

# MODULE 1 — Algorithmic Thinking for Agentic AI

Agentic systems plan, search, reason, and navigate tasks. This module gives you the core CS mental models behind those behaviors.

## Module 1.1 — Time & Space Complexity (Intuition Only)

### Concept

In agents, the cost of exploring options grows quickly. You don't need formal proofs — just intuition for what is:

- cheap (linear,  $O(n)$ )
- expensive (quadratic or worse)
- explosive (exponential branching) Agents often explore multiple possible actions. The number of possibilities grows like:

$O(b^d)$

Where:

- $b$  = branching factor (# of choices per step)
- $d$  = depth This becomes VERY expensive very quickly — crucial for agent planning.

### Example

```
# O(n) example
def find_target(seq):
    for x in seq:
        if x == "target":
            return True
    return False

# O(n^2) example
def all_pairs(seq):
    pairs = []
    for a in seq:
        for b in seq:
            pairs.append((a, b))
    return pairs
```

Q: When I'm collaborating with AI to generate data science solutions, what are things I should be aware of on my end to help keep decision-making more targeted based on what's needed for a given project? Or am I getting ahead of myself here for future topics in this module?

A: In short yes, new Module 2 covers this topic.

---

## 💡 Why this matters for agentic AI

Agents frequently:

- explore possible tool sequences
  - evaluate future steps
  - try reasoning paths (ReAct, ToT)
  - branch into multiple “ideas” Understanding cost helps you:
  - avoid unnecessary branching
  - prune search trees
  - design efficient planning loops
  - reason about feasibility of multi-step agents
- 

## 📝 Mini-Exercise 1.1

Generate all 2-step action sequences from:

```
actions = ["A", "B", "C"]
```

Expected output: A→A, A→B, A→C, B→A, ... (9 sequences total) Your task:

1. Generate the sequences using loops
  2. Print all sequences
  3. Print how many there are
- 

## Module 1.2 — Graphs & Trees (Core Data Structures for Agents)

### Concept

Agents represent tasks and reasoning using:

- Graphs → tasks & dependencies
  - Trees → reasoning steps, decision paths This is foundational for planning, tool selection, workflow routing, etc.
- 

## ❖ Example: A simple task graph (DAG)

```
task_graph = {
    "load_data": ["clean_data"],
    "clean_data": ["analyze"],
    "analyze": ["summarize"],
```

```
    "summarize": []  
}
```

A DAG (Directed Acyclic Graph) is common for pipelines and planning.

Q: Could you explain DAGs in a bit more detail? Along with what graphs & trees are?

## Appendix

- See Module 1, *Graph & Tree Terminology To Be Aware Of*

### Terminology

- **Node** - just a thing, depending on context node could represent
  - A task, step, decision, state, piece of data, thought, tool, file, computation, etc.
- **Edge** - a relationship between two nodes
  - Usually means: *comes before*, *depends on*, *leads to*, *can transition from*, *is connected to*, etc.
- **Graph** - a collection of nodes connected by edges

### Mental Model

- Think of a graph like a map
  - **Cities** = nodes
  - **Roads** = edges
- Map doesn't show you how to travel, just shows what's connected
  - Agent's job is to navigate the map

### Directed vs Undirected Graphs

- **Undirected graph** - edges go both ways
  - "A is connected to B", "B is connected to A"
  - Cases: social networks, similarity graphs, clustering relationships
- **Directed graph** - edges have direction
  - "A must happen before B", "A leads to B", "B depends on A"
  - Cases: workflows, pipelines, task planning, agents
- **Almost everything in data science pipelines and agents uses directed graphs**

### What Is a Tree?

- Special kind of graph with extra structure
  - Single root (starting point)
  - Every node (except for root) has exactly one parent
  - No cycles (can't loop back)
- Trees are perfect for
  - Reasoning paths
  - Decision trees
  - "What if" exploration
  - Tree-of-Thought
  - Recursive problem solving

## Directed Acyclic Graphs (DAGs)

- Directed graph with no cycles
  - Edges have direction
  - Can't loop back
  - Example: Load Data -> Clean Data -> Feature Engineering -> Modeling -> Evaluate
- Guarantee a valid execution order
- Prevent infinite loops
- Model real workflows
- **Most data science workflows are DAGs**
  - Airflow, Prefect, Luigi, Dagster, etc.

## Reasoning (Tree)

- Nodes = thoughts
- Edges = "leads to"

## Why Agents Use Both

- Graphs (DAGs) -> represent *what must be done*
- Trees -> represent *how to decide what to do next*

## Connecting Concepts Back to Module 0

- Nodes -> functions, modules, dataclasses, async tasks
- Edges -> structure, relationships, order, planning

## Why These Terms Matter Going Forward

- "*Agent Planning*" -> graph traversal
- "*Tool selection*" -> choosing edges
- "*Reasoning path*" -> tree branch
- "*Pruning*" -> cutting branches
- "*Execution order*" -> topological sort
- "*Parallelizable steps*" -> independent nodes

Q: In terms of order of operation for AI's approach to creating data science solutions, is code development typically approached bottom-up (e.g. build out tasks first) or top-down (define workflow first)?

## Appendix

- See Module 1, *Where Does Creating DAGs Fit with Development of Tasks In Terms of Order of Operation?*

A: AI systems (and good human–AI collaboration) work best with a top-down → bottom-up loop, not purely one or the other.

- DAGs sit **at the boundary between reasoning and implementation**, which is why you're feeling the connection between Module 1.2 and Module 0.

A: Issue with *pure bottom-up* (task-first only), without clear direction

- Don't know which tasks are actually needed

- May overbuild or misbuild tools
- Run the risk of not ending up with a coherent workflow when everything is connected at the end
- Hard to reason about order of execution
- Run the risk of AI hallucinating extra steps

A: Issue with *pure top-down* (workflow-first only), without tangible logic

- DAGs may be too abstract
- Discovered data quirks force changes later
- Tasks may not be implemented as defined
- AI may produce elegant but unrealistic pipelines

A: Most common approach is *top-down first, bottom-up refinement*

1. **Define the DAG (coarse DAG)** -> No code yet, but nodes & edges are clearly defined
  1. What are the major stages?
  2. What must happen before what?
  3. What can happen in parallel?
  4. What are the decision points?
2. **Implement nodes as tools** -> what was covered in Module 0
  1. Each node becomes -> a function, module, tool, etc.
3. **Feedback loop** -> as tasks are implemented bottom-up
  1. Missing dependencies are discovered
  2. Some nodes are too complex (need further splitting)
  3. Nodes could be merged together (too atomic, oversimplified)
  4. Some edges disappear

Q: How flexible is AI in iterating over previously generated (coded) data science solutions? So let's say in my work setting I need to follow an agile approach to developing a solution. That is, stakeholders expect progress updates from an "end-to-end" point of view, which in turn means iterations of work are represented by MVPs. How capable is AI in refactoring code in this way? Does AI work better in a waterfall setting (build solution one step at a time) or is AI able to handle this code development approach you're describing (e.g., take this hybrid top-down planning + bottom-up implementation + feedback loop for refinement) in an agile setting as well?

## Appendix

- See Module 1, *Does AI Code Generation Work Better In A Waterfall Setting or An Agile Development Setting?*

A: AI works better in an agile, iterative setting than it does in a strict waterfall setting if collaboration is structured correctly.

- AI is very capable of refactoring, but needs anchors to do so safely
- What AI refactors very well when
  - code is modular
  - responsibilities are clearly separated
  - inputs/outputs are typed or structured
  - constraints restated explicitly
  - changes are incremental & scoped -> e.g., what is getting refactored is clearly defined

- Cases where AI may rewrite instead of refactor
  - code is monolithic
  - logic & I/O are tangled
  - there's no explicit "contract" for components
  - goals are restated vaguely ("make it better")
  - earlier directions are preserved intentionally
- What works extremely well
  - Iteration 0 -> skeleton MVP -> top-down, end-to-end
  - Iteration 1 -> harden one stage -> example: refine only the data ingestion step by ... everything else stays the same
  - Etc.
- AI prefers agile (under the hood) because
  - AI models internally reason in incremental deltas, not full rebuilds

A: Rules to follow to keep AI refactoring aligned across sprints

1. Preserve contracts explicitly -> AI respects well
  1. "Do not change the public interface of `run_agent()`."
  2. "This Pydantic model is a stable contract."
2. Scope changes aggressively -> removes room for misinterpretation, hallucination
  1. "Refactor only the feature engineering module to reduce runtime."
3. Restate the current architecture -> re-anchors the model
  1. "Here is the current DAG and module structure. We are modifying only X."
4. Ask for *diff-style* changes conceptually -> forces controlled refactoring
  1. "What changes, what stays the same, and why?"
  2. E.g., treat refactoring with AI the same way a strong engineer would propose a change in a sprint planning or design review — explain the plan and reasoning before writing code.

Takeaways

- Think of DAGs in the same way a network modeling, tasks are nodes, orchestration are edges, directional with no cycling back
- Best approach to collaborating with AI is to take hybrid approach (design top-down, implement bottom-up, refine in feedback loop) -> left unguided AI tends to take bottom-up approach
- AI works best in an agile project management setting (as opposed to waterfall), but only under the right set of controls

## 💡 Why this matters

Agents use graphs for:

- task sequencing
- dependency resolution
- planning
- workflow orchestration
- tracing reasoning A plan is just a path in a graph. A thought process is just a tree.

## 📝 Mini-Exercise 1.2

Represent this structure as a Python dictionary graph:

```
Prepare Dataset
  └── Load CSV
    └── Validate Schema
  └── Clean Missing Values
  └── Compute Features
```

Your graph must map each task → list of child tasks.

Note: just for future reference, every layer in dictionary graph is a node, meaning *Prepare Dataset* is not a title to the structure it's the root node. That & everything seen should be encapsulated in a dictionary called *task\_graph*.

---

## Module 1.3 — BFS & DFS (Search for Agent Reasoning)

### Concept

Agents frequently perform search:

- exploring possible reasoning paths (Tree-of-Thoughts)
  - searching for valid tool sequences
  - evaluating next-step actions
  - trying different solution paths
- The two most common strategies:
- 

### DFS (Depth First Search)

Explore one path deeply, then backtrack. Good for:

- reasoning chains
- recursive strategies
- exploring hypotheses

*"Follow one path as far as possible, then backtrack."*

---

### BFS (Breadth First Search)

Explore all nodes at depth 1, then depth 2... Good for:

- finding shortest plans
- enumerating possible next actions
- structured multi-step searches

*"Show me everything I can do next, before going deeper."*

---

### Example: DFS traversal

```
def dfs(graph, start):
    stack = [start]
    visited = set()

    while stack:
        node = stack.pop()
        if node not in visited:
            print("Visiting:", node)
            visited.add(node)
            stack.extend(graph[node])
```

Note: in this case the while loop is only every 1 element long, the graph dictionary contains all the subsequent steps, next step being tacked on at the end of a given iteration.

Q: In what situations does it make sense to use one approach over the other?

## Appendix

- Under Module 1, see *DFS vs BFS, When Does It Make Sense to Use One Over The Other?*

A: BFS is used when "distance" or "shortest path" matters

- Goes level by level, so naturally finds
  - Shortest number of steps -> the minimal plan
- Application -> workflow planning / orchestration
  - Task schedulers (Airflow, Prefect)
  - Pipeline validation
  - Dependency resolution
- Helps identify **all immediate next steps** -> with respect to current location
- Tradeoffs
  - Uses more memory
  - Explores many shallow options
  - Can explode quickly with high branching factor

A: DFS is used when "depth", "completeness", or "exploration" matters

- Need to fully explore one option -> partial solutions are not useful
- Application -> reasoning & problem-solving agents
  - Tree-of-Thought and recursive reasoning models
    - *"Let me fully explore this line of reasoning before trying another."*
- Similar to hypothesis testing
- Tradeoffs
  - Can get "lost" in deep paths
  - May miss shallow solutions
  - Order-dependent
  - Not guaranteed shortest path

Q: Is it fair to assume planning development relies of BFS while implementation development relies on DFS?

## Appendix

- Under Module 1, See *Is It Fair To Assume Planning Decisions Rely on BFS While Implementation Decisions Rely on DFS?*

A: **BFS is best suited for high-level planning and orchestration decisions, while DFS is best suited for deep, focused exploration within a chosen plan.**

- Note: BFS explores **all options at the same depth** so it gives **global view** of what's possible
- BFS is idea for
  - Valid next steps
  - Parallelized tasks
  - Shortest paths
  - Minimal plans
- DFS is ideal for
  - implementing chosen task
  - exploring data-cleaning strategies
  - refining feature engineering
  - tuning models
  - debugging
  - hypothesis testing

### ✍ A Concrete Example (End-to-End)

#### Planning (BFS)

An agent asks:

- "What's the minimal valid pipeline?"
- "What steps can run in parallel?"
- "What dependencies exist?" → BFS over the DAG

#### Execution (DFS)

For clean\_missing\_values:

- Try strategy A
- If fails → backtrack
- Try strategy B
- Refine
- Validate → DFS within a node

---

### ⚠ Important Refinement (One Subtle Nuance)

The only refinement I'll add is this: DFS should almost always be bounded or guided. Unbounded DFS does get lost. That's why real systems use:

- depth limits
- heuristics
- scoring functions

- early stopping
  - memoization (Module 1.5) But conceptually, your split still holds.
- 

## 🧠 One-Sentence Mental Model (Keep This)

Use BFS to decide what path to take and DFS to figure out how to walk that path. That single sentence will continue to be true as we move into:

- agent orchestration
  - tool calling
  - reflection loops
  - planning agents
  - multi-agent systems
- 

## 💡 Why this matters for agentic AI

- Tree-of-Thought = DFS with pruning
  - Many planners use BFS for breadth exploration
  - Multi-step tool use often requires DFS
  - Tool-selection trees rely on these algorithms
  - Graph-based workflow engines (Prefect, Airflow) use BFS-like reasoning
- 

## 📝 Mini-Exercise 1.3

Given:

```
graph = {
    "A": ["B", "C"],
    "B": ["D"],
    "C": ["E"],
    "D": [],
    "E": []
}
```

Write:

- `bfs(graph, "A")`
  - `dfs(graph, "A")` Each should print nodes in the order they're visited.
- 

## Module 1.4 — Recursion (Subproblem Decomposition for Agents)

### Concept

Agents often break problems into smaller problems:

- hierarchical planning
- recursive reasoning loops
- refining goals
- dividing tasks Recursion is how this “step-by-step refinement” works.

## Terminology

- **base case** - a situation where the answer is known immediately and no further recursion is needed.
  - Think of as: **stopping conditions, smallest solvable subproblems, anchors**

❖ Example: Print all nodes in a tree recursively

```
def walk(tree):
    value, children = tree
    print(value)
    for child in children:
        walk(child)
```

Q: What does *subproblem decomposition* mean?

## Appendix

- Under Module 1, see *What Is Subproblem Decomposition & What Are Other Common Uses of Recursion In AI-Driven Solutions Beyond BFS & DFS?*

A: **Subproblem decomposition means breaking a large goal into smaller, solvable pieces that can be handled independently or sequentially. -> Structured problem-solving.**

- Note: **subproblem decomposition does NOT require recursion** — recursion is just a natural way to express it.
- For agents, typically looks like
  - Start with a goal
  - Break it into subtasks
  - Possibly break those subtasks into even smaller tasks
  - Solve each piece
  - Combine the results

A: Generally speaking, recursion is most helpful when

- A problem is naturally hierarchical
- Each subproblem looks like a smaller version of the same problem
- The depth is unknown or variable
- Examples
  - Trees (reasoning trees, task trees)
  - Nested structures
  - Backtracking problems
  - Divide-and-conquer logic

A: Recursion is not ideal when

- Depth is very large (stack limits)
- State needs to be shared explicitly
- Execution must be paused/resumed
- workflows are event-driven

A concrete example (no code yet)

## High-level goal

"Prepare dataset for modeling"

## Decomposed into subproblems

- Load data
- Validate schema
- Clean missing values
- Compute features

Each of those can be decomposed further:

- Clean missing values
  - Identify missing columns
  - Decide strategy (drop vs impute)
  - Apply strategy
  - Validate result That's subproblem decomposition.

## One-Sentence Takeaway

Subproblem decomposition for agents means breaking goals into smaller tasks that can be reasoned about, solved, and recombined — recursion is just one (very natural) way to express that process in code.

---

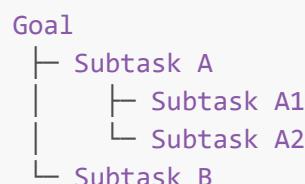
## How Agents Use Subproblem Decomposition (Beyond BFS/DFS)

BFS and DFS are mechanisms. Subproblem decomposition is the strategy. Here are common agent-level uses.

---

### ◊ 1. Hierarchical Task Planning (HTN-style thinking)

Agents often reason like this:



This hierarchy:

- is a **tree**
  - represents decomposition, not just traversal
  - lets agents focus on one level at a time This is widely used in:
    - robotics
    - workflow agents
    - planning systems
    - multi-step reasoning (Tree-of-Thought)
- 

## ◊ 2. Reasoning decomposition (LLM agents)

**When an agent says:** "Let me think step by step..."

What it's actually doing is:

- decomposing a reasoning task
- solving smaller logical steps
- composing a final answer

**Tree-of-Thought is literally:**

- subproblem decomposition
  - represented as a tree
  - explored via DFS-like reasoning
- 

## ◊ 3. Tool decomposition

Agents often turn one request into many tool calls:

"Analyze dataset" → load data → profile schema → compute stats → generate plots

Each tool call is a subproblem.

This is where:

- DAGs define structure
  - DFS handles execution
  - BFS handles orchestration
- 

## ◊ 4. Adaptive replanning

If a subproblem fails:

- agent revises the plan
  - decomposes the task differently
  - retries with a new strategy This is **not** static traversal — it's dynamic decomposition.
-

## 💡 Why this matters

Agents dynamically:

- decompose tasks
  - reflect and retry
  - build multi-step plans
  - expand nodes in a reasoning tree All of these map naturally to recursive structures.
- 

## 📝 Mini-Exercise 1.4

Define a reasoning tree:

```
("root", [
    ("idea1", []),
    ("idea2", [
        ("idea2a", [])
    ])
])
```

Write a recursive function that prints all nodes.

---

## Module 1.5 — Dynamic Programming (Memorization for Agents)

### Concept

Dynamic programming is not just for interviews.

Terminology

- **Memoization** - a programming optimization technique where you store (cache) the results of expensive function calls and return the cached result when the same inputs occur again.

Agents use memoization to:

- avoid recomputing tool results
- remember partial reasoning steps
- prune search trees
- cache expensive computations

The key idea:

**Store results of expensive calls so you never compute them twice.**

---

① What “Dynamic Programming (DP) for Agents” Really Means

At a high level: **Dynamic programming for agents means remembering the results of expensive reasoning or computation so the agent doesn't repeat itself.**

That's it.

For agents, DP is less about math and more about:

- **state reuse**
- **decision reuse**
- **result reuse**

In other words:

"If I've already figured this out once, don't make me think about it again."

---

## ② Why Agents Need Caching (More Than Traditional Code)

Agents differ from normal programs because they:

- explore many possible paths
- revisit similar states
- reason recursively
- branch and backtrack
- retry after failures
- operate over DAGs and trees

Without caching:

- DFS explodes
- reasoning loops repeat
- costs scale exponentially
- latency becomes unacceptable

Dynamic programming is what makes **deep reasoning feasible**.

---

## ③ What "Caching" Means in Agentic Systems

Caching is simply:

A mapping from "problem state" → "known result"

In code terms:

```
cache[state] = result
```

But the definition of "state" is the key design decision.

---

### 3] What “Caching” Means in Agentic Systems

Caching is simply: A mapping from “problem state” → “known result” In code terms:

```
cache[state] = result
```

But the definition of “state” is the key design decision.

### 4] What Gets Cached in AI-Driven Data Science Systems

Here are the most common and valuable things to cache, ordered roughly by impact.

---

#### ◊ 1. Tool Outputs (Very Common)

If an agent calls a tool like:

- load data
- compute summary stats
- generate embeddings
- profile schema
- fetch API results ...and those inputs haven't changed — **cache the result**.

Example:

```
(load_csv, dataset_path) → rows
```

Why:

- tools are often expensive
  - I/O dominates runtime
  - repeated calls are wasteful
- 

#### ◊ 2. Intermediate Pipeline Results

In data science workflows:

- cleaned datasets
- transformed features
- filtered subsets
- preprocessed matrices

These are perfect cache targets.

Example:

```
(clean_missing_values, raw_dataset_hash) → cleaned_dataset
```

This is how:

- feature stores work
  - ML pipelines scale
  - experiments remain reproducible
- 

### ◊ 3. Reasoning Subresults (Agent Thinking)

Agents often ask:

- “Is this schema valid?”
- “Is this transformation acceptable?”
- “Have I already evaluated this option?”

If the **same question** arises again:

- reuse the answer
- skip recomputation

This is especially important in:

- Tree-of-Thought
  - backtracking agents
  - reflection loops
- 

### ◊ 4. Planning Decisions

In planning:

- “Given this DAG state, what tasks are runnable?”
- “Given these constraints, what’s the next step?”

If the **graph hasn’t changed**, cache the decision. This is DP applied to planning, not computation.

---

### ◊ 5. Evaluation Results

Common in:

- model evaluation
- cross-validation
- simulation loops

If you evaluate:

```
(model X, dataset Y, metric Z)
```

Cache the score. This prevents recomputing identical evaluations.

### ❖ Example: Memoized Fibonacci

```
memo = {}

def fib(n):
    if n in memo:
        return memo[n]
    if n <= 1:
        return n
    memo[n] = fib(n-1) + fib(n-2)
    return memo[n]
```

## Takeaways

- Top-level, cache things that are expensive to run & that don't change (data cleansing, feature engineering, etc.)
- Caching provides mapping between reoccurring "problem states" and their affiliated resolve (don't solve the same problem twice!)
- **Don't cache results that depend on** -> randomness, timing, external state, change across runs
  - Examples: *stochastic model training, live API calls without versioning, streaming data, time-dependent queries*
  - E.g., think of BLS labor data, corrections are made to former publications through the year (state of data can change between queries)
  - **Unless you freeze the randomness or snapshot inputs, caching is unsafe.**

## How Caching Changes DFS (Why Module 1.5 Exists)

Without caching:

- DFS recomputes the same subtrees
- exponential blow-up happens fast

With caching:

- DFS becomes almost linear
- repeated subtrees collapse into one computation
- reasoning becomes scalable

This is why DP and DFS are paired conceptually.

In agent terms:

## DFS explores; DP remembers.

---

### A Mental Model That Works Well

Think of an agent as having **short-term memory**:

- "I've already seen this"
- "I already tried that"
- "I know the answer here"

Dynamic programming is just formalizing that memory.

---

### How This Ties Back to Module 0 (Important)

Your Module 0 work laid the groundwork for safe caching:

- **dataclasses** → stable keys
- **Pydantic models** → structured states
- **modular functions** → cacheable units
- **pure functions** → safe reuse
- **async tasks** → cache-friendly concurrency

You've already been preparing for this.

---

### One-Sentence Takeaway

**Dynamic programming for agents means caching the results of expensive, repeatable reasoning or computation so the agent can think deeply without repeating itself.**

---

### 💡 Why this matters for agentic AI

Agents often:

- revisit similar states
- recompute similar thought paths
- repeatedly call tools on similar inputs
- explore graphs with overlapping subproblems

Memoization massively reduces wasted compute.

---

### 📝 Mini-Exercise 1.5

Write a memoized function:

```
count_paths(n)
```

---

Where you can climb **1 or 2 steps**. E.g.,

- $n=1 \rightarrow 1$
- $n=2 \rightarrow 2$
- $n=3 \rightarrow 3$
- $n=4 \rightarrow 5$

Use a dictionary to memoize results. (No brute-force recursion.)

---

## After You Finish All Mini-Exercises:

We will complete the Module 1 Capstone Project:

**Build a Planning Agent that takes a task DAG and produces a valid execution plan using BFS, DFS, or topological logic.**

It will:

- read a task graph
- search for valid task orderings
- validate the plan with a Pydantic model
- simulate agent decision steps
- output a structured reasoning trace

Just like Module 0, you'll build a real agent-like system at the end.

---

Perfect — that's exactly what we want: clean plan printout, clean execution, and cache hits with input-aware keys.  Now let's do the **Module 1 Post-Capstone Reflection** in a way you can reuse later as your agents get more complex.

---

## Module 1 Post-Capstone Reflection

### 1) What is the "agent" actually responsible for?

In this capstone, your agent layer (now split across agent.py + planner.py + executor.py) owns the control plane:

- deciding *what to do* (planning)
- deciding *in what order* (traversal / dependency logic)
- deciding *how to avoid repeated work* (caching/memoization)
- coordinating tools (task functions)

The "work" itself lives in tasks.py (tool layer).

**This is the core agent architecture pattern you'll keep reusing.**

---

### 2) What counts as "state" in this project?

Your state is the artifact that moves through the pipeline (a dict in this capstone):

- initially: `{"dataset_path": "data.csv"}`
- after load: adds rows/cols/missing values
- after cleaning: modifies rows/missing values
- after features: adds feature list

In real AI-for-data-science systems, “state” is often:

- dataset reference (path/table/version)
- schema fingerprint
- feature spec
- model config / hyper params
- metrics so far
- “what has already been attempted”

State is what you use to:

- choose next actions
  - decide what is safe to reuse
  - explain results to stakeholders
- 

### 3) Planner vs Executor (why splitting them was worth it)

You now have a clean separation:

`planner.py` (policy-ish)

- BFS traversal gives a reachable plan
- in more advanced versions, planner might:
  - score options
  - choose among branches
  - incorporate constraints
  - select tools dynamically

`executor.py` (mechanism-ish)

- DFS traversal “runs the pipeline”
- caching lives here because repetition happens here
- later it grows to include:
  - retries
  - timeouts
  - parallel execution
  - logging and telemetry
  - partial re-runs

`agent.py` (orchestration glue)

- wires planner + executor together

- becomes your stable “public interface” This is exactly the refactor you wanted for “recognizable components” later.
- 

#### 4) What you cached — and why it was “dynamic programming”

Your caching strategy was: `cache key = task_name + dataset_path signature`

That’s DP in agent form:

- recursive/DFS execution creates repeated subproblems
- memoization makes each subproblem solved once
- repeated runs become near-instant

**Why this is realistic:** In real pipelines you rarely cache “everything for all time.” You cache outputs that are:

- expensive
  - deterministic given inputs
  - reusable across retries/sprints
- 

#### 5) What you should log for stakeholder confidence (lightweight, but powerful)

If you add just three logging concepts, your agent becomes demo-ready:

1. **Plan**
  - “These are the steps I intend to run.”
2. **Execution trace**
  - “These steps ran successfully.”
  - “These steps were skipped because cached.”
3. **Provenance / cache key**
  - “This result corresponds to dataset X/version Y.”

You already have #1 and #2 implicitly:

- plan list printed
- cache hits printed
- results printed

Later we’ll formalize this into structured run reports (Module 3 and beyond).

---

#### 6) The biggest “Module 1” concept you just internalized

You built the classic agent loop:

- **Represent the world as a graph**
- **Plan broadly (BFS intuition)**
- **Execute deeply (DFS intuition)**
- **Avoid repeated work (DP/memoization)**

That is the practical essence of Module 1.

