A WORKING METHOD · MISS AI

The One-Shot Plan

How to write a plan detailed enough that an AI agent builds the whole thing first try, without you sitting there correcting it.

Keira Nesdale @realmissai Built on a real 19-task plan

Your agents aren't failing. Your plan is.

Everyone tuning their prompts is working on the wrong thing. You can write the most beautiful prompt in the world and still get back something almost right, because the model isn't short of instruction. It's short of *information*.

Every gap you leave, it fills with a decision you never made. Then the next step gets built on that decision, and the one after that inherits it. By the time you notice, you're ten steps into somebody else's idea of your project, and the only way back is to start again.

A plan is just the thing that removes the guessing. Here's how to write one that holds.

01

STATE IT ONCE

Global constraints go at the very top

Before a single task, write the block of things that are true for *every* task. The stack. What isn't installed. The conventions. The things that have already gone wrong before.

Every task inherits it, so no task has to work it out again and no task gets to guess. This one section is the difference between a plan and a to-do list.

The test for what belongs in it

Would you be annoyed if it got this wrong twice? Then it goes in the block.

Here are real ones, lifted straight out of the plan behind the feature this method built:

- The exact versions, and pointedly, *what is not installed*. "No new dependencies. PyYAML is not available."
- "Read keys through `config.env_value(NAME)`, never `os.getenv`" — plus one line on *why*, because the thing that runs it doesn't load the env file and every model call silently fails.
- Which test suite has to still pass, and which failures in it are already known and are *not* yours to fix.
- The writing rules that apply to anything a person will read.
- What's permanently out of scope.

> THE PART PEOPLE MISS
>
> Half the constraints worth writing are **scar tissue**. They exist because something went wrong once and you don't want to explain it a second time.
>
> Which means your constraints block should get longer over time, and a plan written after five builds should be visibly better than a plan written after one. If yours isn't growing, you're re-explaining the same thing every week and calling it prompting.

Why it can't live inside the tasks

The tempting version is to repeat the important bits in each task where they matter. Don't. Repeated constraints drift, the copies start disagreeing with each other, and you find out at task fourteen when two finished pieces won't connect.

02

THEN
DIVIDE

Break it into phases, and phases into tasks

A phase is a group of work that makes sense to finish together. A task is one unit inside it, and a task is only a task if it has all five of these.

EVERY TASK CARRIES	BECAUSE WITHOUT IT
`Files`	It invents a file layout, and the next task can't find anything
`Interfaces`	It guesses what to call things, and two tasks build two halves that don't meet
`A failing test`	You have no way to tell "built" from "built correctly"
`The implementation`	—
`A verification command`	It reports done and is sincerely wrong

Here's one from the real plan, trimmed to fit:

```
TASK 1 OF 19                                  PHASE 1 · ENGINE SKELETON

Files:
  Create: tools/guest-scout/config.py
  Create: tools/guest-scout/run_tests.py
  Create: tools/guest-scout/tests/test_config.py

Interfaces:
  Consumes: nothing
  Produces: config.REPO_ROOT, config.DB_PATH,
            config.env_value(name)

Step 1: write the failing test
        (the whole test file, in full)

Step 2: implement until it passes
```

The size rule

No task should be big enough that the agent would have to make a judgement call halfway through it. If it would, split it. That mid-task judgement call is exactly where the invention happens, and every task after it inherits the invention.

Nineteen tasks sounds like a lot to write. It's less work than it looks, because once the constraints block exists each task is mostly naming files and interfaces. And it's dramatically less work than debugging a build that went sideways at step four.

03

DEFINE DONE

Every phase ends in a test of what done looks like

This is the one that makes the whole method work, and it's the one almost everybody skips.

A task that ends in *"make sure it works"* is not a task. It's a wish. The agent has no way to check it, so it will tell you it's finished and it will be sincerely, confidently wrong. It isn't lying. You just never gave it a way to find out.

A task that ends in a command either passes or it doesn't. Nothing to interpret, nothing to be optimistic about:

```
$ python3 tools/guest-scout/tests/test_config.py
5 passed  ✓ phase 1 clear
```

That's the done metric. It's a gate, not a note. And it's what turns a plan from a description of the work into something the agent can actually run itself against.

What counts as a done metric

- A test command with a non-zero exit code on failure. Agents read exit codes.
- A build or typecheck that either completes or doesn't.
- A script that prints an expected value you specified in advance.

What doesn't count: "check the page loads", "confirm the output looks right", "verify it works end to end". Every one of those needs a human, and the whole point is that you're not there.

> IF A PHASE HAS NO VERIFICATION, IT CAN'T RUN UNATTENDED
>
> This is worth treating as a hard rule rather than a nice-to-have. On the build behind this guide, one phase had no test runner in the codebase at all, so building the runner became a *prerequisite task* inside the plan rather than something to sort out later.
>
> A phase you can't verify is a phase you have to sit and watch. At that point you've bought yourself an expensive autocomplete.

04

THEN LEAVE

Put the agents on a loop

Now the plan can run itself. The loop is simple, and it only works because step 03 gave it something to check against:

1. Do the task.
2. Run its verification command.
3. If it fails, fix it and run it again.
4. Don't move on until it passes.
5. Next task.

```
task 06  the web source         ✓ pass
task 07  the Instagram source   ✗ fail  → retry
task 07  the Instagram source   ✓ pass
task 08  enrichment fan-out     ✓ pass
```

That failure in the middle is the point. Without a done metric, task 07 would have been reported as finished and tasks 08 through 19 would have been built on top of something broken. With one, it just goes round again.

This is what people mean by multi-agent work that actually holds up. Not more agents. Agents that can tell whether they succeeded.

What you stop doing

Micromanaging every step. That's the whole return on writing the plan: you go and do something else, come back, and what's there is what you asked for rather than an approximation of it that you now have to unpick.

05

COPY THESE

The three prompts

In order. The order matters: you can't write a good plan from a vague idea, and you can't loop a plan that has no way to tell whether it worked.

Turn the idea into a spec

PROMPT 1 — BEFORE ANY CODE

```
I want to build [describe it in one sentence].

Before you write anything, interview me. Ask one question at a
time and wait for my answer before you ask the next one. Ask
about the things you would otherwise have to guess: who it is
for, what it does when everything goes right, what it must
never do, and what already exists that this has to fit into.

When you have enough, write a design doc in four sections. Get
my approval on each section before you move to the next one.
Do not write the whole thing and then ask me what I think.

The doc says what we are building and why. It never says how
to code it.
```

One question at a time, because ten questions in one message gets you three real answers and seven you skimmed. Approval section by section, because if section one is wrong then sections two through four are wrong too.

Turn the spec into a plan

PROMPT 2 — THE ONE THAT DOES THE WORK

```
Turn that design doc into an implementation plan.

Start with a Global Constraints section. Everything that is
true for every task goes there, once: the stack, what is not
allowed, the conventions, the things that have already gone
wrong before. Every task inherits it, so no task has to work
it out again and no task gets to guess.

Then break the work into phases, and phases into numbered
tasks.

Every single task has:
  - the exact files it creates or changes
  - what it consumes and what it produces, named
  - a failing test, written first
  - the implementation
  - a verification command I can run that either passes or fails

No task should be big enough that you would have to make a
judgement call halfway through it. If you would, split it.
```

Put it on a loop

PROMPT 3 — THEN GO AND DO SOMETHING ELSE

```
Implement this plan task by task.

After each task, run its verification command and show me the
output. If it fails, fix it and run it again. Do not move on
until it passes.

Do not skip ahead, do not batch tasks together, and never tell
me something works without running the command that proves it.
```

06

WATCH FOR

The four ways a plan quietly fails

None of these throw an error. That's what makes them expensive.

01 **A task with no verification command.**

It will be reported as done. It will not be done. You find out three tasks later, when something built on top of it behaves strangely for reasons that have nothing to do with where you're looking.

02 **Constraints repeated inside tasks instead of stated once at the top.**

The copies drift. The tasks start disagreeing with each other. You find out at task fourteen, when two finished halves won't connect.

03 **A task big enough to need a judgement call.**

That's where the agent invents something, and every task after it inherits the invention. The build isn't wrong so much as it's someone else's.

04 **Skipping the spec and going straight to the plan.**

A plan built from a vague idea is a very detailed, very well-organised description of the wrong thing. This is the most demoralising one, because it looks like it's working right up until it's finished.

> THE SHORT VERSION
>
> Say everything once, at the top. Cut the work small enough that nothing has to be decided mid-task. End every piece in a command that passes or fails. Then let it run.