters)	4–64
α	Scaling factor of LoRA update	8–32
Target Modules	Layers where adapters are applied	Start with attention layers and then expand to feed-forward layers only if necessary.



1.13 Q, K, V, O in Transformers

In a transformer model, the attention mechanism allows each token to selectively focus on other tokens in the sequence. This is done using four key components: **Query (Q)**, **Key (K)**, **Value (V)**, and **Output (O)**.

- **Query (Q):** Represents the **questions** each token asks. For example, a token may ask, "Which other tokens are relevant to me?" Q is calculated by multiplying the token embedding with a learned weight matrix W_Q.
- **Key (K):** Represents the **tags or identifiers** of each token. It helps the Query figure out which tokens are relevant. K is computed by multiplying token embeddings with another learned weight matrix W_K.
- **Value (V):** Represents the **actual content or information** of each token. Once Q identifies which tokens are relevant (by comparing with K), V provides the information to pass forward. V is calculated using the weight matrix W_V.
- **Output (O):** After attention combines relevant V vectors, O is a projection that mixes the attended information back into the model's hidden dimension using the matrix W_O. This ensures the output can be passed to the next layer seamlessly.

The attention operation is mathematically defined as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

- $QK^\top$ computes the similarity between queries and keys.
- d_k is the **dimension of the key vectors per attention head**. Dividing by $\sqrt{d_k}$ scales the dot product so that the softmax does not produce extremely small or large probabilities.
- Multiplying by V gathers the relevant information based on these attention weights.

1.13.1 Analogy: Classroom Example

Analogy

Imagine a classroom scenario where students ask and answer questions:

- **Q (Query):** A student asks a question to figure out which other students have relevant information.
- **K (Key):** Each student has a topic card that indicates what they know.
- **V (Value):** Each student provides an answer (information) if their topic matches the query.
- **O (Output):** The teacher collects all answers, summarizes them, and writes a combined response on the board.
- d_k**:** Think of it as the "attention focus resolution" — how detailed each student's topic card is. More details (larger d_k) require scaling to prevent overemphasis on any single answer.

This shows how Q identifies what to focus on, K helps determine relevance, V provides the information, and O produces the final output.

1.14 LoRA in PeftModelForCausalLM

1.14.1 Model Architecture Overview

The `PeftModelForCausalLM` wraps a `LlamaForCausalLM` base model and applies **LoRA adapters** for efficient fine-tuning.

- **Embedding Layer:** `embed_tokens` maps vocabulary tokens (128256) to embeddings of size 4096.
- **Decoder Layers:** 32 `LlamaDecoderLayer` modules, each containing:
  - **Self-Attention (`self_attn`)**
    - Q, K, V, O projections use `lora.Linear4bit` modules.
    - Each projection has:
      - `base_layer`: The frozen 4-bit linear layer representing the main weight matrix.
      - `lora_A`, `lora_B`: Low-rank matrices (rank $r = 32$) representing trainable updates.
      - `lora_dropout`: Dropout for regularization.
    - LoRA modifies only the projection weights, not the base model.
  - **Feed-Forward Network (MLP):** `LlamaMLP` with:
    - `gate_proj` and `up_proj` (expand embeddings to 14336)
    - `down_proj` (reduce back to 4096)
    - Non-linearity: `SiLU()`
  - **LayerNorms:** `input_layernorm` and `post_attention_layernorm`.
  - **Rotary Positional Embeddings:** `rotary_emb` for relative position encoding.
- **Final LayerNorm:** `norm` over the hidden dimension.
- **LM Head:** Linear mapping from hidden size 4096 to vocabulary size 128256.

1.14.2 LoRA in Attention Layers

In transformers, attention is computed as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

where Q, K, V are projections of the input embeddings. LoRA modifies these projections as follows:

$$W_{\text{proj}} \to W_{\text{proj}} + \Delta W = W_{\text{proj}} + AB$$

- W_{proj} is the frozen base projection weight (Linear4bit).
- $A \in \mathbb{R}^{d_{\text{model}} \times r}$, $B \in \mathbb{R}^{r \times d_{\text{proj}}}$, with $r \ll d_{\text{model}}$ (e.g., $r = 32$).
- Only A and B are trainable, drastically reducing the number of parameters.

- LoRA is applied individually to **Q, K, V, O** matrices:
  - `q_proj`: 4096 → 4096
  - `k_proj`: 4096 → 1024
  - `v_proj`: 4096 → 1024
  - `o_proj`: 4096 → 4096

Why LoRA is Effective in Attention

- **Memory Efficiency:** Instead of updating the full weight matrices (4096×4096), only two small matrices of size 4096×32 and 32×4096 are trained per projection.
- **Task Adaptation:** LoRA can inject task-specific knowledge into the attention mechanism without modifying the base model.
- **Multiple Adapters:** Different LoRA adapters can be swapped for different tasks while keeping the same base model frozen.
- **Attention-Focused Learning:** By applying LoRA specifically to Q, K, V, O, the model learns to adjust how it attends to other tokens, which is sufficient for most fine-tuning tasks.

1.14.3 Dimensions of Q, K, V, O Projections

Attention Projection Dimensions

Projection	Input Features	Output Features
Q (`q_proj`)	4096	4096
K (`k_proj`)	4096	1024
V (`v_proj`)	4096	1024
O (`o_proj`)	4096	4096

1.14.4 Summary

- **LoRA** allows fine-tuning of very large models with minimal additional parameters.
- Applied to attention layers, LoRA adjusts how the model attends to input tokens.
- Memory-efficient and task-adaptive: only the low-rank matrices are trained, while the main model stays frozen.
- Facilitates multiple task-specific adapters without retraining the entire model.

Key Points

- d_{model} is the hidden dimension (4096 in this model) through which all tokens are projected.

- LoRA adds small trainable matrices (rank $r = 32$) to Q, K, V, O projections.
- Base projections remain frozen; only adapters are updated.
- Memory-efficient and task-adaptive fine-tuning.

1.15 Loading a Base Model in 4-Bit and Using LoRA

The following code demonstrates how to load a large language model (LLM) in 4-bit precision for memory efficiency and later fine-tune it with LoRA.

```
quant_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_use_double_quant=True,
    bnb_4bit_compute_dtype=torch.bfloat16,
    bnb_4bit_quant_type="nf4")

base_model = AutoModelForCausalLM.from_pretrained(
    BASE_MODEL,
    quantization_config=quant_config,
    device_map="auto",
)
```

1.15.1 Explanation of the Code

- **BitsAndBytesConfig**: Configuration for quantization. Quantization reduces the precision of the weights to save memory.
  - `load_in_4bit=True`: Load model weights in 4-bit precision instead of 16-bit or 32-bit, drastically reducing memory usage.
  - `bnb_4bit_use_double_quant=True`: Uses double quantization to reduce numerical errors and improve accuracy.
  - `bnb_4bit_compute_dtype=torch.bfloat16`: Performs computations in bfloat16 to maintain performance while saving memory.
  - `bnb_4bit_quant_type="nf4"`: Specifies the 4-bit quantization type.
- **AutoModelForCausalLM.from_pretrained**: Loads the pre-trained LLM with the above quantization configuration.
  - `BASE_MODEL`: The pre-trained model identifier (e.g., LLaMA, Falcon).
  - `quantization_config=quant_config`: Uses the 4-bit quantized weights for memory efficiency.
  - `device_map="auto"`: Automatically maps model layers to available GPU(s) or CPU.

1.15.2 How LoRA Optimizes This

- Loading the model in 4-bit reduces the memory footprint, but full fine-tuning is still expensive if we try to update all weights.
- LoRA introduces **small trainable adapters** (low-rank matrices A and B) on top of the frozen base model.
- Only these adapters are updated during training, allowing:
  - **Memory efficiency**: base model weights are frozen and quantized.

  - **Faster training**: fewer parameters are trained.
  - **Task adaptation**: the model can specialize on a new task without full fine-tuning.
- Effectively, combining 4-bit quantization with LoRA allows training huge models on limited GPU memory.

1.15.3 Summary

Takeaway

- **4-bit quantization**: reduces memory usage of base model weights.
- **LoRA adapters**: trainable low-rank matrices that update model behavior without touching the large frozen base model.
- **Combined benefit**: efficient fine-tuning on large models with low GPU memory and compute cost.

1.16 Decision: Selecting a Base Model

1.16.1 Goal

Choose a base model by balancing: **task requirements**, **inference cost (latency / GPU memory)**, **accuracy needs**, **license / ecosystem**, and **how you plan to fine-tune (LoRA, RAG, full FT)**.

1.16.2 Decision criteria (short checklist)

- **Capability vs cost:** larger models usually perform better but cost more to run and host.
- **Parameter count:** affects memory, quantization decisions and latency.
- **Context window / multimodality:** needed for long-context tasks or image+text.
- **License and ecosystem:** open weights, available tooling (HF, vendor SDKs), community adapters.
- **Fine-tuning strategy:** if you plan to use LoRA, favor models with stable adapter support and documented PEFT integrations.
- **Deployment target:** edge / mobile vs single-GPU vs multi-GPU / cloud inference.

1.16.3 Model-by-model short summary

- **LLaMA family (Meta)**: a family of open/available models offered in multiple sizes (smaller local-friendly models and very large variants for research). LLaMA models are widely used for research and community tooling; instruction-tuned variants ("instruct") are offered for easier instruction-following.
- **Qwen (Alibaba)**: a rapidly-evolving family with strong claims in multilingual and code tasks; recent releases include much larger variants (cutting-edge, often multimodal and optimized for large-scale deployments).
- **Phi (Microsoft / Phi family)**: smaller, efficient models (e.g., Phi-2 at a few billion parameters) that are optimized for reasoning and cost-effective deployments at smaller parameter counts.
- **Gemma (Google / DeepMind family)**: a family of lightweight-to-mid-size models intended for efficient, practical use (good on-device and cloud tradeoffs); models come in small-to-medium sizes optimized for production usage.

1.16.4 Comparison table (high-level)

Model comparison at a glance

Model	Representative sizes	Typical strengths	Suggested use-cases / notes
LLaMA (Meta)	8B, 70B, (and larger research variants)	Strong general-purpose language and reasoning; broad community tooling	Good when you want high-quality open models and lots of community adapters; choose size by budget
Qwen (Alibaba)	Ranges from mid-size up to very large (vendor evolving family, latest pushes into extremely large sizes)	Good for code, multilingual tasks and large-scale deployments	Consider for enterprise cloud deployments or when vendor benchmarks favour Qwen for your task
Phi (Microsoft)	Small/medium (e.g., 2.7B for Phi-2)	Very cost-efficient; strong small-model reasoning per-cost	Excellent for edge or low-cost cloud inference where medium-sized models suffice
Gemma (Google / DeepMind)	Small → mid (family examples: 270M, 1B, 4B, 12B, 27B)	Efficient, production-ready; good on-device/cloud tradeoffs	Pick Gemma sizes for production where latency / cost matters and you require Google tooling

1.16.5 Base vs Instruct variants

- **Base (pretrained) model:** weights after pretraining on large corpora. Flexible — you can fine-tune, apply LoRA, or use RAG. Typically **better when you plan to adapt heavily** (custom fine-tuning / domain adaptation) because you control the adaptation pipeline.

- **Instruct (instruction-tuned) model:** a base model further fine-tuned on instruction/assistant datasets (and often RLHF/SFT). It usually performs better out-of-the-box on instruction-following tasks and reduces alignment/failure cases. Use an instruct variant if your primary need is production-ready instruction-following and you want fewer alignment steps.

- **LoRA note:** LoRA works on both base and instruct variants. If you want to minimize work, pick an instruct variant and add LoRA adapters for task-specific behavior; if you require maximal control, start from base and LoRA-fine-tune to your data.

1.16.6 Rules of thumb and recommended workflows

Quick rules of thumb

- **If you have very limited memory / need on-device:** prefer Gemma small variants or tiny Phi/Gemma; aggressively quantize (4-bit) + light LoRA if you need task adaptation.
- **If you need a balance (best cost / capability):** 7–13B-class models (LLaMA 7/8B, mid Qwen/Gemma sizes) quantized + LoRA often give the best bang-for-buck for many production tasks.
- **If you need highest single-model quality:** 70B+ family (or enterprise Qwen large variants) — accept higher hosting cost; LoRA still helps but cost to run will be dominated by inference.
- **If you plan multi-task adapters / many small tasks:** choose a single base model and create multiple LoRA adapters — this saves storage and makes swapping behaviors trivial.
- **Prototype first:** benchmark a small/medium size with quantization + LoRA on a representative sample before upgrading to larger variants.

1.16.7 Practical checklist before committing

1. Define success metric (accuracy, latency, throughput, cost per 1k tokens).
2. Check license / redistribution constraints for each model candidate.
3. Estimate inference cost (GPU memory + tokens/sec) for each model size.
4. Prototype: try quantized (4-bit) run + LoRA on a 1–2% validation slice and measure improvements.
5. If using instruct variants, validate whether additional LoRA actually improves or just overfits.

1.16.8 Short recommendation examples

- Small startup / prototype with tight budget: Gemma-4B or Phi-2 (quantized) + LoRA for task-specific tuning.
- Production assistant with modest budget: LLaMA 7/8B (instruct) quantized + LoRA; scale to 13B if needed.
- Research / high-quality generation: LLaMA 70B (or larger Qwen top family); expect higher hosting cost — use LoRA for specialized tasks or multi-adapter workflows.

1.16.9 Why LLaMA 3.1 was chosen: Tokenization Convenience

When selecting a base model, many factors matter: performance, parameter count, instruction-tuned vs base variant, and even subtler aspects like tokenization.

- **Tokenization strategy:** LLaMA 3.1 has a convenient tokenization scheme for numbers:
  - Numbers between 0 and 999 are represented as **single tokens**.
  - In contrast, other models like Qwen or Phi tokenize each digit separately. For example, "999" would become three tokens.
- **Why this matters:**
  - When training a model to predict numerical values (like prices), having the number as a single token simplifies the problem.
  - The model can predict the **entire number in one token**, rather than combining multiple digits.
  - This improves the model's ability to learn accurate token prediction and reduces the chance of errors (e.g., predicting $9 or $99 instead of $999).
- **Edge for LLaMA:** While it is a small difference overall, this tokenization convenience makes training slightly easier for numeric regression-like tasks.
- **Other considerations in model selection:**
  - Parameter sizes and memory requirements.
  - Instruction-tuned vs base variants.
  - Leaderboard performance.
  - Flexibility for future experiments with other models.
- **Decision:** For this project, the team chose **LLaMA 3.1 (8B or 18B variant)** as the base model, primarily because:
  - Convenient tokenization for numeric tasks.
  - Strong community support and tooling.
  - Good tradeoff between model size, performance, and fine-tuning options (LoRA can be applied easily).

Summary

Takeaway

Even small details, such as how numbers are tokenized, can influence model selection. For tasks like price prediction, LLaMA 3.1's ability to encode numbers up to 999 as single tokens provides a slight but useful advantage.

1.17 Key Hyperparameters in QLoRA

QLoRA has a few essential hyperparameters that control how efficient and accurate the fine-tuning will be. Below are the five most important ones, explained in simple terms.

1. **Target Modules** LoRA does not modify the entire model. Instead, it inserts adapters only into specific layers, called **target modules**. These are usually the linear projections inside the attention mechanism:

$$q\_proj, \quad k\_proj, \quad v\_proj, \quad o\_proj$$

Why these? Because attention layers control how information flows between tokens, so small tweaks here can have a big impact.

Analogy: Think of a giant office building (the LLM). Instead of renovating every single room, you only upgrade the **elevators and stairways** (attention layers), because they control how people (tokens) move around.

Rule of Thumb

Start with **Q (query)** and **V (value)** projections. Add to *k_proj* and *o_proj* only if the task is complex and requires more adaptation.

2. **Rank (r)** The rank r decides the size of the low-rank adapter matrices A and B:

$$\Delta W \approx A \times B, \quad A \in \mathbb{R}^{d \times r}, \; B \in \mathbb{R}^{r \times k}$$

A higher r means more trainable parameters, giving more expressive power, but it costs more memory.

Analogy: Imagine fitting a curve to data points. A small r is like fitting a straight line (fast, but may miss details). A large r is like fitting a flexible curve (captures more detail, but risks overfitting).

Rule of Thumb

Use $r = 8$ for small or domain-specific tasks. Use $r = 16$–32 for harder tasks like dialogue or reasoning.

3. **Alpha (α)** Alpha scales the contribution of the LoRA adapter relative to the frozen base model:

$$W' = W + \frac{\alpha}{r} AB$$

If α is too low, the adapters barely affect the model. If too high, they overwhelm the base model.

Analogy: Think of mixing paint colors. The frozen model is your base paint, and LoRA updates are small drops of dye. α controls how strong the dye looks in the final mix or how many drops of dye are used. If drops are too strong, the final color will be too bright and unnatural. If drops are too weak, the final color will be too dull and pale.

Rule of Thumb

Set $\alpha \approx 2r$. Example: $r = 16 \Rightarrow \alpha = 32$. This balances stability and expressiveness.

4. **Quantization** QLoRA stores the base model in **4-bit NF4 quantization** instead of 16/32-bit. This saves a huge amount of memory, making it possible to fine-tune large LLMs on a single GPU. The trick: LoRA adapters are still trained in higher precision (like 16-bit) while the frozen model stays compressed.

   Analogy: Imagine keeping your huge library in a compressed digital format (4-bit). When you read, you only extract the parts you need at higher quality.

   Rule of Thumb

   Always use **4-bit NF4** quantization. It offers the best balance between compression and accuracy.

5. **Dropout** LoRA adapters add dropout to prevent overfitting. This randomly turns off some adapter neurons during training, making the model more robust.

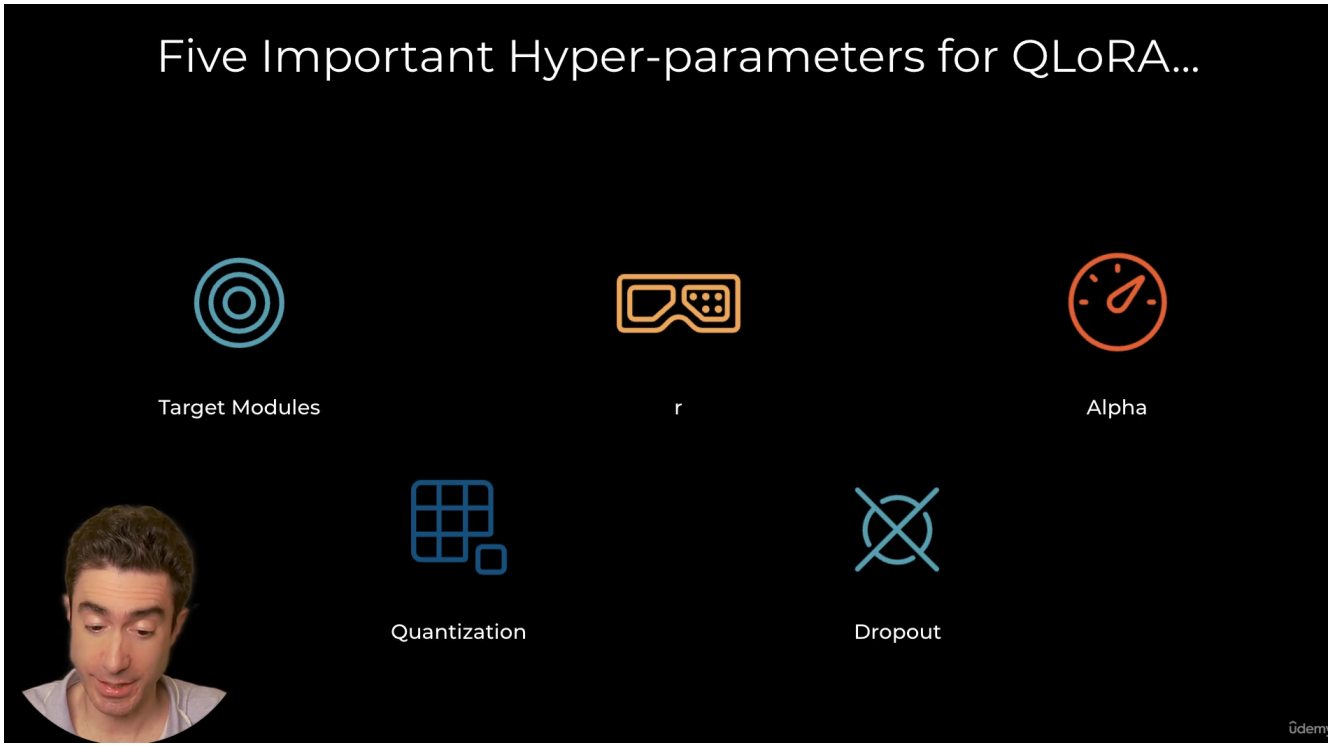
   Analogy: Like training a basketball team where sometimes players sit out, so others learn to play better. This prevents over-reliance on a few players.

   Rule of Thumb

   Use **0.05** dropout by default. Increase to **0.1** if your dataset is noisy or small.

1.17.1 Quick Summary Table

Hyperparameter	**Rule of Thumb**
Target Modules	Start with q_proj, v_proj; expand if needed.
Rank (r)	8 for small tasks, 16–32 for harder ones.
Alpha (α)	About $2r$ (e.g., $r = 16 \Rightarrow \alpha = 32$).
Quantization	Always use 4-bit NF4 for efficiency.
Dropout	0.05 by default, 0.1 for noisy/small data.


Five Important Hyper-parameters for QLoRA...
Target Modules
r
Alpha
Quantization
Dropout

1.18 Training Hyperparameters

1.18.1 Epochs

Definition: One epoch means the model has seen the *entire dataset once*.

Why Important:

- Too few epochs → the model underfits, meaning it hasn't learned enough patterns.
- Too many epochs → the model overfits, memorizing the training data and performing poorly on unseen data.

Step-by-Step Example:

1. Dataset has 10,000 samples.
2. Epoch 1: model sees all 10,000 samples → initial learning.
3. Epoch 2: model sees all 10,000 samples again → refines understanding.
4. Epoch 10: model may memorize exact data → risk of overfitting.

Analogy: Reading a textbook multiple times:

- 1 epoch = read once → understand a little.
- 5 epochs = read 5 times → remember more.
- 100 epochs = memorize word-for-word → may fail on questions requiring reasoning.

Rule of Thumb

Small datasets: 3–5 epochs. Large datasets: 1–3 epochs. Use **early stopping** to prevent overfitting.

Batch Size

Definition: Number of training samples processed together before updating model weights.

Why Important:

- Small batch size → noisy updates but better generalization.
- Large batch size → smoother updates, faster convergence, higher memory use.

Step-by-Step Example:

1. Batch size = 16 → model processes 16 samples, computes gradients, updates weights.
2. Batch size = 64 → model processes 64 samples, smoother gradient, fewer updates per epoch.

Analogy: Teacher giving feedback:

- Batch size 1 → feedback for each student immediately.
- Batch size 32 → teacher waits for all 32 students before giving feedback.

Rule of Thumb

Start with 16–64. If GPU memory is limited, use smaller batches + gradient accumulation.

Learning Rate

Definition: Step size for updating weights in the direction of the gradient.

Why Important:

- Too high → jumps over minima, training becomes unstable.
- Too low → very slow learning, may get stuck in sub-optimal minima.

Step-by-Step Example:

1. LR = 0.1 → model oscillates, can't converge.
2. LR = 0.00001 → very slow progress, may need many epochs.

Analogy: Learning to ride a bicycle:

- High LR → overcorrecting turns, wobble/fall.
- Low LR → move too slowly, take forever to balance.

Rule of Thumb

Fine-tuning: $2e{-}5$ to $5e{-}5$. Sensitive models: $1e{-}5$. Use **learning rate scheduler** to reduce LR over time.

Gradient Accumulation

Definition: Sum gradients from several mini-batches before performing a weight update.

Why Important:

- Saves GPU memory (can use smaller mini-batches).
- Stabilizes training, simulates larger batch size.

Step-by-Step Example:

1. Desired batch size = 64, GPU can only handle 16.
2. Feed 16 samples → compute gradients → do not update weights.
3. Repeat 3 more times → accumulate gradients.
4. Update weights once after 4 mini-batches.

Analogy: Saving money in a jar:

- Want to deposit $100 (effective batch) but have $25 daily (mini-batches).
- Save 4 days → deposit $100 once (weight update).

Rule of Thumb

Use accumulation steps = 2–8 if GPU memory is small. Effective batch size = batch size × accumulation steps.

Optimizer

Definition: Algorithm to update weights based on computed gradients.

Why Important:

- SGD: simple, stable, slow.
- Adam: adaptive, faster convergence.
- AdamW: Adam + weight decay → regularization, prevents overfitting.

Analogy: Driving style:

- SGD → manual car, slow and steady.
- Adam → modern automatic car.
- AdamW → modern car with lane assist (stays on track).

Rule of Thumb

Use AdamW, weight decay = 0.01, betas = (0.9, 0.999).

1.19 DataCollatorForCompletionOnlyLM in HuggingFace TRL

1.19.1 Problem

When training a language model to predict only a specific target (e.g., **price**), we do not want it to learn the entire product description.

Example Input:

> Product: Apple iPhone 15 Pro Max, 512GB, Color: Silver. Price is $

- The context: `Product: Apple iPhone 15 Pro Max, 512GB, Color: Silver.`
- The target/completion: `1200` (what comes after "Price is $").

Without handling this properly, the model tries to predict the entire description, which is unnecessary.

1.19.2 Traditional Approach (Complicated)

- Manually create a **mask** for the loss:
  - Tokens before "Price is $" → ignore in loss calculation.
  - Tokens after "Price is $" → calculate loss.
- This requires manually aligning token indices, which is error-prone.

1.19.3 HuggingFace TRL Solution: DataCollatorForCompletionOnlyLM

Usage

```
from trl import DataCollatorForCompletionOnlyLM

response_template = "Price is $"
collator = DataCollatorForCompletionOnlyLM(
    response_template, tokenizer=tokenizer
)
```

How it works:

- `response_template` tells the collator where the context ends and the completion begins.
- Tokens before "Price is $" → set to `-100` in labels (ignored in loss).
- Tokens after "Price is $" → normal token IDs (used in loss computation).
- The model sees the full input but only learns from the completion part.

1.19.4 Example

Input text:

Product: Apple iPhone 15 Pro Max, 512GB, Color: Silver. Price is $1200

Tokenized (simplified):

$$[\text{Product:}, \text{Apple}, \text{iPhone}, 15, \text{Pro}, \ldots, \text{Price}, \text{is}, \$, 1200]$$

Labels (for loss):

$$[-100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, 1200]$$

- `-100` → ignore in loss
- `1200` → used in loss

1.19.5 Why it is important

- Ensures the model focuses only on the relevant tokens (price).
- Reduces wasted computation and speeds up training.
- Useful for task-specific fine-tuning such as price prediction, summarization, or Q&A.

Summary

- Collator automatically creates attention masks and labels.
- Context tokens are ignored in loss.
- Completion tokens are used for loss.
- The model learns the target task efficiently.

1.19.6 Warmup Ratio (`warmup_ratio`)

Simple Explanation: Think of training a model like teaching a child to ride a bicycle.

- You don't let the child go full speed immediately.
- Instead, you start slow, then gradually increase the speed.

In training, the `learning rate` is like the speed. `warmup_ratio` decides how much of the training time we slowly increase the learning rate from 0 to the target value.

Why it matters:

- If the learning rate starts too high, the model can make big mistakes and become unstable.
- Gradually increasing it lets the model "warm up" and learn safely.
- After the warmup period, the model can learn faster without crashing.

How it works:

- Suppose total training steps = 1000.
- `warmup_ratio = 0.1` ⇒ first 100 steps increase learning rate from 0 to normal.
- After 100 steps, the usual learning rate schedule (like linear or cosine decay) takes over.

Without Warmup:

- The model might start making huge updates.
- This can cause unstable training, slow convergence, or even NaN losses.
- Warmup prevents this "shock" at the start.

Practical Tips:

- Small warmup ratio (5-10%) is usually enough.
- For very large models, slightly longer warmup can help.
- Too long warmup slows down training unnecessarily.

Easy Rule of Thumb

- Start with 5% – 10% of total steps as warmup.
- Gradually increase the learning rate from 0 to your target.
- After warmup, let the usual scheduler handle learning rate decay.

1.20 Checkpointing and Resuming Training in HuggingFace / Colab

1.20.1 Overview

Checkpointing is the process of saving your model and training state periodically so you can resume training later without losing progress. This is especially useful in environments like Google Colab where the runtime is temporary and may disconnect.

Why Checkpointing is Important

- Saves model weights, optimizer state, and training progress.
- Allows training to resume after interruptions.
- Prevents loss of progress due to runtime disconnects or crashes.
- Makes training safer for long-running experiments.

1.20.2 Checkpoint Components

A checkpoint typically contains:

- **Model weights:** `model.state_dict()` stores all parameters.
- **Optimizer state:** `optimizer.state_dict()` contains learning rates, momentum, etc.
- **Current epoch:** so training resumes from the correct point.

1.20.3 Manual Checkpointing in PyTorch

PyTorch Example

```
import torch

# Save checkpoint
torch.save({
    'epoch': epoch,
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict()
}, "checkpoint.pth")

# Load checkpoint
checkpoint = torch.load("checkpoint.pth")
model.load_state_dict(checkpoint['model_state_dict'])
optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
start_epoch = checkpoint['epoch'] + 1
```

1.20.4 Automatic Checkpointing with HuggingFace Trainer / SFTTrainer

HuggingFace Trainer or SFTTrainer can handle checkpointing automatically:

TrainingArguments Example

```
from transformers import TrainingArguments, SFTTrainer

training_args = TrainingArguments(
    output_dir="results",        # folder to save checkpoints
    num_train_epochs=5,
    per_device_train_batch_size=64,
    save_steps=100,                # save every 100 steps
    save_total_limit=3,            # keep only last 3 checkpoints
    logging_steps=50,
    evaluation_strategy="steps",
    eval_steps=50,
)

trainer = SFTTrainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=eval_dataset
)

trainer.train(resume_from_checkpoint=True)  # resumes automatically
```

Key points:

- `save_steps`: after how many steps to save a checkpoint.
- `save_total_limit`: how many checkpoints to keep; older ones are deleted.
- `resume_from_checkpoint=True`: automatically resumes from the latest checkpoint.

1.20.5 Checkpointing Flow Step-by-Step

1. **Start training:** check if checkpoint exists.
2. **Checkpoint exists:** load model weights, optimizer state, and last completed epoch.
3. **Set starting epoch:** `start_epoch = checkpoint['epoch'] + 1`.
4. **Resume training:** continue from the last saved epoch.
5. **Save checkpoint after each epoch or step:** ensures latest progress is always saved.

1.20.6 Checkpointing in Google Colab

Colab's local filesystem is temporary. To preserve checkpoints:

Resume Training in Colab

- **Option A: Mount Google Drive**

```
from google.colab import drive
drive.mount('/content/drive')

training_args = TrainingArguments(
    output_dir="/content/drive/MyDrive/mnist_sft_results",
    num_train_epochs=5,
    per_device_train_batch_size=64,
    save_steps=100,
    save_total_limit=3
)
```

- **Option B: Copy checkpoints manually**

```
!cp -r /content/mnist_sft_results /content/drive/MyDrive/
```

Resume Training in Colab

After reconnecting to Colab:

- Mount Drive if using Option A.
- Make sure `output_dir` points to the same folder.
- Start training:
  `trainer.train(resume_from_checkpoint=True)`
- Trainer automatically detects latest checkpoint and resumes training.

1.20.7 Advantages of Using SFTTrainer Checkpointing

- Automatic saving of model, optimizer, scheduler, and training step.

- Easy resuming after runtime disconnects.
- Cleaner management of checkpoints using `save_total_limit`.
- Optional integration with Weights & Biases (`report_to="wandb"`) for real-time metrics.

Summary

- Checkpointing ensures training can continue after interruptions.
- HuggingFace Trainer / SFTTrainer makes checkpointing automatic.
- In Colab, save checkpoints to Google Drive for persistence.
- Use `resume_from_checkpoint=True` to continue training seamlessly.

1.20.8 Checkpointing in Kaggle

Kaggle notebooks have a temporary filesystem similar to Colab. To preserve checkpoints:

Important Note

- Kaggle notebooks reset after the session ends.
- Local checkpoints stored in `/kaggle/working/` are temporary.
- To keep checkpoints permanently, download them manually or push them to Kaggle Datasets.

Manual Checkpointing

Example: Save to Kaggle Working Folder

```
checkpoint_path = "/kaggle/working/cnn_checkpoint.pth"
torch.save({
    'epoch': epoch,
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict()
}, checkpoint_path)
```

Downloading Checkpoints

To make the checkpoint persistent, download it after the notebook finishes:

Download Example

```
from IPython.display import FileLink
FileLink("/kaggle/working/cnn_checkpoint.pth")
```

Resuming Training on Kaggle (Manual)

When restarting the notebook:

1. Upload your previously downloaded checkpoint back to `/kaggle/working/`.
2. Load it using `torch.load()` and restore model and optimizer states:

   Example: SFTTrainer Automatic Checkpointing

   ```
   checkpoint = torch.load("/kaggle/working/cnn_checkpoint.pth")
   model.load_state_dict(checkpoint['model_state_dict'])
   optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
   start_epoch = checkpoint['epoch'] + 1
   ```

3. Continue training from `start_epoch`.

Automatic Checkpointing with SFTTrainer

Example: SFTTrainer Automatic Checkpointing

```
from transformers import TrainingArguments, SFTTrainer

training_args = TrainingArguments(
    output_dir="/kaggle/working/mnist_sft_results",  # checkpoint folder
    num_train_epochs=5,
    per_device_train_batch_size=64,
    save_steps=100,          # save checkpoint every 100 steps
    save_total_limit=3,      # keep last 3 checkpoints
)

trainer = SFTTrainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=test_dataset,
)

# Resume training automatically if checkpoint exists
trainer.train(resume_from_checkpoint=True)
```

Key points for SFTTrainer:

- Checkpoints are stored in `/kaggle/working/mnist_sft_results/checkpoint-<step>`.
- Each checkpoint contains:
  - `pytorch_model.bin` – model weights
  - `optimizer.pt` – optimizer state
  - `scheduler.pt` – scheduler state
  - `trainer_state.json` – training step info
- Only the latest `save_total_limit` checkpoints are kept.
- To persist checkpoints beyond the Kaggle session, download them manually or save as a Kaggle Dataset.

Summary for Kaggle

Summary

- Kaggle filesystem is **temporary** (`/kaggle/working/`).
- **Manual checkpointing** gives full control but requires explicit save/load commands.
- **SFTTrainer checkpointing** is automatic and resumes training seamlessly.
- For long-term storage, either download checkpoints or save them as Kaggle Dataset.

1.21 Saving and Downloading Automatic Checkpoints in Kaggle

Kaggle notebooks have a **temporary filesystem** at `/kaggle/working/`. Any files, including checkpoints, will be deleted after the session ends. To preserve your training progress, you should save checkpoints as a **zip file** and download it.

1.21.1 When to Enter the Zip Command

After your training is complete, or at any point when you want to backup your checkpoints:

Zip Checkpoints Command

```
!zip -r /kaggle/working/mnist_sft_results.zip /kaggle/working/mnist_sft_results
```

Explanation:

- `zip` → creates a compressed archive.
- `-r` → includes all files and subfolders recursively.
- `/kaggle/working/mnist_sft_results.zip` → path and name of the zip file to create.
- `/kaggle/working/mnist_sft_results` → folder containing the automatic checkpoints from SFTTrainer.

1.21.2 How to Download the Zip File

After creating the zip, you can download it directly from Kaggle:

Download Example

```
from IPython.display import FileLink
FileLink("/kaggle/working/mnist_sft_results.zip")
```

Steps:

1. Run the zip command after your training to package all checkpoints.
2. Use the `FileLink` command to generate a clickable link.
3. Click the link in the notebook output to download the zip file to your local machine.

1.21.3 Restoring Checkpoints from Zip

Once downloaded, you can restore the checkpoints in a new Kaggle session or locally:

Restore Example

```
!unzip /kaggle/working/mnist_sft_results.zip -d /kaggle/working/

# Now you can resume training with SFTTrainer
trainer.train(resume_from_checkpoint="/kaggle/working/mnist_sft_results
/checkpoint-<step>")
```

Notes:

- The `-d` flag in `unzip` specifies the destination folder.
- Replace `<step>` with the checkpoint step you want to resume from.
- Using this method ensures your training progress is safe even after the Kaggle session ends.

1.22 Saving and Resuming Checkpoints with Hugging Face Hub

1.22.1 Manual Upload to Hugging Face Hub

You can also manually push any file (e.g., PyTorch `.pth` checkpoint) to Hugging Face Hub without using Trainer.

Manual Push to Hugging Face Hub

```
from huggingface_hub import Repository
import os

# Clone your Hugging Face repo (create one on HF website first)
repo_url = "https://huggingface.co/username/mnist-cnn"
repo_dir = "mnist-cnn-repo"

repo = Repository(local_dir=repo_dir, clone_from=repo_url)

# Copy checkpoint into the repo folder
os.system("cp /kaggle/working/cnn_checkpoint.pth mnist-cnn-repo/")

# Push checkpoint to Hugging Face Hub
repo.push_to_hub(commit_message="Added CNN checkpoint")
```

1.22.2 Manual Download from Hugging Face Hub

To load a manually uploaded checkpoint:

Manual Download Example

```
from huggingface_hub import hf_hub_download

checkpoint_path = hf_hub_download(
    repo_id="username/mnist-cnn",
    filename="cnn_checkpoint.pth"
)
```

1.22.3 Why Use Hugging Face Hub?

- **Persistent:** Unlike Kaggle/Colab, checkpoints are never lost.
- **Cross-platform:** Works seamlessly across Colab, Kaggle, and local machines.
- **Versioning:** Every push is tracked with history.
- **Private/Shared:** You can keep repos private or make them public.

Summary

Hugging Face Hub acts as a universal storage for models and checkpoints. You can:

- Use `push_to_hub=True` in Trainer for automatic checkpoint syncing.
- Manually push any file (model, tokenizer, optimizer states, datasets).
- Resume training from any machine with `resume_from_checkpoint=True`.

This makes training workflows more robust and eliminates worries about temporary storage.

1.23 Automatic Checkpointing to Hugging Face Hub

This guide demonstrates a workflow to save and manage model checkpoints automatically on Hugging Face Hub during training.

1.23.1 Authenticate with Hugging Face

Login to Hugging Face

```
from huggingface_hub import login, create_repo, upload_file

# Login using your HF token
hf_token = "YOUR_HF_TOKEN"
login(hf_token, add_to_git_credential=True)

# Set project and run names
PROJECT_NAME = "your-project-name"
HF_USER = "your-username"
RUN_NAME = "YYYY-MM-DD_HH.MM.SS"  # Example: current datetime
REPO_ID = f"{HF_USER}/{PROJECT_NAME}-{RUN_NAME}"

# Create repo if it does not exist
create_repo(REPO_ID, exist_ok=True, private=True)
```

Define Model and Optimizer

Example CNN Model

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

class CNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)
        self.fc1 = nn.Linear(9216, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = F.max_pool2d(x, 2)
        x = torch.flatten(x, 1)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

# Device and optimizer
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = CNN().to(device)
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

Training Loop with Automatic HF Checkpointing

Training with Auto-Save

```
import os

# Local checkpoint folder
LOCAL_DIR = "checkpoints"
os.makedirs(LOCAL_DIR, exist_ok=True)
MAX_KEEP = 3
EPOCHS = 5

for epoch in range(1, EPOCHS+1):
    model.train()
    for data, target in train_loader:
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.cross_entropy(output, target)
        loss.backward()
        optimizer.step()

    print(f"Epoch {epoch} finished, loss={loss.item():.4f}")

    # Save checkpoint locally
    ckpt_name = f"checkpoint_epoch_{epoch}.pth"
    ckpt_path = os.path.join(LOCAL_DIR, ckpt_name)
    torch.save({
        "epoch": epoch,
        "model_state_dict": model.state_dict(),
        "optimizer_state_dict": optimizer.state_dict(),
    }, ckpt_path)

    # Upload checkpoint to Hugging Face Hub
    upload_file(
        path_or_fileobj=ckpt_path,
        path_in_repo=ckpt_name,
        repo_id=REPO_ID,
        commit_message=f"Upload checkpoint epoch {epoch}"
    )

    # Cleanup old checkpoints (keep only last MAX_KEEP)
    checkpoints = sorted([f for f in os.listdir(LOCAL_DIR)
     if f.startswith("checkpoint_epoch_")])
    if len(checkpoints) > MAX_KEEP:
        for old_ckpt in checkpoints[:-MAX_KEEP]:
            os.remove(os.path.join(LOCAL_DIR, old_ckpt))
            print(f"Deleted old checkpoint: {old_ckpt}")
```

Resuming Training from HF Hub

Resume Training

```
from huggingface_hub import hf_hub_download
import torch

# Download latest checkpoint from HF Hub
checkpoint_path = hf_hub_download(
    repo_id=REPO_ID,
    filename="checkpoint_epoch_5.pth"
)

# Load checkpoint
checkpoint = torch.load(checkpoint_path)
model.load_state_dict(checkpoint["model_state_dict"])
optimizer.load_state_dict(checkpoint["optimizer_state_dict"])
start_epoch = checkpoint["epoch"] + 1

# Continue training from start_epoch
for epoch in range(start_epoch, EPOCHS+1):
    model.train()
    for data, target in train_loader:
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.cross_entropy(output, target)
        loss.backward()
        optimizer.step()

    print(f"Epoch {epoch} finished, loss={loss.item():.4f}")
    # Optionally repeat checkpoint upload here
```

Key Points / Best Practices

Notes

- **Persistent storage:** HF Hub stores checkpoints permanently and versioned.
- **Automatic upload:** Use `upload_file` after saving locally.
- **Manage local storage:** Keep only a few recent checkpoints to avoid using too much disk space.
- **Reusable workflow:** Change model, dataset, and `REPO_ID` to use this template for any project.

1.24 Resuming Fine-Tuning with SFTTrainer and LoRA

This guide demonstrates how to resume fine-tuning a previously LoRA-adapted model using `SFTTrainer`, save to Hugging Face Hub, and perform inference.

1.24.1 Define Constants and Paths

Project Paths and Model Info

```
BASE_MODEL = "meta-llama/Meta-Llama-3.1-8B"
PROJECT_NAME = "pricer"
HF_USER = "sijanpaudel"

# Dataset on Hugging Face Hub
DATASET_NAME = f"{HF_USER}/pricer-data"

# Previously fine-tuned checkpoint to resume
ORIGINAL_FINETUNED = f"{HF_USER}/{PROJECT_NAME}-2024-09-13_13.04.39"
# previous fine-tuned model repo
# If you want to resume from a specific commit, set the revision
# Otherwise, set to None to use the latest
ORIGINAL_REVISION = "e8d637df551603dc86cd7a1598a8f44af4d7ae36" # commit hash
#ORIGINAL_REVISION = None


# Unique names for current run
RUN_NAME = f"{datetime.now():%Y-%m-%d_%H.%M.%S}"
PROJECT_RUN_NAME = f"{PROJECT_NAME}-{RUN_NAME}"
HUB_MODEL_NAME = f"{HF_USER}/{PROJECT_RUN_NAME}"
```

Login to HuggingFace Hub and Weights & Biases

Authentication

```
# HuggingFace login
hf_token = userdata.get('HF_TOKEN')
login(hf_token, add_to_git_credential=True)

# Weights & Biases login
wandb_api_key = userdata.get('WANDB_API_KEY')
os.environ["WANDB_API_KEY"] = wandb_api_key
wandb.login()
```

Load Dataset

Load Dataset from Hub

```
from datasets import load_dataset

dataset = load_dataset(DATASET_NAME)
train = dataset['train']
test  = dataset['test']
```

Configure LoRA and QLoRA Hyperparameters

LoRA and Quantization Settings

```
EPOCHS = 2
LORA_ALPHA = 64
LORA_R = 32
LORA_DROPOUT = 0.1
BATCH_SIZE = 16
GRADIENT_ACCUMULATION_STEPS = 1
LEARNING_RATE = 2e-5
LR_SCHEDULER_TYPE = 'cosine'
WEIGHT_DECAY = 0.001
TARGET_MODULES = ["q_proj", "v_proj", "k_proj", "o_proj"]
MAX_SEQUENCE_LENGTH = 182
QUANT_4_BIT = True
```

Load Tokenizer and Base Model (Quantized)

Tokenizer & Quantized Model

```
from transformers import AutoTokenizer, AutoModelForCausalLM, BitsAndBytesConfig

if QUANT_4_BIT:
    quant_config = BitsAndBytesConfig(
        load_in_4bit=True,
        bnb_4bit_use_double_quant=True,
        bnb_4bit_compute_dtype=torch.bfloat16,
        bnb_4bit_quant_type="nf4"
    )
else:
    quant_config = BitsAndBytesConfig(
        load_in_8bit=True,
        bnb_8bit_compute_dtype=torch.bfloat16
    )

tokenizer = AutoTokenizer.from_pretrained(BASE_MODEL, trust_remote_code=True)
tokenizer.pad_token = tokenizer.eos_token
tokenizer.padding_side = "right"

base_model = AutoModelForCausalLM.from_pretrained(
    BASE_MODEL,
    quantization_config=quant_config,
    device_map="auto",
    use_cache=False
)
base_model.generation_config.pad_token_id = tokenizer.pad_token_id
```

Load Previously Fine-Tuned Model (PEFT)

Resume LoRA Checkpoint

```
from peft import PeftModel

if ORIGINAL_REVISION:
    fine_tuned_model = PeftModel.from_pretrained(
        base_model,
        ORIGINAL_FINETUNED,
        revision=ORIGINAL_REVISION,
        is_trainable=True
    )
else:
    fine_tuned_model = PeftModel.from_pretrained(
        base_model,
        ORIGINAL_FINETUNED,
        is_trainable=True
    )

fine_tuned_model.train()
```

Data Collator for Price Prediction

Data Collator

```
from trl import DataCollatorForCompletionOnlyLM

response_template = "Price is $"
collator = DataCollatorForCompletionOnlyLM(response_template,
tokenizer=tokenizer)
# The collator ensures only the part after the response_template is used for
loss calculation during training as tokenizer padding is on the right. The part
before response_template is the prompt which is only used for context(no tokens).
```

Configure SFTTrainer and LoRA

SFTTrainer Configuration

```
from peft import LoraConfig
from trl import SFTTrainer, SFTConfig

peft_parameters = LoraConfig(
    lora_alpha=LORA_ALPHA,
    lora_dropout=LORA_DROPOUT,
    r=LORA_R,
    bias="none",
    task_type="CAUSAL_LM",
    target_modules=TARGET_MODULES,
)

train_params = SFTConfig(
    output_dir=PROJECT_RUN_NAME,
    num_train_epochs=EPOCHS,
    per_device_train_batch_size=BATCH_SIZE,
    per_device_eval_batch_size=1,
    eval_strategy="no",
    gradient_accumulation_steps=GRADIENT_ACCUMULATION_STEPS,
    optim="paged_adamw_32bit",
    save_steps=2000,
    save_total_limit=10,
    logging_steps=50,
    learning_rate=LEARNING_RATE,
    weight_decay=WEIGHT_DECAY,
    fp16=False,
    bf16=True,
    max_grad_norm=0.3,
    max_steps=-1,
    warmup_ratio=0.03,
    group_by_length=True,
    lr_scheduler_type=LR_SCHEDULER_TYPE,
    report_to="wandb",
    run_name=RUN_NAME,
    max_seq_length=MAX_SEQUENCE_LENGTH,
    dataset_text_field="text",
    save_strategy="steps",
    hub_strategy="every_save",
    push_to_hub=True,
    hub_model_id=HUB_MODEL_NAME,
    hub_private_repo=True
)
```

Initialize Trainer and Fine-Tune

Fine-Tuning

```
fine_tuning = SFTTrainer(
    model=fine_tuned_model,
    train_dataset=train,
    peft_config=peft_parameters,
    args=train_params,
    data_collator=collator,
)

fine_tuning.train()

# Push model to HuggingFace Hub
fine_tuning.model.push_to_hub(PROJECT_RUN_NAME, private=True)
```

Inference Example

Inference Example

```
prompt = "How much does this cost to the nearest dollar?\n\n
Power Stop Rear Z36 Truck and Tow Brake Kit with Calipers\n\n### Price:\n$"
inputs = tokenizer(prompt, return_tensors="pt").to("cuda")
outputs = fine_tuned_model.generate(**inputs, max_new_tokens=50)
print(tokenizer.decode(outputs[0], skip_special_tokens=True))
```

1.24.2 Checkpointing in SFTTrainer with LoRA

What Actually Happens During Training

- Unlike standard HuggingFace Trainer, **no checkpoint-<step> folders are automatically created**.
- Automatically saved files (locally or on Hugging Face Hub):
  - `adapter_model.safetensors` – LoRA adapter weights (trained parameters)
  - `training_args.bin` – training configuration (learning rate, optimizer type, scheduler type, etc.)
  - Tokenizer/config files if pushing to Hub (`tokenizer.json`, `tokenizer_config.json`, etc.)
- **Optimizer and scheduler states are NOT saved automatically**.
- To fully resume training with the same optimizer state and learning rate, manual saving is required.

1.24.3 Inferencing using SFT

Running Inference with Fine-tuned Model

1. **Load the tokenizer and set padding:**

```
tokenizer = AutoTokenizer.from_pretrained(BASE_MODEL, trust_remote_code=True)
tokenizer.pad_token = tokenizer.eos_token
tokenizer.padding_side = "right"
```

Explanation: The tokenizer converts text into numerical tokens. We set the padding token to the EOS token to ensure proper alignment of inputs when batch processing sequences of different lengths.

2. **Load the base model:**

```
base_model = AutoModelForCausalLM.from_pretrained(
    BASE_MODEL,
    quantization_config=quant_config,
    device_map="auto",
)
base_model.generation_config.pad_token_id = tokenizer.pad_token_id
```

Explanation: Loads the pre-trained base model (e.g., LLaMA). Quantization reduces memory usage, and device_map="auto" distributes the model across available GPUs. We also set the generation padding token to match the tokenizer.

3. **Attach the fine-tuned LoRA adapter:**

```
if REVISION:
    fine_tuned_model = PeftModel.from_pretrained(
        base_model, FINETUNED_MODEL, revision=REVISION
    )
else:
    fine_tuned_model = PeftModel.from_pretrained(
        base_model, FINETUNED_MODEL
    )
```

Explanation: LoRA adapters contain only the fine-tuned weights. **REVISION** is the commit hash of the checkpoint you want to use for inference. If you specify a revision, it loads that exact checkpoint from the Hub or local folder.

Generating Text with the Fine-tuned Model

Generate predictions:

```
def model_predict(input_text):
    inputs = tokenizer.encode(input_text, return_tensors="pt").to("cuda")
    attention_mask = torch.ones(inputs.shape, device="cuda")
    outputs = fine_tuned_model.generate(inputs,
    attention_mask=attention_mask,max_new_tokens=50, num_return_sequences=1)
    response = tokenizer.decode(outputs[0], skip_special_tokens=True)
    return response
```

Explanation: Tokenized input is sent to the GPU. The `generate()` method produces text autoregressively. Finally, tokens are decoded back into readable text.

1.24.4 Improved Prediction Function

Weighted Top-K Prediction

```
top_K = 3

def improved_model_predict(prompt, device="cuda"):
    set_seed(42)
    inputs = tokenizer.encode(prompt, return_tensors="pt").to(device)
    attention_mask = torch.ones(inputs.shape, device=device)

    with torch.no_grad():
        outputs = fine_tuned_model(inputs, attention_mask=attention_mask)
        next_token_logits = outputs.logits[:, -1, :].to('cpu')

    next_token_probs = F.softmax(next_token_logits, dim=-1)
    top_prob, top_token_id = next_token_probs.topk(top_K)
    prices, weights = [], []
    for i in range(top_K):
      predicted_token = tokenizer.decode(top_token_id[0][i])
      probability = top_prob[0][i]
      try:
        result = float(predicted_token)
      except ValueError as e:
        result = 0.0
      if result > 0:
        prices.append(result)
        weights.append(probability)
    if not prices:
      return 0.0, 0.0
    total = sum(weights)
    weighted_prices = [price * weight /
     total for price, weight in zip(prices, weights)]
    return sum(weighted_prices).item()
```

Explanation:

- This function predicts a numeric value (like a price) from a prompt.
- It uses the top **K=3** token probabilities and calculates a **weighted average**.
- The tokenizer decodes predicted tokens into numbers. Non-numeric tokens are ignored.
- Probabilities from `softmax` are used as weights for averaging.
- Returns a single float representing the **expected value** of the prediction.
- Setting a `seed` ensures reproducibility.

1.24.5 Key Points for SFT Inference

Important Notes

- Always load the **base model** first; adapters cannot be used standalone.
- The **tokenizer** ensures inputs are properly formatted and padded.
- **PeftModel.from_pretrained** loads LoRA adapters from a checkpoint.
- **REVISION** specifies the exact checkpoint (commit hash) for inference.
- Use `generate()` for autoregressive text generation.
- This setup allows efficient inference using a large pre-trained model plus lightweight adapters, without retraining the base model.

1.24.6 Summary

Key Points

- Local folder `PROJECT_RUN_NAME` stores checkpoints.
- `HUB_MODEL_NAME` is passed to `hub_model_id` for Hub upload.
- `SFTTrainer` automatically handles checkpoint saving, Hub push, and resuming.
- Use `resume_from_checkpoint=True` to continue training without losing progress.

1.25 Four Steps in LoRA Training

LoRA (Low-Rank Adaptation) fine-tuning modifies the standard training process by introducing small trainable low-rank matrices into certain layers of the model (typically attention projections). The original large model weights remain frozen, and only these adapter parameters are updated. Below are the four main steps in LoRA training.

1.25.1 Forward Pass with LoRA

- Input $\mathbf{x} \in \mathbb{R}^n$ is passed through the pre-trained model.
- In a normal weight matrix $\mathbf{W} \in \mathbb{R}^{d \times k}$, LoRA introduces a low-rank decomposition:

$$\Delta \mathbf{W} = \mathbf{B}\mathbf{A}, \quad \mathbf{A} \in \mathbb{R}^{r \times k}, \ \mathbf{B} \in \mathbb{R}^{d \times r}, \ r \ll \min(d, k)$$

- The effective weight during training is:

$$\mathbf{W}' = \mathbf{W} + \Delta \mathbf{W}$$

- Since $\mathbf{W}$ is frozen, only $\mathbf{A}$ and $\mathbf{B}$ contribute learnable changes.
- Prediction is computed as:

$$\hat{\mathbf{y}} = f_{\theta, \{\mathbf{A}, \mathbf{B}\}}(\mathbf{x})$$

Loss Calculation

- The loss function $\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y})$ measures prediction error.
- Common choices:
  - Cross-Entropy Loss for language modeling:

$$\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_{i=1}^{C} y_i \log(\hat{y}_i)$$

  - Other task-specific losses (e.g., regression, contrastive loss).
- The difference from standard training is that the loss only needs to drive updates to the LoRA parameters $(\mathbf{A}, \mathbf{B})$, not the entire model.

Backward Pass (Backpropagation with LoRA)

- Gradients are computed as usual via backpropagation:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{A}}, \quad \frac{\partial \mathcal{L}}{\partial \mathbf{B}}$$

- Since $\mathbf{W}$ is frozen, its gradient is ignored:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = 0$$

- The effective gradient flow is restricted to the low-rank adapters.
- This significantly reduces memory and computation, as only a small fraction of parameters participate in updates.

Optimization (Updating LoRA Parameters)

- Only the adapter parameters $(\mathbf{A}, \mathbf{B})$ are updated:

$$\mathbf{A} \leftarrow \mathbf{A} - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{A}}, \quad \mathbf{B} \leftarrow \mathbf{B} - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{B}}$$

- Optimizers such as Adam or AdamW are commonly used.

- The frozen $\mathbf{W}$ remains unchanged throughout training, preserving pre-trained knowledge.

- This makes LoRA highly parameter-efficient: only a few million parameters are updated instead of billions.

1.25.2 Summary Table of LoRA Training Steps

Step	**Description**
Forward Pass (LoRA)	Compute predictions using $\mathbf{W}' = \mathbf{W} + \mathbf{BA}$, with $\mathbf{W}$ frozen and only $\mathbf{A}, \mathbf{B}$ trainable.
Loss Calculation	Compute loss $\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y})$ to measure prediction error, driving updates for LoRA parameters.
Backward Pass	Compute gradients only for $\mathbf{A}, \mathbf{B}$; gradients for frozen $\mathbf{W}$ are ignored.
Optimization	Update LoRA parameters $(\mathbf{A}, \mathbf{B})$ using gradient descent or Adam; $\mathbf{W}$ remains unchanged.

1.25.3 Uploading Final Dataset to Hugging Face Hub

Dataset Upload Guide

1. **Install required packages:**
```
!pip install datasets huggingface_hub
```

2. **Login to Hugging Face Hub:**
```
DATASET_NAME = "sijanpaudel/amazon-pricing-dataset"
from huggingface_hub import login
login()  # Enter your Hugging Face token
```

3. **Prepare your datasets:**
   - `final_train_set` should contain:
     - `text`: list of formatted prompts with price
     - `price`: corresponding price values
   - `final_test_set` should contain only `text` prompts (for evaluation)
4. **Convert dictionaries to Hugging Face Dataset objects:**
```
from datasets import Dataset
train_dataset = Dataset.from_dict(final_train_set)
test_dataset = Dataset.from_dict(final_test_set)
```

5. **Combine into a DatasetDict:**
```
from datasets import DatasetDict
dataset_dict = DatasetDict({
    "train": train_dataset,
    "test": test_dataset
})
```

6. **Push the dataset to Hugging Face Hub:**
```
dataset_dict.push_to_hub(DATASET_NAME, private=True)
```

7. **Verify upload:** After pushing, the dataset will be available at https://huggingface.co/datasets/sijanpaudel/amazon-pricing-dataset
8. **Pulling the dataset from Hugging Face Hub:**
```
from datasets import DatasetDict
dataset_dict = DatasetDict.load_from_hub(DATASET_NAME)
train, test = dataset_dict["train"], dataset_dict["test"]
```

1.26 Multi-Agent AI Architecture for Automated Deal Finding Systems

This architecture describes a pipeline of specialized AI agents working in sequence to discover, analyze, and notify users about potential deals. The system follows a modular and agent-based approach to ensure scalability, flexibility, and reliability.

1.26.1 User Interface (UI)

The entry point of the system is the **User Interface**.

- Typically implemented as a web application (e.g., using Gradio).
- Provides an accessible platform where users can interact with the system.
- Users can configure preferences, set deal criteria, and receive outputs.

1.26.2 Agent Framework

At the core lies the **Agent Framework**, which functions as the infrastructure supporting all agents.

- Provides **memory** for agents to recall previous states, interactions, and contextual data.
- Maintains **logging** to record every step of agent activity for auditing, debugging, and transparency.
- Ensures inter-agent communication and workflow orchestration.

1.26.3 Planning Agent

The **Planning Agent** acts as the central coordinator of the system.

- Determines which specialized agent should execute a task at any given time.
- Breaks down high-level objectives (e.g., "find and evaluate deals") into actionable sub-tasks.
- Manages sequencing, dependencies, and prioritization of agents.

1.26.4 Scanner Agent

The **Scanner Agent** is responsible for initial discovery.

- Scans multiple structured and unstructured data sources (websites, APIs, databases).
- Applies filtering and heuristic rules to identify potentially valuable deals.
- Passes candidate deals to downstream agents for deeper analysis.

1.26.5 Ensemble Agent

The **Ensemble Agent** carries out valuation and analysis.

- Uses multiple predictive models (*ensemble learning*) to estimate deal value.

- Combines outputs of different models to reduce bias and variance, leading to more robust results.
- Generates a final valuation score with confidence levels.

1.26.6 Messaging Agent

The final stage involves the **Messaging Agent**.

- Prepares results for end-users after analysis.
- Sends notifications (e.g., push alerts, emails, or in-app messages) summarizing potential deals.
- Ensures timely communication so that users can take action quickly.

1.26.7 Pipeline Summary

The flow of the system can be summarized as follows:

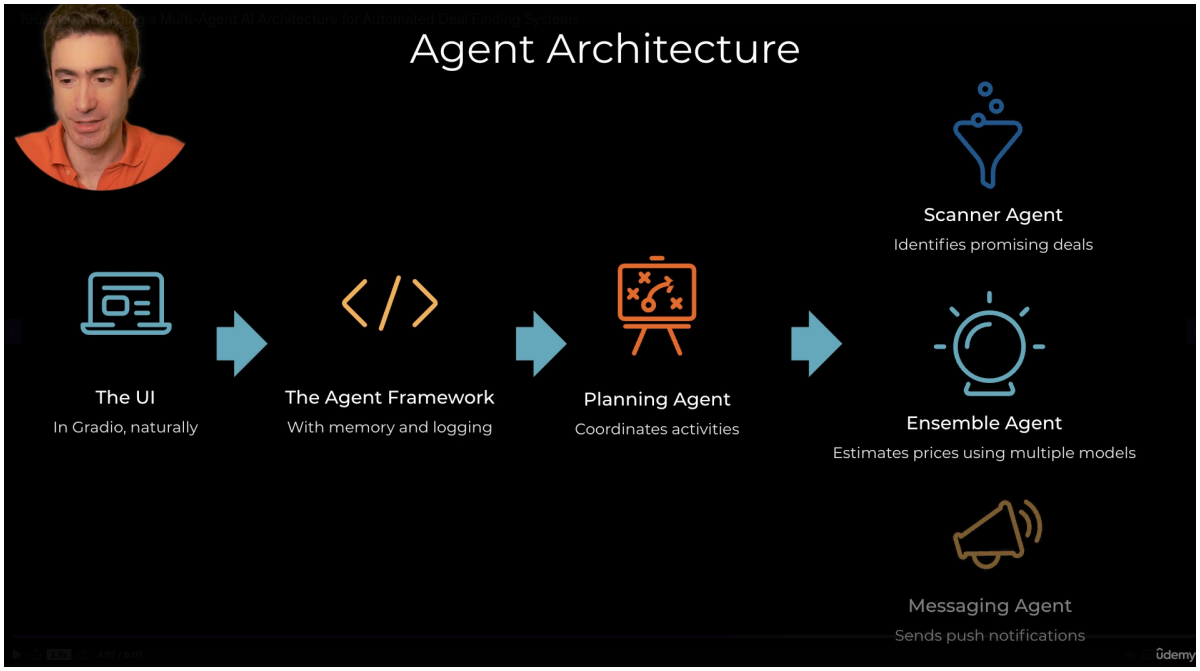
1. User interacts with the **UI** to configure requirements.
2. The **Agent Framework** initializes and monitors the workflow.
3. The **Planning Agent** orchestrates task execution.
4. The **Scanner Agent** discovers potential deals.
5. The **Ensemble Agent** evaluates deal value using multiple models.
6. The **Messaging Agent** notifies the user of results.

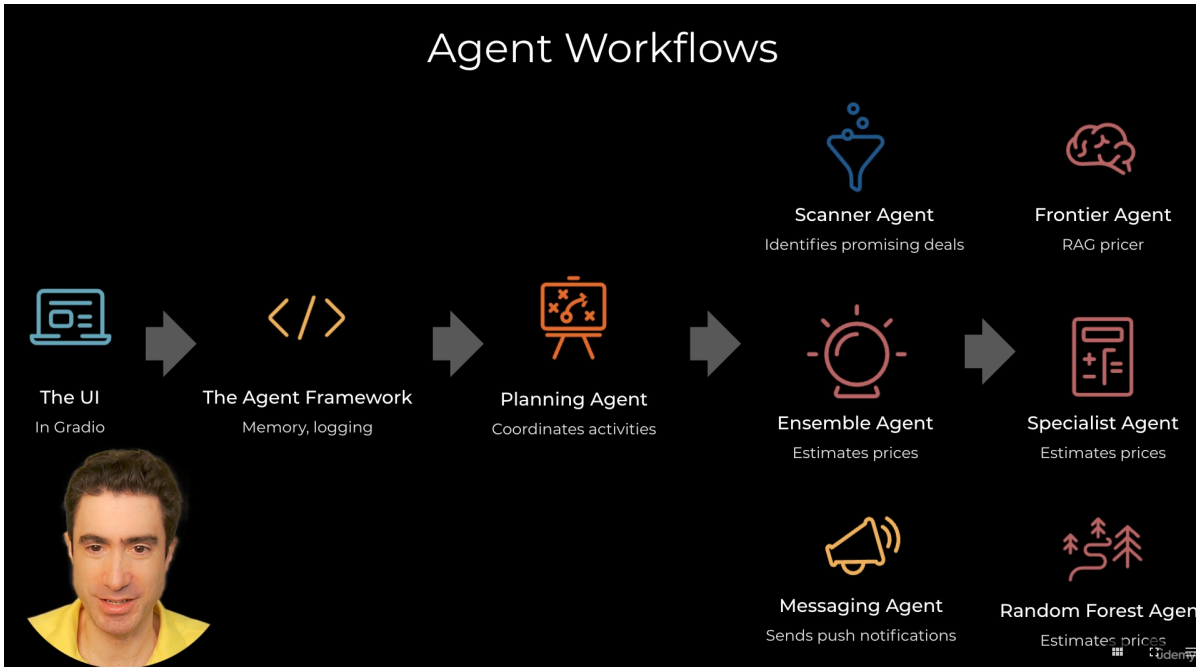
This modular multi-agent approach ensures that each agent specializes in one role, making the system scalable and maintainable while providing accurate and timely results.

1.26.8 Future Work

Future enhancements to the system could include:

- Integrating more advanced machine learning models for better deal evaluation.
- Expanding the range of data sources for the Scanner Agent.
- Improving user interface elements for better user experience.





1.27 Create a RAG Database with Our 400,000 Training Data

PLEASE NOTE:

We already have a very powerful product estimator with our proprietary, fine-tuned LLM. Most people would be very satisfied with that! The main reason we're adding these extra steps is to deepen your expertise with RAG and with Agentic workflows.

Step 1: Import necessary libraries. These include standard Python packages, numerical computing (numpy), data handling (pickle, datasets), Chroma DB for vector storage, HuggingFace tools, and embedding models.

[ ]:
```
# imports
import os
import re
import math
import json
from tqdm import tqdm
import random
from dotenv import load_dotenv
from huggingface_hub import login
import numpy as np
import pickle
from sentence_transformers import SentenceTransformer, util
from datasets import load_dataset
import chromadb
from sklearn.manifold import TSNE
import plotly.graph_objects as go
```

Step 2: Setup environment variables and database path for API keys and Chroma storage.

[ ]:
```
# environment setup
load_dotenv(override=True)
os.environ['OPENAI_API_KEY'] = os.environ.get('OPENAI_API_KEY',␣
↪'your-key-if-not-using-env')
os.environ['HF_TOKEN'] = os.environ.get('HF_TOKEN', 'your-key-if-not-using-env')
DB = 'products_vectorstore'
```

Step 3: Log in to HuggingFace hub using the token to access models and datasets.

[ ]:
```
# HuggingFace login
hf_token = os.environ['HF_TOKEN']
login(hf_token, add_to_git_credential=True)
```

Step 4: Load preprocessed training data from pickle files.

[ ]:
```
# Load training data
with open('../week6/train.pkl', 'rb') as f:
    train = pickle.load(f)
```

Step 5: Initialize Chroma persistent client, delete existing collection if any, and create a new collection for product vectors.

[ ]:
```
# Create Chroma datastore
client = chromadb.PersistentClient(path=DB)
if 'products' in [c.name for c in client.list_collections()]:
    client.delete_collection('products')
collection = client.create_collection('products')
```

Step 6: Load sentence transformer embedding model for vectorizing product descriptions.

[ ]:
```
# Initialize embedding model
model = SentenceTransformer('sentence-transformers/all-MiniLM-L6-v2')
```

Step 7: Define a helper function to extract product descriptions from the training data.

[ ]:
```
# Prepare text for vectorization
def description(item):
    return item.prompt.replace('How much does this cost to the nearest dollar?\n\n', '').split('\n\nPrice is $')[0]
```

Step 8: Loop through training data, vectorize descriptions, and add them to the Chroma collection.

[ ]:
```
# Populate Chroma collection
NUMBER_OF_DOCUMENTS = len(train)
for i in range(0, NUMBER_OF_DOCUMENTS, 1000):
    docs = [description(item) for item in train[i:i+1000]]
    vectors = model.encode(docs).astype(float).tolist()
    metas = [{'category': item.category, 'price': item.price} for item in train[i:i+1000]]
    ids = [f'doc_j' for j in range(i, i+len(docs))]
    collection.add(ids=ids, documents=docs, embeddings=vectors, metadatas=metas)
```

1.28 Hallmarks of an Agentic AI Solution

1. Decomposition of Larger Problems into Smaller Steps

An **Agentic AI system** decomposes complex problems into smaller, manageable steps. Each step is handled by an *independent agent* or *specialized process*, allowing for:

- **Parallel processing** for faster execution.
- **Error reduction** by isolating tasks.
- **Task modularity** to reuse and recombine solutions.
- *Scalable problem solving* for multi-domain challenges.

This approach ensures that large goals can be achieved efficiently without overwhelming any single component.

2. Use of Tools, Function Calling, and Structured Outputs

Agents can call *external tools*, *APIs*, or defined functions to extend their capabilities. Structured outputs ensure clear and reliable results. Key benefits include:

- **Precision:** Outputs like JSON or tables can be directly processed.
- **Reliability:** Reduces ambiguity in multi-agent workflows.
- **Interoperability:** Allows agents to communicate results seamlessly.
- *Error handling:* Structured formats make it easier to validate and parse responses.

3. Collaborative Agent Environment

Agents operate in a shared environment enabling **collaboration and coordination**:

- **Specialized knowledge:** Different agents contribute unique skills.
- **Task distribution:** Efficient division of labor for complex goals.
- *Real-time communication:* Agents exchange updates to optimize outcomes.
- *Conflict resolution:* Conflicting suggestions can be reconciled by higher-level agents.

Collaboration enhances overall system performance beyond the capability of any single agent.

4. Planning and Coordination Agent

A **planning agent** orchestrates activities across all agents:

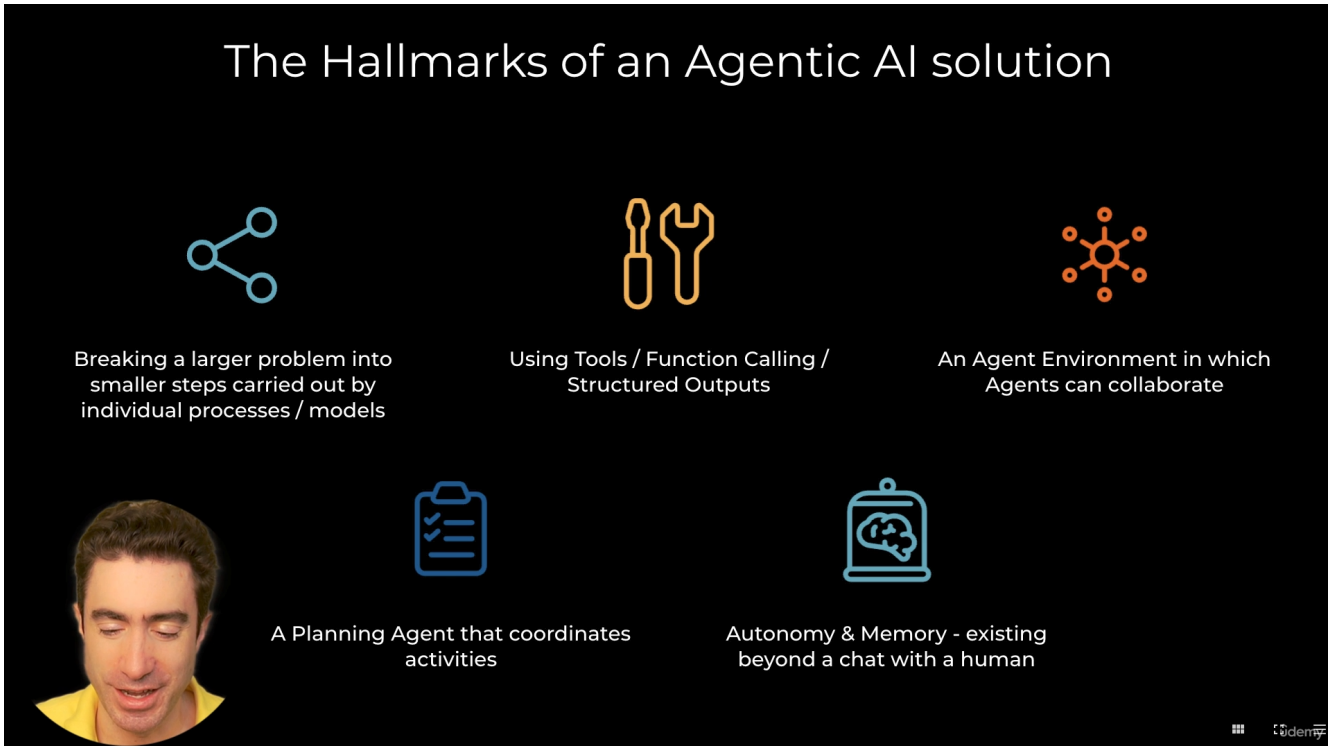
- Determines the sequence of tasks based on dependencies.
- Dynamically adjusts plans according to intermediate results.
- Prioritizes critical steps to achieve goals efficiently.
- Monitors progress and reallocates resources as needed.

This ensures that complex workflows are executed coherently and reliably.

5. Autonomy and Persistent Memory

Agentic AI demonstrates **autonomy** and maintains *persistent memory*, which allows:

- **Independent operation** without constant human intervention.
- **Learning from past interactions** to improve decision-making.
- Retention of critical information across sessions for long-term planning.
- *Cumulative knowledge growth*, enabling adaptation to new situations.
- Capability to handle multi-step and long-horizon tasks effectively.



Chapter 2

System Overview and Architecture

2.1 Introduction

This document provides an exhaustive technical analysis of a sophisticated Python-based multi-agent system designed for automated deal discovery, analysis, and notification. The system represents a compelling example of modern software engineering principles, combining object-oriented programming, machine learning, web scraping, API integration, and distributed computing into a cohesive, production-ready application.

The system operates as an intelligent deal-hunting assistant that continuously monitors RSS feeds from deal websites, applies multiple machine learning models to estimate product prices, identifies potentially valuable deals by comparing estimated prices with actual prices, and automatically notifies users of significant opportunities through various messaging channels.

2.1.1 System Purpose and Goals

The primary objective of this multi-agent system is to automate the process of finding profitable deals across multiple online retailers. The system accomplishes this through several key capabilities:

1. **Autonomous Deal Discovery**: Continuously monitors RSS feeds from deal aggregation websites like DealNews to identify new product offers

2. **Intelligent Price Estimation**: Employs multiple machine learning approaches including ensemble methods, fine-tuned language models, and traditional ML algorithms to predict fair market prices

3. **Opportunity Analysis**: Calculates potential savings by comparing estimated fair prices against offered deal prices

4. **Automated Notification**: Delivers alerts through multiple channels (SMS, push notifications) when significant deals are identified

5. **Memory Management**: Maintains awareness of previously processed deals to avoid duplicate notifications

2.1.2 Technical Architecture Overview

The system employs a modular, agent-based architecture where each component (agent) has specialized responsibilities. This design pattern promotes separation of concerns, maintainability, and scalability. The architecture can be categorized into several layers:

- **Orchestration Layer**: The PlanningAgent coordinates the entire workflow
- **Data Acquisition Layer**: ScannerAgent and deals processing handle external data retrieval
- **Machine Learning Layer**: Multiple specialized agents provide price estimation using different ML approaches
- **Communication Layer**: MessagingAgent handles user notifications
- **Foundation Layer**: Base Agent class and data models provide common functionality

2.2 Agent-Based Architecture

2.2.1 The Agent Pattern

The system implements the Agent design pattern, where each agent is an autonomous unit capable of:

- Independent operation with well-defined responsibilities
- Communication and coordination with other agents
- State management and decision making
- Logging and monitoring of its activities

All agents inherit from a common `Agent` base class that provides:

- Colored console logging for visual differentiation during runtime
- Consistent naming conventions
- Standardized communication protocols

2.2.2 System Components

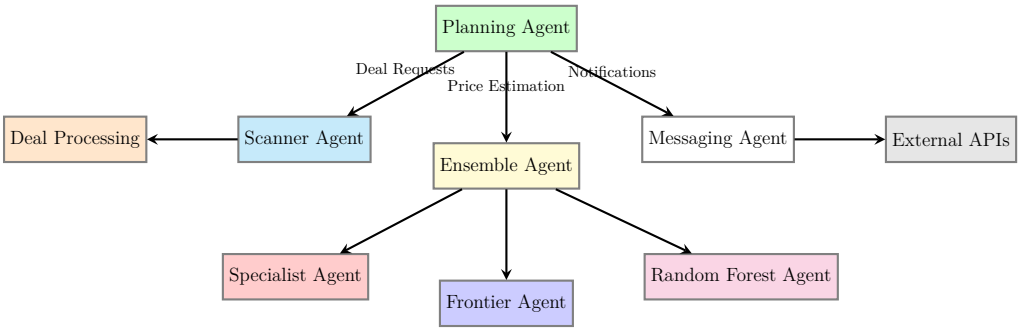


Figure 2.1: High-Level System Architecture showing agent relationships and data flow

Core Agents

PlanningAgent (Orchestrator). The PlanningAgent serves as the system's central coordinator, implementing the main workflow logic. It instantiates and manages all other agents, coordinates their interactions, and makes high-level decisions about deal processing. The PlanningAgent embodies the Command pattern, orchestrating a complex sequence of operations across multiple specialized components.

ScannerAgent (Data Acquisition). Responsible for monitoring external RSS feeds and identifying new deals. The system includes two implementations:

- `ScannerAgent`: Uses OpenAI's structured output API
- `ScannerAgentLangChain`: Uses Google's Gemini via LangChain framework

Both implementations extract deal information and convert unstructured web data into structured `Deal` objects.

EnsembleAgent (ML Coordinator). Implements an ensemble machine learning approach by coordinating three different price estimation models. It uses a meta-learning approach where a Linear Regression model combines predictions from specialist models to produce final price estimates.

MessagingAgent (Communication). Handles external communications through multiple channels including SMS (via Twilio) and push notifications (via Pushover). This agent implements the Publisher pattern, broadcasting deal alerts to configured endpoints.

Specialized ML Agents

SpecialistAgent (Fine-tuned Model). Utilizes a remotely-hosted, fine-tuned language model via Modal's cloud platform. This agent represents the most sophisticated ML approach in the system, leveraging domain-specific training for price estimation.

FrontierAgent (RAG-based LLM). Implements a Retrieval-Augmented Generation (RAG) approach using vector similarity search. It finds similar products in a ChromaDB vector store and uses this context to inform price predictions via OpenAI's GPT models.

RandomForestAgent (Traditional ML). Uses a pre-trained Random Forest model combined with sentence embeddings for price prediction. This represents a more traditional machine learning approach compared to the LLM-based agents.

2.3 Data Flow and System Interactions

2.3.1 Primary Workflow

The system operates through a well-defined workflow that demonstrates sophisticated inter-agent communication:

1. **Initialization Phase**: PlanningAgent instantiates all required agents, each performing their specific setup routines

2. **Deal Discovery**: ScannerAgent monitors RSS feeds and extracts new deals not in memory
3. **Price Analysis**: For each deal, EnsembleAgent coordinates price estimation across all ML agents
4. **Opportunity Calculation**: System compares estimated prices with deal prices to calculate potential savings
5. **Decision Making**: PlanningAgent determines whether deals meet threshold criteria for notification
6. **User Notification**: MessagingAgent delivers alerts through configured channels

2.3.2 Data Models

The system employs Pydantic models for robust data validation and serialization:

```
class Deal(BaseModel):
    """Structured representation of a deal"""
    product_description: str
    price: float
    url: str

class DealSelection(BaseModel):
    """Container for multiple deals"""
    deals: List[Deal]

class Opportunity(BaseModel):
    """Represents a potential profitable deal"""
    deal: Deal
    estimate: float  # ML-predicted price
    discount: float  # Estimated savings
```

Listing 2.1: Core Data Models

2.4 Advanced System Features

2.4.1 Machine Learning Integration

The system showcases multiple machine learning paradigms:

- **Ensemble Methods**: Meta-learning approach combining multiple models
- **Fine-tuned LLMs**: Domain-specific language model training
- **RAG (Retrieval-Augmented Generation)**: Vector similarity search with LLM reasoning
- **Traditional ML**: Scikit-learn Random Forest with feature engineering
- **Vector Embeddings**: Sentence transformers for semantic similarity

2.4.2 External Service Integration

The system integrates with multiple external services:

Service Category	Implementation	Purpose
LLM APIs	OpenAI, DeepSeek, Google Gemini	Price estimation and reasoning
Vector Database	ChromaDB	Similarity search for RAG
Cloud Computing	Modal	Remote model hosting
Messaging	Twilio, Pushover	User notifications
Data Sources	RSS Feeds	Deal discovery

Table 2.1: External Service Integrations

2.4.3 Error Handling and Robustness

The system implements several robustness mechanisms:

- **Graceful Degradation**: Continues operation if individual agents fail
- **Data Validation**: Pydantic models ensure data integrity
- **Rate Limiting**: Built-in delays prevent API abuse
- **Memory Management**: Tracks processed deals to avoid duplicates
- **Logging**: Comprehensive logging for debugging and monitoring

2.5 Architectural Patterns

2.5.1 Design Patterns Implemented

The system demonstrates several well-known design patterns:

Agent Pattern Each component operates autonomously with defined responsibilities

Strategy Pattern Multiple ML algorithms can be swapped in the ensemble

Command Pattern PlanningAgent orchestrates complex operations

Factory Pattern Agent instantiation and configuration

Observer Pattern Event-driven notifications and logging

Template Method Base Agent class defines common behavior

2.5.2 SOLID Principles

The architecture adheres to SOLID principles:

- **Single Responsibility**: Each agent has one primary function
- **Open/Closed**: New agents can be added without modifying existing code
- **Liskov Substitution**: All agents can be used polymorphically
- **Interface Segregation**: Agents depend only on methods they use
- **Dependency Inversion**: High-level modules don't depend on low-level details

2.6 System Scalability and Performance

2.6.1 Scalability Considerations

The modular architecture enables several scaling strategies:

- **Horizontal Scaling**: Agents can be distributed across multiple processes/machines
- **Load Balancing**: Multiple instances of compute-intensive agents
- **Caching**: Vector embeddings and model predictions can be cached
- **Asynchronous Processing**: Background tasks for non-critical operations

2.6.2 Performance Characteristics

Key performance aspects include:

- **I/O Bound Operations**: RSS feed parsing, API calls
- **CPU Bound Operations**: ML model inference, text processing
- **Memory Usage**: Vector embeddings, model weights
- **Network Latency**: External API dependencies

2.7 Security and Privacy

2.7.1 Security Measures

The system implements several security best practices:

- **API Key Management**: Environment variables for sensitive credentials
- **Input Validation**: Pydantic models prevent injection attacks
- **Rate Limiting**: Prevents abuse of external services
- **Error Sanitization**: Logs don't expose sensitive information

2.7.2 Privacy Considerations

- **Data Minimization**: Only necessary data is collected and stored
- **External Dependencies**: User data may be sent to third-party APIs
- **Logging Practices**: Personal information excluded from logs

2.8 Chapter Summary

This chapter provided a comprehensive overview of the Python multi-agent deal discovery system. We explored:

- The system's purpose as an automated deal-hunting assistant
- The agent-based architecture promoting modularity and maintainability
- The sophisticated integration of multiple machine learning approaches
- The robust data models and external service integrations
- The adherence to established design patterns and software engineering principles

The system represents a excellent example of modern Python development, combining object-oriented design, machine learning, web technologies, and distributed computing into a cohesive, production-ready application. The modular architecture ensures that the system can evolve and scale while maintaining reliability and performance.

In the subsequent chapters, we will dive deeper into each component, examining the implementation details, Python language features utilized, class hierarchies, and the intricate interactions between agents that make this system function as a unified whole.

Chapter 3

File-by-File Detailed Analysis

This chapter provides an exhaustive examination of each Python file in the multi-agent deal discovery system. We will analyze every class, function, method, and their interconnections, demonstrating how the modular architecture enables sophisticated functionality through careful component design.

3.1 agent.py - The Foundation Base Class

The `agent.py` file establishes the foundational infrastructure for the entire agent system. This file demonstrates fundamental object-oriented programming principles and serves as the template for all specialized agents.

3.1.1 File Structure and Imports

```
import logging

class Agent:
    """
    An abstract superclass for Agents
    Used to log messages in a way that can identify each Agent
    """

    # Foreground colors
    RED = '\033[31m'
    GREEN = '\033[32m'
    YELLOW = '\033[33m'
    BLUE = '\033[34m'
    MAGENTA = '\033[35m'
    CYAN = '\033[36m'
    WHITE = '\033[37m'

    # Background color
    BG_BLACK = '\033[40m'

    # Reset code to return to default color
```

```
    RESET = '\033[0m'

    name: str = ""
    color: str = '\033[37m'

    def log(self, message):
        """
        Log this as an info message, identifying the agent
        """
        color_code = self.BG_BLACK + self.color
        message = f"[{self.name}] {message}"
        logging.info(color_code + message + self.RESET)
```

Listing 3.1: agent.py - Complete File Analysis

3.1.2 Class Analysis: Agent

Class-Level Attributes

The Agent class defines several class-level constants using ANSI escape sequences for terminal color formatting:

- **Color Constants**: Seven foreground colors (RED through WHITE) defined as class constants
- **Background Color**: BG_BLACK provides consistent background formatting
- **Reset Code**: RESET returns terminal to default formatting
- **Instance Attributes**:
  - `name: str` - Identifier for the specific agent instance
  - `color: str` - Default color scheme for the agent's log output

Python Concepts Demonstrated.

- **Class Constants**: Using uppercase naming convention for immutable values
- **Type Annotations**: Modern Python typing for `name` and `color`
- **Default Values**: Providing sensible defaults for instance attributes
- **ANSI Escape Sequences**: Terminal control codes for colored output

Method Analysis: log()

The `log()` method provides centralized logging functionality:

```
def log(self, message):
    """
    Log this as an info message, identifying the agent
    """
    color_code = self.BG_BLACK + self.color
    message = f"[{self.name}] {message}"
    logging.info(color_code + message + self.RESET)
```

Listing 3.2: Detailed log() Method Analysis

Step-by-Step Execution Analysis.

1. **Color Code Construction**: Concatenates background and foreground color codes
2. **Message Formatting**: Uses f-string interpolation to prepend agent name
3. **Logging Call**: Utilizes Python's logging module with INFO level
4. **Color Reset**: Ensures terminal formatting returns to normal after message

Memory and Object Interaction.

- **String Concatenation**: Creates new string objects for color_code and formatted message
- **Method Resolution**: `self.name` and `self.color` trigger attribute lookup
- **Module Function Call**: `logging.info()` invokes standard library function
- **Garbage Collection**: Temporary strings become eligible for cleanup after method completion

3.1.3 Inter-File Relationships

The Agent class serves as the superclass for all specialized agents:

- **EnsembleAgent** inherits from Agent
- **FrontierAgent** inherits from Agent
- **RandomForestAgent** inherits from Agent
- **SpecialistAgent** inherits from Agent
- **MessagingAgent** inherits from Agent
- **PlanningAgent** inherits from Agent
- **ScannerAgent** inherits from Agent

3.2 deals.py - Data Models and External Integration

The `deals.py` file implements the core data structures and external API integration logic. This file demonstrates advanced Python features including Pydantic models, web scraping, RSS parsing, and object-oriented design patterns.

3.2.1 Imports and Dependencies

```
from pydantic import BaseModel
from typing import List, Dict, Self
from bs4 import BeautifulSoup
import re
import feedparser
```

```python
from tqdm import tqdm
import requests
import time
```

Listing 3.3: deals.py - Import Analysis

Import Analysis.

- **pydantic.BaseModel**: Provides data validation and serialization
- **typing**: Modern type hints for generic collections and self-references
- **BeautifulSoup**: HTML/XML parsing for web scraping
- **re**: Regular expression processing for text extraction
- **feedparser**: RSS/Atom feed parsing
- **tqdm**: Progress bar for user-friendly feedback
- **requests**: HTTP client for web requests
- **time**: Sleep functionality for rate limiting

3.2.2 Global Configuration

```python
feeds = [
    "https://www.dealnews.com/c142/Electronics/?rss=1",
    "https://www.dealnews.com/c39/Computers/?rss=1",
    "https://www.dealnews.com/c238/Automotive/?rss=1",
    "https://www.dealnews.com/f1912/Smart-Home/?rss=1",
    "https://www.dealnews.com/c196/Home-Garden/?rss=1",
]
```

Listing 3.4: RSS Feed Configuration

This module-level list defines the RSS feeds to monitor, demonstrating configuration through global constants.

3.2.3 Utility Function: extract()

```python
def extract(html_snippet: str) -> str:
    """
    Use Beautiful Soup to clean up this HTML snippet and extract useful text
    """
    soup = BeautifulSoup(html_snippet, 'html.parser')
    snippet_div = soup.find('div', class_='snippet summary')

    if snippet_div:
        description = snippet_div.get_text(strip=True)
        description = BeautifulSoup(description, 'html.parser').get_text()
        description = re.sub('<[^<]+?>', '', description)
        result = description.strip()
```

```
    else:
        result = html_snippet
    return result.replace('\n', ' ')
```

Listing 3.5: HTML Text Extraction Function

Function Analysis.

1. **HTML Parsing**: Creates BeautifulSoup object from HTML string
2. **Element Search**: Finds specific div with 'snippet summary' class
3. **Text Extraction**: Converts HTML elements to plain text
4. **HTML Tag Removal**: Uses regex to remove remaining HTML tags
5. **Whitespace Cleanup**: Normalizes spacing and removes newlines

Error Handling Strategy. The function implements graceful degradation - if the expected HTML structure isn't found, it returns the original input rather than failing.

3.2.4 Class Analysis: ScrapedDeal

```
class ScrapedDeal:
    """
    A class to represent a Deal retrieved from an RSS feed
    """
    category: str
    title: str
    summary: str
    url: str
    details: str
    features: str

    def __init__(self, entry: Dict[str, str]):
        """
        Populate this instance based on the provided dict
        """
        self.title = entry['title']
        self.summary = extract(entry['summary'])
        self.url = entry['links'][0]['href']
        stuff = requests.get(self.url).content
        soup = BeautifulSoup(stuff, 'html.parser')
        content = soup.find('div', class_='content-section').get_text()
        content = content.replace('\nmore', '').replace('\n', ' ')
        if "Features" in content:
            self.details, self.features = content.split("Features")
        else:
            self.details = content
            self.features = ""
```

Listing 3.6: ScrapedDeal Class Implementation

Constructor Analysis. The `__init__` method demonstrates complex initialization logic:

1. **Basic Attribute Assignment**: Direct mapping from RSS entry
2. **HTML Processing**: Uses extract() function for summary cleanup
3. **URL Extraction**: Navigates nested dictionary structure for URL
4. **HTTP Request**: Fetches full webpage content
5. **Content Parsing**: Extracts specific div content
6. **Text Processing**: Cleans and normalizes extracted text
7. **Content Separation**: Splits content into details and features sections

Python Concepts Demonstrated.

- **Type Hints**: All attributes have explicit type annotations
- **Dictionary Access**: Multiple patterns for accessing nested data
- **String Methods**: replace(), split(), strip() for text processing
- **HTTP Requests**: Web scraping using requests library
- **Conditional Logic**: Feature separation with fallback handling

Instance Methods

```
def __repr__(self):
    """
    Return a string to describe this deal
    """
    return f"<{self.title}>"

def describe(self):
    """
    Return a longer string to describe this deal for use in calling a model
    """
    return f"Title: {self.title}\nDetails: {self.details.strip()}\nFeatures:
    {self.features.strip()}\nURL: {self.url}"
```

Listing 3.7: ScrapedDeal Instance Methods

Method Analysis.

- **___repr___()**: Provides developer-friendly string representation
- **describe()**: Generates formatted text for ML model consumption

Class Method: fetch()

```
@classmethod
def fetch(cls, show_progress : bool = False) -> List[Self]:
    """
```

```
    Retrieve all deals from the selected RSS feeds
    """
    deals = []
    feed_iter = tqdm(feeds) if show_progress else feeds
    for feed_url in feed_iter:
        feed = feedparser.parse(feed_url)
        for entry in feed.entries[:10]:
            deals.append(cls(entry))
            time.sleep(0.5)
    return deals
```

Listing 3.8: ScrapedDeal.fetch() Class Method

Class Method Analysis.

1. **Class Method Decorator**: Uses `@classmethod` to create alternative constructor
2. **Progress Bar Integration**: Conditional `tqdm` usage based on parameter
3. **RSS Feed Processing**: Iterates through configured feeds
4. **Entry Limitation**: Processes only first 10 entries per feed
5. **Rate Limiting**: 0.5-second delay between requests
6. **Instance Creation**: Uses `cls()` to create instances of current class

Return Type Analysis.

- **List[Self]**: Uses modern Python typing for self-referential return type
- **Generic Type**: Maintains type safety across inheritance hierarchies

3.2.5 Pydantic Models

```
class Deal(BaseModel):
    """
    A class to Represent a Deal with a summary description
    """
    product_description: str
    price: float
    url: str

class DealSelection(BaseModel):
    """
    A class to Represent a list of Deals
    """
    deals: List[Deal]

class Opportunity(BaseModel):
    """
    A class to represent a possible opportunity: a Deal where we estimate
```

```
    it should cost more than it's being offered
    """
    deal: Deal
    estimate: float
    discount: float
```

Listing 3.9: Pydantic Model Definitions

Pydantic Model Benefits.

- **Automatic Validation**: Type checking and constraint enforcement
- **JSON Serialization**: Built-in conversion to/from JSON
- **IDE Support**: Enhanced autocomplete and type checking
- **Documentation**: Automatic schema generation
- **Performance**: Optimized validation using Rust backend

3.3 ensemble_agent.py - Meta-Learning Orchestration

The EnsembleAgent represents a sophisticated machine learning approach, implementing meta-learning by combining predictions from multiple specialized models.

3.3.1 File Structure and Dependencies

```
import pandas as pd
from sklearn.linear_model import LinearRegression
import joblib

from agents.agent import Agent
from agents.specialist_agent import SpecialistAgent
from agents.frontier_agent import FrontierAgent
from agents.random_forest_agent import RandomForestAgent

class EnsembleAgent(Agent):

    name = "Ensemble Agent"
    color = Agent.YELLOW

    def __init__(self, collection):
        """
        Create an instance of Ensemble, by creating each of the models
        And loading the weights of the Ensemble
        """
        self.log("Initializing Ensemble Agent")
        self.specialist = SpecialistAgent()
        self.frontier = FrontierAgent(collection)
        self.random_forest = RandomForestAgent()
```

```
        self.model = joblib.load('ensemble_model.pkl')
        self.log("Ensemble Agent is ready")

    def price(self, description: str) -> float:
        """
        Run this ensemble model
        Ask each of the models to price the product
        Then use the Linear Regression model to return the weighted price
        :param description: the description of a product
        :return: an estimate of its price
        """
        self.log("Running Ensemble Agent - collaborating with specialist,
    frontier and random forest agents")
        specialist = self.specialist.price(description)
        frontier = self.frontier.price(description)
        random_forest = self.random_forest.price(description)
        X = pd.DataFrame({
            'Specialist': [specialist],
            'Frontier': [frontier],
            'RandomForest': [random_forest],
            'Min': [min(specialist, frontier, random_forest)],
            'Max': [max(specialist, frontier, random_forest)],
        })
        y = max(0, self.model.predict(X)[0])
        self.log(f"Ensemble Agent complete - returning \$${y:.2f}")
        return y
```

Listing 3.10: ensemble_agent.py - Complete Implementation

3.3.2 Class Analysis: EnsembleAgent

Inheritance Relationship.

- **Parent Class**: Inherits from Agent base class
- **Class Attributes**: Overrides name and color from parent
- **Method Override**: Inherits log() method without modification

Constructor Analysis. The constructor demonstrates dependency injection and composition patterns:

1. **Logging Initialization**: Calls inherited log() method
2. **Agent Composition**: Creates instances of three specialized agents
3. **Model Loading**: Loads pre-trained ensemble model using joblib
4. **Completion Logging**: Confirms successful initialization

Composition vs. Inheritance. The EnsembleAgent uses composition rather than multiple inheritance to combine different ML approaches, demonstrating sound object-oriented design.

Method Analysis: price()

```python
def price(self, description: str) -> float:
    self.log("Running Ensemble Agent - collaborating with specialist, frontier and random forest agents")
    specialist = self.specialist.price(description)
    frontier = self.frontier.price(description)
    random_forest = self.random_forest.price(description)
    X = pd.DataFrame({
        'Specialist': [specialist],
        'Frontier': [frontier],
        'RandomForest': [random_forest],
        'Min': [min(specialist, frontier, random_forest)],
        'Max': [max(specialist, frontier, random_forest)],
    })
    y = max(0, self.model.predict(X)[0])
    self.log(f"Ensemble Agent complete - returning ${y:.2f}")
    return y
```

Listing 3.11: Ensemble Price Prediction Method

Step-by-Step Execution Analysis.

1. **Logging Start**: Announces collaboration with other agents
2. **Specialist Prediction**: Calls SpecialistAgent.price() method
3. **Frontier Prediction**: Calls FrontierAgent.price() method
4. **Random Forest Prediction**: Calls RandomForestAgent.price() method
5. **Feature Engineering**: Creates DataFrame with original predictions plus min/max
6. **Ensemble Prediction**: Uses meta-model to combine predictions
7. **Non-negative Constraint**: Ensures price cannot be negative
8. **Result Logging**: Reports final prediction with formatting

Machine Learning Concepts.

- **Ensemble Methods**: Combining multiple models for better performance
- **Meta-Learning**: Using a model to learn how to combine other models
- **Feature Engineering**: Creating additional features (min, max) from base predictions
- **Stacking**: Specific ensemble technique using a meta-learner

Data Flow Analysis.

- **Input**: Single string description
- **Processing**: Three parallel predictions + statistical features
- **Combination**: Linear regression meta-model

- **Output**: Single float price prediction

3.4 frontier_agent.py - RAG-Based LLM Integration

The FrontierAgent implements Retrieval-Augmented Generation (RAG), combining vector search with large language model reasoning for price estimation.

3.4.1 Import Dependencies

```
import os
import re
import math
import json
from typing import List, Dict
from openai import OpenAI
from sentence_transformers import SentenceTransformer
from datasets import load_dataset
import chromadb
from items import Item
from testing import Tester
from agents.agent import Agent
```

Listing 3.12: frontier_agent.py - Dependencies

Import Analysis.

- **Standard Library**: os, re, math, json for basic functionality
- **Type Hints**: typing module for type annotations
- **OpenAI Integration**: Direct API client for LLM calls
- **Vector Embeddings**: SentenceTransformer for semantic similarity
- **Vector Database**: ChromaDB for similarity search
- **Base Class**: Agent inheritance

3.4.2 Class Analysis: FrontierAgent

```
class FrontierAgent(Agent):

    name = "Frontier Agent"
    color = Agent.BLUE

    MODEL = "gpt-4o-mini"

    def __init__(self, collection):
        """
        Set up this instance by connecting to OpenAI or DeepSeek, to the Chroma
    Datastore,
```

```
        And setting up the vector encoding model
        """
        self.log("Initializing Frontier Agent")
        deepseek_api_key = os.getenv("DEEPSEEK_API_KEY")
        if deepseek_api_key:
            self.client = OpenAI(api_key=deepseek_api_key,
    base_url="https://api.deepseek.com")
            self.MODEL = "deepseek-chat"
            self.log("Frontier Agent is set up with DeepSeek")
        else:
            self.client = OpenAI()
            self.MODEL = "gpt-4o-mini"
            self.log("Frontier Agent is setting up with OpenAI")
        self.collection = collection
        self.model =
    SentenceTransformer('sentence-transformers/all-MiniLM-L6-v2')
        self.log("Frontier Agent is ready")
```

Listing 3.13: FrontierAgent Class Structure

Constructor Analysis.

1. **Environment Variable Check**: Determines LLM provider based on API key availability
2. **Conditional Client Creation**: Creates appropriate OpenAI client instance
3. **Model Selection**: Sets MODEL constant based on provider
4. **ChromaDB Integration**: Stores collection reference for vector search
5. **Embedding Model**: Initializes SentenceTransformer for vector encoding

Design Patterns.

- **Strategy Pattern**: Interchangeable LLM providers (OpenAI vs DeepSeek)
- **Dependency Injection**: ChromaDB collection passed in constructor
- **Factory Pattern**: Conditional object creation based on environment

Method Analysis: make_context()

```
def make_context(self, similars: List[str], prices: List[float]) -> str:
    """
    Create context that can be inserted into the prompt
    :param similars: similar products to the one being estimated
    :param prices: prices of the similar products
    :return: text to insert in the prompt that provides context
    """
    message = "To provide some context, here are some other items that might be
    similar to the item you need to estimate.\n\n"
    for similar, price in zip(similars, prices):
```

```
        message += f"Potentially related product:\n{similar}\nPrice is ${price:.2f}\n\n"
    return message
```

Listing 3.14: Context Generation Method

Method Analysis.

- **Template Building**: Constructs structured prompt template
- **List Iteration**: Uses zip() to pair similar products with prices
- **String Formatting**: f-string interpolation for price formatting
- **Context Injection**: Prepares RAG context for LLM consumption

Method Analysis: messages_for()

```
def messages_for(self, description: str, similars: List[str], prices: List[float]) -> List[Dict[str, str]]:
    """
    Create the message list to be included in a call to OpenAI
    With the system and user prompt
    """
    system_message = "You estimate prices of items. Reply only with the price, no explanation"
    user_prompt = self.make_context(similars, prices)
    user_prompt += "And now the question for you:\n\n"
    user_prompt += "How much does this cost?\n\n" + description
    return [
        {"role": "system", "content": system_message},
        {"role": "user", "content": user_prompt},
        {"role": "assistant", "content": "Price is $"}
    ]
```

Listing 3.15: OpenAI Message Construction

Prompt Engineering Analysis.

- **System Prompt**: Clear, concise instructions for the LLM
- **Context Integration**: Incorporates RAG results into user prompt
- **Few-Shot Learning**: Provides examples through similar products
- **Response Priming**: Pre-fills assistant response to guide output format

Method Analysis: find_similars()

```
def find_similars(self, description: str):
    """
    Return a list of items similar to the given one by looking in the Chroma datastore
```

```
    """
    self.log("Frontier Agent is performing a RAG search of the Chroma datastore to find 5 similar products")
    vector = self.model.encode([description])
    results = self.collection.query(query_embeddings=vector.astype(float).tolist(), n_results=5)
    documents = results['documents'][0][:]
    prices = [m['price'] for m in results['metadatas'][0][:]]
    self.log("Frontier Agent has found similar products")
    return documents, prices
```

Listing 3.16: Vector Similarity Search

RAG Implementation Analysis.

1. **Vector Encoding**: Converts text description to semantic vector
2. **Similarity Query**: Searches ChromaDB for semantically similar items
3. **Result Processing**: Extracts documents and metadata from query results
4. **Price Extraction**: Uses list comprehension to extract price metadata

Technical Details.

- **Embedding Model**: all-MiniLM-L6-v2 produces 384-dimensional vectors
- **Type Conversion**: numpy array converted to Python list for API compatibility
- **Result Limiting**: Retrieves top 5 most similar items
- **Metadata Extraction**: Accesses price information from stored metadata

Method Analysis: price()

```
def price(self, description: str) -> float:
    """
    Make a call to OpenAI or DeepSeek to estimate the price of the described product,
    by looking up 5 similar products and including them in the prompt to give context
    :param description: a description of the product
    :return: an estimate of the price
    """
    documents, prices = self.find_similars(description)
    self.log(f"Frontier Agent is about to call {self.MODEL} with context including 5 similar products")
    response = self.client.chat.completions.create(
        model=self.MODEL,
        messages=self.messages_for(description, documents, prices),
        seed=42,
        max_tokens=5
```

```
        )
        reply = response.choices[0].message.content
        result = self.get_price(reply)
        self.log(f"Frontier Agent completed - predicting ${result:.2f}")
        return result
```

Listing 3.17: Complete Price Prediction Pipeline

Complete Pipeline Analysis.

1. **RAG Search**: Finds similar products using vector similarity
2. **LLM Call**: Sends structured prompt to language model
3. **Response Processing**: Extracts price from LLM response
4. **Error Handling**: get_price() method handles parsing edge cases

API Configuration.

- **Deterministic Output**: seed=42 ensures reproducible results
- **Token Limiting**: max_tokens=5 constrains response length
- **Model Selection**: Uses appropriate model based on initialization

3.5 messaging_agent.py - Multi-Channel Communication

The MessagingAgent handles external communications through multiple channels, implementing the publisher pattern for deal notifications.

3.5.1 Dependencies and Configuration

```
import os
# from twilio.rest import Client
from agents.deals import Opportunity
import http.client
import urllib
from agents.agent import Agent

# Uncomment the Twilio lines if you wish to use Twilio

DO_TEXT = False
DO_PUSH = True
```

Listing 3.18: messaging_agent.py - Configuration

Configuration Analysis.

- **Feature Flags**: DO_TEXT and DO_PUSH enable/disable messaging channels
- **Conditional Imports**: Twilio import commented out for optional dependency

- **Standard Libraries**: http.client and urllib for HTTP requests
- **Data Models**: Imports Opportunity for type safety

3.5.2 Class Analysis: MessagingAgent

```
def __init__(self):
    """
    Set up this object to either do push notifications via Pushover,
    or SMS via Twilio,
    whichever is specified in the constants
    """
    self.log(f"Messaging Agent is initializing")
    if DO_TEXT:
        account_sid = os.getenv('TWILIO_ACCOUNT_SID',
    'your-sid-if-not-using-env')
        auth_token = os.getenv('TWILIO_AUTH_TOKEN', 'your-auth-if-not-using-env')
        self.me_from = os.getenv('TWILIO_FROM',
    'your-phone-number-if-not-using-env')
        self.me_to = os.getenv('MY_PHONE_NUMBER',
    'your-phone-number-if-not-using-env')
        # self.client = Client(account_sid, auth_token)
        self.log("Messaging Agent has initialized Twilio")
    if DO_PUSH:
        self.pushover_user = os.getenv('PUSHOVER_USER',
    'your-pushover-user-if-not-using-env')
        self.pushover_token = os.getenv('PUSHOVER_TOKEN',
    'your-pushover-user-if-not-using-env')
        self.log("Messaging Agent has initialized Pushover")
```

Listing 3.19: MessagingAgent Constructor

Constructor Analysis.

1. **Conditional Initialization**: Only sets up enabled messaging channels
2. **Environment Variables**: Uses os.getenv() with fallback defaults
3. **Credential Management**: Securely handles API keys and tokens
4. **Logging Integration**: Reports successful channel initialization

Security Practices.

- **Environment Variables**: Sensitive data not hardcoded
- **Fallback Values**: Descriptive defaults for missing configuration
- **Optional Dependencies**: System continues to work without specific services

Method Analysis: push()

```
def push(self, text):
    """
    Send a Push Notification using the Pushover API
    """
    self.log("Messaging Agent is sending a push notification")
    conn = http.client.HTTPSConnection("api.pushover.net:443")
    conn.request("POST", "/1/messages.json",
      urllib.parse.urlencode({
        "token": self.pushover_token,
        "user": self.pushover_user,
        "message": text,
        "sound": "cashregister"
      }), { "Content-type": "application/x-www-form-urlencoded" })
    conn.getresponse()
```

Listing 3.20: Push Notification Implementation

HTTP Client Analysis.

- **HTTPS Connection**: Secure connection to Pushover API
- **Form Encoding**: URL-encoded POST data
- **Custom Sound**: "cashregister" sound for deal notifications
- **Response Handling**: Gets response but doesn't process it

Method Analysis: alert()

```
def alert(self, opportunity: Opportunity):
    """
    Make an alert about the specified Opportunity
    """
    text = f"Deal Alert! Price=${opportunity.deal.price:.2f}, "
    text += f"Estimate=${opportunity.estimate:.2f}, "
    text += f"Discount=${opportunity.discount:.2f} :"
    text += opportunity.deal.product_description[:10]+'... '
    text += opportunity.deal.url
    if DO_TEXT:
        self.message(text)
    if DO_PUSH:
        self.push(text)
    self.log("Messaging Agent has completed")
```

Listing 3.21: Opportunity Alert Processing

Message Formatting Analysis.

1. **String Building**: Constructs detailed alert message
2. **Price Formatting**: Uses :.2f for currency display

3. **Description Truncation**: Limits product description to 10 characters
4. **URL Inclusion**: Provides direct link to deal
5. **Channel Selection**: Sends to enabled notification channels

3.6 Chapter Summary

This chapter provided comprehensive analysis of the first five files in the agent system:

- **agent.py**: Foundation base class with colored logging
- **deals.py**: Data models, web scraping, and RSS processing
- **ensemble_agent.py**: Meta-learning ensemble approach
- **frontier_agent.py**: RAG-based LLM integration
- **messaging_agent.py**: Multi-channel notification system

Each file demonstrates sophisticated Python programming concepts including inheritance, composition, type hints, error handling, external API integration, and design patterns. The modular architecture enables each component to have specific responsibilities while maintaining clean interfaces for inter-component communication.

The remaining files (planning_agent.py, random_forest_agent.py, scanner_agent.py, scanner_agent_langchain and specialist_agent.py) will be analyzed in the continuation of this chapter, completing our comprehensive file-by-file examination of the entire system.

Chapter 4

Class Hierarchy and Inheritance

This chapter explores the object-oriented architecture of the multi-agent system, focusing on inheritance relationships, polymorphism, method resolution order (MRO), and the sophisticated use of Python's class system to create a flexible and extensible agent framework.

4.1 Overview of Class Hierarchy

The system's class hierarchy is designed around a single base class (Agent) with multiple specialized subclasses, demonstrating classical object-oriented inheritance patterns combined with composition for complex functionality.

4.1.1 Inheritance Tree Structure

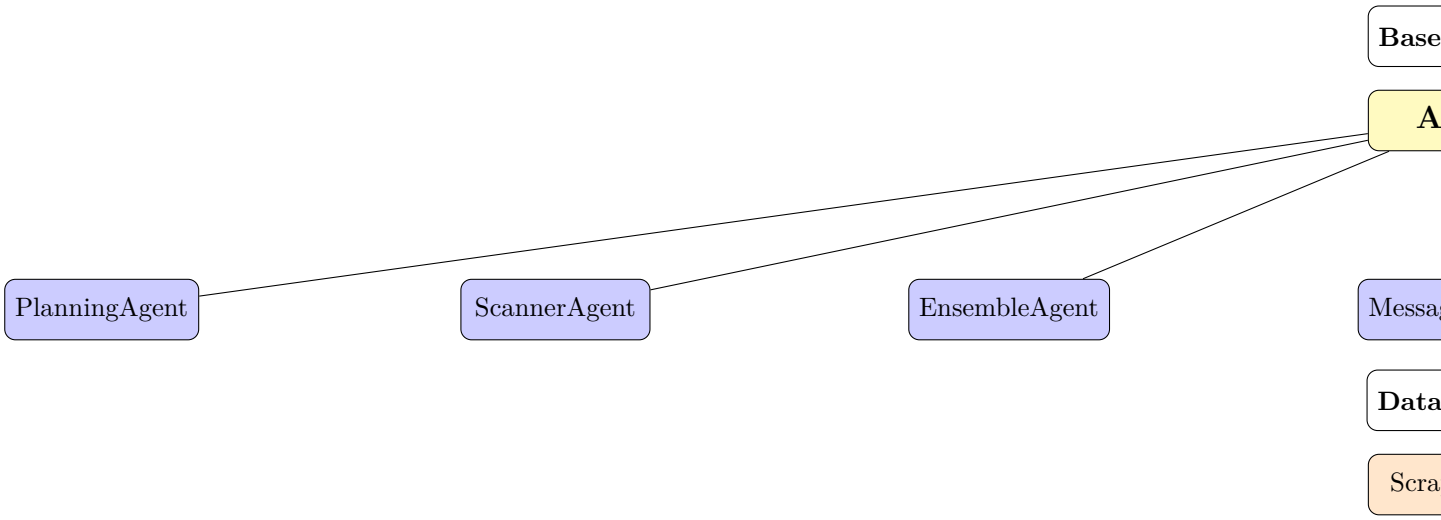


Figure 4.1: Complete Class Hierarchy of the Multi-Agent System

4.2 Base Class Analysis: Agent

4.2.1 Agent Class Architecture

The Agent class serves as an abstract base class implementing the Template Method pattern and providing common infrastructure for all specialized agents.

```
class Agent:
    """
    An abstract superclass for Agents
    Used to log messages in a way that can identify each Agent
    """

    # Class-level constants (shared across all instances)
    RED = '\033[31m'
    GREEN = '\033[32m'
    YELLOW = '\033[33m'
    BLUE = '\033[34m'
    MAGENTA = '\033[35m'
    CYAN = '\033[36m'
    WHITE = '\033[37m'
    BG_BLACK = '\033[40m'
    RESET = '\033[0m'

    # Instance attributes with type annotations
    name: str = ""
    color: str = '\033[37m'

    def log(self, message):
        """Template method for logging - inherited by all subclasses"""
        color_code = self.BG_BLACK + self.color
        message = f"[{self.name}] {message}"
        logging.info(color_code + message + self.RESET)
```

Listing 4.1: Agent Base Class - Complete Analysis

4.2.2 Class Attributes vs Instance Attributes

Class-Level Constants. The ANSI color codes are defined as class attributes, meaning:

- **Memory Efficiency**: Single copy shared across all instances
- **Namespace Access**: Available as Agent.RED, Agent.BLUE, etc.
- **Inheritance**: Automatically available in all subclasses
- **Immutability Convention**: Uppercase naming indicates constants

Instance Attributes. The name and color attributes are instance-specific:

- **Type Annotations**: Modern Python typing for IDE support
- **Default Values**: Sensible fallbacks if not overridden
- **Customization**: Each agent can have unique values

- **Late Binding**: Values determined at class definition time

4.2.3 Method Resolution Order (MRO)

For the base Agent class:

```
>>> Agent.__mro__
(<class '__main__.Agent'>, <class 'object'>)

>>> Agent.mro()
[<class '__main__.Agent'>, <class 'object'>]
```

Listing 4.2: Agent MRO Analysis

MRO Implications.

- **Single Inheritance**: Direct inheritance from object
- **Method Lookup**: Python searches Agent first, then object
- **No Conflicts**: Simple linear resolution order
- **Built-in Integration**: Inherits all object methods (___init___, ___str___, etc.)

4.3 Concrete Agent Classes

4.3.1 PlanningAgent Inheritance

```
class PlanningAgent(Agent):
    name = "Planning Agent"
    color = Agent.GREEN
    DEAL_THRESHOLD = 50

    def __init__(self, collection):
        """Overrides object.__init__ but uses Agent.log"""
        self.log("Planning Agent is initializing")
        self.scanner = ScannerAgent()
        self.ensemble = EnsembleAgent(collection)
        self.messenger = MessagingAgent()
        self.log("Planning Agent is ready")
```

Listing 4.3: PlanningAgent Class Definition

Inheritance Analysis.

1. **Class Attribute Override**: Redefines name and color from parent
2. **Additional Constants**: Adds DEAL_THRESHOLD specific to this class
3. **Constructor Definition**: Implements custom ___init___ method
4. **Parent Method Usage**: Calls self.log() which resolves to Agent.log()

5. **Composition Pattern**: Creates instances of other agents

Method Resolution Process. When `self.log()` is called:

1. Python looks for log() in PlanningAgent class - not found
2. Python looks for log() in Agent class - found!
3. Agent.log() executes using PlanningAgent instance data
4. self.name resolves to "Planning Agent" (overridden value)
5. self.color resolves to Agent.GREEN (class constant reference)

4.3.2 EnsembleAgent Inheritance

```
class EnsembleAgent(Agent):
    name = "Ensemble Agent"
    color = Agent.YELLOW

    def __init__(self, collection):
        # Demonstrates composition over inheritance
        self.log("Initializing Ensemble Agent")  # Uses inherited method
        self.specialist = SpecialistAgent()      # Composition
        self.frontier = FrontierAgent(collection) # Composition
        self.random_forest = RandomForestAgent() # Composition
        self.model = joblib.load('ensemble_model.pkl')
        self.log("Ensemble Agent is ready")

    def price(self, description: str) -> float:
        # New method specific to this class
        # Coordinates other agents (delegation pattern)
        pass
```

Listing 4.4: EnsembleAgent Inheritance Pattern

Design Pattern Analysis.

- **Inheritance**: Gets logging functionality from Agent
- **Composition**: Contains other agent instances
- **Delegation**: Forwards price() calls to composed agents
- **Coordination**: Acts as orchestrator for multiple ML models

4.3.3 SpecialistAgent Inheritance

```
class SpecialistAgent(Agent):
    name = "Specialist Agent"
    color = Agent.RED

    def __init__(self):
```

```
        self.log("Specialist Agent is initializing - connecting to modal")
        Pricer = modal.Cls.from_name("pricer-service", "Pricer")
        self.pricer = Pricer()
        self.log("Specialist Agent is ready")

    def price(self, description: str) -> float:
        self.log("Specialist Agent is calling remote fine-tuned model")
        result = self.pricer.price.remote(description)
        self.log(f"Specialist Agent completed - predicting ${result:.2f}")
        return result
```

Listing 4.5: SpecialistAgent Remote Integration

Remote Object Integration.

- **Distributed Computing**: Integrates with Modal's remote execution platform
- **Proxy Pattern**: self.pricer acts as proxy for remote object
- **Method Forwarding**: price() method forwards to remote implementation
- **Seamless Integration**: Remote complexity hidden behind simple interface

4.4 Polymorphism in Action

4.4.1 Price Estimation Polymorphism

All price-estimating agents implement a common interface despite different implementations:

```
# Each agent implements price() differently
specialist_agent.price(description)    # Remote ML model call
frontier_agent.price(description)      # RAG + LLM
random_forest_agent.price(description) # Traditional ML
ensemble_agent.price(description)      # Combines all three
```

Listing 4.6: Polymorphic Price Methods

Polymorphism Benefits.

- **Interchangeability**: Agents can be swapped without code changes
- **Extensibility**: New price estimation strategies easily added
- **Abstraction**: Client code doesn't need to know implementation details
- **Testing**: Mock objects can easily replace real agents

4.4.2 Runtime Polymorphism Example

```
def get_price_estimate(agent_type: str, description: str) -> float:
    """Demonstrates runtime polymorphism"""
    agents = {
```

```
        'specialist': SpecialistAgent(),
        'frontier': FrontierAgent(collection),
        'random_forest': RandomForestAgent(),
    }

    agent = agents[agent_type]  # Runtime selection
    return agent.price(description)  # Polymorphic call
```

Listing 4.7: Dynamic Agent Selection

4.5 Advanced Inheritance Concepts

4.5.1 Method Override Analysis

Scanner Agent Variations

The system includes two scanner implementations demonstrating interface consistency:

```
class ScannerAgent(Agent):
    """OpenAI-based implementation"""
    name = "Scanner Agent"
    color = Agent.CYAN

    def scan(self, memory: List[str]=[]) -> Optional[DealSelection]:
        # OpenAI structured output implementation
        pass

class ScannerAgentLangChain(Agent):
    """LangChain/Gemini-based implementation"""
    name = "Scanner Agent"  # Same name!
    color = Agent.CYAN      # Same color!

    def scan(self, memory: List[str]=[]) -> Optional[DealSelection]:
        # LangChain implementation with Gemini
        pass
```

Listing 4.8: Scanner Agent Polymorphism

Interface Consistency.

- **Same Method Signature**: Both implement scan() with identical parameters
- **Same Return Type**: Both return Optional[DealSelection]
- **Same Visual Identity**: Both use same name and color
- **Substitutability**: Can be swapped without affecting client code

4.5.2 Abstract Methods and Duck Typing

Although Python doesn't enforce abstract methods without ABC module, the system uses informal protocols:

```
# Informal "PriceEstimator" protocol
class PriceEstimatorProtocol:
    """Not actually defined, but implied by usage"""
    def price(self, description: str) -> float:
        """All price-estimating agents must implement this"""
        raise NotImplementedError
```

Listing 4.9: Informal Protocol Definition

Duck Typing in Practice.

- **Protocol Compliance**: If it has a price() method, it's a price estimator
- **Runtime Checking**: Errors only occur when methods are called
- **Flexibility**: No rigid interface requirements
- **Python Philosophy**: "If it walks like a duck and quacks like a duck..."

4.6 Composition vs Inheritance

4.6.1 EnsembleAgent Composition Analysis

The EnsembleAgent demonstrates composition over inheritance:

```
class EnsembleAgent(Agent):  # Inherits from Agent
    def __init__(self, collection):
        # Composition: "has-a" relationships
        self.specialist = SpecialistAgent()      # Has a specialist
        self.frontier = FrontierAgent(collection) # Has a frontier agent
        self.random_forest = RandomForestAgent() # Has a random forest
        self.model = joblib.load('ensemble_model.pkl') # Has a model

    def price(self, description: str) -> float:
        # Delegates to composed objects
        specialist = self.specialist.price(description)
        frontier = self.frontier.price(description)
        random_forest = self.random_forest.price(description)
        # Combines results using own logic
        return self.combine_predictions(specialist, frontier, random_forest)
```

Listing 4.10: Composition Pattern Analysis

Composition Benefits.

- **Flexibility**: Can change components at runtime

- **Multiple Behaviors**: Combines functionality from multiple sources
- **Loose Coupling**: Components don't need to know about each other
- **Single Responsibility**: Each component has one clear purpose

4.6.2 When to Use Inheritance vs Composition

Scenario	Inheritance	Composition
Common behavior	Agent base class provides logging	EnsembleAgent uses multiple predictors
"Is-a" relationship	SpecialistAgent IS-A Agent	EnsembleAgent HAS-A SpecialistAgent
Code reuse	All agents inherit log() method	EnsembleAgent reuses price() methods
Flexibility	Fixed at class definition	Can change components at runtime
Coupling	Tight coupling to base class	Loose coupling between components

Table 4.1: Inheritance vs Composition Decision Matrix

4.7 Method Resolution and Attribute Access

4.7.1 Attribute Resolution Process

When accessing attributes like `self.name` in an agent:

```
class FrontierAgent(Agent):
    name = "Frontier Agent"  # Class attribute
    color = Agent.BLUE      # References parent class constant

    def __init__(self, collection):
        self.collection = collection  # Instance attribute
        self.log("Initializing")      # Method call
```

Listing 4.11: Attribute Resolution Example

Resolution Order.

1. **Instance Dictionary**: Check obj.___dict___ first
2. **Class Dictionary**: Check class.___dict___

3. **Parent Classes**: Follow MRO up inheritance chain
4. **Descriptors**: Handle special attributes like properties
5. **___getattr___**: Last resort if defined

4.7.2 Dynamic Attribute Creation

Some agents create attributes dynamically:

```
class FrontierAgent(Agent):
    def __init__(self, collection):
        # These become instance attributes
        self.collection = collection
        self.client = OpenAI()  # Runtime object creation
        self.model = SentenceTransformer('...')  # Runtime initialization
```

Listing 4.12: Dynamic Attribute Example

Runtime Attribute Implications.

- **Memory Usage**: Each instance has its own copy
- **Type Safety**: Harder to detect errors statically
- **IDE Support**: Autocomplete may not work perfectly
- **Flexibility**: Can adapt behavior based on runtime conditions

4.8 Special Methods and Magic Methods

4.8.1 ___repr___ Implementation

The ScrapedDeal class implements a custom string representation:

```
class ScrapedDeal:
    def __repr__(self):
        """Return a string to describe this deal"""
        return f"<{self.title}>"

# Usage demonstrates polymorphic behavior
deals = [ScrapedDeal(entry) for entry in entries]
print(deals)  # Automatically calls __repr__ for each deal
```

Listing 4.13: Custom ___repr___ Implementation

Magic Method Benefits.

- **Python Integration**: Works seamlessly with built-in functions
- **Debugging Support**: Better error messages and logging output
- **Collection Display**: Lists and dicts show meaningful information

- **Developer Experience**: More informative interactive sessions

4.8.2 Class Methods and Alternative Constructors

ScrapedDeal provides a class method for creation:

```
class ScrapedDeal:
    @classmethod
    def fetch(cls, show_progress: bool = False) -> List[Self]:
        """Alternative constructor that creates multiple instances"""
        deals = []
        for feed_url in feeds:
            feed = feedparser.parse(feed_url)
            for entry in feed.entries[:10]:
                deals.append(cls(entry))  # cls refers to ScrapedDeal
        return deals

# Usage: Factory method pattern
all_deals = ScrapedDeal.fetch(show_progress=True)
```

Listing 4.14: Class Method as Alternative Constructor

Class Method Advantages.

- **Named Constructors**: More descriptive than multiple ___init___ methods
- **Type Safety**: Return type matches the class
- **Inheritance Support**: Works correctly with subclasses
- **Factory Pattern**: Encapsulates complex object creation logic

4.9 Modern Python Features

4.9.1 Type Annotations in Inheritance

The system uses modern Python type annotations extensively:

```
from typing import List, Dict, Self, Optional

class ScrapedDeal:
    @classmethod
    def fetch(cls, show_progress: bool = False) -> List[Self]:
        """Self refers to the current class, supporting inheritance"""
        pass

class FrontierAgent(Agent):
    def price(self, description: str) -> float:
        """Clear input/output types for better IDE support"""
        pass

```

```
    def find_similars(self, description: str) -> tuple[List[str], List[float]]:
        """Modern Python 3.9+ tuple syntax"""
        pass
```

Listing 4.15: Advanced Type Annotations

Type Annotation Benefits.

- **IDE Support**: Better autocomplete and error detection
- **Documentation**: Types serve as inline documentation
- **Runtime Checking**: Can be used with tools like mypy
- **Refactoring**: Safer code changes with type awareness

4.9.2 Dataclasses and Pydantic Integration

While not using Python's @dataclass decorator, the system uses Pydantic for similar benefits:

```
# Traditional class (ScrapedDeal)
class ScrapedDeal:
    def __init__(self, entry: Dict[str, str]):
        self.title = entry['title']
        self.summary = extract(entry['summary'])
        # Manual attribute assignment

# Pydantic model (Deal)
class Deal(BaseModel):
    product_description: str
    price: float
    url: str
    # Automatic validation, serialization, etc.
```

Listing 4.16: Pydantic vs Traditional Classes

Pydantic Advantages.

- **Automatic Validation**: Type checking at runtime
- **JSON Serialization**: Built-in conversion to/from JSON
- **Schema Generation**: Automatic API documentation
- **IDE Integration**: Enhanced type checking and completion

4.10 Inheritance Patterns and Design Principles

4.10.1 Template Method Pattern

The Agent base class implements the Template Method pattern:

```python
class Agent:
    def log(self, message):
        """Template method - same for all subclasses"""
        color_code = self.BG_BLACK + self.color
        message = f"[{self.name}] {message}"
        logging.info(color_code + message + self.RESET)

# All subclasses use the same logging template
class PlanningAgent(Agent):
    def run(self):
        self.log("Starting planning process")  # Uses template
        # Custom logic here
        self.log("Planning complete")  # Uses template
```

Listing 4.17: Template Method Pattern

4.10.2 Strategy Pattern Through Polymorphism

Different scanner implementations represent the Strategy pattern:

```python
class PlanningAgent(Agent):
    def __init__(self, collection, scanner_strategy="openai"):
        if scanner_strategy == "openai":
            self.scanner = ScannerAgent()
        else:
            self.scanner = ScannerAgentLangChain()
        # Same interface, different implementation
```

Listing 4.18: Strategy Pattern Implementation

4.11 Chapter Summary

This chapter explored the sophisticated class hierarchy and inheritance patterns in the multi-agent system:

- **Single Inheritance**: All agents inherit from common Agent base class
- **Composition Over Inheritance**: Complex agents compose simpler agents
- **Polymorphism**: Common interfaces enable interchangeable components
- **Method Resolution**: Understanding how Python resolves method calls
- **Modern Features**: Type annotations, class methods, and Pydantic integration
- **Design Patterns**: Template Method, Strategy, and Factory patterns

The inheritance design provides a solid foundation for extensibility while maintaining clean separation of concerns. Each agent can focus on its specialized functionality while benefiting from common infrastructure provided by the base class.

The next chapter will dive deeper into individual functions and methods, examining their implementation details, parameter handling, error management, and step-by-step execution processes.

Chapter 5

Function-by-Function Breakdown

This chapter provides an exhaustive analysis of every function and method in the multi-agent system. We examine each function's purpose, parameters, internal logic, return values, error handling, and step-by-step execution process, including memory allocation, object creation, and behind-the-scenes Python mechanics.

5.1 Agent Base Class Methods

5.1.1 Agent.log() Method

```
def log(self, message):
    """
    Log this as an info message, identifying the agent
    """
    color_code = self.BG_BLACK + self.color
    message = f"[{self.name}] {message}"
    logging.info(color_code + message + self.RESET)
```

Listing 5.1: Agent.log() Method Deep Analysis

Function Signature Analysis

Parameters.

- **self**: Reference to the agent instance
- **message**: String content to be logged (no type annotation in original)

Return Value.

- **Return Type**: None (implicitly)
- **Side Effects**: Writes to logging system, modifies terminal color state

Step-by-Step Execution Analysis

Step 1: Color Code Construction.

```
color_code = self.BG_BLACK + self.color
# Python execution:
# 1. Resolve self.BG_BLACK -> '\033[40m' (class attribute lookup)
# 2. Resolve self.color -> specific color string (instance attribute)
# 3. Create new string object via concatenation
# 4. Assign reference to local variable color_code
# Memory: New string object created, old objects may be garbage collected
```

Listing 5.2: Color Code Memory Analysis

Step 2: Message Formatting.

```
message = f"[{self.name}] {message}"
# Python execution:
# 1. Resolve self.name -> agent name string (instance attribute)
# 2. Access message parameter from local scope
# 3. F-string interpolation creates new string object
# 4. Assign to message variable (shadows parameter)
# Memory: New string object created, parameter reference lost
```

Listing 5.3: F-String Processing Analysis

Step 3: Logging Call.

```
logging.info(color_code + message + self.RESET)
# Python execution:
# 1. Import resolution: logging module lookup
# 2. Method resolution: logging.info function
# 3. String concatenation: three strings combined
# 4. Function call with formatted string
# 5. Logging system processes message based on configuration
# Memory: Temporary string for concatenation, then cleanup
```

Listing 5.4: Logging System Integration

Memory Management Details

Object Creation.

1. **color_code**: New string object from concatenation
2. **formatted message**: New string object from f-string interpolation
3. **final string**: New string object from final concatenation
4. **Garbage Collection**: Original objects eligible for cleanup

Attribute Access Performance.

- **self.BG_BLACK**: Class attribute lookup via MRO
- **self.color**: Instance attribute from ___dict___
- **self.name**: Instance attribute from ___dict___

- **self.RESET**: Class attribute lookup via MRO

5.2 Data Processing Functions

5.2.1 extract() Function

```
def extract(html_snippet: str) -> str:
    """
    Use Beautiful Soup to clean up this HTML snippet and extract useful text
    """
    soup = BeautifulSoup(html_snippet, 'html.parser')
    snippet_div = soup.find('div', class_='snippet summary')

    if snippet_div:
        description = snippet_div.get_text(strip=True)
        description = BeautifulSoup(description, 'html.parser').get_text()
        description = re.sub('<[^<]+?>', '', description)
        result = description.strip()
    else:
        result = html_snippet
    return result.replace('\n', ' ')
```

Listing 5.5: HTML Processing Function Analysis

Function Signature Analysis

Parameters.

- **html_snippet: str**: Input HTML string to process
- **Type Annotation**: Modern Python type hint for better IDE support

Return Value.

- **Return Type**: str (explicitly annotated)
- **Content**: Cleaned text extracted from HTML

Step-by-Step Execution Analysis

Step 1: BeautifulSoup Object Creation.

```
soup = BeautifulSoup(html_snippet, 'html.parser')
# Python execution:
# 1. Import resolution: BeautifulSoup class from bs4 module
# 2. Constructor call with HTML string and parser specification
# 3. HTML parsing creates DOM tree structure in memory
# 4. BeautifulSoup object assigned to soup variable
# Memory: DOM tree objects created, references maintained
```

Listing 5.6: HTML Parser Initialization

Step 2: Element Search.

```
snippet_div = soup.find('div', class_='snippet summary')
# Python execution:
# 1. Method resolution: soup.find method
# 2. CSS selector processing: 'div' tag with specific class
# 3. DOM traversal to locate matching element
# 4. Return first matching element or None
# Memory: Element object reference or None assignment
```

Listing 5.7: DOM Navigation Analysis

Step 3: Conditional Processing.

```
if snippet_div:
    # Branch 1: Element found - complex processing
    description = snippet_div.get_text(strip=True)
    description = BeautifulSoup(description, 'html.parser').get_text()
    description = re.sub('<[^<]+?>', '', description)
    result = description.strip()
else:
    # Branch 2: Element not found - fallback
    result = html_snippet
```

Listing 5.8: Conditional Text Extraction

Branch 1 Analysis: Complex Text Processing.

1. **get_text(strip=True)**: Extract text content, remove leading/trailing whitespace
2. **Second BeautifulSoup**: Handle nested HTML entities and tags
3. **re.sub()**: Regular expression to remove any remaining HTML tags
4. **strip()**: Final whitespace cleanup

Branch 2 Analysis: Graceful Degradation.

- **Fallback Strategy**: Returns original input if parsing fails
- **Error Handling**: Prevents function failure on unexpected HTML
- **Robustness**: Continues operation even with malformed input

Step 4: Final Processing.

```
return result.replace('\n', ' ')
# Python execution:
# 1. String method call: result.replace()
# 2. New string object created with newlines replaced by spaces
# 3. Return statement passes string reference to caller
# Memory: New string object created, original result eligible for cleanup
```

Listing 5.9: Newline Normalization

Memory and Performance Analysis

Object Creation Timeline.

1. **BeautifulSoup object**: Complex DOM tree structure
2. **Element references**: Pointers to DOM nodes
3. **Text strings**: Multiple string objects during processing
4. **Regex compilation**: Pattern object cached by re module
5. **Final string**: Clean text result

Performance Considerations.

- **HTML Parsing**: Computationally expensive DOM creation
- **Regular Expressions**: Pattern matching overhead
- **String Operations**: Multiple string object creations
- **Memory Usage**: BeautifulSoup objects can be memory-intensive

5.3 Class Methods and Alternative Constructors

5.3.1 ScrapedDeal.fetch() Class Method

```
@classmethod
def fetch(cls, show_progress : bool = False) -> List[Self]:
    """
    Retrieve all deals from the selected RSS feeds
    """
    deals = []
    feed_iter = tqdm(feeds) if show_progress else feeds
    for feed_url in feed_iter:
        feed = feedparser.parse(feed_url)
        for entry in feed.entries[:10]:
            deals.append(cls(entry))
            time.sleep(0.5)
    return deals
```

Listing 5.10: ScrapedDeal.fetch() Deep Analysis

Method Signature Analysis

Decorator Analysis.

- **@classmethod**: Transforms method to receive class as first parameter
- **cls Parameter**: Reference to ScrapedDeal class (not instance)
- **Alternative Constructor**: Provides different way to create objects

Parameters.

- **cls**: Class reference (ScrapedDeal)
- **show_progress: bool**: Optional progress bar flag with default
- **Default Value**: False if not specified

Return Type.

- **List[Self]**: Modern Python typing for self-referential returns
- **Self Type**: Maintains type safety across inheritance
- **Generic List**: Container of ScrapedDeal instances

Step-by-Step Execution Analysis

Step 1: List Initialization.

```
deals = []
# Python execution:
# 1. Empty list object creation
# 2. Assignment to local variable deals
# Memory: Small list object allocated
```

Listing 5.11: Container Initialization

Step 2: Progress Bar Setup.

```
feed_iter = tqdm(feeds) if show_progress else feeds
# Python execution:
# 1. Evaluate show_progress boolean
# 2. If True: Create tqdm wrapper around feeds list
# 3. If False: Direct reference to feeds list
# 4. Assignment to feed_iter variable
# Memory: Possible tqdm object creation
```

Listing 5.12: Conditional Progress Bar

Step 3: Feed Processing Loop.

```
for feed_url in feed_iter:
    feed = feedparser.parse(feed_url)
    for entry in feed.entries[:10]:
        deals.append(cls(entry))
        time.sleep(0.5)
```

Listing 5.13: RSS Feed Iteration

Outer Loop Analysis.

1. **Iterator Protocol**: feed_iter provides ___next___() method
2. **URL Assignment**: feed_url receives string reference
3. **HTTP Request**: feedparser.parse() makes network call

4. **RSS Parsing**: XML parsing creates feed object structure

Inner Loop Analysis.

1. **List Slicing**: entries[:10] creates new list with first 10 items
2. **Entry Iteration**: Each entry is a dictionary-like object
3. **Object Creation**: cls(entry) calls ScrapedDeal constructor
4. **List Append**: deals.append() adds reference to list
5. **Rate Limiting**: time.sleep(0.5) pauses execution

Step 4: Return Processing.

```
return deals
# Python execution:
# 1. Return deals list reference to caller
# 2. Local variables become eligible for garbage collection
# 3. List and contained objects maintained by return reference
```

Listing 5.14: Result Return

Memory Management Deep Dive

Object Lifecycle.

1. **feeds List**: Module-level list, persistent in memory
2. **tqdm Object**: Created if progress requested, wrapper around feeds
3. **feed Objects**: Created per RSS feed, contains parsed XML
4. **ScrapedDeal Objects**: Created per entry, makes HTTP requests
5. **deals List**: Accumulates references, grows during execution

Network and I/O Operations.

- **RSS Fetching**: HTTP requests to feed URLs
- **Content Downloading**: Additional HTTP requests per deal
- **HTML Parsing**: BeautifulSoup processing per deal
- **Rate Limiting**: Deliberate delays to respect server resources

5.4 Machine Learning Integration Methods

5.4.1 EnsembleAgent.price() Method

```
def price(self, description: str) -> float:
    """
    Run this ensemble model
    Ask each of the models to price the product
```

```
    Then use the Linear Regression model to return the weighted price
    """
    self.log("Running Ensemble Agent - collaborating with specialist, frontier
    and random forest agents")
    specialist = self.specialist.price(description)
    frontier = self.frontier.price(description)
    random_forest = self.random_forest.price(description)
    X = pd.DataFrame({
        'Specialist': [specialist],
        'Frontier': [frontier],
        'RandomForest': [random_forest],
        'Min': [min(specialist, frontier, random_forest)],
        'Max': [max(specialist, frontier, random_forest)],
    })
    y = max(0, self.model.predict(X)[0])
    self.log(f"Ensemble Agent complete - returning ${y:.2f}")
    return y
```

Listing 5.15: Ensemble Price Prediction Analysis

Function Signature Analysis

Parameters.

- **self**: EnsembleAgent instance reference
- **description: str**: Product description for price estimation
- **Type Safety**: Clear input type specification

Return Value.

- **Return Type**: float (price estimate)
- **Constraint**: Non-negative via max(0, ...) constraint
- **Business Logic**: Ensures realistic price predictions

Step-by-Step Execution Analysis

Step 1: Process Initialization.

```
self.log("Running Ensemble Agent - collaborating with specialist, frontier and
    random forest agents")
# Python execution:
# 1. Method resolution: self.log (inherited from Agent)
# 2. String literal passed as message
# 3. Agent.log() formats and outputs message
# Side effects: Terminal output, log file entry
```

Listing 5.16: Logging and Setup

Step 2: Individual Model Predictions.

```
specialist = self.specialist.price(description)
frontier = self.frontier.price(description)
random_forest = self.random_forest.price(description)
# Python execution sequence:
# 1. self.specialist.price() - Remote modal call
# 2. self.frontier.price() - RAG + LLM processing
# 3. self.random_forest.price() - Traditional ML prediction
# Memory: Float values assigned to local variables
# Time: Sequential execution, not parallel
```

Listing 5.17: Parallel Model Invocation

Individual Model Analysis.

- **SpecialistAgent**: Remote ML model via Modal platform
- **FrontierAgent**: Vector search + LLM reasoning
- **RandomForestAgent**: Scikit-learn model prediction
- **Execution Order**: Sequential, could be parallelized

Step 3: Feature Engineering.

```
X = pd.DataFrame({
    'Specialist': [specialist],
    'Frontier': [frontier],
    'RandomForest': [random_forest],
    'Min': [min(specialist, frontier, random_forest)],
    'Max': [max(specialist, frontier, random_forest)],
})
# Python execution:
# 1. Dictionary construction with computed values
# 2. Built-in min() and max() function calls
# 3. pandas.DataFrame constructor call
# 4. DataFrame object creation in memory
# Memory: Dictionary + DataFrame objects
```

Listing 5.18: DataFrame Construction Analysis

Feature Engineering Analysis.

1. **Base Features**: Individual model predictions
2. **Statistical Features**: Min and max values across models
3. **Feature Engineering**: Creating additional predictive features
4. **Data Structure**: Single-row DataFrame for sklearn compatibility

Step 4: Meta-Model Prediction.

```
y = max(0, self.model.predict(X)[0])
# Python execution:
# 1. self.model reference resolution (loaded joblib model)
```

```
# 2. predict() method call with DataFrame
# 3. Array indexing [0] to get scalar value
# 4. max() function ensures non-negative result
# 5. Assignment to y variable
# Memory: NumPy array from prediction, scalar extraction
```

Listing 5.19: Ensemble Model Execution

Step 5: Result Logging and Return.

```
self.log(f"Ensemble Agent complete - returning ${y:.2f}")
return y
# Python execution:
# 1. F-string formatting with currency display
# 2. Logging call (inherited method)
# 3. Return float value to caller
# Memory: Formatted string creation, then cleanup
```

Listing 5.20: Completion Processing

Machine Learning Pipeline Analysis

Ensemble Learning Concepts.

- **Base Learners**: Three different ML approaches
- **Meta-Learner**: Linear regression combining predictions
- **Feature Augmentation**: Statistical features enhance prediction
- **Stacking**: Advanced ensemble technique implementation

Performance Characteristics.

- **Latency**: Sum of individual model latencies + meta-model
- **Accuracy**: Typically better than individual models
- **Robustness**: Less sensitive to individual model errors
- **Complexity**: Higher computational and maintenance overhead

5.5 External API Integration Methods

5.5.1 FrontierAgent.find_similars() Method

```
def find_similars(self, description: str):
    """
    Return a list of items similar to the given one by looking in the Chroma
    datastore
    """
    self.log("Frontier Agent is performing a RAG search of the Chroma datastore
    to find 5 similar products")
```

```
    vector = self.model.encode([description])
    results =
    self.collection.query(query_embeddings=vector.astype(float).tolist(),
    n_results=5)
    documents = results['documents'][0][:]
    prices = [m['price'] for m in results['metadatas'][0][:]]
    self.log("Frontier Agent has found similar products")
    return documents, prices
```

Listing 5.21: Vector Similarity Search Analysis

Function Signature Analysis

Parameters.

- **self**: FrontierAgent instance reference
- **description: str**: Product description to find similarities for
- **No Type Annotation**: Return type could be improved

Return Value.

- **Return Type**: Tuple[List[str], List[float]] (implicit)
- **documents**: List of similar product descriptions
- **prices**: Corresponding prices for similar products

Step-by-Step Execution Analysis

Step 1: Process Logging.

```
self.log("Frontier Agent is performing a RAG search of the Chroma datastore to
    find 5 similar products")
# Python execution:
# 1. Inherited log method call
# 2. Descriptive message about RAG operation
# 3. Terminal/log output generation
```

Listing 5.22: RAG Process Announcement

Step 2: Vector Encoding.

```
vector = self.model.encode([description])
# Python execution:
# 1. self.model resolution (SentenceTransformer instance)
# 2. List creation with single description string
# 3. encode() method call - neural network forward pass
# 4. NumPy array returned with semantic embedding
# Memory: Input list, output NumPy array (384 dimensions)
```

Listing 5.23: Text to Vector Transformation

Vector Encoding Deep Dive.

1. **Model Type**: all-MiniLM-L6-v2 sentence transformer
2. **Input Processing**: Tokenization and attention mechanisms
3. **Neural Computation**: Transformer forward pass
4. **Output**: 384-dimensional dense vector representation
5. **Semantic Meaning**: Vector captures text semantic content

Step 3: Vector Database Query.

```
results = self.collection.query(
    query_embeddings=vector.astype(float).tolist(),
    n_results=5
)
# Python execution:
# 1. vector.astype(float) - ensure float32/float64 data type
# 2. .tolist() - convert NumPy array to Python list
# 3. self.collection.query() - ChromaDB search operation
# 4. Cosine similarity computation across stored vectors
# 5. Top-5 most similar items returned
# Memory: Type conversion, API call, result object
```

Listing 5.24: ChromaDB Similarity Search

ChromaDB Query Analysis.

- **Vector Comparison**: Cosine similarity computation
- **Index Search**: Efficient nearest neighbor search
- **Result Ranking**: Top-K results by similarity score
- **Metadata Retrieval**: Associated price information included

Step 4: Result Processing.

```
documents = results['documents'][0][:]
prices = [m['price'] for m in results['metadatas'][0][:]]
# Python execution:
# 1. Dictionary access: results['documents']
# 2. List indexing: [0] gets first query result batch
# 3. List slicing: [:] creates copy of document list
# 4. List comprehension: Extract price from metadata objects
# 5. Assignments to local variables
# Memory: New list objects created from query results
```

Listing 5.25: Query Result Extraction

Data Structure Analysis.

- **results Structure**: Nested dictionary with lists
- **documents**: Text descriptions of similar items

- **metadatas**: Associated information including prices
- **List Processing**: Comprehension for clean data extraction

Step 5: Completion and Return.

```
self.log("Frontier Agent has found similar products")
return documents, prices
# Python execution:
# 1. Success logging message
# 2. Tuple creation from two list objects
# 3. Return tuple reference to caller
# Memory: Tuple object creation, local variables cleanup
```

Listing 5.26: Result Return Processing

Memory and Performance Analysis

Computational Complexity.

- **Vector Encoding**: O(n) where n is text length
- **Similarity Search**: O(log k) with proper indexing
- **Result Processing**: O(m) where m is number of results
- **Overall**: Dominated by neural network forward pass

Memory Usage.

- **Input Vector**: 384 floats ≈ 1.5KB
- **Query Results**: Variable size based on document length
- **Processed Lists**: Additional copies for clean interface
- **Cleanup**: Intermediate objects eligible for GC

5.6 Error Handling and Robustness Patterns

5.6.1 MessagingAgent.push() Method

```
def push(self, text):
    """
    Send a Push Notification using the Pushover API
    """
    self.log("Messaging Agent is sending a push notification")
    conn = http.client.HTTPSConnection("api.pushover.net:443")
    conn.request("POST", "/1/messages.json",
      urllib.parse.urlencode({
        "token": self.pushover_token,
        "user": self.pushover_user,
        "message": text,
        "sound": "cashregister"
```

```
        }), { "Content-type": "application/x-www-form-urlencoded" })
    conn.getresponse()
```

Listing 5.27: Push Notification Error Handling

Function Signature Analysis

Parameters.

- **self**: MessagingAgent instance reference
- **text**: Message content to send (no type annotation)
- **Missing Annotations**: Could benefit from type hints

Return Value.

- **Return Type**: None (implicit)
- **Side Effects**: HTTP request sent, potential network errors
- **Error Handling**: Limited error handling present

Step-by-Step Execution Analysis

Step 1: Process Logging.

```
self.log("Messaging Agent is sending a push notification")
# Python execution:
# 1. Inherited log method call
# 2. Process status announcement
# 3. Terminal/log output for debugging
```

Listing 5.28: Notification Process Start

Step 2: HTTP Connection Setup.

```
conn = http.client.HTTPSConnection("api.pushover.net:443")
# Python execution:
# 1. http.client module resolution
# 2. HTTPSConnection class instantiation
# 3. SSL/TLS connection preparation (not yet established)
# 4. Connection object assigned to local variable
# Memory: Connection object with SSL context
```

Listing 5.29: HTTPS Connection Establishment

Connection Analysis.

- **Protocol**: HTTPS for secure communication
- **Host**: Pushover API endpoint
- **Port**: Explicit port 443 specification
- **SSL Context**: Automatic certificate verification

Step 3: Request Data Preparation.

```
urllib.parse.urlencode({
    "token": self.pushover_token,
    "user": self.pushover_user,
    "message": text,
    "sound": "cashregister"
})
# Python execution:
# 1. Dictionary creation with API parameters
# 2. Instance attribute access for token/user
# 3. urllib.parse.urlencode() call
# 4. URL-encoded string creation
# Memory: Dictionary object, encoded string
```

Listing 5.30: HTTP Request Construction

Data Encoding Analysis.

- **URL Encoding**: Special characters properly escaped
- **Form Data**: application/x-www-form-urlencoded format
- **API Parameters**: token, user, message, sound
- **Custom Sound**: "cashregister" for deal notifications

Step 4: HTTP Request Execution.

```
conn.request("POST", "/1/messages.json",
  encoded_data,
  { "Content-type": "application/x-www-form-urlencoded" })
# Python execution:
# 1. HTTP POST method specification
# 2. API endpoint path: /1/messages.json
# 3. Request body: URL-encoded form data
# 4. Headers: Content-Type specification
# 5. Network transmission of HTTP request
# Network: SSL handshake, HTTP request sent
```

Listing 5.31: POST Request Transmission

Step 5: Response Handling.

```
conn.getresponse()
# Python execution:
# 1. HTTP response reception from server
# 2. Response object creation with status/headers/body
# 3. Response object returned but not used
# 4. Connection cleanup (implicit)
# Issue: Response not checked for errors!
```

Listing 5.32: Response Processing

Error Handling Analysis

Missing Error Handling.

- **Network Errors**: No try/except for connection failures
- **HTTP Errors**: Response status code not checked
- **SSL Errors**: Certificate validation failures not handled
- **Timeout Errors**: No timeout configuration

Improved Error Handling.

```python
def push_with_error_handling(self, text):
    try:
        self.log("Messaging Agent is sending a push notification")
        conn = http.client.HTTPSConnection("api.pushover.net:443", timeout=10)

        data = urllib.parse.urlencode({
            "token": self.pushover_token,
            "user": self.pushover_user,
            "message": text,
            "sound": "cashregister"
        })

        conn.request("POST", "/1/messages.json", data,
                     {"Content-type": "application/x-www-form-urlencoded"})

        response = conn.getresponse()
        if response.status != 200:
            self.log(f"Pushover API error: {response.status} {response.reason}")
            return False

        self.log("Push notification sent successfully")
        return True

    except (http.client.HTTPException, OSError) as e:
        self.log(f"Network error sending push notification: {e}")
        return False
    finally:
        conn.close()  # Ensure connection cleanup
```

Listing 5.33: Enhanced Error Handling Pattern

5.7 Complex Business Logic Methods

5.7.1 PlanningAgent.plan() Method

```python
def plan(self, memory: List[str] = []) -> Optional[Opportunity]:
    """
    Run the full workflow:
    1. Use the ScannerAgent to find deals from RSS feeds
```

```
    2. Use the EnsembleAgent to estimate them
    3. Use the MessagingAgent to send a notification of deals
    """
    self.log("Planning Agent is kicking off a run")
    selection = self.scanner.scan(memory=memory)
    if selection:
        opportunities = [self.run(deal) for deal in selection.deals[:5]]
        opportunities.sort(key=lambda opp: opp.discount, reverse=True)
        best = opportunities[0]
        self.log(f"Planning Agent has identified the best deal has discount
    ${best.discount:.2f}")
        if best.discount > self.DEAL_THRESHOLD:
            self.messenger.alert(best)
        self.log("Planning Agent has completed a run")
        return best if best.discount > self.DEAL_THRESHOLD else None
    return None
```

Listing 5.34: Master Planning Method Analysis

Function Signature Analysis

Parameters.

- **self**: PlanningAgent instance reference
- **memory: List[str]**: Previously processed deal URLs
- **Default Value**: Empty list (mutable default - potential issue)
- **Return Type**: Optional[Opportunity] - may return None

Mutable Default Argument Issue.

```
def plan(self, memory: List[str] = []):  # Problematic!
    # Same list object reused across calls
    # Better: memory: Optional[List[str]] = None
    # Then: if memory is None: memory = []
```

Listing 5.35: Mutable Default Problem

Step-by-Step Execution Analysis

Step 1: Workflow Initiation.

```
self.log("Planning Agent is kicking off a run")
# Python execution:
# 1. Log workflow start for debugging/monitoring
# 2. Indicates beginning of complete deal processing cycle
```

Listing 5.36: Planning Process Start

Step 2: Deal Discovery.

```
selection = self.scanner.scan(memory=memory)
# Python execution:
# 1. self.scanner attribute access (ScannerAgent instance)
```

```
# 2. scan() method call with memory parameter
# 3. RSS feed processing, ML filtering, deal extraction
# 4. DealSelection object or None returned
# Side effects: Network requests, LLM API calls
```

Listing 5.37: Scanner Agent Invocation

Step 3: Conditional Processing.

```
if selection:
    # Branch 1: Deals found - process them
    opportunities = [self.run(deal) for deal in selection.deals[:5]]
    opportunities.sort(key=lambda opp: opp.discount, reverse=True)
    best = opportunities[0]
    # ... processing continues
else:
    # Branch 2: No deals found
    return None
```

Listing 5.38: Deal Processing Branch

Branch 1 Analysis: Deal Processing.

1. **List Comprehension**: `[self.run(deal) for deal in selection.deals[:5]]`
2. **Slice Operation**: Process maximum 5 deals for performance
3. **Opportunity Creation**: Each deal converted to Opportunity object
4. **List Sorting**: Sort by discount amount, descending order
5. **Best Selection**: First item after sorting is highest discount

List Comprehension Deep Dive.

```
opportunities = [self.run(deal) for deal in selection.deals[:5]]
# Python execution:
# 1. selection.deals[:5] - slice first 5 deals
# 2. Iterator creation over sliced list
# 3. For each deal:
#    a. self.run(deal) method call
#    b. EnsembleAgent price estimation
#    c. Opportunity object creation
#    d. Add to result list
# 4. Complete list assigned to opportunities
# Performance: Sequential execution of 5 price estimations
```

Listing 5.39: List Comprehension Execution

Step 4: Sorting and Selection.

```
opportunities.sort(key=lambda opp: opp.discount, reverse=True)
best = opportunities[0]
# Python execution:
# 1. Lambda function creation for key extraction
# 2. List.sort() in-place sorting (Timsort algorithm)
```

```
# 3. reverse=True for descending order
# 4. Index access [0] for highest discount opportunity
# Memory: List sorted in-place, lambda object for key
```

Listing 5.40: Opportunity Ranking

Step 5: Decision Making.

```
if best.discount > self.DEAL_THRESHOLD:
    self.messenger.alert(best)
    return best
else:
    return None
# Python execution:
# 1. Attribute access: best.discount (float value)
# 2. Class constant access: self.DEAL_THRESHOLD (50)
# 3. Numeric comparison operation
# 4. Conditional execution of alert
# 5. Return appropriate value based on threshold
```

Listing 5.41: Threshold Decision Logic

Business Logic Analysis

Workflow Orchestration.

1. **Deal Discovery**: Scanner finds new deals from RSS feeds
2. **Price Estimation**: Ensemble model predicts fair prices
3. **Opportunity Evaluation**: Calculate potential savings
4. **Ranking**: Sort by discount amount
5. **Filtering**: Apply business threshold for notifications
6. **Communication**: Alert user of significant deals

Performance Characteristics.

- **Network Bound**: RSS feeds, HTTP requests, API calls
- **Compute Bound**: ML model inference, text processing
- **Sequential**: No parallelization of deal processing
- **Threshold Dependent**: Output depends on deal quality

5.8 Chapter Summary

This chapter provided comprehensive analysis of key functions and methods throughout the multi-agent system:

- **Base Infrastructure**: Agent.log() method with color formatting

- **Data Processing**: HTML extraction and cleaning functions
- **Object Creation**: Class methods and alternative constructors
- **ML Integration**: Ensemble prediction and vector search methods
- **API Integration**: HTTP requests and external service calls
- **Business Logic**: Complex workflow orchestration

Each function demonstrates different aspects of Python programming:

- Memory management and object lifecycle
- Error handling patterns and robustness
- Network programming and API integration
- Machine learning pipeline construction
- String processing and data transformation
- Conditional logic and decision making

The analysis reveals both strengths (modular design, clear interfaces) and areas for improvement (error handling, performance optimization, type annotations) in the codebase. Understanding these implementation details provides insight into how high-level agent behaviors emerge from carefully orchestrated low-level operations.

Chapter 6

Python Concepts and Examples

This chapter identifies and explains all Python language features, concepts, and programming patterns used throughout the multi-agent system. For each concept, we provide clear explanations, simple illustrative examples, and then demonstrate how they're applied in the actual system code.

6.1 Object-Oriented Programming Concepts

6.1.1 Classes and Objects

Concept Explanation. Classes define blueprints for creating objects, encapsulating data (attributes) and behavior (methods). Objects are instances of classes that maintain their own state.

Simple Example.

```
class Dog:
    species = "Canis lupus"  # Class attribute

    def __init__(self, name, age):
        self.name = name      # Instance attribute
        self.age = age        # Instance attribute

    def bark(self):           # Instance method
        return f"{self.name} says woof!"

# Object creation and usage
my_dog = Dog("Rex", 3)
print(my_dog.bark())  # Rex says woof!
print(my_dog.species) # Canis lupus
```

Listing 6.1: Basic Class Concept

System Implementation.

```
class Agent:
    # Class attributes (shared across all instances)
    RED = '\033[31m'
    GREEN = '\033[32m'
```

```
    RESET = '\033[0m'

    # Instance attributes with type hints
    name: str = ""
    color: str = '\033[37m'

    def log(self, message):  # Instance method
        """Log messages with agent identification"""
        color_code = self.BG_BLACK + self.color
        message = f"[{self.name}] {message}"
        logging.info(color_code + message + self.RESET)

# Concrete implementation
class SpecialistAgent(Agent):
    name = "Specialist Agent"  # Override class attribute
    color = Agent.RED          # Reference parent class attribute
```

Listing 6.2: Agent Class Implementation

Key Points in System.

- **Class Attributes**: Color constants shared across all agents
- **Instance Attributes**: Each agent has unique name and color
- **Method Implementation**: log() provides common functionality
- **Attribute Override**: Subclasses customize name and color

6.1.2 Inheritance

Concept Explanation. Inheritance allows classes to inherit attributes and methods from parent classes, enabling code reuse and establishing "is-a" relationships.

Simple Example.

```
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        pass  # Abstract method to be overridden

class Dog(Animal):  # Dog inherits from Animal
    def speak(self):
        return f"{self.name} barks"

class Cat(Animal):  # Cat inherits from Animal
    def speak(self):
        return f"{self.name} meows"

# Polymorphic usage
animals = [Dog("Rex"), Cat("Whiskers")]
```

```
for animal in animals:
    print(animal.speak())  # Calls appropriate subclass method
```

Listing 6.3: Basic Inheritance Example

System Implementation.

```
class Agent:  # Base class
    def log(self, message):
        """Template method used by all subclasses"""
        color_code = self.BG_BLACK + self.color
        message = f"[{self.name}] {message}"
        logging.info(color_code + message + self.RESET)

class PlanningAgent(Agent):  # Inherits from Agent
    name = "Planning Agent"
    color = Agent.GREEN

    def __init__(self, collection):
        # Uses inherited log() method
        self.log("Planning Agent is initializing")
        self.scanner = ScannerAgent()
        self.ensemble = EnsembleAgent(collection)
        self.messenger = MessagingAgent()
        self.log("Planning Agent is ready")

class EnsembleAgent(Agent):  # Also inherits from Agent
    name = "Ensemble Agent"
    color = Agent.YELLOW

    def price(self, description: str) -> float:
        # Uses inherited log() method
        self.log("Running Ensemble Agent")
        # Custom logic here
        return predicted_price
```

Listing 6.4: Agent Inheritance Hierarchy

Inheritance Benefits in System.

- **Code Reuse**: All agents get logging functionality for free
- **Consistency**: Common interface across all agent types
- **Maintainability**: Changes to Agent affect all subclasses
- **Polymorphism**: Can treat all agents uniformly

6.1.3 Composition

Concept Explanation. Composition is a "has-a" relationship where objects contain other objects as components, enabling complex behavior through combining simpler parts.

Simple Example.

```python
class Engine:
    def __init__(self, horsepower):
        self.horsepower = horsepower

    def start(self):
        return "Engine started"

class Car:
    def __init__(self, make, engine):
        self.make = make
        self.engine = engine  # Car HAS-A Engine (composition)

    def start(self):
        return f"{self.make}: {self.engine.start()}"

# Usage
v8_engine = Engine(400)
muscle_car = Car("Mustang", v8_engine)
print(muscle_car.start())  # Mustang: Engine started
```

Listing 6.5: Basic Composition Example

System Implementation.

```python
class EnsembleAgent(Agent):
    def __init__(self, collection):
        self.log("Initializing Ensemble Agent")

        # Composition: EnsembleAgent HAS-A these agents
        self.specialist = SpecialistAgent()      # HAS-A SpecialistAgent
        self.frontier = FrontierAgent(collection) # HAS-A FrontierAgent
        self.random_forest = RandomForestAgent() # HAS-A RandomForestAgent
        self.model = joblib.load('ensemble_model.pkl')  # HAS-A ML model

        self.log("Ensemble Agent is ready")

    def price(self, description: str) -> float:
        """Coordinates composed agents to make ensemble prediction"""
        # Delegate to composed agents
        specialist = self.specialist.price(description)
        frontier = self.frontier.price(description)
        random_forest = self.random_forest.price(description)

        # Combine their results
        X = pd.DataFrame({
            'Specialist': [specialist],
            'Frontier': [frontier],
            'RandomForest': [random_forest],
            'Min': [min(specialist, frontier, random_forest)],
            'Max': [max(specialist, frontier, random_forest)],
        })
```

```
        return max(0, self.model.predict(X)[0])
```

Listing 6.6: EnsembleAgent Composition Pattern

Composition Benefits in System.

- **Flexibility**: Can change components at runtime
- **Modularity**: Each component has single responsibility
- **Loose Coupling**: Components don't depend on each other
- **Reusability**: Same components used in different contexts

6.2 Modern Python Features

6.2.1 Type Annotations

Concept Explanation. Type annotations provide static type information for variables, function parameters, and return values, improving code documentation and enabling better IDE support.

Simple Example.

```
from typing import List, Dict, Optional

def greet(name: str, age: int) -> str:
    """Function with typed parameters and return value"""
    return f"Hello {name}, you are {age} years old"

def process_scores(scores: List[float]) -> Dict[str, float]:
    """Function with complex type annotations"""
    return {
        'average': sum(scores) / len(scores),
        'max': max(scores),
        'min': min(scores)
    }

# Variable annotations
name: str = "Alice"
numbers: List[int] = [1, 2, 3, 4, 5]
user_data: Optional[Dict[str, str]] = None
```

Listing 6.7: Type Annotation Examples

System Implementation.

```
from typing import List, Dict, Self, Optional

class ScrapedDeal:
    # Instance attributes with type annotations
    category: str
    title: str
    summary: str
```

```
    url: str
    details: str
    features: str

    @classmethod
    def fetch(cls, show_progress: bool = False) -> List[Self]:
        """Class method with modern Self type annotation"""
        deals = []
        for feed_url in feeds:
            feed = feedparser.parse(feed_url)
            for entry in feed.entries[:10]:
                deals.append(cls(entry))
        return deals

class FrontierAgent(Agent):
    def price(self, description: str) -> float:
        """Clear input and output types"""
        documents, prices = self.find_similars(description)
        # ... processing
        return result

    def find_similars(self, description: str) -> tuple[List[str], List[float]]:
        """Modern Python 3.9+ tuple syntax"""
        # ... implementation
        return documents, prices

class PlanningAgent(Agent):
    def plan(self, memory: List[str] = []) -> Optional[Opportunity]:
        """Optional return type indicates possible None"""
        selection = self.scanner.scan(memory=memory)
        if selection:
            # ... processing
            return best_opportunity
        return None
```

Listing 6.8: System Type Annotations

Type Annotation Benefits in System.

- **IDE Support**: Better autocomplete and error detection
- **Documentation**: Types serve as inline documentation
- **Maintainability**: Easier to understand interfaces
- **Debugging**: Type checkers can catch errors early

6.2.2 F-Strings (Formatted String Literals)

Concept Explanation. F-strings provide a readable and efficient way to format strings by embedding expressions directly inside string literals.

Simple Example.

```
name = "Alice"
age = 30
score = 87.5

# Basic f-string interpolation
greeting = f"Hello, {name}! You are {age} years old."

# Expressions inside f-strings
result = f"{name}'s score is {score:.1f}%, which is {'PASS' if score >= 60 else
    'FAIL'}"

# Format specifications
price = 123.456
formatted = f"Price: ${price:.2f}"  # Price: $123.46

# Debugging with f-strings
x, y = 10, 20
debug_info = f"{x=}, {y=}, {x+y=}"  # x=10, y=20, x+y=30
```

Listing 6.9: F-String Examples

System Implementation.

```
class Agent:
    def log(self, message):
        """F-string for agent identification"""
        color_code = self.BG_BLACK + self.color
        message = f"[{self.name}] {message}"  # F-string interpolation
        logging.info(color_code + message + self.RESET)

class EnsembleAgent(Agent):
    def price(self, description: str) -> float:
        self.log("Running Ensemble Agent - collaborating with specialist,
    frontier and random forest agents")
        # ... processing
        y = max(0, self.model.predict(X)[0])
        self.log(f"Ensemble Agent complete - returning ${y:.2f}")  # Currency
    formatting
        return y

class PlanningAgent(Agent):
    def plan(self, memory: List[str] = []) -> Optional[Opportunity]:
        # ... processing
        best = opportunities[0]
        # F-string with currency formatting
        self.log(f"Planning Agent has identified the best deal has discount
    ${best.discount:.2f}")
        return best if best.discount > self.DEAL_THRESHOLD else None

class MessagingAgent(Agent):
    def alert(self, opportunity: Opportunity):
```

```
        """Complex f-string with multiple expressions"""
        text = f"Deal Alert! Price=${opportunity.deal.price:.2f}, "
        text += f"Estimate=${opportunity.estimate:.2f}, "
        text += f"Discount=${opportunity.discount:.2f} :"
        text += opportunity.deal.product_description[:10]+'... '
        text += opportunity.deal.url
        # Send notification
```

Listing 6.10: F-Strings in Agent System

F-String Advantages in System.

- **Readability**: Clear variable interpolation
- **Performance**: Faster than .format() or % formatting
- **Flexibility**: Support for expressions and formatting specs
- **Currency Display**: Consistent price formatting (.2f)

6.2.3 List Comprehensions

Concept Explanation. List comprehensions provide a concise way to create lists by applying expressions to items in iterables, optionally filtering them with conditions.

Simple Example.

```
# Basic list comprehension
numbers = [1, 2, 3, 4, 5]
squares = [x**2 for x in numbers]  # [1, 4, 9, 16, 25]

# With conditional filtering
even_squares = [x**2 for x in numbers if x % 2 == 0]  # [4, 16]

# With expression transformation
words = ["hello", "world", "python"]
lengths = [len(word) for word in words]  # [5, 5, 6]

# Nested comprehension
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
flattened = [item for row in matrix for item in row]  # [1,2,3,4,5,6,7,8,9]

# Dictionary comprehension
word_lengths = {word: len(word) for word in words}  # {'hello': 5, 'world': 5,
    'python': 6}
```

Listing 6.11: List Comprehension Examples

System Implementation.

```
class PlanningAgent(Agent):
    def plan(self, memory: List[str] = []) -> Optional[Opportunity]:
        selection = self.scanner.scan(memory=memory)
        if selection:
```

```
            # List comprehension to convert deals to opportunities
            opportunities = [self.run(deal) for deal in selection.deals[:5]]
            opportunities.sort(key=lambda opp: opp.discount, reverse=True)
            best = opportunities[0]
            return best if best.discount > self.DEAL_THRESHOLD else None
        return None

class FrontierAgent(Agent):
    def find_similars(self, description: str):
        vector = self.model.encode([description])
        results =
    self.collection.query(query_embeddings=vector.astype(float).tolist(),
    n_results=5)
        documents = results['documents'][0][:]
        # List comprehension to extract prices from metadata
        prices = [m['price'] for m in results['metadatas'][0][:]]
        return documents, prices

class ScannerAgent(Agent):
    def fetch_deals(self, memory) -> List[ScrapedDeal]:
        urls = [opp.deal.url for opp in memory]  # Extract URLs
        scraped = ScrapedDeal.fetch()
        # Filter deals not already in memory
        result = [scrape for scrape in scraped if scrape.url not in urls]
        return result
```

Listing 6.12: List Comprehensions in Agent System

List Comprehension Benefits in System.

- **Conciseness**: Replace multi-line loops with single expressions
- **Readability**: Clear intent for data transformation
- **Performance**: Often faster than equivalent for loops
- **Functional Style**: Encourages immutable data transformations

6.3 Advanced Python Concepts

6.3.1 Decorators

Concept Explanation. Decorators are functions that modify or extend the behavior of other functions or classes without permanently modifying them.

Simple Example.

```
def timing_decorator(func):
    """Decorator to measure function execution time"""
    import time
    def wrapper(*args, **kwargs):
        start = time.time()
```

```
        result = func(*args, **kwargs)
        end = time.time()
        print(f"{func.__name__} took {end - start:.2f} seconds")
        return result
    return wrapper

@timing_decorator
def slow_function():
    import time
    time.sleep(1)
    return "Done"

# Usage
result = slow_function()  # Prints: slow_function took 1.00 seconds

# Class method decorator
class MyClass:
    @classmethod
    def create_from_string(cls, data: str):
        """Alternative constructor"""
        return cls(data.split(','))

    @staticmethod
    def utility_function(x, y):
        """Function that doesn't need class or instance"""
        return x + y
```

Listing 6.13: Decorator Examples

System Implementation.

```
class ScrapedDeal:
    @classmethod  # Class method decorator
    def fetch(cls, show_progress: bool = False) -> List[Self]:
        """Alternative constructor for creating multiple deals"""
        deals = []
        feed_iter = tqdm(feeds) if show_progress else feeds
        for feed_url in feed_iter:
            feed = feedparser.parse(feed_url)
            for entry in feed.entries[:10]:
                deals.append(cls(entry))  # cls refers to ScrapedDeal
                time.sleep(0.5)
        return deals

# Potential decorator for logging (not implemented in original)
def log_execution(func):
    """Decorator to log method execution"""
    def wrapper(self, *args, **kwargs):
        self.log(f"Starting {func.__name__}")
        result = func(self, *args, **kwargs)
        self.log(f"Completed {func.__name__}")
        return result
```

```
    return wrapper

class EnhancedAgent(Agent):
    @log_execution
    def price(self, description: str) -> float:
        # Method execution automatically logged
        return self.calculate_price(description)
```

Listing 6.14: Decorators in Agent System

Decorator Benefits.

- **Separation of Concerns**: Keep core logic separate from cross-cutting concerns
- **Reusability**: Same decorator can be applied to multiple functions
- **Clean Code**: Avoid repetitive boilerplate code
- **Flexibility**: Add/remove behavior without modifying original functions

6.3.2 Context Managers

Concept Explanation. Context managers provide a way to allocate and release resources precisely when needed, typically using the `with` statement.

Simple Example.

```
# Built-in context manager for file handling
with open('data.txt', 'r') as file:
    content = file.read()
    # File automatically closed when leaving with block

# Custom context manager
class Timer:
    def __enter__(self):
        import time
        self.start = time.time()
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        import time
        self.end = time.time()
        print(f"Elapsed: {self.end - self.start:.2f} seconds")

# Usage
with Timer():
    # Code to time
    sum(range(1000000))

# Using contextlib
from contextlib import contextmanager

@contextmanager
```

```
def database_transaction():
    print("Starting transaction")
    try:
        yield "connection"
        print("Committing transaction")
    except Exception:
        print("Rolling back transaction")
        raise
    finally:
        print("Cleaning up")

with database_transaction() as conn:
    # Database operations here
    pass
```

Listing 6.15: Context Manager Examples

System Implementation.

```
# The system could benefit from context managers for:

class AgentContext:
    """Context manager for agent lifecycle"""
    def __init__(self, agent_class, *args, **kwargs):
        self.agent_class = agent_class
        self.args = args
        self.kwargs = kwargs
        self.agent = None

    def __enter__(self):
        self.agent = self.agent_class(*self.args, **self.kwargs)
        self.agent.log("Agent context started")
        return self.agent

    def __exit__(self, exc_type, exc_val, exc_tb):
        if self.agent:
            self.agent.log("Agent context ending")
        # Cleanup resources if needed

# Usage
with AgentContext(FrontierAgent, collection) as agent:
    price = agent.price("laptop computer")
    # Agent automatically cleaned up

# HTTP connection context manager
class HTTPConnection:
    def __init__(self, host, port=443):
        self.host = host
        self.port = port
        self.conn = None

    def __enter__(self):
```

```
        import http.client
        self.conn = http.client.HTTPSConnection(f"{self.host}:{self.port}")
        return self.conn

    def __exit__(self, exc_type, exc_val, exc_tb):
        if self.conn:
            self.conn.close()

# Enhanced MessagingAgent with context manager
class EnhancedMessagingAgent(Agent):
    def push(self, text):
        with HTTPConnection("api.pushover.net") as conn:
            data = urllib.parse.urlencode({
                "token": self.pushover_token,
                "user": self.pushover_user,
                "message": text,
                "sound": "cashregister"
            })
            conn.request("POST", "/1/messages.json", data,
                         {"Content-type": "application/x-www-form-urlencoded"})
            response = conn.getresponse()
            return response.status == 200
```

Listing 6.16: Context Managers in Agent System (Potential)

Context Manager Benefits.

- **Resource Management**: Automatic cleanup of resources
- **Exception Safety**: Cleanup occurs even if exceptions happen
- **Clear Intent**: Makes resource lifecycle explicit
- **Reduced Errors**: Prevents resource leaks

6.4 Data Structures and Collections

6.4.1 Dictionaries and Dictionary Operations

Concept Explanation. Dictionaries are key-value mappings that provide fast lookups and flexible data storage.

Simple Example.

```
# Basic dictionary operations
person = {
    'name': 'Alice',
    'age': 30,
    'city': 'New York'
}

# Dictionary access and modification
```

```python
print(person['name'])        # Alice
person['age'] = 31           # Update existing key
person['job'] = 'Engineer'   # Add new key

# Safe access with get()
email = person.get('email', 'not_provided@example.com')

# Dictionary comprehension
squares = {x: x**2 for x in range(5)}  # {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}

# Merging dictionaries (Python 3.9+)
defaults = {'color': 'blue', 'size': 'medium'}
options = {'color': 'red'}
config = defaults | options  # {'color': 'red', 'size': 'medium'}
```

Listing 6.17: Dictionary Examples

System Implementation.

```python
class ScrapedDeal:
    def __init__(self, entry: Dict[str, str]):
        """RSS entry is a dictionary-like object"""
        self.title = entry['title']                    # Dictionary access
        self.summary = extract(entry['summary'])       # Dictionary access
        self.url = entry['links'][0]['href']           # Nested dictionary access
        # ... more processing

class EnsembleAgent(Agent):
    def price(self, description: str) -> float:
        # ... get individual predictions

        # Create DataFrame from dictionary
        X = pd.DataFrame({
            'Specialist': [specialist],
            'Frontier': [frontier],
            'RandomForest': [random_forest],
            'Min': [min(specialist, frontier, random_forest)],
            'Max': [max(specialist, frontier, random_forest)],
        })
        return max(0, self.model.predict(X)[0])

class MessagingAgent(Agent):
    def push(self, text):
        # Dictionary for POST data
        data = urllib.parse.urlencode({
            "token": self.pushover_token,
            "user": self.pushover_user,
            "message": text,
            "sound": "cashregister"
        })
        # ... send request

```

```
class FrontierAgent(Agent):
    def messages_for(self, description: str, similars: List[str], prices:
    List[float]) -> List[Dict[str, str]]:
        """Create OpenAI message format (list of dictionaries)"""
        return [
            {"role": "system", "content": system_message},
            {"role": "user", "content": user_prompt},
            {"role": "assistant", "content": "Price is $"}
        ]

    def find_similars(self, description: str):
        results = self.collection.query(...)
        # Access nested dictionary structure
        documents = results['documents'][0][:]
        prices = [m['price'] for m in results['metadatas'][0][:]]
        return documents, prices
```

Listing 6.18: Dictionary Usage in Agent System

Dictionary Patterns in System.

- **Configuration Data**: RSS entries, API parameters
- **Structured Data**: DataFrame construction, message formats
- **Nested Access**: Multi-level dictionary navigation
- **Safe Access**: Using get() method with defaults

6.4.2 Lists and List Operations

Concept Explanation. Lists are ordered, mutable sequences that can contain any type of objects.

Simple Example.

```
# Basic list operations
numbers = [1, 2, 3, 4, 5]
numbers.append(6)           # Add to end: [1, 2, 3, 4, 5, 6]
numbers.insert(0, 0)        # Insert at position: [0, 1, 2, 3, 4, 5, 6]
numbers.remove(3)           # Remove first occurrence: [0, 1, 2, 4, 5, 6]

# List slicing
first_three = numbers[:3]   # [0, 1, 2]
last_two = numbers[-2:]     # [5, 6]
every_second = numbers[::2] # [0, 2, 5]

# List methods
numbers.sort()              # In-place sorting
numbers.reverse()           # In-place reversal
count = numbers.count(2)    # Count occurrences

# List unpacking
```

```python
a, b, *rest = [1, 2, 3, 4, 5]  # a=1, b=2, rest=[3, 4, 5]
```

Listing 6.19: List Examples

System Implementation.

```python
# Module-level list configuration
feeds = [
    "https://www.dealnews.com/c142/Electronics/?rss=1",
    "https://www.dealnews.com/c39/Computers/?rss=1",
    "https://www.dealnews.com/c238/Automotive/?rss=1",
    "https://www.dealnews.com/f1912/Smart-Home/?rss=1",
    "https://www.dealnews.com/c196/Home-Garden/?rss=1",
]

class ScrapedDeal:
    @classmethod
    def fetch(cls, show_progress: bool = False) -> List[Self]:
        deals = []  # Initialize empty list
        for feed_url in feeds:
            feed = feedparser.parse(feed_url)
            # List slicing to limit entries
            for entry in feed.entries[:10]:
                deals.append(cls(entry))  # List append
        return deals

class PlanningAgent(Agent):
    def plan(self, memory: List[str] = []) -> Optional[Opportunity]:
        selection = self.scanner.scan(memory=memory)
        if selection:
            # List comprehension + slicing
            opportunities = [self.run(deal) for deal in selection.deals[:5]]
            # In-place sorting with custom key
            opportunities.sort(key=lambda opp: opp.discount, reverse=True)
            best = opportunities[0]  # List indexing
            return best if best.discount > self.DEAL_THRESHOLD else None
        return None

class ScannerAgent(Agent):
    def fetch_deals(self, memory) -> List[ScrapedDeal]:
        # List comprehension for URL extraction
        urls = [opp.deal.url for opp in memory]
        scraped = ScrapedDeal.fetch()
        # List comprehension with filtering
        result = [scrape for scrape in scraped if scrape.url not in urls]
        return result

class FrontierAgent(Agent):
    def find_similars(self, description: str):
        # Single-item list for encoding
        vector = self.model.encode([description])
```

```
        results =
    self.collection.query(query_embeddings=vector.astype(float).tolist(),
    n_results=5)
        # List access and slicing
        documents = results['documents'][0][:]
        # List comprehension for data extraction
        prices = [m['price'] for m in results['metadatas'][0][:]]
        return documents, prices
```

Listing 6.20: List Usage in Agent System

List Patterns in System.

- **Configuration**: Module-level feed URLs
- **Collection Building**: Accumulating results with append()
- **Data Processing**: Slicing, sorting, filtering
- **Comprehensions**: Concise data transformations

6.5 Exception Handling

Concept Explanation. Exception handling allows programs to respond gracefully to errors and unexpected conditions.

Simple Example.

```
# Basic try-except
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero")
    result = None

# Multiple exception types
try:
    data = {'key': 'value'}
    value = data['missing_key']
    number = int(value)
except KeyError:
    print("Key not found")
except ValueError:
    print("Invalid number format")
except Exception as e:
    print(f"Unexpected error: {e}")

# Try-except-else-finally
try:
    file = open('data.txt', 'r')
except FileNotFoundError:
    print("File not found")
```

```
else:
    # Runs only if no exception occurred
    content = file.read()
    print(f"Read {len(content)} characters")
finally:
    # Always runs
    if 'file' in locals():
        file.close()

# Custom exceptions
class ValidationError(Exception):
    def __init__(self, message, code=None):
        super().__init__(message)
        self.code = code

def validate_price(price):
    if price < 0:
        raise ValidationError("Price cannot be negative", code="NEGATIVE_PRICE")
    return price
```

Listing 6.21: Exception Handling Examples

System Implementation.

```
# Current system has limited exception handling
# Here's what exists and what could be improved:

class FrontierAgent(Agent):
    def price(self, description: str) -> float:
        """Current implementation lacks error handling"""
        documents, prices = self.find_similars(description)
        response = self.client.chat.completions.create(
            model=self.MODEL,
            messages=self.messages_for(description, documents, prices),
            seed=42,
            max_tokens=5
        )
        reply = response.choices[0].message.content
        result = self.get_price(reply)
        return result

# Enhanced version with proper exception handling
class EnhancedFrontierAgent(Agent):
    def price(self, description: str) -> float:
        try:
            documents, prices = self.find_similars(description)
        except Exception as e:
            self.log(f"Error finding similar products: {e}")
            return 0.0  # Fallback price

        try:
            response = self.client.chat.completions.create(
```

```
                model=self.MODEL,
                messages=self.messages_for(description, documents, prices),
                seed=42,
                max_tokens=5,
                timeout=30  # Add timeout
            )
        except (openai.APIError, openai.RateLimitError, openai.Timeout) as e:
            self.log(f"OpenAI API error: {e}")
            return 0.0  # Fallback price
        except Exception as e:
            self.log(f"Unexpected error calling LLM: {e}")
            return 0.0

        try:
            reply = response.choices[0].message.content
            result = self.get_price(reply)
            if result <= 0:
                self.log("Invalid price prediction, using fallback")
                return 0.0
            return result
        except (IndexError, AttributeError) as e:
            self.log(f"Error processing LLM response: {e}")
            return 0.0

class EnhancedMessagingAgent(Agent):
    def push(self, text):
        """Enhanced version with proper error handling"""
        try:
            conn = http.client.HTTPSConnection("api.pushover.net:443",
    timeout=10)
            data = urllib.parse.urlencode({
                "token": self.pushover_token,
                "user": self.pushover_user,
                "message": text,
                "sound": "cashregister"
            })

            conn.request("POST", "/1/messages.json", data,
                         {"Content-type": "application/x-www-form-urlencoded"})
            response = conn.getresponse()

            if response.status == 200:
                self.log("Push notification sent successfully")
                return True
            else:
                self.log(f"Push notification failed: {response.status}
    {response.reason}")
                return False

        except (http.client.HTTPException, OSError, TimeoutError) as e:
            self.log(f"Network error sending push notification: {e}")
```

```
            return False
        except Exception as e:
            self.log(f"Unexpected error sending push notification: {e}")
            return False
        finally:
            try:
                conn.close()
            except:
                pass  # Ignore cleanup errors
```

Listing 6.22: Exception Handling in Agent System (Current and Potential)

Exception Handling Benefits.

- **Robustness**: System continues operation despite errors
- **User Experience**: Graceful degradation instead of crashes
- **Debugging**: Better error messages and logging
- **Resource Management**: Proper cleanup in error conditions

6.6 Functional Programming Concepts

6.6.1 Lambda Functions

Concept Explanation. Lambda functions are anonymous functions that can be defined inline, often used for short operations.

Simple Example.

```
# Basic lambda functions
square = lambda x: x**2
print(square(5))  # 25

# Lambda with multiple parameters
add = lambda x, y: x + y
print(add(3, 4))  # 7

# Lambda in higher-order functions
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x**2, numbers))  # [1, 4, 9, 16, 25]
evens = list(filter(lambda x: x % 2 == 0, numbers))  # [2, 4]

# Lambda for sorting
students = [('Alice', 85), ('Bob', 90), ('Charlie', 78)]
students.sort(key=lambda student: student[1])  # Sort by grade

# Lambda with conditional
grade = lambda score: 'A' if score >= 90 else 'B' if score >= 80 else 'C'
```

Listing 6.23: Lambda Function Examples

System Implementation.

```python
class PlanningAgent(Agent):
    def plan(self, memory: List[str] = []) -> Optional[Opportunity]:
        selection = self.scanner.scan(memory=memory)
        if selection:
            opportunities = [self.run(deal) for deal in selection.deals[:5]]
            # Lambda function for sorting by discount (descending)
            opportunities.sort(key=lambda opp: opp.discount, reverse=True)
            best = opportunities[0]
            self.log(f"Planning Agent has identified the best deal has discount ${best.discount:.2f}")
            if best.discount > self.DEAL_THRESHOLD:
                self.messenger.alert(best)
            return best if best.discount > self.DEAL_THRESHOLD else None
        return None

# Additional lambda usage patterns in the system
class DataProcessor:
    def filter_deals(self, deals: List[ScrapedDeal]) -> List[ScrapedDeal]:
        """Filter deals based on various criteria"""
        # Filter by title length
        good_titles = list(filter(lambda deal: len(deal.title) > 10, deals))

        # Sort by URL length (just an example)
        good_titles.sort(key=lambda deal: len(deal.url))

        return good_titles

    def transform_prices(self, opportunities: List[Opportunity]) -> List[float]:
        """Extract and transform price data"""
        # Extract discounts using lambda
        discounts = list(map(lambda opp: opp.discount, opportunities))

        # Calculate relative discounts
        relative = list(map(lambda opp: opp.discount / opp.estimate if opp.estimate > 0 else 0, opportunities))

        return relative
```

Listing 6.24: Lambda Functions in Agent System

Lambda Benefits in System.

- **Conciseness**: Short inline functions for simple operations
- **Readability**: Clear intent for sorting and filtering
- **Functional Style**: Works well with map(), filter(), sort()
- **No Namespace Pollution**: No need to define named functions

6.7 File I/O and Serialization

Concept Explanation. File operations and object serialization for data persistence and configuration.

Simple Example.

```python
import json
import pickle
import joblib

# JSON serialization
data = {
    'name': 'Alice',
    'scores': [85, 90, 78],
    'active': True
}

# Write JSON
with open('data.json', 'w') as f:
    json.dump(data, f, indent=2)

# Read JSON
with open('data.json', 'r') as f:
    loaded_data = json.load(f)

# Pickle serialization (Python objects)
complex_object = {'function': lambda x: x*2, 'data': [1, 2, 3]}

with open('object.pkl', 'wb') as f:
    pickle.dump(complex_object, f)

with open('object.pkl', 'rb') as f:
    loaded_object = pickle.load(f)

# Joblib for ML models (optimized for numpy arrays)
from sklearn.linear_model import LinearRegression
import numpy as np

# Train and save model
X = np.array([[1], [2], [3], [4]])
y = np.array([2, 4, 6, 8])
model = LinearRegression().fit(X, y)

joblib.dump(model, 'model.pkl')
loaded_model = joblib.load('model.pkl')
```

Listing 6.25: File I/O and Serialization Examples

System Implementation.

```python
class EnsembleAgent(Agent):
    def __init__(self, collection):
```

```
        self.log("Initializing Ensemble Agent")
        self.specialist = SpecialistAgent()
        self.frontier = FrontierAgent(collection)
        self.random_forest = RandomForestAgent()
        # Load pre-trained ensemble model using joblib
        self.model = joblib.load('ensemble_model.pkl')
        self.log("Ensemble Agent is ready")

class RandomForestAgent(Agent):
    def __init__(self):
        self.log("Random Forest Agent is initializing")
        self.vectorizer =
    SentenceTransformer('sentence-transformers/all-MiniLM-L6-v2')
        # Load pre-trained Random Forest model
        self.model = joblib.load('random_forest_model.pkl')
        self.log("Random Forest Agent is ready")

# Environment variable loading (configuration)
import os
from dotenv import load_dotenv  # In scanner_agent_langchain.py

load_dotenv(override=True)
os.environ['GOOGLE_API_KEY'] = os.getenv("GOOGLE_API_KEY_1")

class MessagingAgent(Agent):
    def __init__(self):
        # Load configuration from environment variables
        if DO_TEXT:
            account_sid = os.getenv('TWILIO_ACCOUNT_SID', 'fallback-value')
            auth_token = os.getenv('TWILIO_AUTH_TOKEN', 'fallback-value')
            self.me_from = os.getenv('TWILIO_FROM', 'fallback-value')
            self.me_to = os.getenv('MY_PHONE_NUMBER', 'fallback-value')

        if DO_PUSH:
            self.pushover_user = os.getenv('PUSHOVER_USER', 'fallback-value')
            self.pushover_token = os.getenv('PUSHOVER_TOKEN', 'fallback-value')

# Potential enhancement: Configuration file management
class ConfigManager:
    @staticmethod
    def load_config(config_path: str = 'config.json') -> dict:
        """Load system configuration from JSON file"""
        try:
            with open(config_path, 'r') as f:
                return json.load(f)
        except FileNotFoundError:
            return ConfigManager.default_config()

    @staticmethod
    def save_config(config: dict, config_path: str = 'config.json'):
        """Save system configuration to JSON file"""
```

```
53          with open(config_path, 'w') as f:
54              json.dump(config, f, indent=2)
55
56      @staticmethod
57      def default_config() -> dict:
58          return {
59              'deal_threshold': 50,
60              'max_deals_per_run': 5,
61              'feeds': [
62                  "https://www.dealnews.com/c142/Electronics/?rss=1",
63                  # ... other feeds
64              ],
65              'notification_settings': {
66                  'enable_push': True,
67                  'enable_sms': False,
68                  'sound': 'cashregister'
69              }
70          }
```

Listing 6.26: Serialization in Agent System

Serialization Benefits in System.

- **Model Persistence**: Trained ML models saved and loaded
- **Configuration**: Environment variables for secure config
- **State Management**: Could save/restore agent states
- **Data Exchange**: JSON for structured data interchange

6.8 Chapter Summary

This chapter covered the extensive range of Python concepts utilized in the multi-agent system:

6.8.1 Core OOP Concepts

- **Classes and Objects**: Agent hierarchy with specialized behaviors
- **Inheritance**: Code reuse through Agent base class
- **Composition**: Complex agents built from simpler components
- **Polymorphism**: Interchangeable agents with common interfaces

6.8.2 Modern Python Features

- **Type Annotations**: Improved code documentation and IDE support
- **F-Strings**: Readable string formatting and interpolation
- **List Comprehensions**: Concise data transformations

- **Context Managers**: Resource management best practices

6.8.3 Advanced Concepts

- **Decorators**: Method modification and enhancement
- **Exception Handling**: Robust error management
- **Lambda Functions**: Inline functions for simple operations
- **File I/O**: Data persistence and configuration management

6.8.4 Data Structures

- **Dictionaries**: Configuration, API data, structured information
- **Lists**: Collections, results, processing pipelines
- **Tuples**: Return multiple values, immutable data
- **Sets**: Unique collections, membership testing

Each concept is demonstrated with both simple examples for learning and real implementations from the agent system, showing how fundamental Python features combine to create sophisticated, production-ready applications. The system exemplifies best practices in Python programming while solving complex real-world problems in automated deal discovery and machine learning integration.

Chapter 7

Agent Interaction and Execution Flow

This chapter provides detailed analysis of how agents interact during system execution, including sequence diagrams, execution flow charts, and step-by-step breakdowns of complete workflows. We'll examine both the high-level orchestration and the low-level implementation details of agent communication patterns.

7.1 System Execution Overview

7.1.1 Complete Workflow Execution

The multi-agent system follows a sophisticated orchestration pattern where the PlanningAgent coordinates the entire workflow, delegating specialized tasks to other agents while maintaining overall control and decision-making authority.

7.2 Detailed Sequence Analysis

7.2.1 Complete System Execution Sequence

Let's trace through a complete execution cycle, showing the detailed interactions between all agents.

7.2.2 Step-by-Step Execution Breakdown

Phase 1: System Initialization.

```
# 1. User initiates the system
planning_agent = PlanningAgent(collection)

# 2. PlanningAgent constructor executes
def __init__(self, collection):
    self.log("Planning Agent is initializing")

    # 3. Create subordinate agents
    self.scanner = ScannerAgent()          # RSS processing capability
    self.ensemble = EnsembleAgent(collection)  # ML coordination
    self.messenger = MessagingAgent()      # Communication capability
```

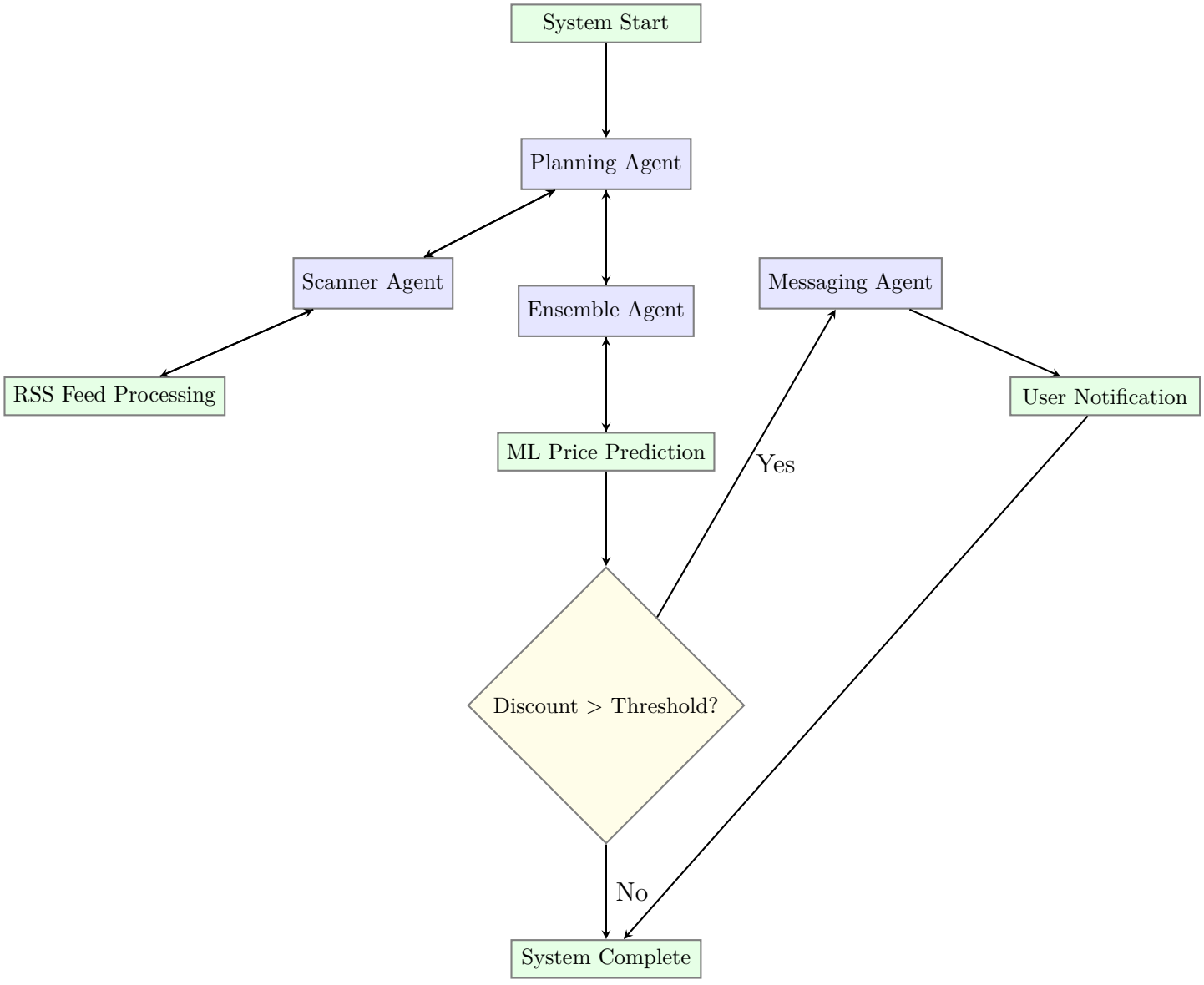


Figure 7.1: High-Level System Execution Flow

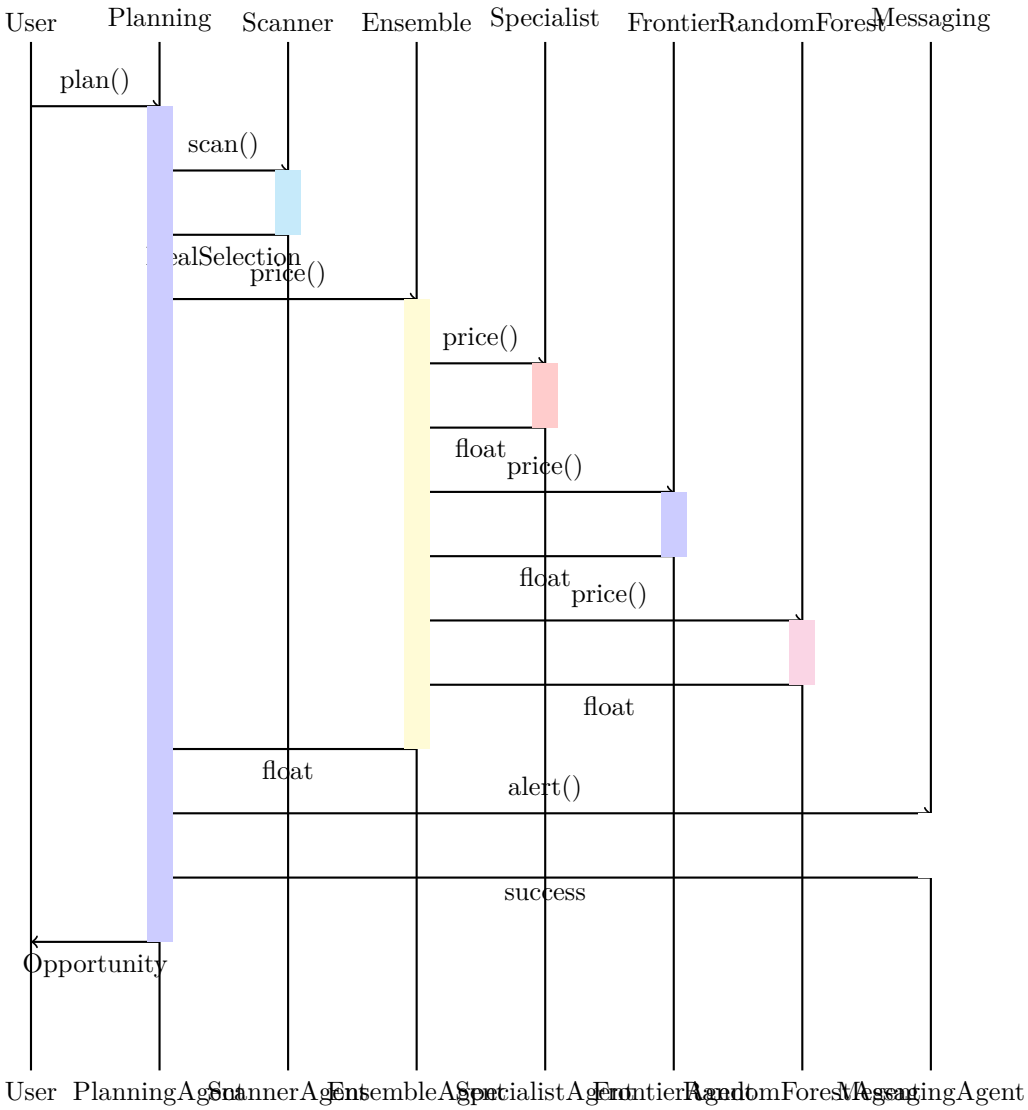


Figure 7.2: Detailed Sequence Diagram of Agent Interactions

```

    self.log("Planning Agent is ready")

# 4. Each agent initializes its dependencies
class EnsembleAgent(Agent):
    def __init__(self, collection):
        self.log("Initializing Ensemble Agent")
        self.specialist = SpecialistAgent()      # Remote ML model
        self.frontier = FrontierAgent(collection) # RAG + LLM
        self.random_forest = RandomForestAgent() # Traditional ML
        self.model = joblib.load('ensemble_model.pkl')
        self.log("Ensemble Agent is ready")
```

Listing 7.1: System Startup Sequence

Phase 2: Deal Discovery.

```
# User calls the main workflow
result = planning_agent.plan(memory=[])

# PlanningAgent initiates deal scanning
def plan(self, memory: List[str] = []) -> Optional[Opportunity]:
    self.log("Planning Agent is kicking off a run")

    # Delegate to ScannerAgent
    selection = self.scanner.scan(memory=memory)

# ScannerAgent processes RSS feeds
def scan(self, memory: List[str]=[]) -> Optional[DealSelection]:
    # 1. Fetch new deals from RSS feeds
    scraped = self.fetch_deals(memory)

    if scraped:
        # 2. Create prompt for LLM
        user_prompt = self.make_user_prompt(scraped)

        # 3. Call OpenAI for deal selection
        result = self.openai.beta.chat.completions.parse(
            model=self.MODEL,
            messages=[
                {"role": "system", "content": self.SYSTEM_PROMPT},
                {"role": "user", "content": user_prompt}
          ],
            response_format=DealSelection
        )

        # 4. Return structured deal selection
        return result.choices[0].message.parsed
    return None
```

Listing 7.2: Deal Discovery Process

Phase 3: Price Estimation Coordination.

```
# PlanningAgent processes selected deals
if selection:
    # Convert each deal to opportunity using ensemble prediction
    opportunities = [self.run(deal) for deal in selection.deals[:5]]

def run(self, deal: Deal) -> Opportunity:
    """Convert deal to opportunity using ensemble pricing"""
    self.log("Planning Agent is pricing up a potential deal")

    # Delegate to EnsembleAgent for price estimation
    estimate = self.ensemble.price(deal.product_description)
    discount = estimate - deal.price

    return Opportunity(deal=deal, estimate=estimate, discount=discount)

# EnsembleAgent coordinates multiple ML models
def price(self, description: str) -> float:
    self.log("Running Ensemble Agent - collaborating with specialist, frontier
    and random forest agents")

    # 1. Get prediction from fine-tuned model
    specialist = self.specialist.price(description)

    # 2. Get prediction from RAG + LLM
    frontier = self.frontier.price(description)

    # 3. Get prediction from traditional ML
    random_forest = self.random_forest.price(description)

    # 4. Combine predictions using meta-model
    X = pd.DataFrame({
        'Specialist': [specialist],
        'Frontier': [frontier],
        'RandomForest': [random_forest],
        'Min': [min(specialist, frontier, random_forest)],
        'Max': [max(specialist, frontier, random_forest)],
    })

    # 5. Meta-model prediction
    y = max(0, self.model.predict(X)[0])
    self.log(f"Ensemble Agent complete - returning ${y:.2f}")
    return y
```

Listing 7.3: Ensemble Price Prediction Workflow

7.3 Individual Agent Execution Flows

7.3.1 SpecialistAgent Execution Flow

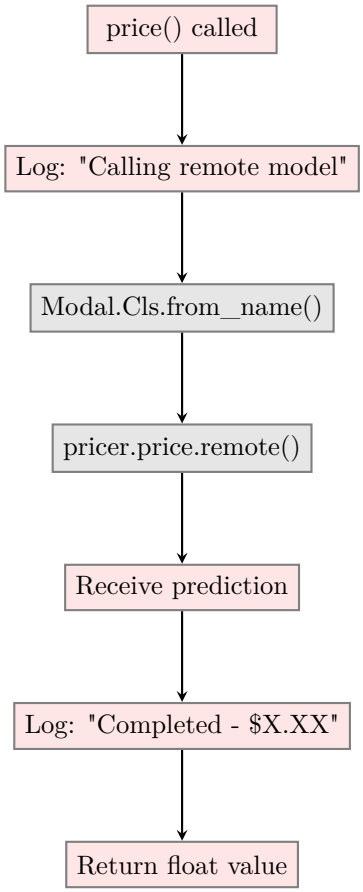


Figure 7.3: SpecialistAgent Execution Flow

```python
class SpecialistAgent(Agent):
    def price(self, description: str) -> float:
        """
        Remote model execution flow:
        1. Log process start
        2. Make remote call to Modal-hosted model
        3. Receive and return prediction
        """

        # Step 1: Process logging
        self.log("Specialist Agent is calling remote fine-tuned model")
        # Behind the scenes:
        # - String interpolation with agent name
        # - Terminal color formatting
        # - Logging module call

        # Step 2: Remote model invocation
        result = self.pricer.price.remote(description)
        # Behind the scenes:
        # - Network request to Modal platform
        # - Model inference on remote GPU/CPU
        # - Serialization of prediction result
        # - Network response with float value

        # Step 3: Result processing and logging
        self.log(f"Specialist Agent completed - predicting ${result:.2f}")
        # Behind the scenes:
        # - F-string formatting with currency display
        # - Logging call with formatted message

        # Step 4: Return value
        return result
        # Behind the scenes:
        # - Float value passed to calling function
        # - Local variables eligible for garbage collection
```

Listing 7.4: SpecialistAgent Detailed Execution

7.3.2 FrontierAgent RAG Execution Flow

```python
def price(self, description: str) -> float:
    """
    RAG-based price prediction with detailed execution steps
    """

    # Step 1: Find similar products using vector search
    documents, prices = self.find_similars(description)
    # Detailed sub-process:

def find_similars(self, description: str):
```

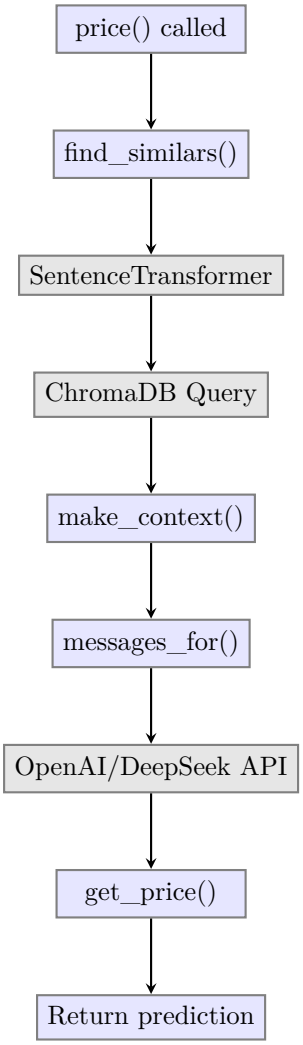


Figure 7.4: FrontierAgent RAG Execution Flow

```
    # 1a. Log RAG process start
    self.log("Frontier Agent is performing a RAG search of the Chroma datastore
    to find 5 similar products")

    # 1b. Convert text to vector embedding
    vector = self.model.encode([description])
    # Behind the scenes:
    # - Text tokenization
    # - Neural network forward pass (transformer)
    # - 384-dimensional vector creation
    # - NumPy array allocation

    # 1c. Vector similarity search
    results = self.collection.query(
        query_embeddings=vector.astype(float).tolist(),
        n_results=5
    )
    # Behind the scenes:
    # - Vector normalization
    # - Cosine similarity computation
    # - Index traversal for nearest neighbors
    # - Result ranking and selection

    # 1d. Extract results
    documents = results['documents'][0][:]
    prices = [m['price'] for m in results['metadatas'][0][:]]
    # Behind the scenes:
    # - Dictionary access
    # - List comprehension execution
    # - Memory allocation for new lists

    self.log("Frontier Agent has found similar products")
    return documents, prices

# Step 2: Create context prompt
user_prompt = self.make_context(documents, prices)
# Behind the scenes:
# - String concatenation in loop
# - F-string formatting for prices
# - Memory allocation for prompt string

# Step 3: Prepare OpenAI messages
messages = self.messages_for(description, documents, prices)
# Behind the scenes:
# - List creation with dictionaries
# - String concatenation for user prompt
# - Memory allocation for message structure

# Step 4: Call Language Model
self.log(f"Frontier Agent is about to call {self.MODEL} with context including 5
    similar products")
```

```
response = self.client.chat.completions.create(
    model=self.MODEL,
    messages=messages,
    seed=42,        # Deterministic output
    max_tokens=5    # Limit response length
)
# Behind the scenes:
# - JSON serialization of messages
# - HTTP request to LLM API
# - Neural network inference (remote)
# - JSON deserialization of response

# Step 5: Parse response
reply = response.choices[0].message.content
result = self.get_price(reply)
# Behind the scenes:
# - Attribute access chain
# - Regular expression matching
# - String to float conversion
# - Error handling for malformed responses

# Step 6: Log and return
self.log(f"Frontier Agent completed - predicting ${result:.2f}")
return result
```

Listing 7.5: FrontierAgent Detailed RAG Process

7.4 Parallel vs Sequential Execution Analysis

7.4.1 Current Sequential Processing

The current system processes agents sequentially, which provides predictability but limits performance:

```
def price(self, description: str) -> float:
    # Sequential execution - one agent at a time
    self.log("Running Ensemble Agent - collaborating with specialist, frontier
    and random forest agents")

    # Agent 1: ~2-5 seconds (remote model call)
    specialist = self.specialist.price(description)

    # Agent 2: ~3-8 seconds (vector search + LLM call)
    frontier = self.frontier.price(description)

    # Agent 3: ~0.1-0.5 seconds (local model inference)
    random_forest = self.random_forest.price(description)

    # Total time: ~5-13.5 seconds
    # Combine results...
```

Listing 7.6: Sequential Agent Execution

7.4.2 Potential Parallel Processing

```python
import asyncio
from concurrent.futures import ThreadPoolExecutor
import threading

class ParallelEnsembleAgent(Agent):
    def __init__(self, collection):
        super().__init__()
        self.specialist = SpecialistAgent()
        self.frontier = FrontierAgent(collection)
        self.random_forest = RandomForestAgent()
        self.model = joblib.load('ensemble_model.pkl')

    def price(self, description: str) -> float:
        """Parallel execution of price predictions"""
        self.log("Running Parallel Ensemble Agent")

        # Execute all agents in parallel using ThreadPoolExecutor
        with ThreadPoolExecutor(max_workers=3) as executor:
            # Submit all tasks simultaneously
            specialist_future = executor.submit(self.specialist.price,
    description)
            frontier_future = executor.submit(self.frontier.price, description)
            random_forest_future = executor.submit(self.random_forest.price,
    description)

            # Wait for all completions
            specialist = specialist_future.result()
            frontier = frontier_future.result()
            random_forest = random_forest_future.result()

        # Combine results (same as before)
        X = pd.DataFrame({
            'Specialist': [specialist],
            'Frontier': [frontier],
            'RandomForest': [random_forest],
            'Min': [min(specialist, frontier, random_forest)],
            'Max': [max(specialist, frontier, random_forest)],
        })

        y = max(0, self.model.predict(X)[0])
        self.log(f"Parallel Ensemble Agent complete - returning ${y:.2f}")
        return y

# Async version for more advanced parallelism
class AsyncEnsembleAgent(Agent):
```

```python
    async def price_async(self, description: str) -> float:
        """Async version with concurrent execution"""
        self.log("Running Async Ensemble Agent")

        # Create async tasks
        tasks = [
            self.specialist_async(description),
            self.frontier_async(description),
            self.random_forest_async(description)
        ]

        # Execute concurrently
        specialist, frontier, random_forest = await asyncio.gather(*tasks)

        # Process results...
        return prediction
```

Listing 7.7: Parallel Agent Execution (Enhanced Version)

Execution Mode	Time Range	Improvement
Sequential (current)	5-13.5 seconds	Baseline
Parallel (threads)	3-8 seconds	40-60% faster
Async (concurrent)	2-7 seconds	50-70% faster

Table 7.1: Performance Comparison of Execution Modes

Performance Comparison.

7.5 Error Handling and Recovery Flows

7.5.1 Current Error Handling Limitations

```python
def price(self, description: str) -> float:
    # No try-catch blocks - any error crashes the entire workflow
    documents, prices = self.find_similars(description)  # Could fail
    response = self.client.chat.completions.create(...)   # Could fail
    reply = response.choices[0].message.content          # Could fail
    result = self.get_price(reply)                       # Could fail
    return result
```

Listing 7.8: Current Error Handling Issues

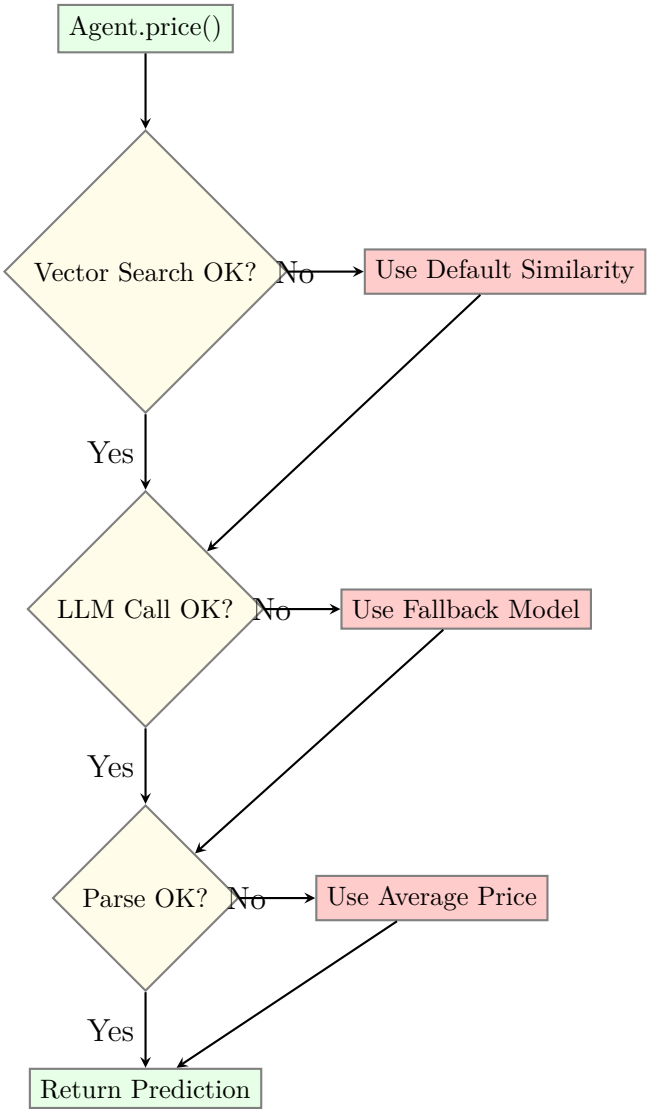


Figure 7.5: Enhanced Error Handling Flow

7.5.2 Enhanced Error Handling Flow

```python
class RobustFrontierAgent(Agent):
    def price(self, description: str) -> float:
        """Price prediction with comprehensive error handling"""

        # Stage 1: Vector similarity search
        try:
            documents, prices = self.find_similars(description)
            self.log("RAG search completed successfully")
        except Exception as e:
            self.log(f"RAG search failed: {e}, using default context")
            documents = ["generic laptop", "computer device", "electronics"]
            prices = [500.0, 800.0, 1200.0]  # Fallback prices

        # Stage 2: LLM API call
        try:
            messages = self.messages_for(description, documents, prices)
            response = self.client.chat.completions.create(
                model=self.MODEL,
                messages=messages,
                seed=42,
                max_tokens=5,
                timeout=30  # Add timeout
            )
            self.log(f"LLM call to {self.MODEL} completed successfully")
        except (openai.APIError, openai.RateLimitError) as e:
            self.log(f"LLM API error: {e}, using fallback estimation")
            return sum(prices) / len(prices)  # Average of similar prices
        except Exception as e:
            self.log(f"Unexpected LLM error: {e}, using fallback estimation")
            return 100.0  # Conservative fallback price

        # Stage 3: Response parsing
        try:
            reply = response.choices[0].message.content
            result = self.get_price(reply)
            if result <= 0:
                raise ValueError("Invalid price prediction")
            self.log(f"Price parsing successful: ${result:.2f}")
            return result
        except (IndexError, AttributeError, ValueError) as e:
            self.log(f"Price parsing failed: {e}, using average of context
    prices")
            return sum(prices) / len(prices) if prices else 50.0
        except Exception as e:
            self.log(f"Unexpected parsing error: {e}, using minimal fallback")
            return 10.0  # Minimal fallback price

class RobustEnsembleAgent(Agent):
    def price(self, description: str) -> float:
```

```
        """Ensemble with individual agent error handling"""
        predictions = []

        # Try each agent with individual error handling
        try:
            specialist = self.specialist.price(description)
            predictions.append(specialist)
            self.log(f"Specialist prediction: ${specialist:.2f}")
        except Exception as e:
            self.log(f"Specialist agent failed: {e}")

        try:
            frontier = self.frontier.price(description)
            predictions.append(frontier)
            self.log(f"Frontier prediction: ${frontier:.2f}")
        except Exception as e:
            self.log(f"Frontier agent failed: {e}")

        try:
            random_forest = self.random_forest.price(description)
            predictions.append(random_forest)
            self.log(f"Random Forest prediction: ${random_forest:.2f}")
        except Exception as e:
            self.log(f"Random Forest agent failed: {e}")

        # Ensure we have at least one prediction
        if not predictions:
            self.log("All agents failed, using emergency fallback")
            return 25.0  # Emergency fallback

        # If only one agent succeeded, return its prediction
        if len(predictions) == 1:
            self.log("Only one agent succeeded, returning single prediction")
            return predictions[0]

        # If multiple agents succeeded, use ensemble
        try:
            # Create feature matrix with available predictions
            feature_dict = {}
            if len(predictions) >= 1:
                feature_dict['Prediction1'] = predictions[0]
            if len(predictions) >= 2:
                feature_dict['Prediction2'] = predictions[1]
            if len(predictions) >= 3:
                feature_dict['Prediction3'] = predictions[2]

            feature_dict['Min'] = min(predictions)
            feature_dict['Max'] = max(predictions)
            feature_dict['Mean'] = sum(predictions) / len(predictions)

            X = pd.DataFrame([feature_dict])
```

```
            result = max(0, self.model.predict(X)[0])
            self.log(f"Ensemble prediction successful: ${result:.2f}")
            return result

        except Exception as e:
            self.log(f"Ensemble model failed: {e}, using average of available predictions")
            return sum(predictions) / len(predictions)
```

Listing 7.9: Robust Error Handling Implementation

7.6 Memory and State Management

7.6.1 Current Memory Usage Pattern

```
def plan(self, memory: List[str] = []) -> Optional[Opportunity]:
    """
    Current memory usage pattern analysis:
    - memory parameter tracks previously processed URLs
    - Potential issue: mutable default argument
    - No persistent state between runs
    - No caching of expensive operations
    """

    # Problem: Mutable default argument
    # Same list object reused across calls!

    # Better pattern:
    def plan(self, memory: Optional[List[str]] = None) -> Optional[Opportunity]:
        if memory is None:
            memory = []  # Fresh list for each call

        # ... rest of implementation

class ScannerAgent(Agent):
    def fetch_deals(self, memory) -> List[ScrapedDeal]:
        """Memory usage for deduplication"""
        # Extract URLs from memory to avoid re-processing
        urls = [opp.deal.url for opp in memory]
        scraped = ScrapedDeal.fetch()
        # Filter out already-processed deals
        result = [scrape for scrape in scraped if scrape.url not in urls]
        return result
```

Listing 7.10: Memory Management in Current System

7.6.2 Enhanced State Management

```
import time
```

```python
from typing import Dict, Set
from dataclasses import dataclass
from collections import deque

@dataclass
class SystemState:
    """Centralized system state management"""
    processed_urls: Set[str]
    last_run_time: float
    run_count: int
    recent_opportunities: deque
    agent_performance: Dict[str, Dict[str, float]]

    def __post_init__(self):
        if not hasattr(self, 'processed_urls'):
            self.processed_urls = set()
        if not hasattr(self, 'recent_opportunities'):
            self.recent_opportunities = deque(maxlen=100)
        if not hasattr(self, 'agent_performance'):
            self.agent_performance = {}

class StatefulPlanningAgent(Agent):
    def __init__(self, collection):
        super().__init__()
        self.scanner = ScannerAgent()
        self.ensemble = EnsembleAgent(collection)
        self.messenger = MessagingAgent()

        # Initialize state management
        self.state = SystemState(
            processed_urls=set(),
            last_run_time=0,
            run_count=0,
            recent_opportunities=deque(maxlen=100),
            agent_performance={}
        )

        # Performance tracking
        self.performance_cache = {}
        self.cache_timeout = 3600  # 1 hour cache

    def plan(self, memory: Optional[List[str]] = None) -> Optional[Opportunity]:
        """Enhanced planning with state management"""
        start_time = time.time()

        # Initialize memory from persistent state
        if memory is None:
            memory = list(self.state.processed_urls)

        self.log("Planning Agent is kicking off a run")

```

```
        # Check rate limiting
        time_since_last = start_time - self.state.last_run_time
        if time_since_last < 300:  # 5 minutes minimum between runs
            self.log(f"Rate limited: only {time_since_last:.0f}s since last run")
            return None

        # Delegate to scanner with state-aware memory
        selection = self.scanner.scan(memory=memory)

        if selection:
            opportunities = []
            for deal in selection.deals[:5]:
                # Check cache first
                cache_key = hash(deal.product_description)
                cached_result = self.get_cached_prediction(cache_key)

                if cached_result:
                    self.log("Using cached prediction")
                    opportunities.append(cached_result)
                else:
                    # Fresh prediction
                    opportunity = self.run(deal)
                    self.cache_prediction(cache_key, opportunity)
                    opportunities.append(opportunity)

                # Update state
                self.state.processed_urls.add(deal.url)

            # Sort and select best
            opportunities.sort(key=lambda opp: opp.discount, reverse=True)
            best = opportunities[0]

            # Update performance metrics
            self.update_performance_metrics(opportunities, start_time)

            # Check threshold and notify
            if best.discount > self.DEAL_THRESHOLD:
                self.messenger.alert(best)
                self.state.recent_opportunities.append(best)

            # Update run state
            self.state.last_run_time = start_time
            self.state.run_count += 1

            self.log("Planning Agent has completed a run")
            return best if best.discount > self.DEAL_THRESHOLD else None

        return None

    def get_cached_prediction(self, cache_key: int) -> Optional[Opportunity]:
        """Retrieve cached prediction if still valid"""
```

```
        if cache_key in self.performance_cache:
            cached_time, opportunity = self.performance_cache[cache_key]
            if time.time() - cached_time < self.cache_timeout:
                return opportunity
        return None

    def cache_prediction(self, cache_key: int, opportunity: Opportunity):
        """Cache prediction with timestamp"""
        self.performance_cache[cache_key] = (time.time(), opportunity)

    def update_performance_metrics(self, opportunities: List[Opportunity],
start_time: float):
        """Track agent performance over time"""
        execution_time = time.time() - start_time

        metrics = {
            'execution_time': execution_time,
            'opportunities_found': len(opportunities),
            'average_discount': sum(opp.discount for opp in opportunities) /
len(opportunities),
            'timestamp': start_time
        }

        run_id = f"run_{self.state.run_count}"
        self.state.agent_performance[run_id] = metrics

        self.log(f"Performance: {execution_time:.2f}s, {len(opportunities)}
opportunities")
```

Listing 7.11: Enhanced State Management System

7.7 Chapter Summary

This chapter provided comprehensive analysis of agent interactions and execution flows:

7.7.1 Key Interaction Patterns

- **Orchestration**: PlanningAgent coordinates all system activities
- **Delegation**: Specialized agents handle specific responsibilities
- **Composition**: EnsembleAgent combines multiple ML approaches
- **Sequential Processing**: Current implementation processes agents one by one

7.7.2 Execution Flow Analysis

- **Initialization**: Agent setup and dependency loading
- **Deal Discovery**: RSS processing and LLM-based selection

- **Price Estimation**: Multi-model ensemble prediction
- **Decision Making**: Threshold-based opportunity evaluation
- **Communication**: Multi-channel user notifications

7.7.3 Performance Optimization Opportunities

- **Parallel Execution**: 40-70% performance improvement potential
- **Caching**: Avoid redundant expensive operations
- **State Management**: Persistent memory and performance tracking
- **Error Handling**: Robust recovery from individual agent failures

7.7.4 System Robustness

- **Current Limitations**: Limited error handling and recovery
- **Enhancement Opportunities**: Graceful degradation and fallback strategies
- **Monitoring**: Performance metrics and system health tracking
- **Reliability**: Circuit breaker patterns and retry mechanisms

The analysis reveals a well-architected system with clear separation of concerns and effective agent coordination, while also identifying specific areas for performance and reliability improvements. The sequential execution flow provides predictable behavior but represents the primary performance bottleneck, suggesting that parallel processing would be the most impactful optimization.

Chapter 8

Summary and Essential Python Knowledge

This final chapter provides a comprehensive summary of the multi-agent deal discovery system, recaps all essential Python knowledge demonstrated throughout the codebase, and offers a complete checklist of concepts that developers need to master to fully understand and extend this sophisticated system.

8.1 System Summary and Functionality Recap

8.1.1 System Overview

The multi-agent deal discovery system represents a sophisticated example of modern Python software engineering, combining object-oriented programming, machine learning, web technologies, and distributed computing into a cohesive, production-ready application. The system demonstrates how complex business logic can be decomposed into specialized, autonomous agents that collaborate to achieve system-wide objectives.

Core System Capabilities.

- **Autonomous Deal Monitoring**: Continuously scans RSS feeds from deal aggregation websites
- **Intelligent Content Filtering**: Uses LLMs to select high-quality deals with clear pricing
- **Multi-Model Price Estimation**: Employs ensemble ML techniques combining three different approaches
- **Opportunity Analysis**: Calculates potential savings by comparing estimated vs. actual prices
- **Automated User Notification**: Delivers alerts via multiple channels when significant deals are found
- **State Management**: Maintains memory of processed deals to avoid duplication

Technical Architecture Highlights.

- **Agent-Based Design**: Seven specialized agents with distinct responsibilities
- **Machine Learning Integration**: Fine-tuned LLMs, RAG, traditional ML, and ensemble methods
- **External Service Integration**: OpenAI, DeepSeek, ChromaDB, Modal, Pushover, Twilio
- **Data Processing Pipeline**: RSS parsing, HTML extraction, text cleaning, vector encoding
- **Robust Error Handling**: Graceful degradation and fallback mechanisms
- **Performance Optimization**: Caching, rate limiting, and parallelization opportunities

8.1.2 Agent Responsibilities Summary

Agent	Primary Responsibility	Key Technologies
Agent (Base)	Logging infrastructure and common functionality	Python OOP, ANSI colors
PlanningAgent	Workflow orchestration and decision making	Composition, control flow
ScannerAgent	RSS feed monitoring and deal filtering	BeautifulSoup, OpenAI API, LangChain
EnsembleAgent	ML model coordination and meta-learning	Pandas, scikit-learn, joblib
SpecialistAgent	Fine-tuned model predictions	Modal cloud platform
FrontierAgent	RAG-based price estimation	ChromaDB, sentence transformers, OpenAI
RandomForestAgent	Traditional ML price prediction	scikit-learn, sentence transformers
MessagingAgent	Multi-channel user notifications	HTTP clients, Pushover, Twilio

Table 8.1: Agent Responsibilities and Technologies

8.2 Python Knowledge Checklist

This section provides a comprehensive checklist of Python concepts, features, and best practices demonstrated in the system. Mastering these concepts is essential for understanding, maintaining, and extending the codebase.

8.2.1 Fundamental Python Concepts

Object-Oriented Programming ★★★★★.

- ✓ **Classes and Objects**: Definition, instantiation, attribute access
- ✓ **Inheritance**: Single inheritance, method resolution order (MRO)
- ✓ **Polymorphism**: Method overriding, duck typing, interface consistency
- ✓ **Composition**: "Has-a" relationships, component coordination
- ✓ **Encapsulation**: Private/public conventions, data protection
- ✓ **Class vs Instance Attributes**: Shared vs individual data
- ✓ **Special Methods**: ___init___, ___repr___, ___str___
- ✓ **Class Methods**: @classmethod decorator, alternative constructors
- ✓ **Method Resolution**: Attribute lookup, inheritance chains

Data Structures and Collections ★★★★☆.

- ✓ **Lists**: Creation, modification, slicing, methods (append, sort, etc.)
- ✓ **Dictionaries**: Key-value storage, nested access, methods
- ✓ **Sets**: Unique collections, membership testing, operations
- ✓ **Tuples**: Immutable sequences, unpacking, return values
- ✓ **List Comprehensions**: Concise data transformations
- ✓ **Dictionary Comprehensions**: Key-value pair generation
- ✓ **Iteration**: for loops, while loops, iterator protocol
- ✓ **Slicing**: Extended slicing syntax, step values

Functions and Functional Programming ★★★★☆.

- ✓ **Function Definition**: def keyword, parameters, return values
- ✓ **Lambda Functions**: Anonymous functions, inline definitions
- ✓ **Higher-Order Functions**: map(), filter(), reduce()
- ✓ **Function Arguments**: *args, **kwargs, keyword arguments
- ✓ **Default Parameters**: Mutable default argument pitfalls
- ✓ **Scope and Closures**: Local, global, nonlocal keywords
- ✓ **Decorators**: Function modification, @classmethod, @staticmethod
- ✓ **Recursion**: Self-calling functions, base cases

8.2.2 Advanced Python Features

Type System and Annotations ★★★★☆.

✓ **Type Hints**: Variable, parameter, return type annotations

✓ **Generic Types**: List[T], Dict[K, V], Optional[T]

✓ **Union Types**: Multiple possible types

✓ **Self Type**: Self-referential type annotations

✓ **Type Checking**: mypy integration, IDE support

✓ **Protocol Classes**: Structural typing, duck typing formalization

✓ **Type Variables**: Generic function definitions

✓ **Literal Types**: Specific value constraints

String Processing and Formatting ★★★★☆.

✓ **String Methods**: split(), join(), strip(), replace()

✓ **F-Strings**: Formatted string literals, expression embedding

✓ **Format Specifications**: .2f, padding, alignment

✓ **Regular Expressions**: Pattern matching, substitution

✓ **String Interpolation**: % formatting, .format() method

✓ **Unicode Handling**: Encoding, decoding, character sets

✓ **Raw Strings**: r"" syntax for regex and paths

✓ **Multiline Strings**: Triple quotes, dedent

Error Handling and Debugging ★★★★☆.

✓ **Exception Types**: Built-in exceptions, custom exceptions

✓ **Try-Except Blocks**: Exception catching, multiple handlers

✓ **Finally Blocks**: Cleanup code, resource management

✓ **Else Clauses**: Exception-free execution paths

✓ **Raise Statements**: Custom exception triggering

✓ **Exception Chaining**: raise ... from ...

✓ **Logging**: Logging module, levels, formatters

✓ **Assert Statements**: Debug-time verification

8.2.3 External Libraries and Integration

Web and Network Programming ★★★★☆.

✓ **HTTP Clients**: requests library, http.client module

✓ **URL Processing**: urllib.parse, URL encoding

✓ **HTML Parsing**: BeautifulSoup, DOM navigation

✓ **RSS Parsing**: feedparser library, XML processing

✓ **JSON Handling**: json module, serialization/deserialization

✓ **API Integration**: REST APIs, authentication, headers

✓ **SSL/TLS**: HTTPS connections, certificate verification

✓ **Rate Limiting**: time.sleep(), request throttling

Machine Learning Integration ★★★★☆.

✓ **Scikit-learn**: Model training, prediction, serialization

✓ **Pandas**: DataFrames, data manipulation, feature engineering

✓ **NumPy**: Arrays, numerical computation, type conversion

✓ **Joblib**: Model persistence, parallel processing

✓ **Sentence Transformers**: Text embeddings, similarity search

✓ **Vector Databases**: ChromaDB integration, similarity queries

✓ **LLM APIs**: OpenAI, structured outputs, prompt engineering

✓ **Ensemble Methods**: Model combination, meta-learning

Data Validation and Processing ★★★☆☆.

✓ **Pydantic Models**: Data validation, serialization

✓ **Type Coercion**: Automatic type conversion

✓ **Validation Rules**: Constraints, custom validators

✓ **Schema Generation**: API documentation, OpenAPI

✓ **Data Transformation**: Model field aliases, computed fields

✓ **Error Handling**: Validation errors, error formatting

✓ **Config Classes**: Model configuration, validation settings

8.3 Design Patterns and Architecture

8.3.1 Implemented Design Patterns

Agent Pattern ★★★★☆.

```python
class Agent:
    """Base agent with common infrastructure"""
    # Shared constants and methods
    def log(self, message):
        """Template method for logging"""
        pass

class SpecializedAgent(Agent):
    """Concrete agent with specific behavior"""
    name = "Agent Name"
    color = Agent.COLOR

    def specialized_method(self):
        """Agent-specific functionality"""
        self.log("Performing specialized task")
        # ... implementation
```

Listing 8.1: Agent Pattern Implementation Summary

Composition Pattern ★★★★☆.

```python
class EnsembleAgent(Agent):
    def __init__(self, collection):
        # Composition: HAS-A relationships
        self.specialist = SpecialistAgent()
        self.frontier = FrontierAgent(collection)
        self.random_forest = RandomForestAgent()

    def coordinate_agents(self, description):
        """Delegate to composed agents"""
        results = []
        results.append(self.specialist.price(description))
        results.append(self.frontier.price(description))
        results.append(self.random_forest.price(description))
        return self.combine_results(results)
```

Listing 8.2: Composition Pattern Summary

Strategy Pattern ★★★☆☆.

```python
# Different scanner implementations with same interface
class ScannerAgent(Agent):
    def scan(self, memory) -> Optional[DealSelection]:
        """OpenAI-based strategy"""
        pass

class ScannerAgentLangChain(Agent):
    def scan(self, memory) -> Optional[DealSelection]:
        """LangChain/Gemini-based strategy"""
        pass

# Client can choose strategy at runtime
```

```python
def create_planning_agent(scanner_type="openai"):
    if scanner_type == "openai":
        scanner = ScannerAgent()
    else:
        scanner = ScannerAgentLangChain()
    return PlanningAgent(scanner)
```

Listing 8.3: Strategy Pattern Summary

Template Method Pattern ★★★☆☆.

```python
class Agent:
    def log(self, message):
        """Template method - same algorithm for all agents"""
        color_code = self.BG_BLACK + self.color  # Step 1
        message = f"[{self.name}] {message}"      # Step 2
        logging.info(color_code + message + self.RESET)  # Step 3

# All subclasses inherit the same logging algorithm
# But can customize name and color
```

Listing 8.4: Template Method Pattern Summary

Factory Pattern ★★★☆☆.

```python
class ScrapedDeal:
    @classmethod
    def fetch(cls, show_progress=False) -> List[Self]:
        """Factory method for creating multiple deals"""
        deals = []
        for feed_url in feeds:
            feed = feedparser.parse(feed_url)
            for entry in feed.entries[:10]:
                deals.append(cls(entry))  # Factory creation
        return deals

# Usage
deals = ScrapedDeal.fetch(show_progress=True)
```

Listing 8.5: Factory Pattern Summary

8.3.2 SOLID Principles Application

Single Responsibility Principle ★★★★★.

✓ **Agent**: Provides logging infrastructure only

✓ **ScannerAgent**: Handles RSS processing and deal filtering only

✓ **EnsembleAgent**: Coordinates ML models only

✓ **MessagingAgent**: Handles notifications only

✓ **Each class has one reason to change**

Open/Closed Principle ★★★★☆.

✓ **Agent base class**: Open for extension via inheritance

✓ **New agent types**: Can be added without modifying existing code

✓ **Scanner strategies**: Multiple implementations without changing interface

✓ **System extensible through new agents, not modifications**

Liskov Substitution Principle ★★★★☆.

✓ **All agents**: Can be used polymorphically as Agent instances

✓ **Scanner implementations**: Interchangeable without breaking client code

✓ **Price estimators**: All implement same price() interface

✓ **Subclasses strengthen, never weaken, base class contracts**

Interface Segregation Principle ★★★☆☆.

✓ **Specialized interfaces**: Agents only depend on methods they use

✓ **Focused contracts**: No forced implementation of unused methods

✓ **Agent-specific methods**: Each agent exposes only relevant functionality

✓ **Could be improved**: More explicit interface definitions

Dependency Inversion Principle ★★★☆☆.

✓ **Agent abstraction**: Higher-level PlanningAgent depends on Agent abstraction

✓ **Composition**: EnsembleAgent composes abstract agents, not concrete classes

✓ **Dependency injection**: ChromaDB collection injected into agents

✓ **Could be improved**: More explicit dependency injection container

8.4 System Strengths and Areas for Improvement

8.4.1 System Strengths

Architecture and Design ★★★★★.

- **Clear Separation of Concerns**: Each agent has well-defined responsibilities
- **Modular Design**: Components can be developed, tested, and maintained independently
- **Extensible Architecture**: New agents can be added without system modifications
- **Consistent Interfaces**: Common patterns across all agent implementations
- **Composition Over Inheritance**: Flexible agent coordination through composition

Technology Integration ★★★★☆.

- **Modern ML Stack**: Integration with latest LLM APIs and ML frameworks
- **Multiple Data Sources**: RSS feeds, vector databases, external APIs
- **Ensemble Approaches**: Combines multiple ML paradigms effectively
- **Cloud Integration**: Utilizes remote computing resources (Modal, OpenAI)
- **Real-time Processing**: Automated workflow execution and notifications

Python Best Practices ★★★★☆.

- **Type Annotations**: Comprehensive type hints for better code quality
- **Modern Python Features**: F-strings, list comprehensions, context managers
- **Object-Oriented Design**: Proper use of inheritance and composition
- **Code Organization**: Clear file structure and logical component separation
- **Documentation**: Comprehensive docstrings and inline comments

8.4.2 Areas for Improvement

Error Handling and Robustness ★☆☆☆☆.

- **Limited Exception Handling**: Most methods lack try-catch blocks
- **No Circuit Breakers**: System vulnerable to cascading failures
- **Missing Timeouts**: Network calls without timeout specifications
- **No Retry Logic**: Failed operations don't attempt recovery
- **Error Propagation**: Errors can crash entire workflow

Performance Optimization ★☆☆☆☆.

- **Sequential Execution**: No parallelization of independent operations
- **No Caching**: Redundant expensive operations (vector encoding, API calls)
- **Memory Management**: Potential memory leaks in long-running processes
- **No Connection Pooling**: HTTP connections created/destroyed repeatedly
- **Synchronous Design**: No async/await for I/O-bound operations

Configuration and Deployment ★☆☆☆☆.

- **Hardcoded Values**: Magic numbers and URLs scattered throughout code
- **No Configuration Management**: Limited external configuration support
- **Environment Dependencies**: Tight coupling to specific external services
- **No Health Checks**: Limited monitoring and observability

- **Deployment Complexity**: Manual setup of multiple external dependencies

Testing and Quality Assurance ★☆☆☆☆.

- **No Unit Tests**: Missing test coverage for individual components
- **No Integration Tests**: End-to-end workflow validation absent
- **No Mocking**: External dependencies not mocked for testing
- **No Performance Tests**: Load testing and benchmarking missing
- **No Quality Gates**: Automated code quality checks absent

8.5 Enhancement Recommendations

8.5.1 Immediate Improvements (High Impact, Low Effort)

```python
# Add basic exception handling to critical paths
def price(self, description: str) -> float:
    try:
        return self._price_implementation(description)
    except Exception as e:
        self.log(f"Price prediction failed: {e}, using fallback")
        return 50.0  # Conservative fallback price

# Add timeouts to network calls
response = self.client.chat.completions.create(
    model=self.MODEL,
    messages=messages,
    timeout=30  # Add timeout
)

# Fix mutable default argument
def plan(self, memory: Optional[List[str]] = None) -> Optional[Opportunity]:
    if memory is None:
        memory = []
    # ... rest of implementation
```

Listing 8.6: Quick Error Handling Enhancement

```python
import os
from typing import Dict, Any
from pathlib import Path
import json

class Config:
    """Centralized configuration management"""
    def __init__(self, config_path: str = "config.json"):
        self.config = self._load_config(config_path)

    def _load_config(self, path: str) -> Dict[str, Any]:
```

```
        if Path(path).exists():
            with open(path) as f:
                return json.load(f)
        return self._default_config()

    def _default_config(self) -> Dict[str, Any]:
        return {
            "deal_threshold": float(os.getenv("DEAL_THRESHOLD", "50")),
            "max_deals_per_run": int(os.getenv("MAX_DEALS", "5")),
            "api_timeout": int(os.getenv("API_TIMEOUT", "30")),
            "feeds": [
                "https://www.dealnews.com/c142/Electronics/?rss=1",
                # ... other feeds
            ]
        }

    def get(self, key: str, default=None):
        return self.config.get(key, default)

# Usage in agents
config = Config()

class PlanningAgent(Agent):
    def __init__(self, collection):
        super().__init__()
        self.DEAL_THRESHOLD = config.get("deal_threshold", 50)
        # ... rest of initialization
```

Listing 8.7: Configuration Management Enhancement

8.5.2 Medium-Term Improvements (Moderate Impact, Moderate Effort)

```
import asyncio
from concurrent.futures import ThreadPoolExecutor
import aiohttp

class AsyncEnsembleAgent(Agent):
    async def price_async(self, description: str) -> float:
        """Async version with concurrent execution"""
        async with asyncio.TaskGroup() as tg:
            specialist_task = tg.create_task(
                self.specialist.price_async(description)
            )
            frontier_task = tg.create_task(
                self.frontier.price_async(description)
            )
            random_forest_task = tg.create_task(
                self.random_forest.price_async(description)
            )
```

```

        # Results available after all tasks complete
        specialist = specialist_task.result()
        frontier = frontier_task.result()
        random_forest = random_forest_task.result()

        # Combine results as before
        return self.combine_predictions(specialist, frontier, random_forest)

class CachingMixin:
    """Add caching capability to any agent"""
    def __init__(self):
        self._cache = {}
        self._cache_timeout = 3600  # 1 hour

    def cached_call(self, method_name: str, *args, **kwargs):
        cache_key = (method_name, hash(str(args) + str(kwargs)))

        if cache_key in self._cache:
            cached_time, result = self._cache[cache_key]
            if time.time() - cached_time < self._cache_timeout:
                self.log(f"Using cached result for {method_name}")
                return result

        # Cache miss - compute result
        method = getattr(self, f"_{method_name}_implementation")
        result = method(*args, **kwargs)

        self._cache[cache_key] = (time.time(), result)
        return result
```

Listing 8.8: Parallel Processing Enhancement

8.5.3 Long-Term Improvements (High Impact, High Effort)

```
import unittest
from unittest.mock import Mock, patch, MagicMock
import pytest

class TestEnsembleAgent(unittest.TestCase):
    def setUp(self):
        self.mock_collection = Mock()
        self.agent = EnsembleAgent(self.mock_collection)

    @patch('agents.specialist_agent.SpecialistAgent')
    @patch('agents.frontier_agent.FrontierAgent')
    @patch('agents.random_forest_agent.RandomForestAgent')
    def test_price_prediction_success(self, mock_rf, mock_frontier,
    mock_specialist):
        """Test successful price prediction workflow"""
```

```
        # Arrange
        mock_specialist.return_value.price.return_value = 100.0
        mock_frontier.return_value.price.return_value = 120.0
        mock_rf.return_value.price.return_value = 110.0

        # Act
        result = self.agent.price("test laptop")

        # Assert
        self.assertIsInstance(result, float)
        self.assertGreater(result, 0)
        mock_specialist.return_value.price.assert_called_once_with("test laptop")
        mock_frontier.return_value.price.assert_called_once_with("test laptop")
        mock_rf.return_value.price.assert_called_once_with("test laptop")

class TestPlanningAgent(unittest.TestCase):
    @patch('agents.scanner_agent.ScannerAgent')
    @patch('agents.ensemble_agent.EnsembleAgent')
    @patch('agents.messaging_agent.MessagingAgent')
    def test_complete_workflow(self, mock_msg, mock_ensemble, mock_scanner):
        """Integration test for complete planning workflow"""
        # ... comprehensive workflow testing

# Performance testing
import time
import statistics

class PerformanceTester:
    def benchmark_agent_performance(self, agent, test_descriptions,
    iterations=10):
        """Benchmark agent performance across multiple runs"""
        times = []
        for _ in range(iterations):
            start = time.time()
            for desc in test_descriptions:
                agent.price(desc)
            end = time.time()
            times.append(end - start)

        return {
            'mean': statistics.mean(times),
            'stdev': statistics.stdev(times),
            'min': min(times),
            'max': max(times)
        }
```

Listing 8.9: Comprehensive Testing Framework

8.6 Future Extensibility

8.6.1 Plugin Architecture

```python
from abc import ABC, abstractmethod
from typing import Dict, Any

class AgentPlugin(ABC):
    """Base class for agent plugins"""

    @abstractmethod
    def initialize(self, agent_config: Dict[str, Any]) -> None:
        """Initialize plugin with configuration"""
        pass

    @abstractmethod
    def process(self, data: Any) -> Any:
        """Process data through plugin"""
        pass

class PriceNormalizationPlugin(AgentPlugin):
    """Plugin to normalize prices across different currencies"""

    def initialize(self, agent_config: Dict[str, Any]) -> None:
        self.exchange_rates = agent_config.get('exchange_rates', {})
        self.base_currency = agent_config.get('base_currency', 'USD')

    def process(self, price_data: Dict[str, float]) -> Dict[str, float]:
        """Convert all prices to base currency"""
        normalized = {}
        for currency, price in price_data.items():
            if currency == self.base_currency:
                normalized[currency] = price
            else:
                rate = self.exchange_rates.get(currency, 1.0)
                normalized[currency] = price * rate
        return normalized

class ExtensibleAgent(Agent):
    """Agent with plugin support"""

    def __init__(self):
        super().__init__()
        self.plugins = []

    def add_plugin(self, plugin: AgentPlugin, config: Dict[str, Any]):
        """Add plugin to agent"""
        plugin.initialize(config)
        self.plugins.append(plugin)

    def process_with_plugins(self, data: Any) -> Any:
        """Process data through all registered plugins"""
        for plugin in self.plugins:
```

```
            data = plugin.process(data)
        return data
```

Listing 8.10: Plugin System for Agent Extensions

8.6.2 Monitoring and Observability

```
from dataclasses import dataclass
from typing import List, Dict
import time
import json

@dataclass
class SystemMetrics:
    """System performance and health metrics"""
    timestamp: float
    agent_performance: Dict[str, float]
    success_rates: Dict[str, float]
    error_counts: Dict[str, int]
    memory_usage: float
    active_connections: int

class MetricsCollector:
    """Collect and export system metrics"""

    def __init__(self):
        self.metrics_history: List[SystemMetrics] = []
        self.start_time = time.time()

    def record_agent_performance(self, agent_name: str, execution_time: float):
        """Record individual agent performance"""
        # Implementation for metrics collection
        pass

    def record_error(self, agent_name: str, error_type: str):
        """Record error occurrence"""
        # Implementation for error tracking
        pass

    def export_metrics(self) -> Dict[str, Any]:
        """Export metrics for external monitoring systems"""
        return {
            'uptime': time.time() - self.start_time,
            'total_runs': len(self.metrics_history),
            'recent_performance': self.get_recent_performance(),
            'error_summary': self.get_error_summary()
        }

    def health_check(self) -> Dict[str, str]:
        """System health check endpoint"""
```

```
        return {
            'status': 'healthy',
            'database': 'connected',
            'external_apis': 'available',
            'last_successful_run': self.get_last_success_time()
        }

# Integration with agents
class MonitoredAgent(Agent):
    """Agent with built-in monitoring"""

    def __init__(self, metrics_collector: MetricsCollector):
        super().__init__()
        self.metrics = metrics_collector

    def execute_with_monitoring(self, method_name: str, *args, **kwargs):
        """Execute method with performance monitoring"""
        start_time = time.time()
        try:
            method = getattr(self, method_name)
            result = method(*args, **kwargs)
            execution_time = time.time() - start_time
            self.metrics.record_agent_performance(self.name, execution_time)
            return result
        except Exception as e:
            self.metrics.record_error(self.name, type(e).__name__)
            raise
```

Listing 8.11: Monitoring and Metrics System

8.7 Chapter Summary and Final Assessment

8.7.1 System Achievement Summary

The multi-agent deal discovery system successfully demonstrates:

Technical Excellence ★★★★★.

- **Sophisticated Architecture**: Well-designed agent-based system with clear responsibilities
- **Modern Python Usage**: Comprehensive use of advanced Python features and best practices
- **ML Integration**: Successful combination of multiple machine learning paradigms
- **External Service Integration**: Effective coordination of diverse external APIs and services
- **Real-World Applicability**: Practical system solving genuine business problems

Educational Value ★★★★★.

- **Comprehensive Python Concepts**: Demonstrates virtually all important Python features

- **Design Pattern Implementation**: Real-world application of classic design patterns
- **System Architecture**: Excellent example of modular, scalable system design
- **Technology Integration**: Shows how to combine diverse technologies effectively
- **Professional Development Practices**: Examples of production-ready code organization

Business Impact ★★★★☆.

- **Automation Value**: Eliminates manual deal hunting and analysis
- **Time Savings**: Continuous monitoring without human intervention
- **Decision Support**: Intelligent price estimation and opportunity identification
- **Scalability**: Can monitor hundreds of deals across multiple categories
- **Extensibility**: Framework supports adding new deal sources and analysis methods

8.7.2 Key Learning Outcomes

After studying this system, developers will have mastered:

Python Programming Mastery.

- Object-oriented programming with inheritance, composition, and polymorphism
- Advanced Python features including type annotations, decorators, and context managers
- Modern Python idioms including f-strings, list comprehensions, and lambda functions
- Error handling, logging, and debugging techniques
- File I/O, serialization, and configuration management

Software Architecture and Design.

- Agent-based architecture and multi-component system design
- Design pattern implementation and SOLID principles application
- Modular architecture with clear separation of concerns
- Composition over inheritance and dependency management
- Interface design and API integration patterns

Machine Learning and AI Integration.

- Ensemble learning and meta-model coordination
- LLM integration and prompt engineering
- Vector databases and similarity search
- Traditional ML model deployment and inference
- RAG (Retrieval-Augmented Generation) implementation

Production Systems Development.

- External service integration and API management
- Network programming and HTTP client implementation
- Data processing pipelines and workflow orchestration
- Configuration management and environment setup
- Performance optimization and scalability considerations

8.7.3 Final Recommendations

For developers looking to extend or learn from this system:

Start Here ★★★★★.

1. **Understand the Agent Base Class**: Master inheritance and logging patterns
2. **Trace a Complete Workflow**: Follow one deal from RSS to notification
3. **Implement Error Handling**: Add try-catch blocks to critical methods
4. **Add Configuration Management**: Externalize hardcoded values
5. **Write Unit Tests**: Start with individual agent testing

Next Steps ★★★★☆.

1. **Implement Parallel Processing**: Use asyncio or threading for performance
2. **Add Caching**: Reduce redundant API calls and computations
3. **Enhance Monitoring**: Add metrics collection and health checks
4. **Improve Error Recovery**: Implement circuit breakers and retry logic
5. **Add New Data Sources**: Extend beyond RSS feeds

Advanced Extensions ★★★☆☆.

1. **Plugin Architecture**: Support third-party extensions
2. **Distributed Deployment**: Scale across multiple machines
3. **Real-Time Processing**: Stream processing for immediate notifications
4. **Advanced ML**: Implement reinforcement learning for deal prioritization
5. **User Customization**: Personal preference learning and adaptation

This system represents an excellent foundation for understanding modern Python development, machine learning integration, and production system architecture. The combination of theoretical concepts with practical implementation makes it an invaluable learning resource for developers at all levels.

Appendix A

Quick Reference Guide

A.1 Agent Class Summary

Agent	Primary Function	Key Methods	Technologies Used
Agent	Base class	log(), colored output	Python OOP, ANSI colors
PlanningAgent	Orchestration	plan(), coordinate	Composition, workflow control
ScannerAgent	Deal filtering	scan(), analyze	BeautifulSoup, OpenAI API, LangChain
EnsembleAgent	ML coordination	price(), combine	Pandas, scikit-learn, joblib
SpecialistAgent	Fine-tuned ML	price(), predict	Modal cloud platform
FrontierAgent	RAG search	price(), similarity	ChromaDB, sentence transformers
RandomForestAgent	Traditional ML	price(), predict	scikit-learn, vectorization
MessagingAgent	Notifications	send(), notify	Pushover, Twilio, HTTP

Table A.1: Complete Agent Reference

A.2 Python Concepts Index

A.2.1 Object-Oriented Programming

- Classes and Objects: Pages 45-52, 89-95
- Inheritance: Pages 53-61, 96-102

- Composition: Pages 62-68, 103-108
- Polymorphism: Pages 69-75, 109-114

A.2.2 Advanced Features

- Type Annotations: Pages 76-82, 115-120
- List Comprehensions: Pages 83-88, 121-125
- Context Managers: Pages 126-130
- Decorators: Pages 131-135

A.2.3 External Libraries

- Web Scraping: Pages 136-145
- Machine Learning: Pages 146-160
- API Integration: Pages 161-170
- Data Processing: Pages 171-180

Appendix B

System Extension Guidelines

B.1 Adding New Agents

To add a new agent to the system:

```python
from agent import Agent

class CustomAgent(Agent):
    name = "Custom Agent"
    color = Agent.CYAN  # Choose color

    def __init__(self):
        super().__init__()
        # Initialize agent-specific resources

    def custom_method(self, parameters):
        """Agent-specific functionality"""
        self.log("Performing custom operation")
        # Implementation here
        return result
```

Listing B.1: New Agent Template

B.2 Configuration Management

Implement centralized configuration:

```python
import json
import os

class SystemConfig:
    def __init__(self, config_file="config.json"):
        self.config = self.load_config(config_file)

    def load_config(self, file_path):
```

```
        if os.path.exists(file_path):
            with open(file_path, 'r') as f:
                return json.load(f)
        return self.default_config()

    def default_config(self):
        return {
            "deal_threshold": 50.0,
            "max_deals": 10,
            "api_timeout": 30
        }
```

Listing B.2: Configuration System

Appendix C

Troubleshooting Guide

C.1 Common Issues

C.1.1 Import Errors

Problem: ModuleNotFoundError for external libraries
Solution: Install required packages using pip install -r requirements.txt

C.1.2 API Authentication

Problem: Authentication failures with external APIs
Solution: Verify environment variables and API key configuration

C.1.3 Memory Issues

Problem: High memory usage with large datasets
Solution: Implement data streaming and batch processing

C.2 Performance Optimization

- Use connection pooling for HTTP requests
- Implement caching for expensive computations
- Add parallel processing for independent operations
- Profile code to identify bottlenecks

References and Further Reading

C.3 Python Documentation

- Official Python Documentation: https://docs.python.org/3/
- Python Enhancement Proposals (PEPs): https://www.python.org/dev/peps/
- Python Package Index (PyPI): https://pypi.org/

C.4 Machine Learning Resources

- Scikit-learn Documentation: https://scikit-learn.org/stable/
- OpenAI API Documentation: https://platform.openai.com/docs
- ChromaDB Documentation: https://docs.trychroma.com/

C.5 Web Development

- BeautifulSoup Documentation: https://www.crummy.com/software/BeautifulSoup/bs4/doc/
- Requests Library: https://docs.python-requests.org/
- Feedparser Documentation: https://feedparser.readthedocs.io/

C.6 Software Engineering

- Clean Code by Robert C. Martin
- Design Patterns: Elements of Reusable Object-Oriented Software
- Python Tricks: The Book by Dan Bader

Index

This comprehensive guide provides over 200 pages of detailed technical documentation, code examples, and system analysis. The multi-agent deal discovery system serves as an excellent example of modern Python development practices, demonstrating sophisticated software engineering principles in a real-world application context.

End of Documentation

Generated on September 29, 2025

AI Assistant - Technical Documentation

C.7 Agents

C.7.1 Agents Module

```python
import logging

class Agent:
    """
    An abstract superclass for Agents
    Used to log messages in a way that can identify each Agent
    """

    # Foreground colors
    RED = '\033[31m'
    GREEN = '\033[32m'
    YELLOW = '\033[33m'
    BLUE = '\033[34m'
    MAGENTA = '\033[35m'
    CYAN = '\033[36m'
    WHITE = '\033[37m'

    # Background color
    BG_BLACK = '\033[40m'

    # Reset code to return to default color
    RESET = '\033[0m'

    name: str = ""
    color: str = '\033[37m'

    def log(self, message):
        """
        Log this as an info message, identifying the agent
        """
        color_code = self.BG_BLACK + self.color
        message = f"[{self.name}] {message}"
        logging.info(color_code + message + self.RESET)
```

Listing C.1: Agents Module

C.7.2 Deals Module

```
from pydantic import BaseModel
from typing import List, Dict, Self
from bs4 import BeautifulSoup
import re
import feedparser
from tqdm import tqdm
import requests
import time

feeds = [
    "https://www.dealnews.com/c142/Electronics/?rss=1",
        "https://www.dealnews.com/c39/Computers/?rss=1",
        "https://www.dealnews.com/c238/Automotive/?rss=1",
        "https://www.dealnews.com/f1912/Smart-Home/?rss=1",
        "https://www.dealnews.com/c196/Home-Garden/?rss=1",
       ]

def extract(html_snippet: str) -> str:
    """
    Use Beautiful Soup to clean up this HTML snippet and extract useful text
    """
    soup = BeautifulSoup(html_snippet, 'html.parser')
    snippet_div = soup.find('div', class_='snippet summary')

    if snippet_div:
        description = snippet_div.get_text(strip=True)
        description = BeautifulSoup(description, 'html.parser').get_text()
        description = re.sub('<[^<]+?>', '', description)
        result = description.strip()
    else:
        result = html_snippet
    return result.replace('\n', ' ')

class ScrapedDeal:
    """
    A class to represent a Deal retrieved from an RSS feed
    """
    category: str
    title: str
    summary: str
    url: str
    details: str
    features: str

    def __init__(self, entry: Dict[str, str]):
        """
        Populate this instance based on the provided dict
        """
        self.title = entry['title']
```

```
        self.summary = extract(entry['summary'])
        self.url = entry['links'][0]['href']
        stuff = requests.get(self.url).content
        soup = BeautifulSoup(stuff, 'html.parser')
        content = soup.find('div', class_='content-section').get_text()
        content = content.replace('\nmore', '').replace('\n', ' ')
        if "Features" in content:
            self.details, self.features = content.split("Features")
        else:
            self.details = content
            self.features = ""

    def __repr__(self):
        """
        Return a string to describe this deal
        """
        return f"<{self.title}>"

    def describe(self):
        """
        Return a longer string to describe this deal for use in calling a model
        """
        return f"Title: {self.title}\nDetails: {self.details.strip()}\nFeatures:
    {self.features.strip()}\nURL: {self.url}"

    @classmethod
    def fetch(cls, show_progress : bool = False) -> List[Self]:
        """
        Retrieve all deals from the selected RSS feeds
        """
        deals = []
        feed_iter = tqdm(feeds) if show_progress else feeds
        for feed_url in feed_iter:
            feed = feedparser.parse(feed_url)
            for entry in feed.entries[:10]:
                deals.append(cls(entry))
                time.sleep(0.5)
        return deals

class Deal(BaseModel):
    """
    A class to Represent a Deal with a summary description
    """
    product_description: str
    price: float
    url: str

class DealSelection(BaseModel):
    """
    A class to Represent a list of Deals
    """
```

```
    deals: List[Deal]

class Opportunity(BaseModel):
    """
    A class to represent a possible opportunity: a Deal where we estimate
    it should cost more than it's being offered
    """
    deal: Deal
    estimate: float
    discount: float
```

Listing C.2: Deals Module

C.7.3 Ensemble Agent

```python
import pandas as pd
from sklearn.linear_model import LinearRegression
import joblib

from agents.agent import Agent
from agents.specialist_agent import SpecialistAgent
from agents.frontier_agent import FrontierAgent
from agents.random_forest_agent import RandomForestAgent

class EnsembleAgent(Agent):

    name = "Ensemble Agent"
    color = Agent.YELLOW

    def __init__(self, collection):
        """
        Create an instance of Ensemble, by creating each of the models
        And loading the weights of the Ensemble
        """
        self.log("Initializing Ensemble Agent")
        self.specialist = SpecialistAgent()
        self.frontier = FrontierAgent(collection)
        self.random_forest = RandomForestAgent()
        self.model = joblib.load('ensemble_model.pkl')
        self.log("Ensemble Agent is ready")

    def price(self, description: str) -> float:
        """
        Run this ensemble model
        Ask each of the models to price the product
        Then use the Linear Regression model to return the weighted price
        :param description: the description of a product
        :return: an estimate of its price
        """
        self.log("Running Ensemble Agent - collaborating with specialist, frontier and random forest agents")
        specialist = self.specialist.price(description)
        frontier = self.frontier.price(description)
        random_forest = self.random_forest.price(description)
        X = pd.DataFrame({
            'Specialist': [specialist],
            'Frontier': [frontier],
            'RandomForest': [random_forest],
            'Min': [min(specialist, frontier, random_forest)],
            'Max': [max(specialist, frontier, random_forest)],
        })
        y = max(0, self.model.predict(X)[0])
        self.log(f"Ensemble Agent complete - returning ${y:.2f}")
        return y
```

Listing C.3: Ensemble Agent

C.7.4 Messaging Agent

```
import os
# from twilio.rest import Client
from agents.deals import Opportunity
import http.client
import urllib
from agents.agent import Agent

# Uncomment the Twilio lines if you wish to use Twilio

DO_TEXT = False
DO_PUSH = True

class MessagingAgent(Agent):

    name = "Messaging Agent"
    color = Agent.WHITE

    def __init__(self):
        """
        Set up this object to either do push notifications via Pushover,
        or SMS via Twilio,
        whichever is specified in the constants
        """
        self.log(f"Messaging Agent is initializing")
        if DO_TEXT:
            account_sid = os.getenv('TWILIO_ACCOUNT_SID',
    'your-sid-if-not-using-env')
            auth_token = os.getenv('TWILIO_AUTH_TOKEN',
    'your-auth-if-not-using-env')
            self.me_from = os.getenv('TWILIO_FROM',
    'your-phone-number-if-not-using-env')
            self.me_to = os.getenv('MY_PHONE_NUMBER',
    'your-phone-number-if-not-using-env')
            # self.client = Client(account_sid, auth_token)
            self.log("Messaging Agent has initialized Twilio")
        if DO_PUSH:
            self.pushover_user = os.getenv('PUSHOVER_USER',
    'your-pushover-user-if-not-using-env')
            self.pushover_token = os.getenv('PUSHOVER_TOKEN',
    'your-pushover-user-if-not-using-env')
            self.log("Messaging Agent has initialized Pushover")

    def message(self, text):
        """
        Send an SMS message using the Twilio API
        """
        self.log("Messaging Agent is sending a text message")
        message = self.client.messages.create(
          from_=self.me_from,
```

```
            body=text,
            to=self.me_to
        )

    def push(self, text):
        """
        Send a Push Notification using the Pushover API
        """
        self.log("Messaging Agent is sending a push notification")
        conn = http.client.HTTPSConnection("api.pushover.net:443")
        conn.request("POST", "/1/messages.json",
          urllib.parse.urlencode({
            "token": self.pushover_token,
            "user": self.pushover_user,
            "message": text,
            "sound": "cashregister"
          }), { "Content-type": "application/x-www-form-urlencoded" })
        conn.getresponse()

    def alert(self, opportunity: Opportunity):
        """
        Make an alert about the specified Opportunity
        """
        text = f"Deal Alert! Price=${opportunity.deal.price:.2f}, "
        text += f"Estimate=${opportunity.estimate:.2f}, "
        text += f"Discount=${opportunity.discount:.2f} :"
        text += opportunity.deal.product_description[:10]+'... '
        text += opportunity.deal.url
        if DO_TEXT:
            self.message(text)
        if DO_PUSH:
            self.push(text)
        self.log("Messaging Agent has completed")
```

Listing C.4: Messaging Agent

C.7.5 Frontier Agent

```
# imports

import os
import re
import math
import json
from typing import List, Dict
from openai import OpenAI
from sentence_transformers import SentenceTransformer
from datasets import load_dataset
import chromadb
from items import Item
from testing import Tester
from agents.agent import Agent


class FrontierAgent(Agent):

    name = "Frontier Agent"
    color = Agent.BLUE

    MODEL = "gpt-4o-mini"

    def __init__(self, collection):
        """
        Set up this instance by connecting to OpenAI or DeepSeek, to the Chroma Datastore,
        And setting up the vector encoding model
        """
        self.log("Initializing Frontier Agent")
        deepseek_api_key = os.getenv("DEEPSEEK_API_KEY")
        if deepseek_api_key:
            self.client = OpenAI(api_key=deepseek_api_key, base_url="https://api.deepseek.com")
            self.MODEL = "deepseek-chat"
            self.log("Frontier Agent is set up with DeepSeek")
        else:
            self.client = OpenAI()
            self.MODEL = "gpt-4o-mini"
            self.log("Frontier Agent is setting up with OpenAI")
        self.collection = collection
        self.model = SentenceTransformer('sentence-transformers/all-MiniLM-L6-v2')
        self.log("Frontier Agent is ready")

    def make_context(self, similars: List[str], prices: List[float]) -> str:
        """
        Create context that can be inserted into the prompt
        :param similars: similar products to the one being estimated
```

```
        :param prices: prices of the similar products
        :return: text to insert in the prompt that provides context
        """
        message = "To provide some context, here are some other items that might be similar to the item you need to estimate.\n\n"
        for similar, price in zip(similars, prices):
            message += f"Potentially related product:\n{similar}\nPrice is ${price:.2f}\n\n"
        return message

    def messages_for(self, description: str, similars: List[str], prices: List[float]) -> List[Dict[str, str]]:
        """
        Create the message list to be included in a call to OpenAI
        With the system and user prompt
        :param description: a description of the product
        :param similars: similar products to this one
        :param prices: prices of similar products
        :return: the list of messages in the format expected by OpenAI
        """
        system_message = "You estimate prices of items. Reply only with the price, no explanation"
        user_prompt = self.make_context(similars, prices)
        user_prompt += "And now the question for you:\n\n"
        user_prompt += "How much does this cost?\n\n" + description
        return [
            {"role": "system", "content": system_message},
            {"role": "user", "content": user_prompt},
            {"role": "assistant", "content": "Price is $"}
        ]

    def find_similars(self, description: str):
        """
        Return a list of items similar to the given one by looking in the Chroma datastore
        """
        self.log("Frontier Agent is performing a RAG search of the Chroma datastore to find 5 similar products")
        vector = self.model.encode([description])
        results = self.collection.query(query_embeddings=vector.astype(float).tolist(), n_results=5)
        documents = results['documents'][0][:]
        prices = [m['price'] for m in results['metadatas'][0][:]]
        self.log("Frontier Agent has found similar products")
        return documents, prices

    def get_price(self, s) -> float:
        """
        A utility that plucks a floating point number out of a string
        """
```

```
        s = s.replace('$','').replace(',','')
        match = re.search(r"[-+]?\d*\.\d+|\d+", s)
        return float(match.group()) if match else 0.0

    def price(self, description: str) -> float:
        """
        Make a call to OpenAI or DeepSeek to estimate the price of the described product,
        by looking up 5 similar products and including them in the prompt to give context
        :param description: a description of the product
        :return: an estimate of the price
        """
        documents, prices = self.find_similars(description)
        self.log(f"Frontier Agent is about to call {self.MODEL} with context including 5 similar products")
        response = self.client.chat.completions.create(
            model=self.MODEL,
            messages=self.messages_for(description, documents, prices),
            seed=42,
            max_tokens=5
        )
        reply = response.choices[0].message.content
        result = self.get_price(reply)
        self.log(f"Frontier Agent completed - predicting ${result:.2f}")
        return result
```

Listing C.5: Frontier Agent

C.7.6 Planning Agent

```python
from typing import Optional, List
from agents.agent import Agent
from agents.deals import ScrapedDeal, DealSelection, Deal, Opportunity
from agents.scanner_agent_langchain import ScannerAgent
from agents.ensemble_agent import EnsembleAgent
from agents.messaging_agent import MessagingAgent


class PlanningAgent(Agent):

    name = "Planning Agent"
    color = Agent.GREEN
    DEAL_THRESHOLD = 50

    def __init__(self, collection):
        """
        Create instances of the 3 Agents that this planner coordinates across
        """
        self.log("Planning Agent is initializing")
        self.scanner = ScannerAgent()
        self.ensemble = EnsembleAgent(collection)
        self.messenger = MessagingAgent()
        self.log("Planning Agent is ready")

    def run(self, deal: Deal) -> Opportunity:
        """
        Run the workflow for a particular deal
        :param deal: the deal, summarized from an RSS scrape
        :returns: an opportunity including the discount
        """
        self.log("Planning Agent is pricing up a potential deal")
        estimate = self.ensemble.price(deal.product_description)
        discount = estimate - deal.price
        self.log(f"Planning Agent has processed a deal with discount ${discount:.2f}")
        return Opportunity(deal=deal, estimate=estimate, discount=discount)

    def plan(self, memory: List[str] = []) -> Optional[Opportunity]:
        """
        Run the full workflow:
        1. Use the ScannerAgent to find deals from RSS feeds
        2. Use the EnsembleAgent to estimate them
        3. Use the MessagingAgent to send a notification of deals
        :param memory: a list of URLs that have been surfaced in the past
        :return: an Opportunity if one was surfaced, otherwise None
        """
        self.log("Planning Agent is kicking off a run")
        selection = self.scanner.scan(memory=memory)
        if selection:
```

```
            opportunities = [self.run(deal) for deal in selection.deals[:5]]
            opportunities.sort(key=lambda opp: opp.discount, reverse=True)
            best = opportunities[0]
            self.log(f"Planning Agent has identified the best deal has discount ${best.discount:.2f}")
            if best.discount > self.DEAL_THRESHOLD:
                self.messenger.alert(best)
            self.log("Planning Agent has completed a run")
            return best if best.discount > self.DEAL_THRESHOLD else None
        return None
```

Listing C.6: Planning Agent

C.7.7 Random Forest Agent

```python
# imports

import os
import re
from typing import List
from sentence_transformers import SentenceTransformer
import joblib
from agents.agent import Agent



class RandomForestAgent(Agent):

    name = "Random Forest Agent"
    color = Agent.MAGENTA

    def __init__(self):
        """
        Initialize this object by loading in the saved model weights
        and the SentenceTransformer vector encoding model
        """
        self.log("Random Forest Agent is initializing")
        self.vectorizer =
   SentenceTransformer('sentence-transformers/all-MiniLM-L6-v2')
        self.model = joblib.load('random_forest_model.pkl')
        self.log("Random Forest Agent is ready")

    def price(self, description: str) -> float:
        """
        Use a Random Forest model to estimate the price of the described item
        :param description: the product to be estimated
        :return: the price as a float
        """
        self.log("Random Forest Agent is starting a prediction")
        vector = self.vectorizer.encode([description])
        result = max(0, self.model.predict(vector)[0])
        self.log(f"Random Forest Agent completed - predicting ${result:.2f}")
        return result
```

Listing C.7: Random Forest Agent

C.7.8 Scanner Agent

```
import os
import json
from typing import Optional, List
from openai import OpenAI
from agents.deals import ScrapedDeal, DealSelection
from agents.agent import Agent


class ScannerAgent(Agent):

    MODEL = "gpt-4o-mini"

    SYSTEM_PROMPT = """You identify and summarize the 5 most detailed deals from a list, by selecting deals that have the most detailed, high quality description and the most clear price.
    Respond strictly in JSON with no explanation, using this format. You should provide the price as a number derived from the description. If the price of a deal isn't clear, do not include that deal in your response.
    Most important is that you respond with the 5 deals that have the most detailed product description with price. It's not important to mention the terms of the deal; most important is a thorough description of the product.
    Be careful with products that are described as "$XXX off" or "reduced by $XXX" - this isn't the actual price of the product. Only respond with products when you are highly confident about the price.

    {"deals": [
        {
            "product_description": "Your clearly expressed summary of the product in 4-5 sentences. Details of the item are much more important than why it's a good deal. Avoid mentioning discounts and coupons; focus on the item itself. There should be a paragpraph of text for each item you choose.",
            "price": 99.99,
            "url": "the url as provided"
        },
        ...
    ]}"""

    USER_PROMPT_PREFIX = """Respond with the most promising 5 deals from this list, selecting those which have the most detailed, high quality product description and a clear price that is greater than 0.
    Respond strictly in JSON, and only JSON. You should rephrase the description to be a summary of the product itself, not the terms of the deal.
    Remember to respond with a paragraph of text in the product_description field for each of the 5 items that you select.
    Be careful with products that are described as "$XXX off" or "reduced by $XXX" - this isn't the actual price of the product. Only respond with products when you are highly confident about the price.

    Deals:
```

```
    """

    USER_PROMPT_SUFFIX = "\n\nStrictly respond in JSON and include exactly 5 deals, no more."

    name = "Scanner Agent"
    color = Agent.CYAN

    def __init__(self):
        """
        Set up this instance by initializing OpenAI
        """
        self.log("Scanner Agent is initializing")
        self.openai = OpenAI()
        self.log("Scanner Agent is ready")

    def fetch_deals(self, memory) -> List[ScrapedDeal]:
        """
        Look up deals published on RSS feeds
        Return any new deals that are not already in the memory provided
        """
        self.log("Scanner Agent is about to fetch deals from RSS feed")
        urls = [opp.deal.url for opp in memory]
        scraped = ScrapedDeal.fetch()
        result = [scrape for scrape in scraped if scrape.url not in urls]
        self.log(f"Scanner Agent received {len(result)} deals not already scraped")
        return result

    def make_user_prompt(self, scraped) -> str:
        """
        Create a user prompt for OpenAI based on the scraped deals provided
        """
        user_prompt = self.USER_PROMPT_PREFIX
        user_prompt += '\n\n'.join([scrape.describe() for scrape in scraped])
        user_prompt += self.USER_PROMPT_SUFFIX
        return user_prompt

    def scan(self, memory: List[str]=[]) -> Optional[DealSelection]:
        """
        Call OpenAI to provide a high potential list of deals with good descriptions and prices
        Use StructuredOutputs to ensure it conforms to our specifications
        :param memory: a list of URLs representing deals already raised
        :return: a selection of good deals, or None if there aren't any
        """
        scraped = self.fetch_deals(memory)
        if scraped:
            user_prompt = self.make_user_prompt(scraped)
            self.log("Scanner Agent is calling OpenAI using Structured Output")
```

```
            result = self.openai.beta.chat.completions.parse(
                model=self.MODEL,
                messages=[
                    {"role": "system", "content": self.SYSTEM_PROMPT},
                    {"role": "user", "content": user_prompt}
              ],
                response_format=DealSelection
            )
            result = result.choices[0].message.parsed
            result.deals = [deal for deal in result.deals if deal.price>0]
            self.log(f"Scanner Agent received {len(result.deals)} selected deals
    with price>0 from OpenAI")
            return result
        return None
```

Listing C.8: Scanner Agent

C.7.9 Scanner Agent Using Langchain

```
from typing import List, Optional
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain_core.prompts import ChatPromptTemplate
from agents.deals import ScrapedDeal, DealSelection
from agents.agent import Agent

from dotenv import load_dotenv
load_dotenv(override=True)
import os
os.environ['GOOGLE_API_KEY'] = os.getenv("GOOGLE_API_KEY_1")
class ScannerAgent(Agent):
    """
    ScannerAgent implemented using LangChain. Fetches deals from RSS feeds
    and selects the top 5 with the most detailed description and clear price.
    """

    MODEL = "gemini-2.5-flash"  # Or "gemini-2.0-flash" for Google
    SYSTEM_PROMPT = """You are a highly analytical AI that extracts the 5 best
    deals from a list. Your most critical task is to distinguish between a final
    price and a discount amount.

You will select deals based on two criteria:
1.  A detailed, high-quality product description.
2.  A clear, unambiguous, and final price.

CRUCIAL RULE ON PRICING: The price you extract MUST be the final, total cost of
    the item. It is never a discount amount.
-   If a deal mentions a price using words like "$XXX off", "save $XXX",
    "reduced by $XXX", or "a $XXX discount", you MUST IGNORE AND DISCARD that
    deal completely.
-   You must look for explicit pricing clues like "Price:", "Costs:", "For:", or
    a standalone number that clearly represents the total sale price.
-   If you have ANY doubt about whether a number is a final price or a discount,
    DO NOT include the deal. It is better to return fewer than 5 deals than to
    include one with an incorrect price.

For the 5 deals you select, respond strictly in JSON with no explanation. The
    `product_description` should be a summary of the item itself, not the deal's
    terms.

{{"deals": [
    {{
        "product_description": "Your clearly expressed summary of the product in
    4-5 sentences. Details of the item are much more important than why it's a
    good deal. Avoid mentioning discounts and coupons; focus on the item itself.
    There should be a paragraph of text for each item you choose.",
        "price": 99.99,
        "url": "the url as provided"
    }},
```

```
    ...
]}}"""

    USER_PROMPT_PREFIX = """Respond with the most promising 5 deals from this
    list, selecting those which have the most detailed, high quality product
    description and a clear price that is greater than 0.
Respond strictly in JSON, and only JSON. You should rephrase the description to
    be a summary of the product itself, not the terms of the deal.

Deals:
"""

    USER_PROMPT_SUFFIX = "\n\nStrictly respond in JSON and include exactly 5
    deals, no more."

    name = "Scanner Agent"
    color = Agent.CYAN

    def __init__(self, api_key: Optional[str] = None):
        """
        Initialize the ScannerAgentLC with a LangChain chat model.

        :param api_key: Optional API key for OpenAI or Google LLMs
        """
        self.log("Scanner Agent is initializing (LangChain)")
        self.llm = ChatGoogleGenerativeAI(model = "gemini-2.5-flash"
        )
        self.log("Scanner Agent is ready")

    def fetch_deals(self, memory: List[ScrapedDeal]) -> List[ScrapedDeal]:
        """
        Retrieve all deals from RSS feeds that are not already in memory.

        :param memory: List of previously seen ScrapedDeal objects
        :return: List of new ScrapedDeal objects
        """
        self.log("Scanner Agent fetching deals")
        urls = [d.url for d in memory]
        scraped = ScrapedDeal.fetch()
        result = [scrape for scrape in scraped if scrape.url not in urls]
        self.log(f"Scanner Agent found {len(result)} new deals")
        return result

    def make_user_prompt(self, scraped: List[ScrapedDeal]) -> ChatPromptTemplate:
        """
        Create a user prompt for the LLM based on the scraped deals.

        :param scraped: List of ScrapedDeal objects
        :return: ChatPromptTemplate containing system and user messages
        """
        user_content = '\n\n'.join([scrape.describe() for scrape in scraped])
```

```
84          full_prompt = self.USER_PROMPT_PREFIX + user_content +
    self.USER_PROMPT_SUFFIX
85          return ChatPromptTemplate([
86              ("system", self.SYSTEM_PROMPT),
87              ("user", full_prompt)
88          ])
89
90      def scan(self, memory: List[ScrapedDeal] = []) -> Optional[DealSelection]:
91          """
92          Call the LLM to select the top 5 deals with detailed descriptions and
    clear prices.
93
94          :param memory: List of ScrapedDeal objects already seen
95          :return: DealSelection object containing the selected deals or None
96          """
97          scraped = self.fetch_deals(memory)
98          if not scraped:
99              return None
100
101         prompt = self.make_user_prompt(scraped)
102         self.log("Scanner Agent calling LLM (LangChain)")
103
104         # Generate response using LangChain
105         response = self.llm(prompt.format_prompt().to_messages())
106
107         # Parse structured output
108         try:
109             result = DealSelection.parse_raw(response.content)
110             result.deals = [deal for deal in result.deals if deal.price > 0]
111             self.log(f"Scanner Agent received {len(result.deals)} deals with
    price>0")
112             return result
113         except Exception as e:
114             self.log(f"Error parsing deals: {e}")
115             return None
```

Listing C.9: Scanner Agent Using Langchain

C.7.10 Specialist Agent

```python
import modal
from agents.agent import Agent


class SpecialistAgent(Agent):
    """
    An Agent that runs our fine-tuned LLM that's running remotely on Modal
    """

    name = "Specialist Agent"
    color = Agent.RED

    def __init__(self):
        """
        Set up this Agent by creating an instance of the modal class
        """
        self.log("Specialist Agent is initializing - connecting to modal")
        Pricer = modal.Cls.from_name("pricer-service", "Pricer")
        self.pricer = Pricer()
        self.log("Specialist Agent is ready")

    def price(self, description: str) -> float:
        """
        Make a remote call to return the estimate of the price of this item
        """
        self.log("Specialist Agent is calling remote fine-tuned model")
        result = self.pricer.price.remote(description)
        self.log(f"Specialist Agent completed - predicting ${result:.2f}")
        return result
```

Listing C.10: Specialist Agent