Planning in LLMs: A Dialogue

Core Questions and Answers

Q: Planners ultimately only need to develop one step at a time — much like next-word prediction. Hierarchical planning would help, but based on our Friday conversation, LLMs already do a fair bit of planning in their upper layers during next-word prediction. Couldn't the same apply to planning in a virtual world?

The analogy is tight. Next-token prediction already requires implicit planning — the upper layers pre-compute lexical choices, syntactic structure, and discourse coherence before committing to a single word. Planning in a virtual world is structurally the same problem at a different scale of abstraction: produce a good next action, conditioned on goal state and current context, then repeat.

The "one step at a time" framing is actually liberating. Classical planning research fixates on generating a complete plan upfront. But humans don't do that, and LLMs don't need to either. The autoregressive mechanism is already the right inductive bias — it just needs to be grounded in world-state rather than linguistic context.

The layer hierarchy maps naturally: lower layers handle immediate feasibility and concrete details; upper layers maintain abstract goal representations. This is the same division of labor visible in language generation. And virtual environments — games, structured text worlds — are the ideal proving ground: discrete, observable, verifiable state, no sensor noise, navigable via language.

Q: The architecture probably needs to change, but not by much. In agentic coding systems, the LLM first breaks a problem into manageable tasks, then those tasks are fed back repeatedly. An erasable scratchpad feels intuitively useful, though I'm not committed to it.

The current agentic loop already works — orchestrator decomposes, LLM solves subproblems, results get appended to context, repeat. But the tight coupling between abstraction levels is broken: the high-level plan can influence a subproblem, but the subproblem's results can only feed back in the *next* forward pass, not the current one. The bidirectional interaction is one step delayed.

The scratchpad argument is not primarily about efficiency. An external scratchpad — appended to the prompt — flattens the bidirectional interaction into a linear sequence. What's needed is an *internal* scratchpad: persistent, writable state that lives in the residual stream across forward passes. The upper layers already write plan-like representations into the residual stream during

next-token generation; the modification is giving those writes somewhere to live longer than a single pass.

The KV-cache already does something like this *within* a pass. The scratchpad extends it across passes. This is close to the ATLAS direction: treating memory as an optimizable component rather than passive context.

Q: Now that biological working memory comes up — the scratchpad needs to be inside the LLM. There's an efficiency aspect, but probably more to it. And the transformer architecture has taught us that architecture really matters.

The key point missed by the efficiency framing: human planning is not sequential between levels. We hold a partial, sketchy high-level plan, dive into a subproblem, and the results of that deep dive feed back upward and revise the sketch — simultaneously, not in the next pass. That tight bidirectional coupling is what an external scratchpad can never preserve, because by the time the model reads it, the interaction has already been flattened into text.

An internal scratchpad preserves the coupling because upper layers (abstract, high-level) and lower layers (concrete, syntactic) run *within the same forward pass*, and the residual stream connects them. If the scratchpad lives in that stream, you get the interleaved, mutually-influencing planning that human introspection actually describes.

On architecture mattering: the transformer lesson is not just that attention is useful. It is that the right architecture makes the right computations *easy to learn and easy to compose*. The key inductive bias needed for planning is persistent, writable state at multiple levels of abstraction simultaneously, with learned gating over what gets updated when. That starts to look less like a notepad and more like what biological working memory actually does: not static storage, but an active maintenance process.

Q: I don't care much about benchmarks.

That's the right stance. Benchmarks measure outputs — does the plan solve the task. They say nothing about whether the model has the right internal process. Once a benchmark becomes a target, it stops measuring what you care about (Goodhart's law applies hard here).

What to measure instead: does the model exhibit level-switching during planning? Do upper layers maintain stable-but-revisable abstract representations while lower layers work on a subproblem? Do subproblem failures propagate upward and modify the high-level representation? Those are mechanistic questions, answerable with SAE features and activation patching. The training data is good if it induces the right internal process — independent of whether benchmark scores improve.

This also opens a genuinely unusual research position. Almost all planning research is behavioral. The mechanistic angle on planning is nearly untrodden.

Q: When it comes to training, we may just need to design data that teaches planning directly. Right now LLMs can do some planning, but they seem to have picked it up by reading a lot.

That is exactly the problem. LLMs learned to plan the way a well-read person develops strategic intuition without being explicitly taught strategy. They read polished artifacts — finished proofs, working code, completed arguments — not the process that produced them. The planning machinery is reconstructed archaeologically from traces, which is why it is brittle at the edges: novel domains, long horizons, tasks where surface patterns are stripped.

What deliberate training data would look like is qualitatively different. Not more plans, but data that exposes the *process*: false starts, level-switching, moments where a high-level assumption gets revised because a subproblem came back unsolvable. Programming with execution feedback is the closest existing approximation. Scientific reasoning logs, proof attempts with backtracking, synthetic planning trajectories with explicit revision loops would all be more targeted.

The dangerous case is synthetic data: if you encode your theory of planning into the training distribution and the theory is wrong, you train the wrong thing very efficiently.

The analogy to reading is clarifying in another way too. When we teach children to read, we are not teaching them language — they already have that. We are teaching a new input modality that connects to existing cognitive machinery. Deliberate planning training may be similar: not installing new capability, but teaching the model to apply machinery it already has to a new domain — acting in the world rather than generating language.