

# Toward AGI: Planning, Reasoning, and the Architecture Gap

## A Q&A Exploration

---

**Q: It seems that everything semantic and pragmatic about an input is determined by around layer 30 of an LLM. The remaining layers seem concerned with preparing for possible responses — not next-word prediction. Is this right, and if so, how are those upper layers trained?**

The broad intuition is correct. Work using the "logit lens" technique shows that mid-layer representations often already "know" the correct answer to a factual query — later layers aren't discovering new semantic content so much as shaping how it gets expressed. Upper layers appear to do discourse-level and pragmatic computation: modeling genre, register, conversational role, and the implied trajectory of an exchange. In instruction-tuned models there is evidence of something like implicit response planning occurring before any output is generated.

Crucially, all layers are trained by exactly the same objective — minimizing next-token prediction loss. No separate training signal exists for upper layers. What happens is that optimization pressure causes specialization to emerge. Predicting the next token in rich discourse requires modeling the discourse itself. By layer 60 or so, the marginal value of re-doing semantic computation is low; the marginal value of computing what to do with those semantics is high. The upper layers end up doing anticipatory, pragmatic work because that is what is left to do once the lower layers have handled syntax and semantics — and because doing it well is exactly what next-token prediction on complex text demands.

---

**Q: The upper layers have attention and MLP layers anyway — they may as well do something with their compute rather than sit idle.**

Exactly, and this is a non-trivial point. The network has no mechanism for a layer to pass through unchanged without cost. Every layer applies its transformations regardless. So the optimization pressure is: given that you are going to transform the residual stream anyway, you might as well do something useful.

This explains why upper layers develop genuine specialization rather than adding noise. MLP layers in particular appear to act as key-value memories — where keys are patterns in the residual stream and values are distributions over vocabulary. Upper-layer MLPs store more abstract, contextual associations than lower-layer ones. Anthropic's work on superposition further suggests the network encodes far more features than it has neurons by overlapping

representations in high-dimensional space, meaning upper layers may be doing more computation than their parameter count naively implies.

---

**Q: For AGI, I am interested in world modeling and reasoning. My intuition is this requires a larger working memory spanning residual streams. But based on our discussion, the upper layers already seem to have a very good world model. So how much planning goes into code generation — and how much happens in the LLM versus the agentic system?**

The world model intuition is well-founded. Upper-layer MLP and attention machinery does contain a rich implicit world model. The working memory question is more nuanced.

For code generation, a single forward pass — specifically the upper-layer representations — can handle the full plan for problems within the model's training distribution. Before the first output token is emitted, the upper layers have already formed something like a solution intent: the shape of the answer, the key abstractions, the approach. Chain-of-thought reasoning is partly externalizing a plan that was already partially formed, not constructing it token by token.

For deeper tasks, a single forward pass is bounded in reasoning depth — roughly proportional to the number of layers. This is where the agentic system compensates: it breaks large tasks into subtasks each within the model's effective planning horizon, manages persistent state across passes, and feeds execution results back into context.

---

**Q: The decomposition step in an agentic system — is it written in classical AI or a trained neural network?**

It is the trained neural network doing the decomposition — essentially all the way down. There is no classical AI planning (no STRIPS, no HTN planners, no symbolic decomposition engines) in mainstream agentic coding systems. The decomposition is just the LLM prompted to break a task into subtasks, possibly in structured output format. The agentic scaffold around it — the part that calls tools, manages context, and routes outputs — is ordinary software. But the cognitive work of decomposition is entirely the LLM.

This means decomposition quality is bounded by the same model that does everything else. There is no separate specialized planning module. And unlike a classical planner, the LLM's decomposition is opaque — it may look like a plan but there is no guarantee the internal structure is coherent in the way a symbolic plan would be.

---

**Q: So you don't even have to train the agentic system?**

Correct — and this was one of the most surprising empirical discoveries of the last few years. The agentic capability was largely already present in the pretrained model. Nobody trained a

separate planning module. Nobody designed a decomposition algorithm. The model was trained purely on next-token prediction on internet text and absorbed enough knowledge about problem-solving, software engineering, and task decomposition that you could simply prompt it to do those things.

This explains the pace of progress. Once people realized the capability was already in the weights, the race became about scaffolding design — how to prompt, what tools to provide, how to manage context. That is fast iteration. You are not retraining models; you are writing Python.

---

### **Q: What exactly is missing to get current AI systems to do genuine planning?**

The gaps are structural, not about intelligence or world knowledge. Six specific things are missing:

1. **Persistent hierarchical state.** A real plan is a tree — with goal/subgoal relationships and constraints that propagate up and down. Nothing in the current architecture maintains that structure natively. The model re-infers plan structure from flat text on every single forward pass, which is fragile and lossy.
2. **Genuine backtracking.** Current systems can retry failed subtasks, but cannot recognize that a failure at a low level implies the decomposition itself was wrong and then surgically revise the high-level plan.
3. **Consistency enforcement across subtasks.** When subtasks are generated somewhat independently, there is no mechanism ensuring they are mutually consistent. Two subtasks may make contradictory assumptions about a shared interface or data structure.
4. **Explicit uncertainty tracking.** There is no representation of uncertainty about the plan itself. The model commits confidently even when the problem is underspecified.
5. **Goal preservation over long horizons.** In a long agentic loop the model drifts — immediate context gradually crowds out awareness of the high-level goal. The model starts optimizing locally rather than globally.
6. **Meta-cognitive monitoring.** Knowing the plan is going wrong before it fully fails, by comparing current state against expected state at each step.

The common thread is the absence of a dynamic, structured, persistent working representation of the plan — something the model can read from and write to, that maintains hierarchical relationships, tracks uncertainty, and survives across many forward passes.

---

**Q: I am not sure I subscribe to the statement that there is no persistent structured working memory. All the residual streams together seem like a massive working memory — perhaps**

**not as flexible as a scratchpad, but massive and informed.**

This is a sharp correction. The residual stream working memory is not just the context window. For a large model: roughly 96 layers times 100,000 token context times a 12,288-dimensional residual stream yields on the order of 100 billion active floats per forward pass. And superposition means the effective representational capacity is larger still. Every transformation applied to it is informed by the world model baked into the weights, so it is not just massive — it is massive and intelligent.

The real limitation is more specific: the computation is strictly feedforward and non-iterative within a single pass. Information flows in one direction — layer 1 to layer 96. There is no mechanism to say "given what I computed at layer 80, revise what I did at layer 40." No looping, no backtracking, no revision within a pass.

This reframes the problem. The question is not how to give the model more working memory. It is how to give the model iterative, revisable access to its own rich internal state.

---

**Q: How about looping over the entire 96 layers — not maximally efficient, but a reasonable prototype?**

This is a well-motivated instinct, and it has precedent. Universal Transformers (Dehghani et al., 2018) did essentially this — running the same layer block repeatedly with output feeding back as input. The key finding was that this confers something closer to Turing-completeness in computational expressiveness, because iteration allows depth-unlimited refinement. A standard transformer is a bounded-depth circuit; a looped transformer is not.

Building from scratch is the cleaner path because you avoid the geometric mismatch problem — the weights are designed so that the output of layer 96 is a valid input to layer 1 again. Key design decisions include whether weights are shared across loops (parameter-efficient, forces learning a general refinement operation) or unshared (more expressive, easier to train), and whether looping runs a fixed number of passes or uses an adaptive halting mechanism that allocates more compute to harder problems. The loop counter embedding is critical — without it the model has no way to know which pass it is on, which makes learning pass-specific behavior very difficult.

---

**Q: What are the options for persistent state — the ability to carry a plan across multiple forward passes?**

Six options with increasing sophistication:

1. **Context window as state.** The plan lives in the token stream. Works today, no architectural changes needed. Fundamentally inadequate for deep planning — flat, serial, size-limited, no enforced structure.

2. **External key-value memory.** A memory bank outside the model that the model reads and writes via special tokens or tool calls. Unbounded size, persistent across passes, can be structured. But the interface is clunky — the model must explicitly decide what to store and retrieve.
3. **Persistent residual stream across passes.** Carry a compressed form of the residual stream forward into the next generation step. Directly leverages the rich representational space, natural fit with a looped architecture. Requires learned compression — the model must learn what is worth preserving.
4. **Dedicated state tokens.** A fixed set of special tokens whose embeddings are explicitly designated as writable state, updated by the model on each pass via attention. Stays entirely within the transformer framework, differentiable end to end, trainable naturally. Fixed capacity must be decided upfront. This is the most practical first implementation.
5. **Recurrent state vector.** A fixed-size learned vector updated by a small network after each loop pass. Compact and fast, but the fixed size creates a bottleneck — the model must compress arbitrarily complex plan state into it.
6. **Hierarchical memory.** Explicitly maintained tree structure with goal, subtask, and working-detail levels. Directly mirrors the structure of plans and enforces consistency by construction. The right long-term target, but not where to start.

The recommended approach: start with state tokens, add a recurrent state vector for high-level summary, and move toward hierarchical structure once the basics work.

---

**Q: How would you train a system where state tokens are iteratively refined across loop passes before any output is generated?**

This is the hardest part of the design. Four specific challenges must be solved:

The **bootstrapping problem**: early in training, state tokens contain noise. The model learns to ignore them, so they stay noisy, so the model keeps ignoring them. This deadlock must be broken by forcing state token use before looping is introduced — for example, by masking input mid-sequence and requiring the model to answer from state tokens alone, combined with an auxiliary reconstruction loss that penalizes state tokens that fail to capture key information.

**Gradient flow across loops**: backpropagation must propagate through multiple iterations, analogous to backpropagation through time in RNNs. This is stabilized by making state token updates residual —  $\text{State}(n+1) = \text{State}(n) + \Delta\text{State}(n)$  — so the model learns corrections rather than full rewrites. Gradient checkpointing manages the memory cost.

**Process supervision**: supervising only final output gives no signal that intermediate state token representations should be meaningful. Training on synthetic tasks with known intermediate

states, combined with a process reward model that scores reasoning quality at each pass, provides the necessary intermediate signal.

**Teaching revision:** the model needs to learn that pass 1 is for rough planning and subsequent passes are for refinement and consistency checking. This is best approached through a curriculum: problems solvable in one pass, then problems where a second pass demonstrably catches a flaw in the first, then deeper multi-constraint problems.

The fundamental training strategy is externalization followed by internalization: first train the model to plan explicitly in generated text, then train it to produce equivalent output with progressively less explicit text, pushing the planning into the state tokens. Use the explicit reasoning traces as supervision signal for what the state tokens should encode. The most important practical advice: make state tokens observable during training. Build tooling to inspect what information they contain at each loop pass. Without that visibility, diagnosing failure modes is nearly impossible — and the failure modes will be informative.

---

**Q: For the Fibonacci example — is all the code output after one pass through the LLM? One pass per token? Something in between?**

One full forward pass per token — all 96 layers — for every single output token. For a 60-token Fibonacci function, there are approximately 60 complete forward passes. Each pass takes the entire context — original prompt plus all tokens generated so far — through all layers from scratch.

What makes this remarkable is that on pass 1, the upper layers already represent the entire plan for the function: the base case, the iterative approach, the return type. Each subsequent token generation is not a fresh decision — it is reading out the next piece of a plan that was largely formed on pass 1 and maintained across subsequent passes. The first token emitted already reflects decisions made in the upper layers about the structure of the entire solution.

This also reveals a significant inefficiency: the planning computation in upper layers is nearly identical across all 60 passes. The same planning work is repeated 60 times. The KV cache saves the attention computations for previous tokens but does not cache the upper-layer planning representations. A looped architecture with state tokens could address this directly — do the deep planning work during the looping phase, cache the plan in state tokens, then read from the cached plan during generation rather than recomputing it.

---

**Q: Are people working on a solution to this redundant planning problem?**

Yes, from several directions, though nobody has cleanly solved the specific inefficiency. The KV cache is the most widely deployed solution but addresses attention computation, not planning

redundancy. Speculative decoding reduces the number of large-model passes but does not change what each pass computes.

The most directly relevant research directions are: Mamba and other state space models, which maintain a compressed recurrent state motivated by exactly this redundancy; diffusion language models, which generate the entire sequence simultaneously and refine it iteratively — conceptually the closest to the proposed architecture's goals; and the o-series and extended thinking approaches, which validate the value of process-level training signal by rewarding correct intermediate reasoning rather than just correct output.

The proposed architecture — state tokens refined across loop passes, then consulted during generation — would address the redundancy more directly than any current production approach, and represents a genuine architectural contribution rather than an incremental improvement.