

THE ANATOMY OF MODERN AI

FROM TOKEN VECTORS TO AUTONOMOUS AGENTIC WORKFLOWS

```
import torch
class Transformer(nn.Module):
    def forward(self, x):
```

FINE-TUNING

ADAPTING FOUNDATION MODELS FOR SPECIFIC TASKS AND ALIGNMENT.

ENGINE

CORE NEURAL ARCHITECTURES, PRE-TRAINING, AND MODEL WEIGHTS.

INTERFACES

STANDARD PROTOCOLS AND TOOLS FOR MODEL INTERACTION AND DEPLOYMENT.

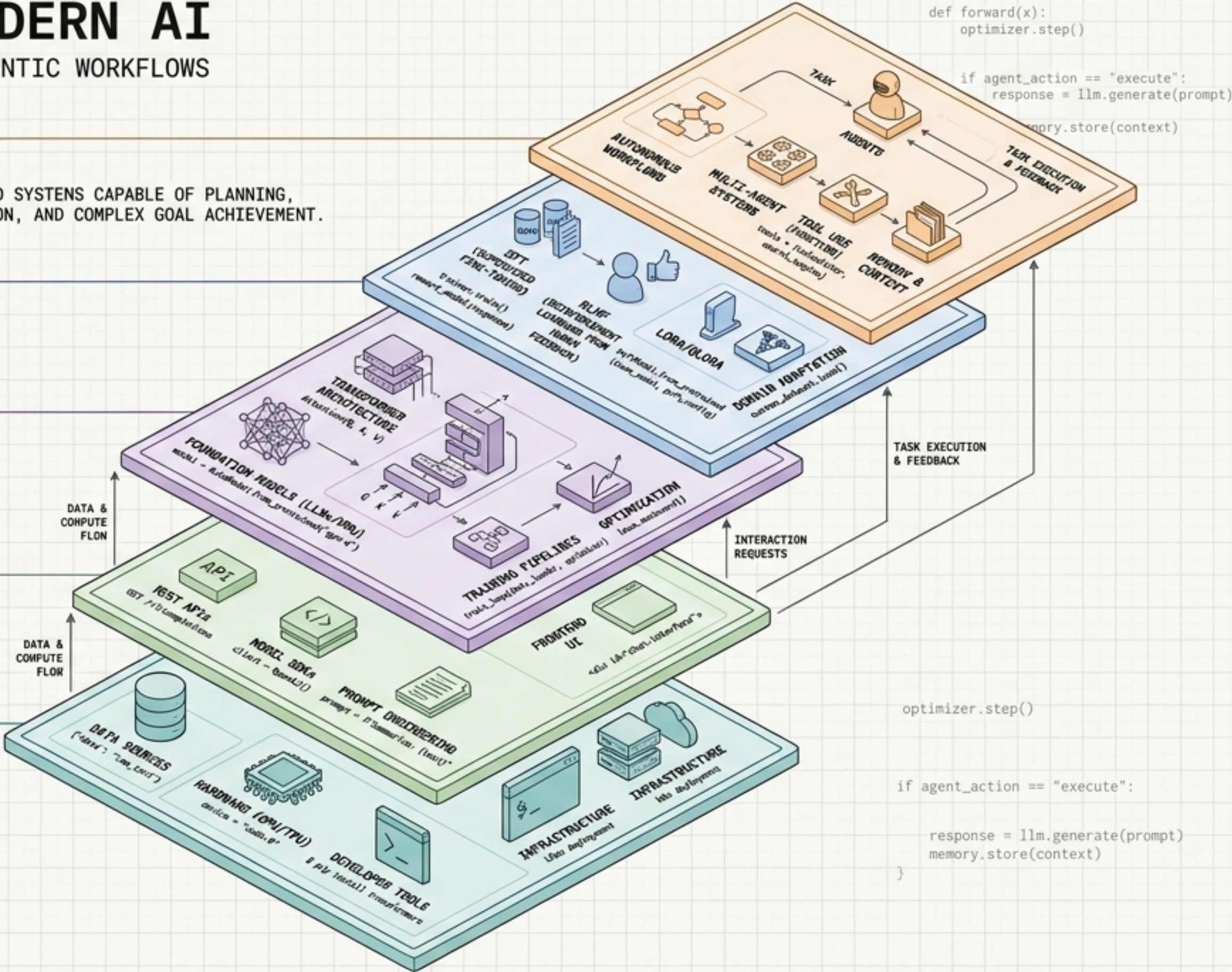
ECOSYSTEM

FOUNDATIONAL INFRASTRUCTURE, DATA LAKES, AND COMPUTE RESOURCES.

```
def forward(self, x):
    agent.step()
}
```

AGENTS

SELF-DIRECTED SYSTEMS CAPABLE OF PLANNING, TOOL EXECUTION, AND COMPLEX GOAL ACHIEVEMENT.



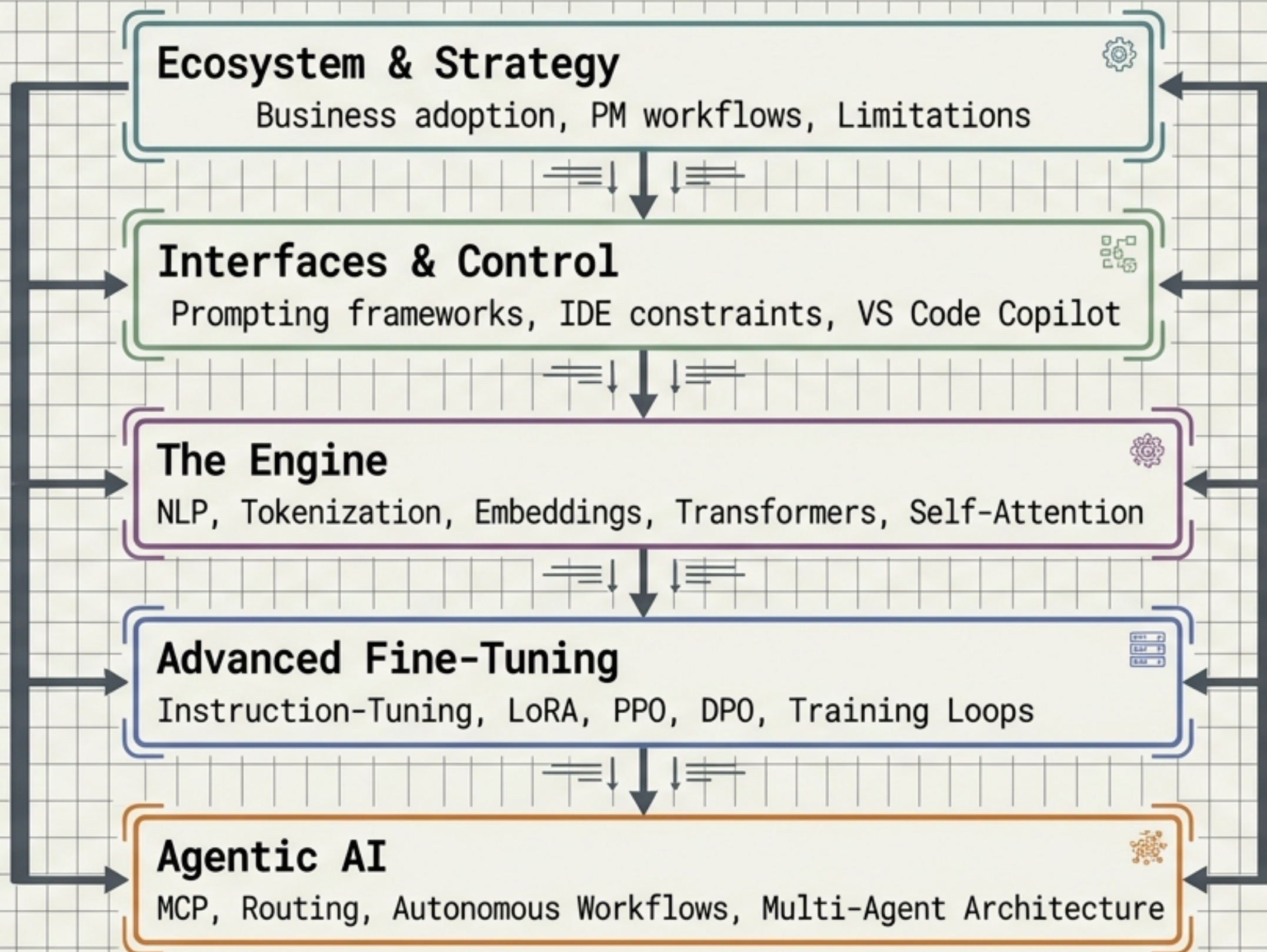
```
def forward(x):
    optimizer.step()

if agent_action == "execute":
    response = llm.generate(prompt)
    memory.store(context)

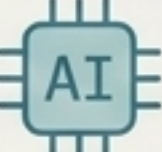
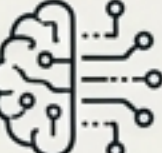
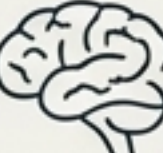
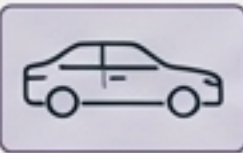
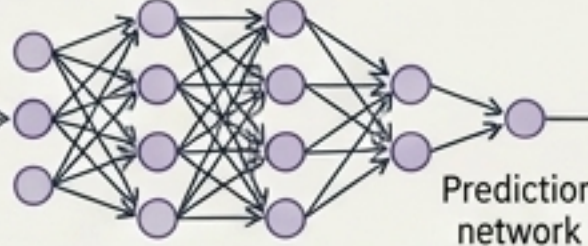
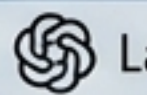
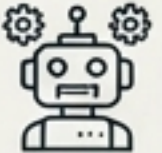
optimizer.step()

if agent_action == "execute":
    response = llm.generate(prompt)
    memory.store(context)
}
```


Navigating the Modern AI Stack

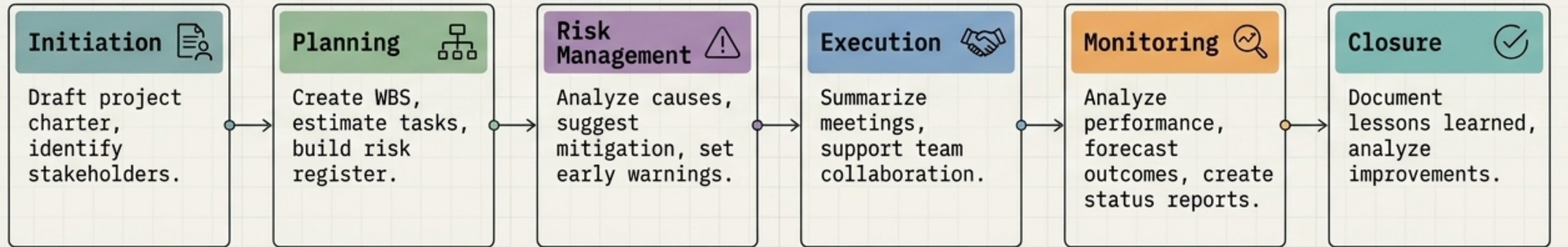


Defining the Boundaries of AI Disciplines

	INPUTS	MECHANISMS	OUTPUTS
 Artificial Intelligence (AI)	Machine Learning Generative AI Deep Learning		Neural Network Computer Vision NLP
 Machine Learning		Detection, Clustering → Unsupervised learning Ranking, Regression → Supervised learning Policy, ModelFree → Reinforcement Learning	
 Deep Learning		 →  Prediction network → This is a car	
 Generative AI	User	 Large Language Model → Tools & Data sources	Output
 RAG	User	Embedding -> Vector Db Data sources Retrieval → Data + System Prompt + User Prompt Tools Augmented	Generation → Output
 AI Agents	User	System Prompt User Prompt → Agent → Database Memory → Feedback Reasoning → Planning Tools	Output

Injecting AI into the Project Lifecycle

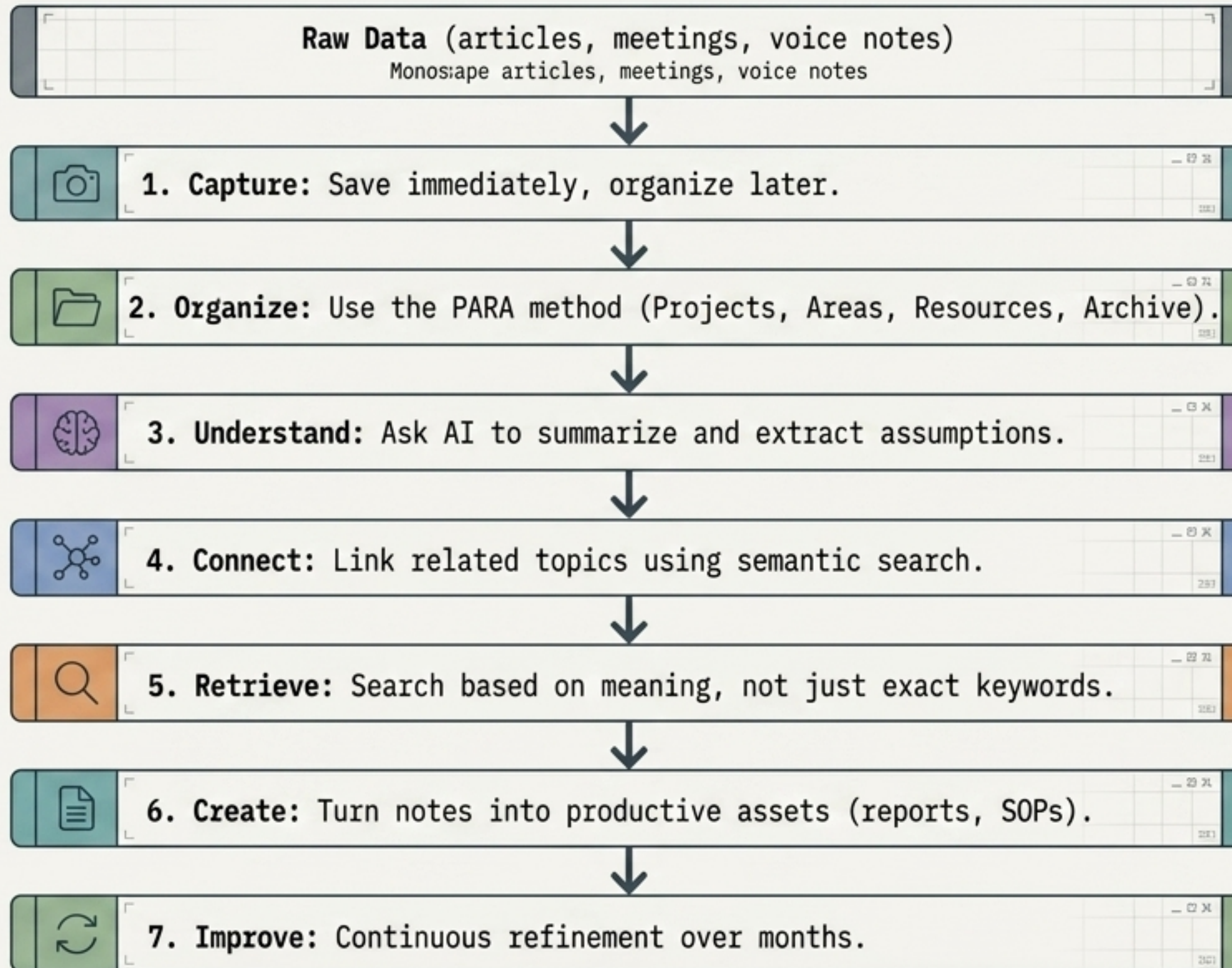
PROJECT LIFECYCLE TIMELINE



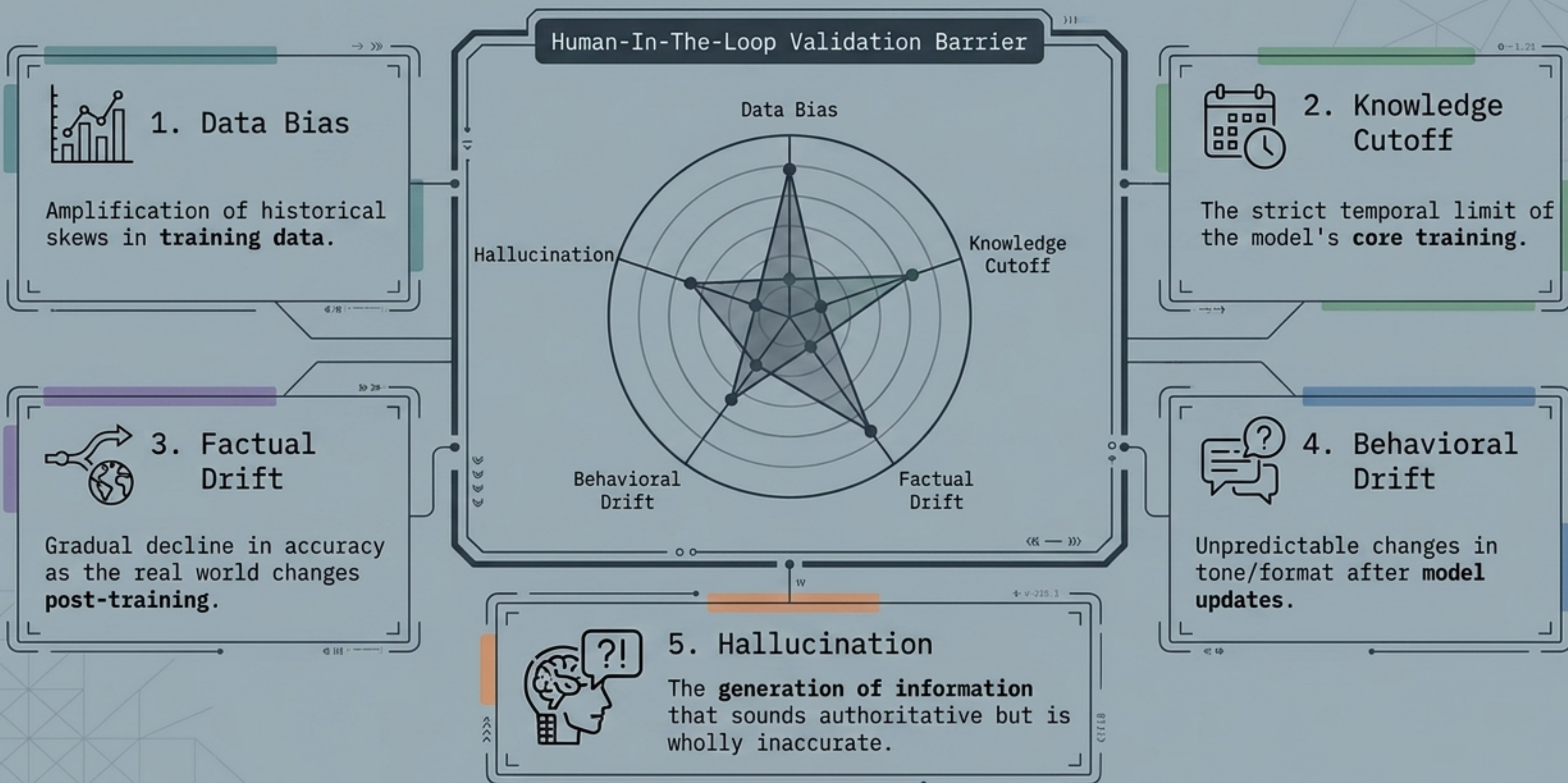
THE PM + AI MINDSET



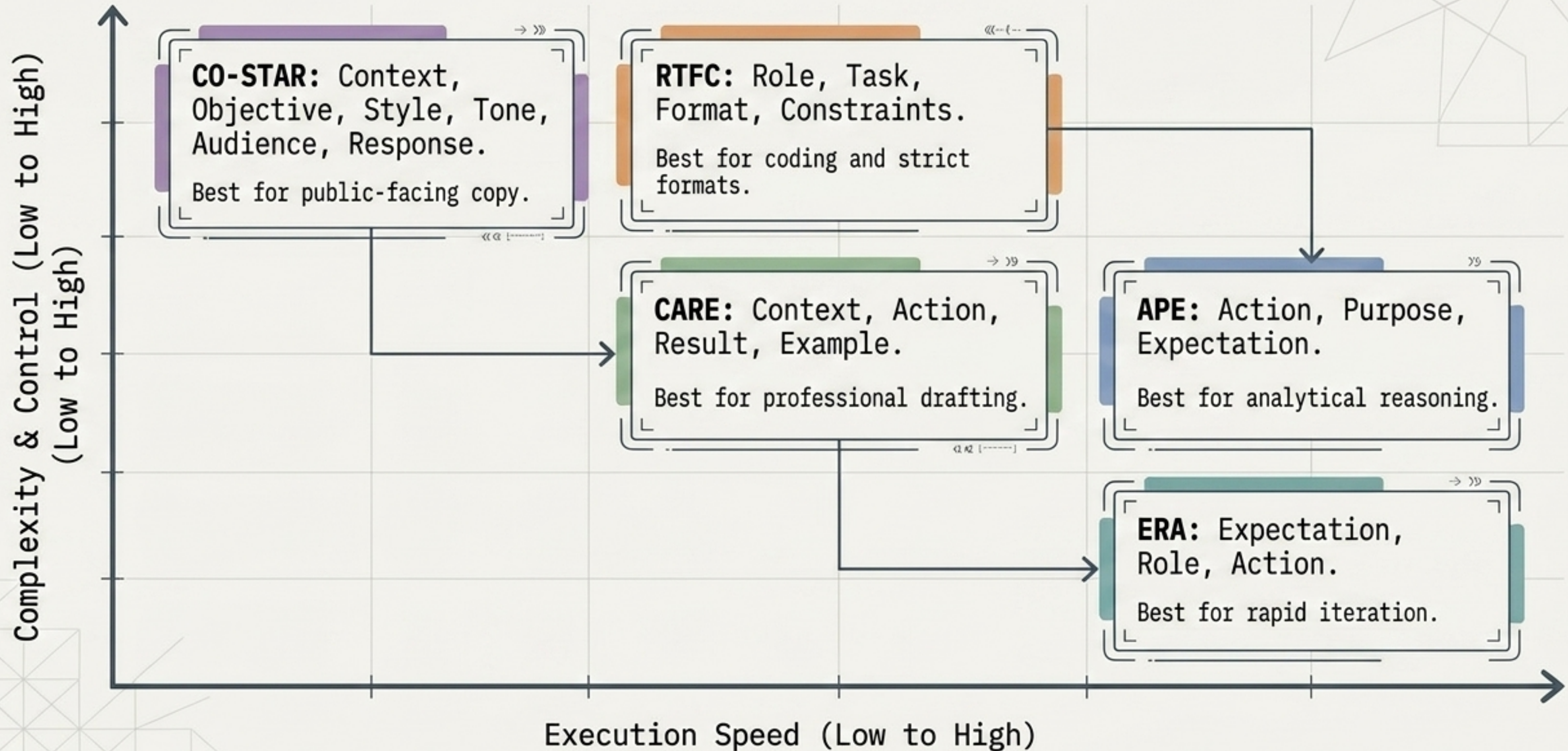
Architecting an AI-Powered Second Brain



Navigating Limitations: Bias, Drift, and Hallucinations



The Prompt Selection Matrix



Constraining the AI Assistant in the IDE

Kills **ghost text** to reduce distraction.

Restricts privacy leaks by hiding **full file context** from the LLM.

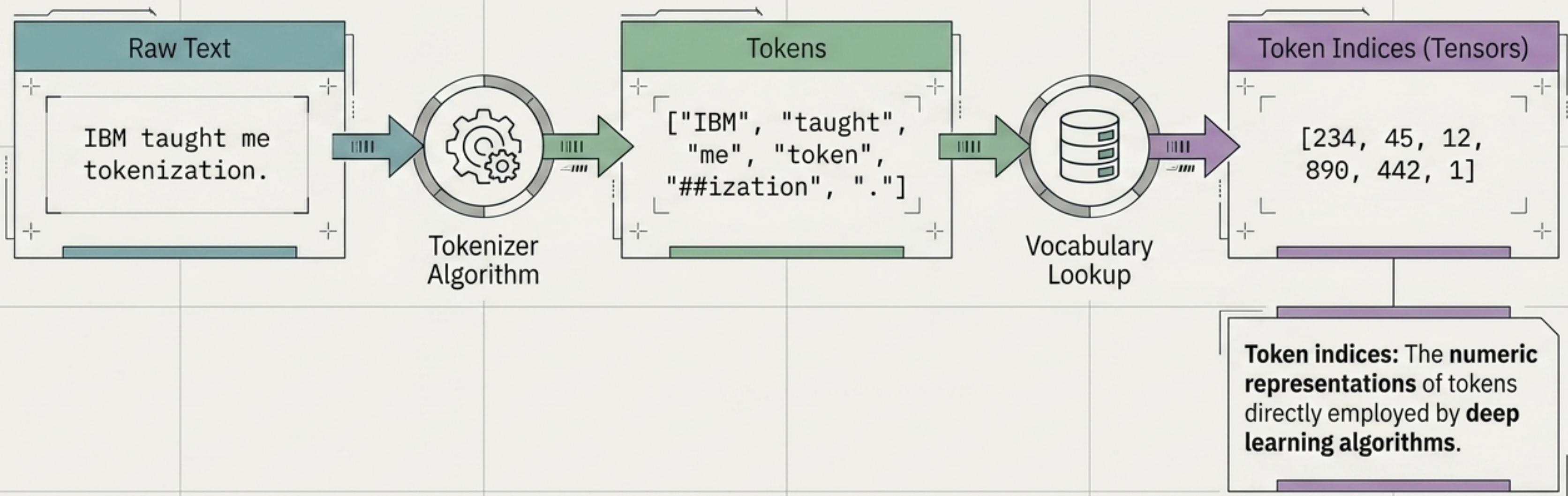
settings.json

```
{
  "github.copilot.inlineSuggest.enable": false,
  "github.copilot.suggestion.length": 20,
  "github.copilot.advanced.editorContext":
    "currentLine",
  "github.copilot.enable": {
    "*": false,
    "python": true,
    "javascript": true
  }
}
```

Imposes a strict **20-line ceiling** on generation to prevent runaway code.

Object block explicitly disabling **'**'** but enabling **'python'** and **'javascript'**.

Tokenization: Shattering Language into Numbers



Algorithm Tradeoffs: Word vs. Character vs. Subword

Word-based	Character-based	Subword-based
Target Word Processing: "Unhappiness" ["Unhappiness"] Description Large vocabulary, misses semantic relationships (e.g., Unicorns vs Unicorn).	Target Word Processing: "Unhappiness" ['U','n','h','a','p','p','i','n','e','s','s'] Description Tiny vocabulary, zero out-of-vocabulary (OOV) tokens, but massive sequence length and loss of meaning.	Target Word Processing: "Unhappiness" ["Un", "happiness"] Description The industry standard. Splits words into meaningful prefixes/suffixes.

Syntax Comparison:

- BERT's WordPiece uses ##ization (maximizing likelihood)
- XLNet's SentencePiece uses ▣IBM (vocabulary reduction)

The PyTorch Tokenization Pipeline

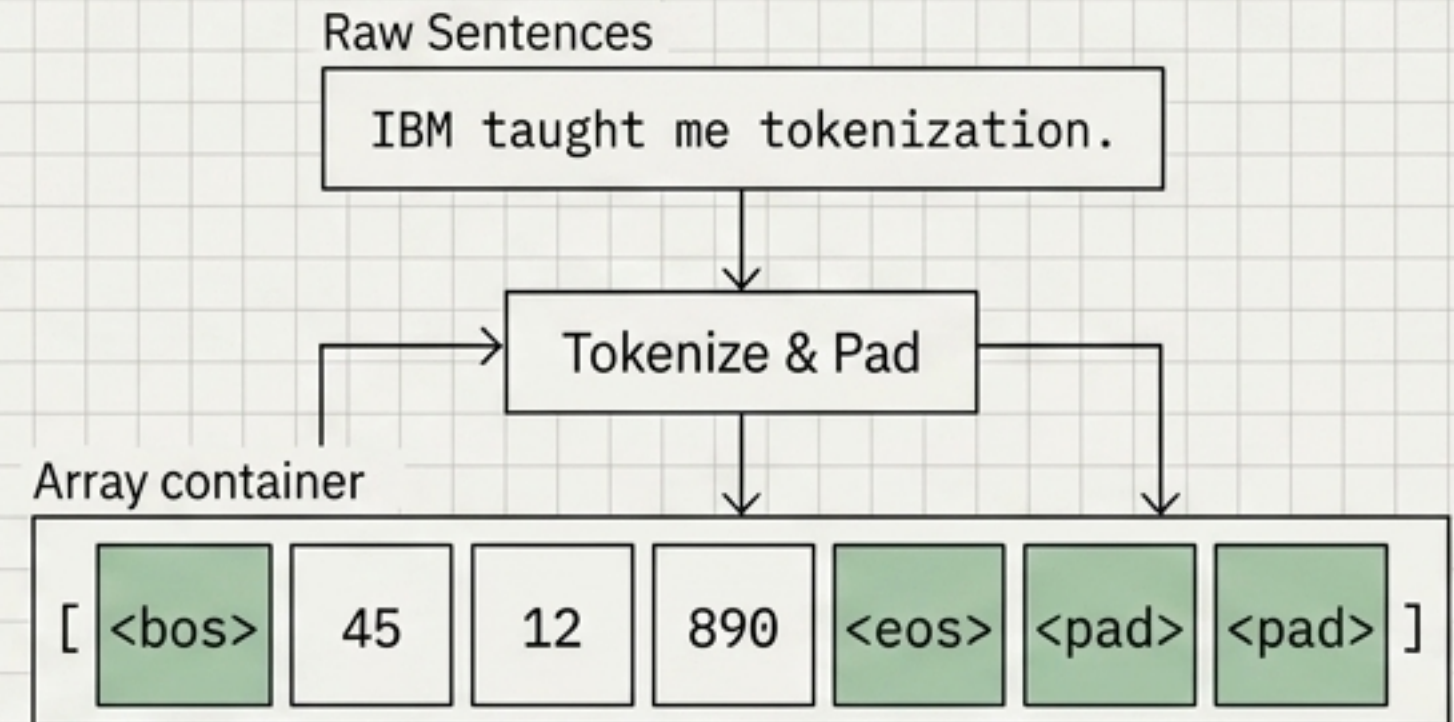
PYTORCH CODE

```
tokenizer = get_tokenizer("basic_english")

def yield_tokens(data_iter):
    for _, text in data_iter:
        yield tokenizer(text)

vocab = build_vocab_from_iterator(
    yield_tokens(dataset),
    specials=["<unk>"])
```

VISUAL CONCEPT DIAGRAM



DEFINITIONS BLOCK

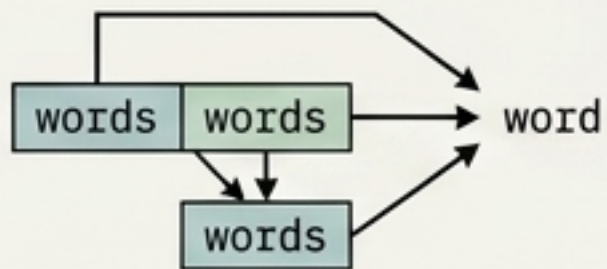
- <bos> (Index 2): Beginning of sequence.
- <eos> (Index 3): End of sequence.
- <pad> (Index 1): Padding to force uniform batch length.
- <unk> (Index 0): Out-of-vocabulary fallback.

The Evolution of Text Generation

The Memory Bottleneck: Sequential processing limits context retention.

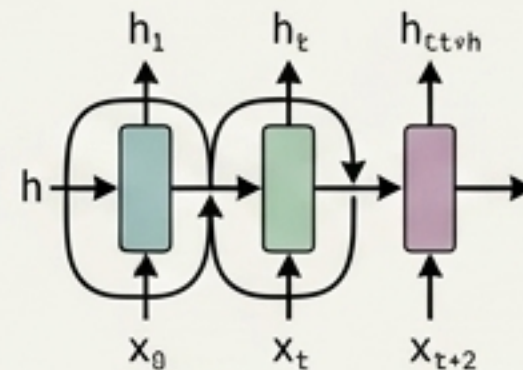
N-Grams (Bi/Tri-gram)

A rigid sliding window predicting only based on the immediate 1-2 previous words.



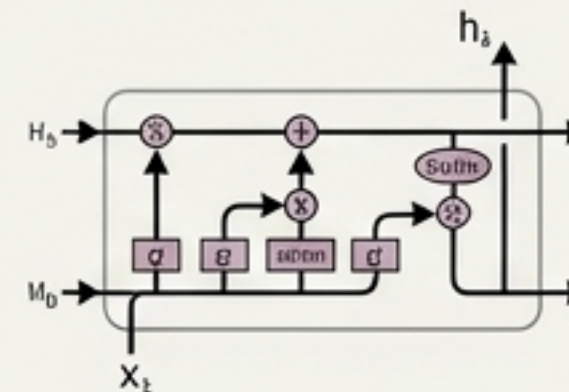
RNNs (Recurrent Neural Networks)

Processes sequential loops to maintain a "hidden state" memory, but suffers on long text.



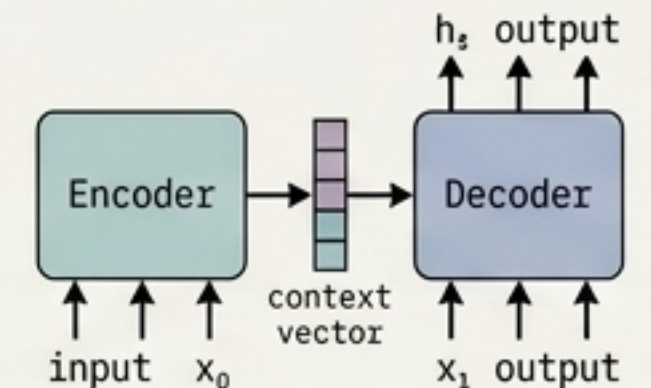
LSTMs & GRUs

Advanced RNNs capturing longer temporal dependencies, but fundamentally choked by sequential processing (one word at a time).

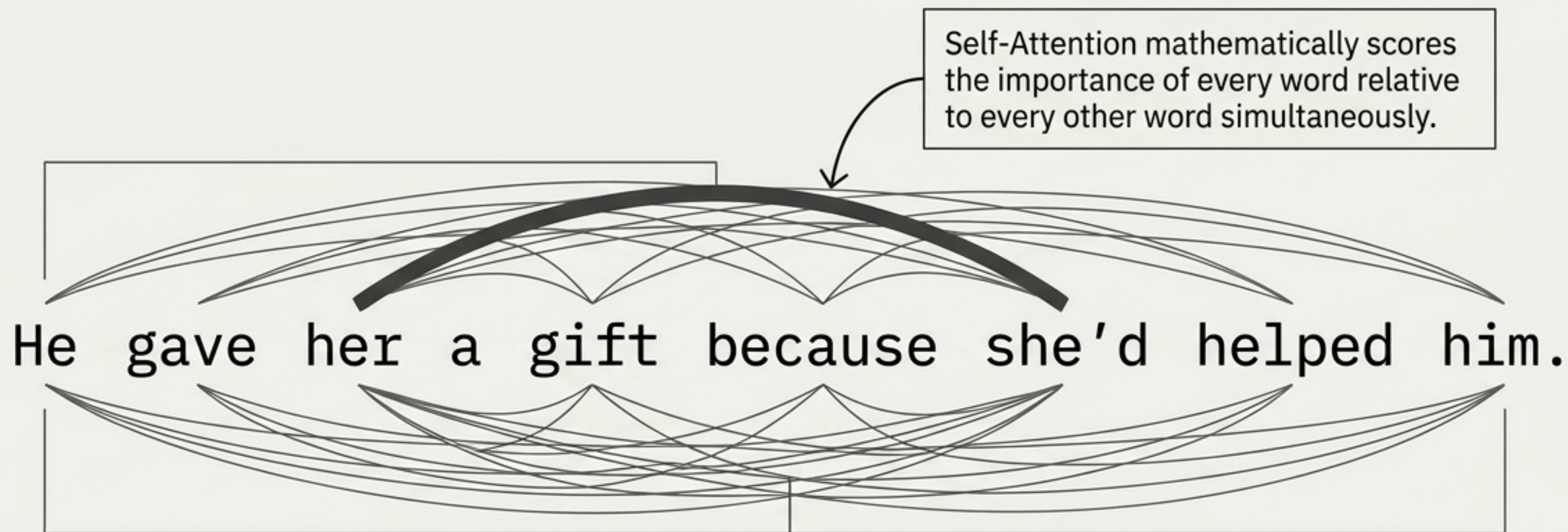


Seq2Seq

Encoder-Decoder setups for translation, setting the stage for attention.



The Transformer Breakthrough: Attention is All You Need



Parallel processing allows simultaneous computation of all relationships, overcoming the sequential bottleneck of RNNs.

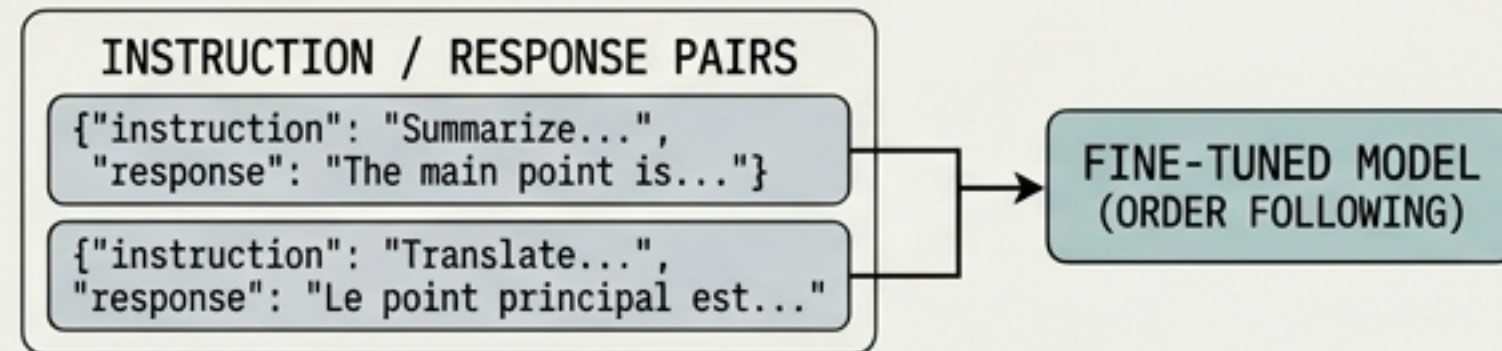
`nn.EmbeddingBag`

Maps tokens to a high-dimensional vector space.

The Paradigms of Advanced Fine-Tuning

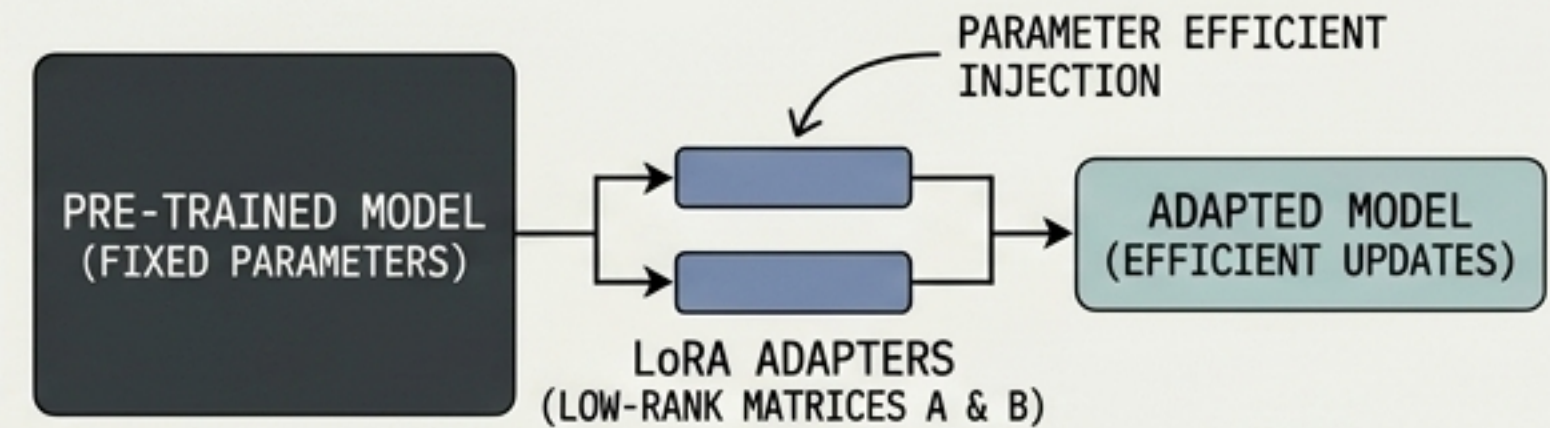
Instruction-Tuning

Feeding diverse instruction/response pairs so the model learns how to follow orders rather than just predict the next word.



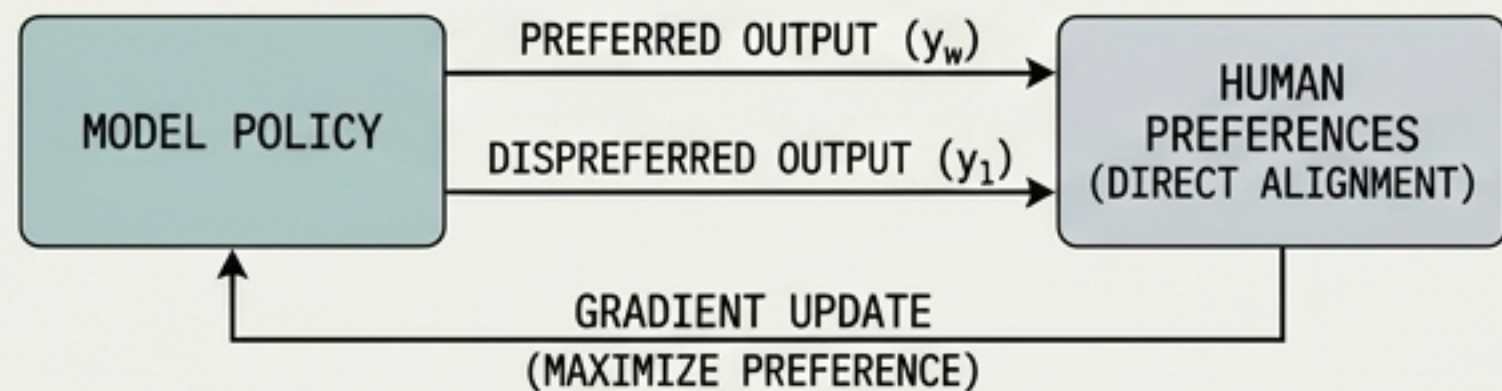
LoRA (Low-Rank Adaptation)

Highly efficient parameter reduction by injecting low-dimensional bottlenecks.



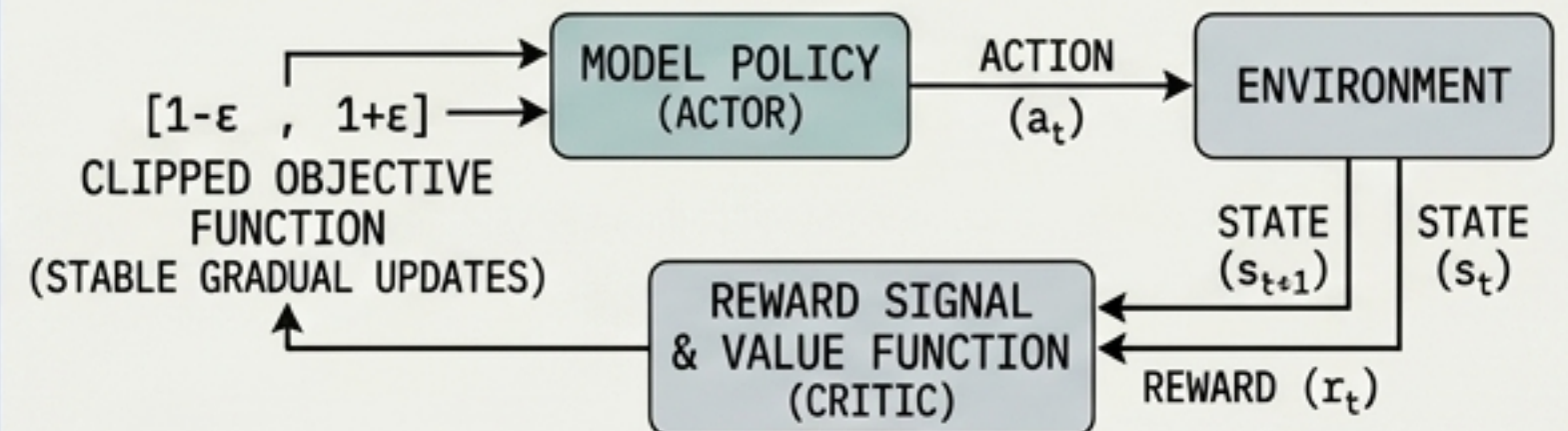
DPO (Direct Preference Optimization)

Aligning model outputs directly to human preferences without complex reward models.



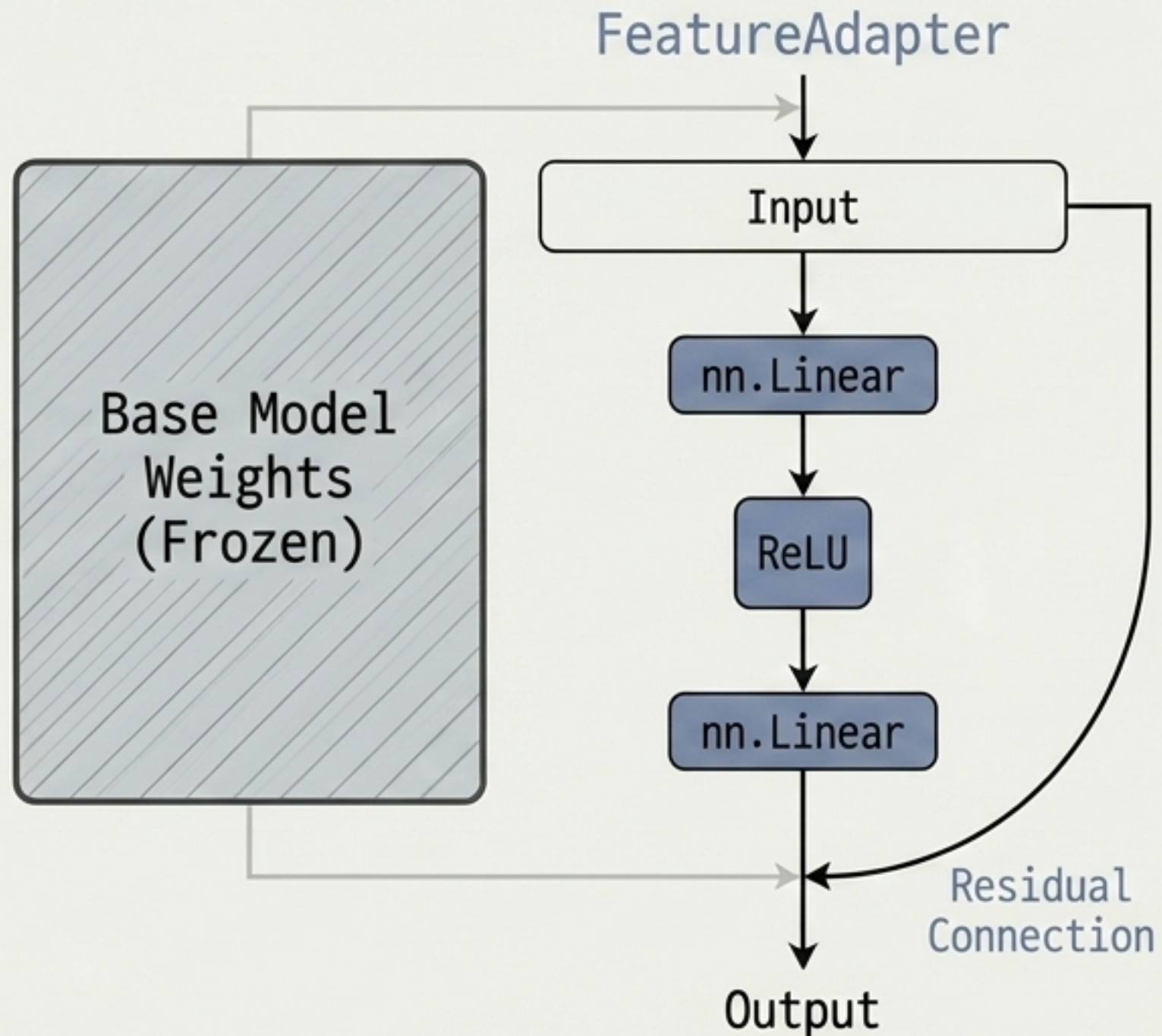
PPO (Proximal Policy Optimization)

Reinforcement learning utilizing a clipped objective function to ensure stable, gradual updates.



Deconstructing LoRA (Low-Rank Adaptation)

Diagram: FeatureAdapter Flow

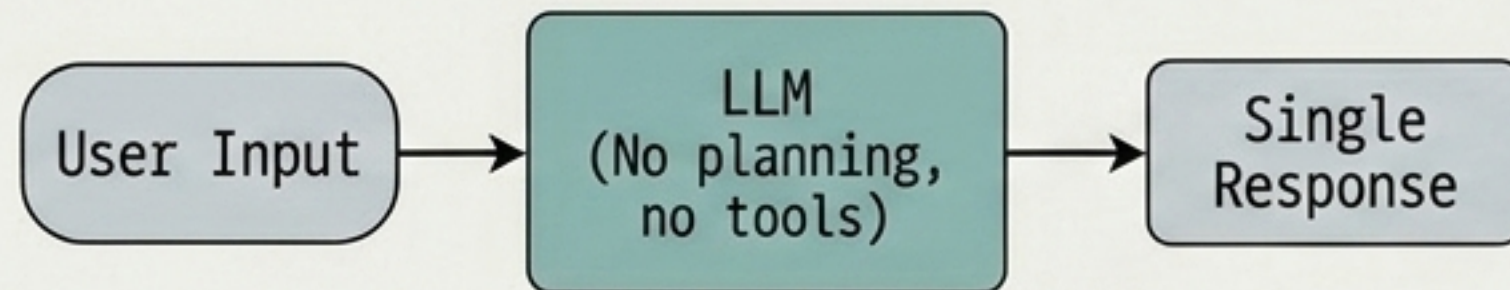


Code Container: PyTorch Implementation

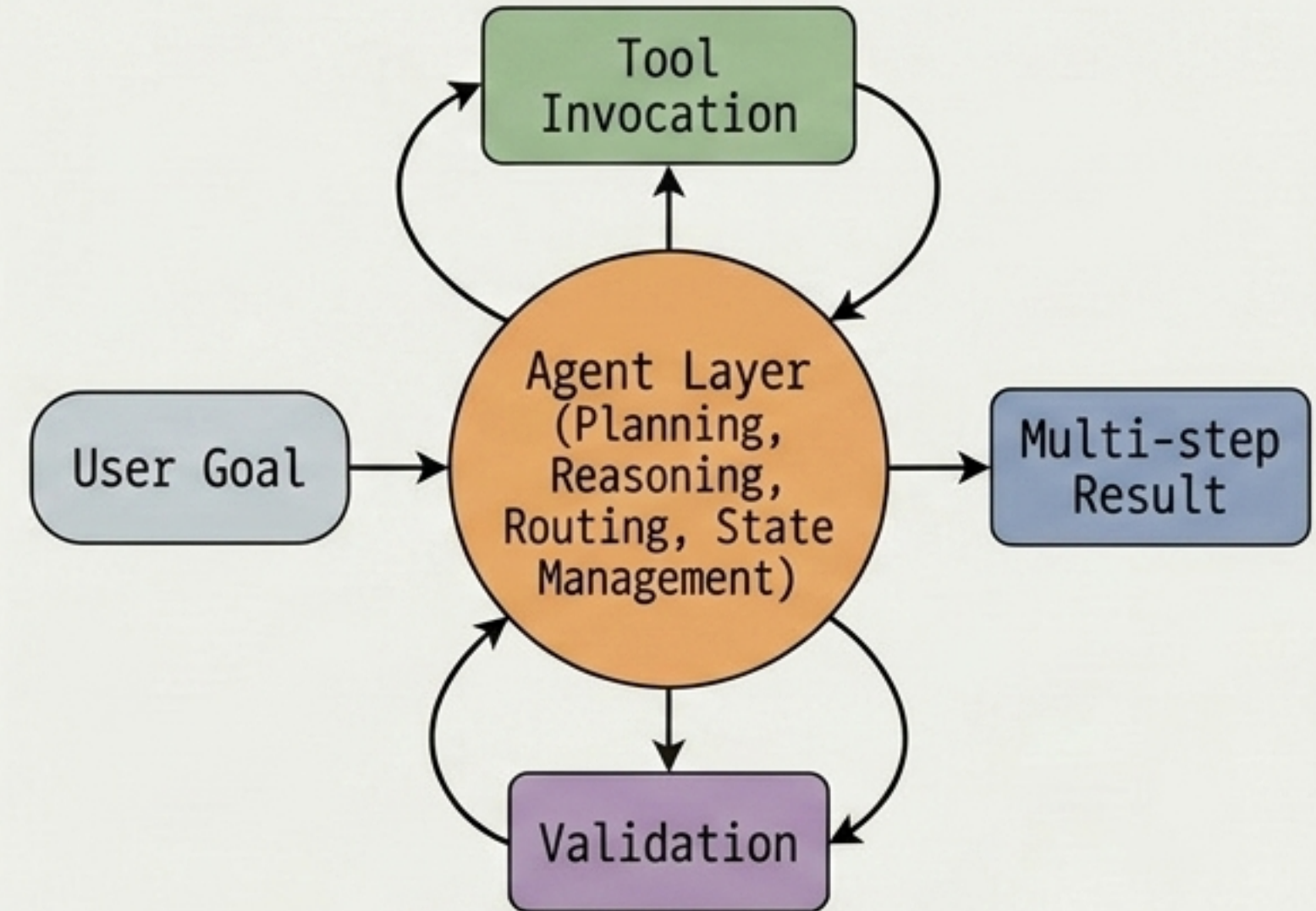
```
class LoRALayer(torch.nn.Module):
    def __init__(self, in_dim, out_dim, rank, alpha):
        self.A = torch.nn.Parameter(torch.randn(in_dim, rank))
        self.B = torch.nn.Parameter(torch.zeros(rank, out_dim))
        # base model set to requires_grad=False
```


Crossing the Threshold: Reactive vs. Agentic AI

Non-Agentic / Reactive

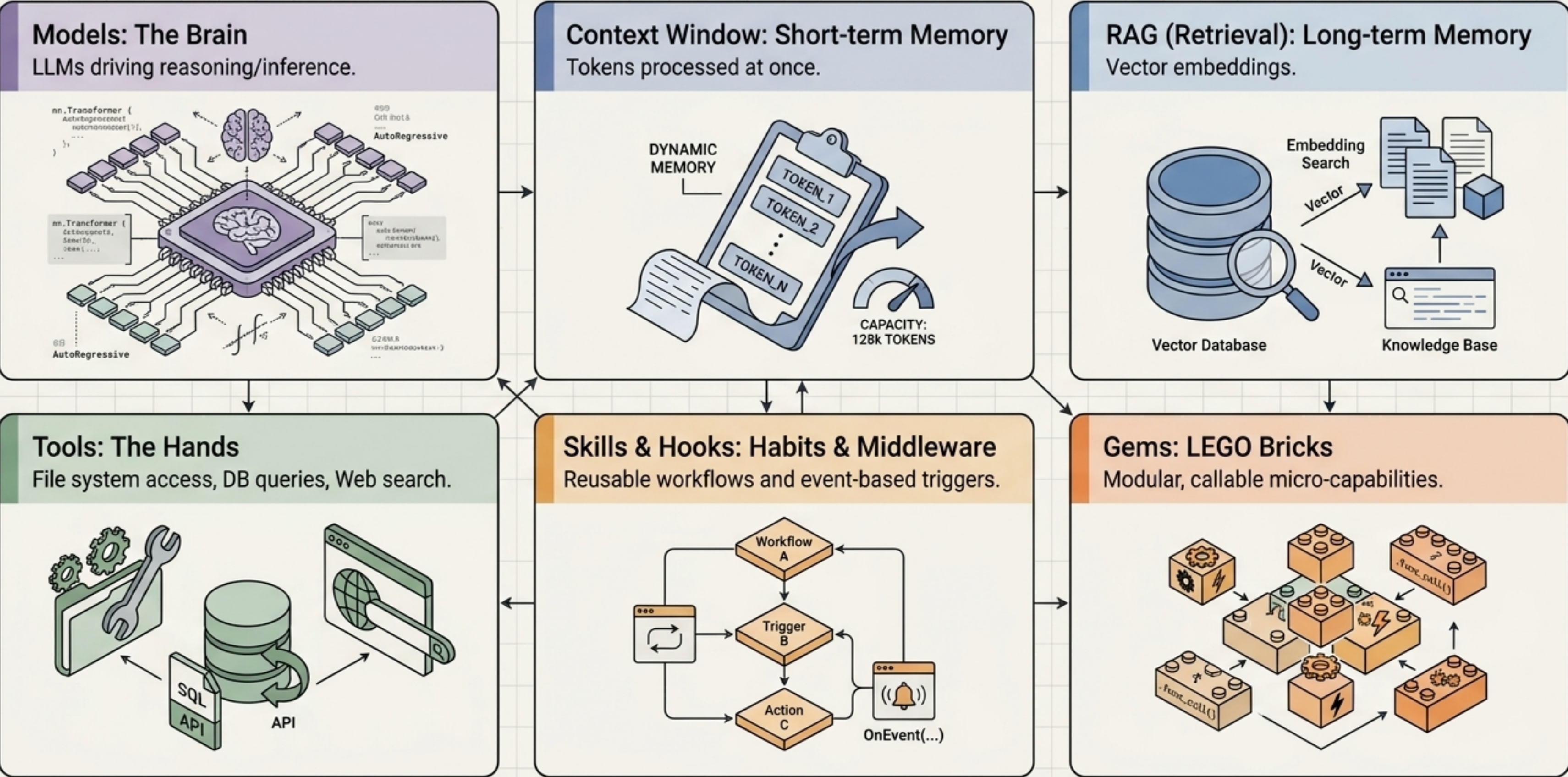


Agentic / Autonomous



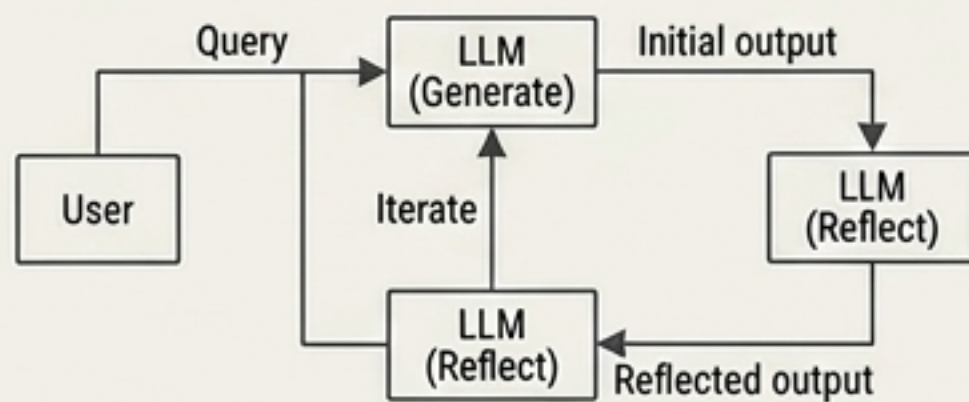
Agentic AI doesn't just respond—it acts. It decides what to do next, calls external functions, and pursues a goal autonomously.

The Anatomy of an AI Agent

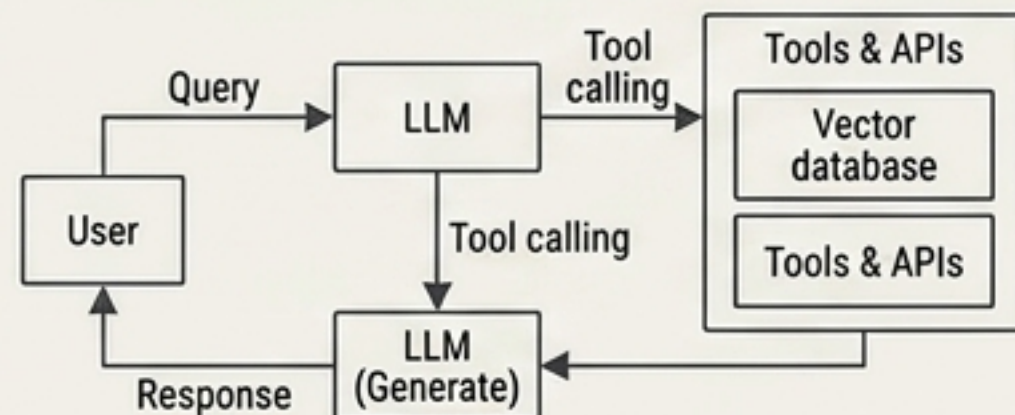


5 Essential Agentic Design Patterns

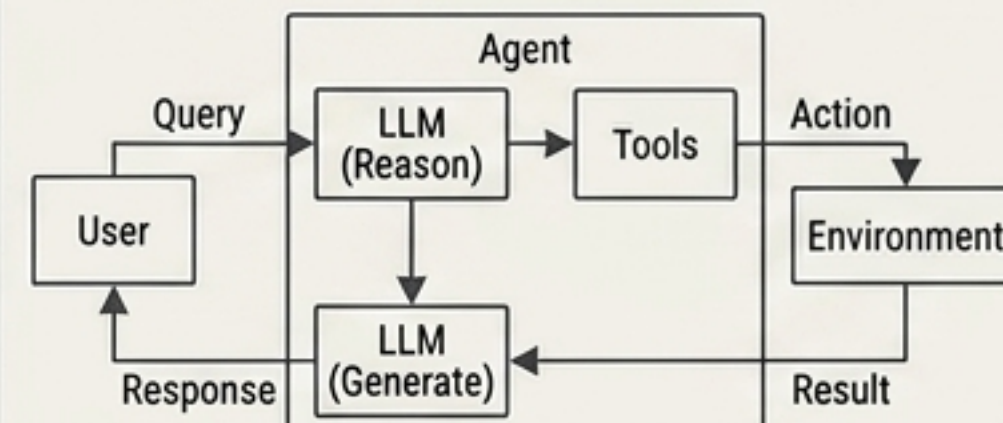
Reflection Pattern



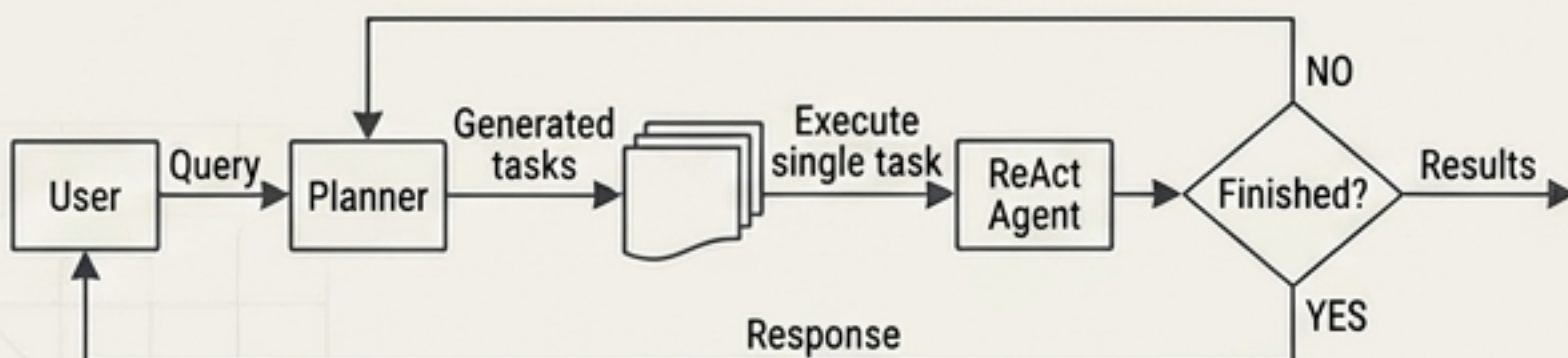
Tool Use Pattern



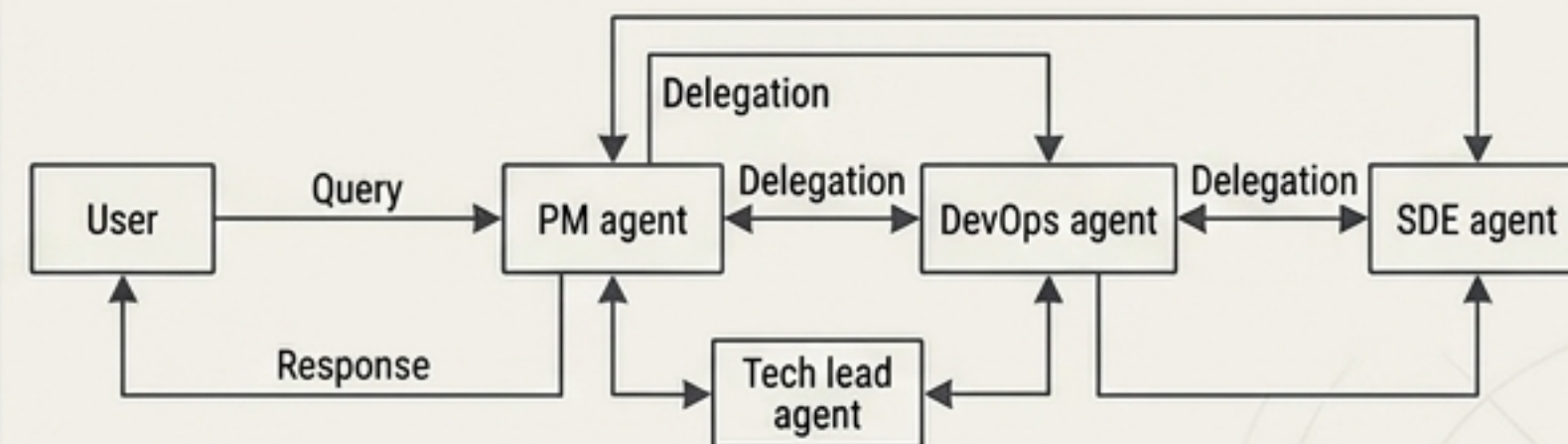
ReAct Pattern



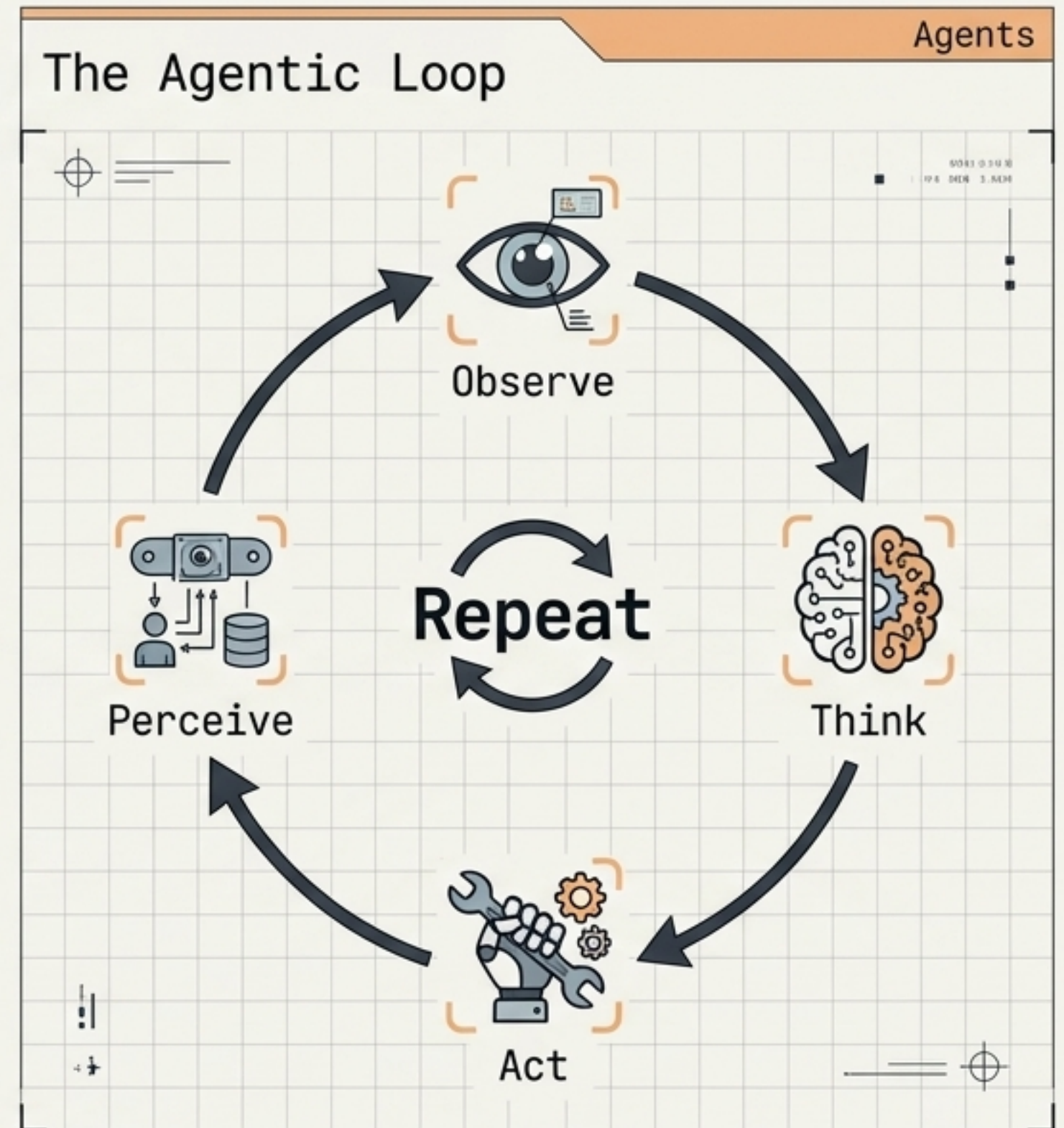
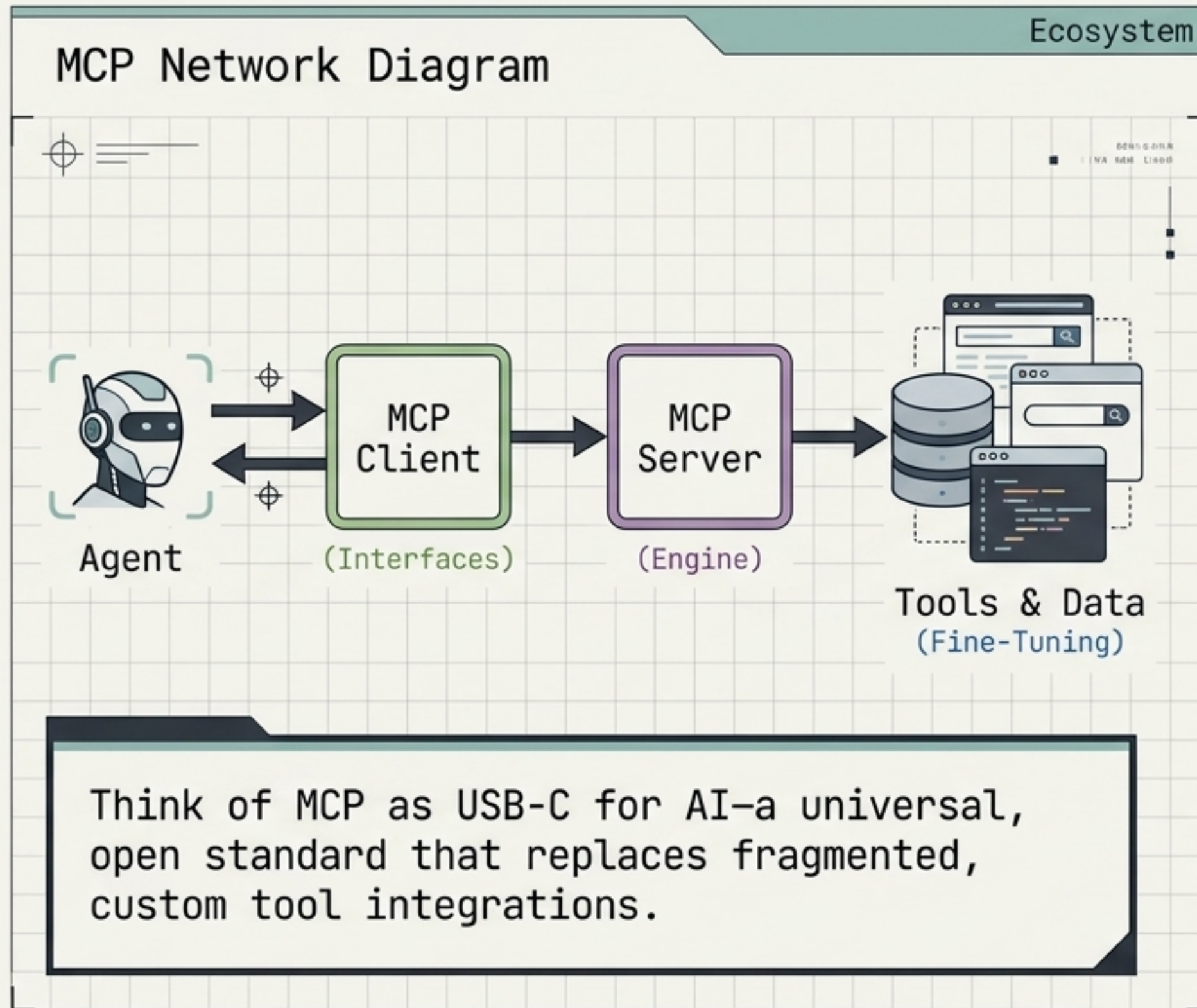
Planning Pattern



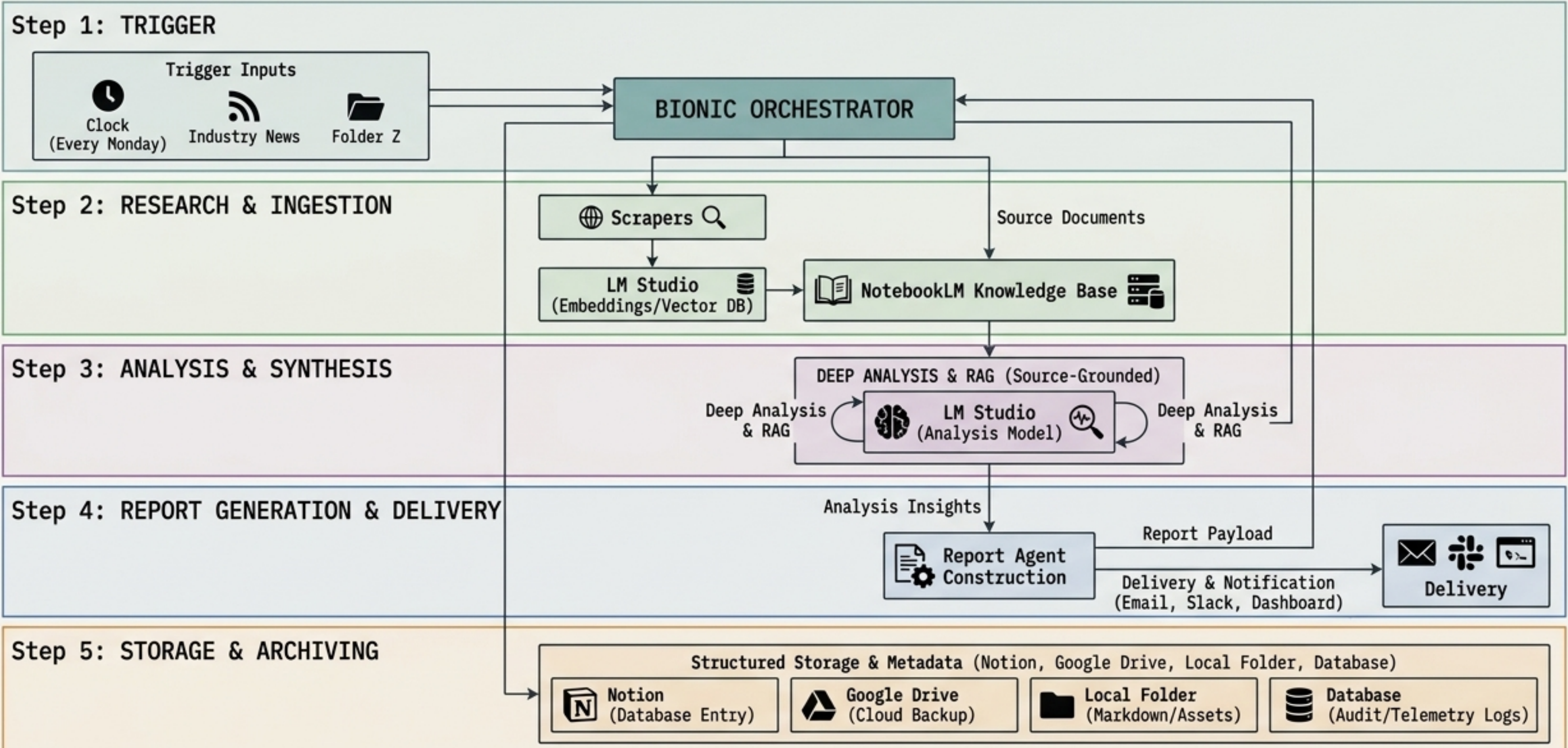
Multi-Agent Pattern



Universal Connections: Model Context Protocol (MCP)



Synthesis: The Fully Automated Agentic Workflow



THE BUILDER'S PLAYBOOK: 10 STEPS TO AGENTIC AI

COLUMN 1

[1] DEFINE ROLE & GOAL
What will it do? Who is it helping?

[2] DESIGN STRUCTURED I/O
Use Pydantic AI or JSON Schemas to define inputs and outputs.

[3] TUNE BEHAVIOR & ADD PROTOCOL
Utilize System prompts, MCP (Model Context Protocol), and Prompt Tuning.

[4] ADD REASONING & TOOL USE
Implement frameworks like ReAct (Reasoning + Action) and Chain-of-Thought.

[5] STRUCTURE MULTI-AGENT LOGIC
Employ orchestration tools such as CrewAI or LangGraph for coordination.

COLUMN 2

[6] ADD MEMORY AND LONG-TERM CONTEXT
Integrate Context Window and Vector RAG for historical awareness.

[7] ADD VOICE OR VISION CAPABILITIES
Let your agent see and speak with multimodal inputs and outputs.

[8] DELIVER THE OUTPUT
Format results into Markdown, PDF, or JSON for usability.

[9] WRAP IN A UI
Build front-end interfaces using Streamlit, Gradio, or FastAPI.

[10] EVALUATE AND MONITOR
Regularly run tests, and use logs and benchmarks to ensure reliability and performance.