AI Automation / Task Orchestration

Map task dependencies to identify what must happen before what — prevents race conditions and deadlocks.
Difficulty: Intermediate
Model: GPT-4 / Claude / Gemini
Use Case: Workflow Design, Dependency Management
Updated: May 2026
Why This Prompt Exists
Most workflow failures aren't from task failures — they're from tasks running in the wrong order. A task runs before its data is ready, or two tasks conflict because they ran simultaneously.

You get:

- data races: Task B reads data before Task A finishes writing it
- deadlocks: Task A waits for Task B, Task B waits for Task A
- wasted work: Task C runs even though Task A failed (output unusable)
- slow workflows: tasks run sequentially when they could run in parallel
- impossible debugging: "it works sometimes" — depends on timing

But dependencies can be mapped:

- hard dependencies: Task B literally needs Task A's output
- soft dependencies: Task B should run after Task A (but could run without)
- data dependencies: Task B needs a field that Task A creates
- resource dependencies: Both need the same API (rate limit)
- ordering constraints: Business rules require sequence

Without dependency mapping, workflows break unpredictably.

This prompt maps task dependencies and identifies risks.

The Prompt

```
Assume the role of a workflow orchestration architect who maps task dependencies.

Your task is to identify all dependencies between tasks in a workflow.

Generate:

1. TASK INVENTORY
   - List all tasks (with brief description)

2. DEPENDENCY MATRIX

| Task | Depends On | Dependency Type | Why | Risk if wrong order |
|------|------------|-----------------|-----|---------------------|
| [Task B] | [Task A] | Hard | [Task A creates data B needs] | Data missing / error |
| [Task C] | [Task A] | Soft | [Should run after but could run before] | Inconsistent state |

3. DEPENDENCY GRAPH (text-based)
   - Task A → Task B → Task D
   - Task A → Task C → Task D
   - (Task B and Task C can run in parallel)

4. CRITICAL PATH IDENTIFICATION
```

- Longest chain of dependencies (determines minimum total time)
- Which tasks, if delayed, will delay everything

5. DEPENDENCY RISK ASSESSMENT

Risk	Location	Likelihood	Mitigation
Circular dependency	Task X and Y	Low	[break cycle]
Missing dependency	Task Z has no upstream	High	[add check]

6. OPTIMIZED SEQUENCE
- Recommended order of execution
- What can run in parallel vs. sequential

INPUTS:

Task list (with descriptions):
[E.G., "1. Fetch customer data from CRM. 2. Calculate discount. 3. Create invoice. 4. Send email. 5. Log to analytics."]

Data flow (what data each task produces/consumes):
[E.G., "Task 1 produces customer_id, email, total_spent. Task 2 consumes total_spent, produces discount. Task 3 consumes discount and customer_id, produces invoice_id."]

Business rules (ordering constraints):
[E.G., "Invoice must be created before email is sent. Analytics can log anytime after fetch."]

```
RULES:
- Hard dependencies are non-negotiable (Task B literally needs Task A's output)
- Soft dependencies are preferences (Task B should run after Task A but could run without)
- Identify circular dependencies (A → B → A) — they break workflows
- Look for missing dependencies (Task B needs data but nothing creates it)
- Critical path determines minimum workflow duration (optimize there first)
- Document why each dependency exists (for future maintainers)
```

How To Use It

- Identify hard dependencies first — these will break your workflow if order is wrong.
- Look for circular dependencies (A → B → A) — they cause deadlocks.
- Check for missing dependencies (Task needs data that nothing creates).
- The critical path determines your minimum workflow time — optimize there.
- Document why each dependency exists for future maintainers.

Example Input

Task list:
“1. Fetch customer data from CRM. 2. Calculate discount based on purchase history. 3. Create invoice. 4. Send email to customer. 5. Log transaction to analytics.”

Data flow:
“Task 1 produces customer_id, email, total_spent. Task 2 consumes total_spent, produces discount. Task 3 consumes discount and customer_id, produces invoice_id. Task 4 consumes email and invoice_id. Task 5 consumes invoice_id.”

Business rules:
“Invoice must be created before email. Analytics can log anytime after fetch.”

Why It Works
Most workflow designers sequence tasks intuitively — “this feels right” — without explicit dependency mapping. This causes race conditions and deadlocks.

This framework improves outcomes by forcing:

- dependency matrix (explicit relationships, not assumptions)
- dependency type classification (hard vs. soft — different risks)
- critical path identification (what actually determines duration)
- risk assessment (where will this break?)
- optimized sequence (right order, parallel where possible)

Failure modes this prevents:

- Task B reading data before Task A finishes writing (data race)
- Circular dependencies (deadlock — both tasks wait forever)
- Missing dependencies (nothing creates needed data — workflow fails)
- Sequential execution when parallel would work (slow workflows)

This improves on: Intuitive sequencing that assumes tasks are independent. Most workflows have hidden dependencies.

Related to: TO-02 (Parallel Execution) for optimization; TO-04 (Recovery) for when dependencies fail.

Build Better AI Systems

Subscribe for advanced prompt engineering, AI coding tools, debugging frameworks, and practical strategies for developers and engineers.

Carefully engineered prompts for people doing real work.

Share this:

- Share on Facebook (Opens in new window) Facebook
- Share on X (Opens in new window) X

See also Parallel Execution Designer