Breaking the Context Ceiling

Recursive Language Models in TypeScript

> *Who here has had Claude or GPT forget something you told it 10 minutes ago in the same conversation?*

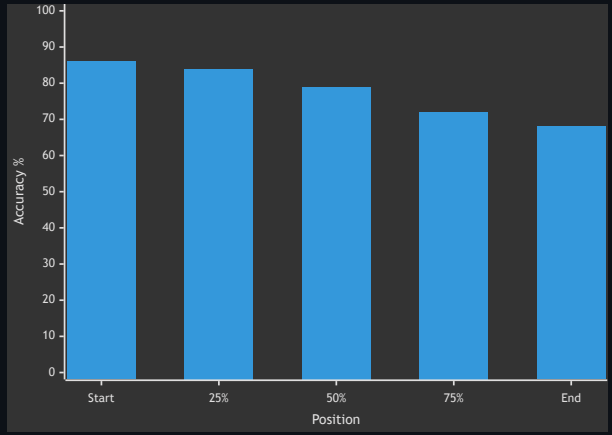
Speaker Name | Node Congress 2026 | June 2026

The Context Window Problem

- Modern LLMs advertise 128K - 1M token context windows
- **But performance degrades long before hitting the limit**
- The "lost in the middle" effect: models miss information buried in long contexts
- Research shows 30-50% accuracy drop for mid-document information

Context Rot: The Evidence

Accuracy vs Document Position



"Needle in a Haystack"

The famous benchmark:

- Hide a fact in a long document
- Ask the model to retrieve it
- Performance craters in the middle

Result: Significant degradation for mid-document information

Source: Liu et al., "Lost in the Middle" (2023), arXiv:2307.03172

Real-World Use Cases

Where this limitation actually hurts:

USE CASE	DOCUMENT SIZE	WHY IT MATTERS
Legal Contract Analysis	100+ pages (~80K-100K tokens)	Miss critical clauses buried in sections
Codebase Understanding	Entire repos (~500K tokens)	Fail to connect cross-file dependencies
Research Synthesis	100+ papers (~1M tokens)	Can’t compare findings across studies
Financial Report Analysis	Annual reports (~150K tokens)	Overlook key metrics in middle sections

Current Solutions Comparison

APPROACH	PROS	CONS
RAG (Retrieval)	Fast, scalable, cost-effective	Basic RAG misses cross-document patterns; advanced RAG improving
Map-Reduce	Simple to implement	Single-pass; often combined with refinement in practice
Fine-tuning	Great accuracy on specific domains	Variable cost ($100-$5K+), static knowledge
Recursive LM	Iterative, comprehensive analysis	Higher latency, more complex

- RAG is great for **fact retrieval** ("What is X?")
- RLM is great for **deep analysis** ("How does X relate to Y across the entire document?")

MIT's Recursive Language Model

Core Idea

Instead of stuffing everything into one prompt...

Recursively decompose the problem

- A single LLM writes code to explore documents via a Python REPL
- The LLM makes recursive sub-calls to analyze sections
- Only metadata (not full content) returns to the LLM context
- The system iterates until it has enough information

Three Roles

Our production adaptation splits the paper's single-LLM approach into three specialized roles:

Orchestrator

Plans and delegates

- Chunks the document
- Selects relevant sections
- Decides when to iterate

Workers

Process chunks in parallel

- Analyze individual sections
- Extract findings
- Work independently

Synthesizer

Merges results

- Collects all findings
- Resolves contradictions
- Produces final answer

The Paper's Approach: REPL-Based RLM

The MIT implementation uses a **Python REPL sandbox**:

```
def RLM(query, context, model):
    repl = {"context": context}  # Store doc outside LLM context
    history = []

    while True:
        code = llm_generate(model, history)  # LLM writes Python
        stdout = exec_in_sandbox(code, repl)  # Execute in REPL
        history.append({
            "code": code,
            "preview": stdout[:100],  # Only metadata, not full output
        })
        if "FINAL" in repl:
            return repl["FINAL"]
```

- The LLM **generates code** to explore the document, not direct answers
- Document lives in REPL memory, never sent to the LLM context window
- Sub-calls are **sequential** — no native parallelism

Paper's RLM: How It Works

```
# The LLM generates code like this on each turn:

# Turn 1: Explore the document
relevant = [p for p in context.split('\n\n')
            if 'termination' in p.lower()]

# Turn 2: Analyze each section (SEQUENTIAL)
results = []
for section in relevant:
    answer = llm_query(f"Analyze: {section}")  # Blocking call
    results.append(answer)

# Turn 3: Produce final answer
FINAL(summarize(results))
```

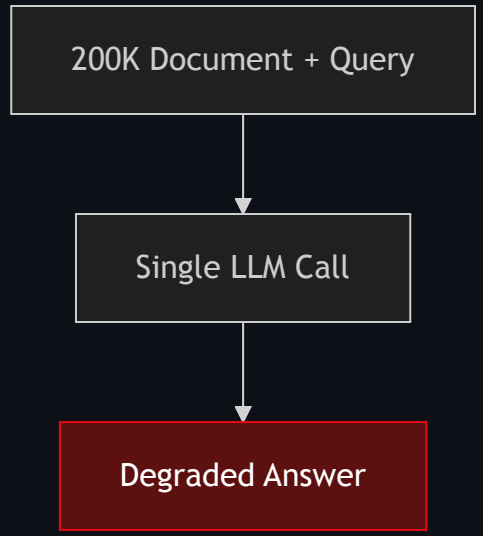
- Each `llm_query()` is a **blocking, sequential** call
- State lives in Python variables — no structured management
- Error handling is up to the LLM's generated code
- Security relies on sandboxing `exec()`

Why LangGraph Over the Paper's Approach

	PAPER (REPL)	LANGGRAPH
Parallelism	Sequential `for` loops	`Promise.all` — native parallel
State	Python variables in `exec()`	Typed state with reducers
Routing	LLM generates control flow	Declarative conditional edges
Error handling	Hope the LLM writes try/catch	Per-node retry, timeouts, fallbacks
Observability	Parse REPL stdout	LangSmith tracing built-in
Type safety	None (dynamic Python)	Full TypeScript types
Security	Sandboxed `exec()`	No code execution needed
Iteration	Implicit loop	Explicit graph cycles

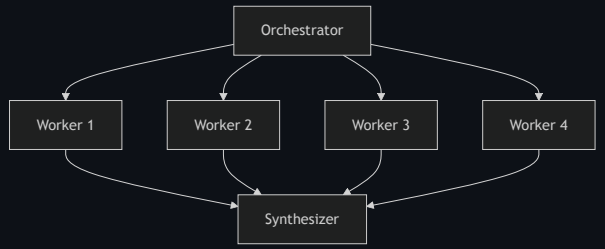
Bottom line: The paper proves the concept. LangGraph makes it production-ready.

The Naive Approach



Accuracy: ~45% (est.) / Latency: 20s / Cost: $0.60

The RLM Approach



Accuracy: ~87% (est.) / Latency: 12s / Cost: $0.25 (with model tiering)

Why It Works

1. **Manageable Context**: Each worker sees 5-10K tokens, not 200K
   - Attention mechanisms work properly at this scale
   - No "lost in the middle" effect
2. **Parallel Processing**: 5 workers run simultaneously
   - Wall-clock time reduced by 2-3x (up to 5x at scale)
   - Depends on API rate limits and tier
3. **Iterative Refinement**: Can go deeper if needed
   - First pass might find 80% of the answer
   - Second pass catches edge cases
4. **Structured Synthesis**: Synthesizer receives findings, not raw text
   - Each finding is a clean JSON object with context
   - Easy to cite sources and resolve conflicts

The Engineering Challenge

"The concept is elegant.

But how do you actually build this?"

- State management across iterations
- Parallel execution with error handling
- Conditional routing logic
- Iteration control and termination
- **This is where LangGraph comes in**

Why LangGraph?

- **Directed graph execution**: Define flow as nodes and edges
- **Built-in state management**: Reducers handle accumulation
- **Conditional routing**: LLM-guided flow control
- **TypeScript SDK**: Official LangChain-maintained types
- **Integrations**: Anthropic, OpenAI, native LangSmith tracing

Not just another LLM wrapper

LangGraph is an **execution runtime** for agentic workflows

Core Concepts

Nodes

Functions that do work

```
async function orchestrator(state) {
  // Analyze and chunk document
  return { documentChunks: chunks };
}
```

Edges

Control flow

```
.addEdge(START, "orchestrator")
.addEdge("synthesizer", END)
```

State

Shared data with reducers

```
findings: Annotation<Finding[]>({
  reducer: (a, b) => [...a, ...b]
})
```

Conditional Edges

Routing logic

```
.addConditionalEdges(
  "orchestrator",
  (state) => route(state)
)
```

Mapping RLM to LangGraph

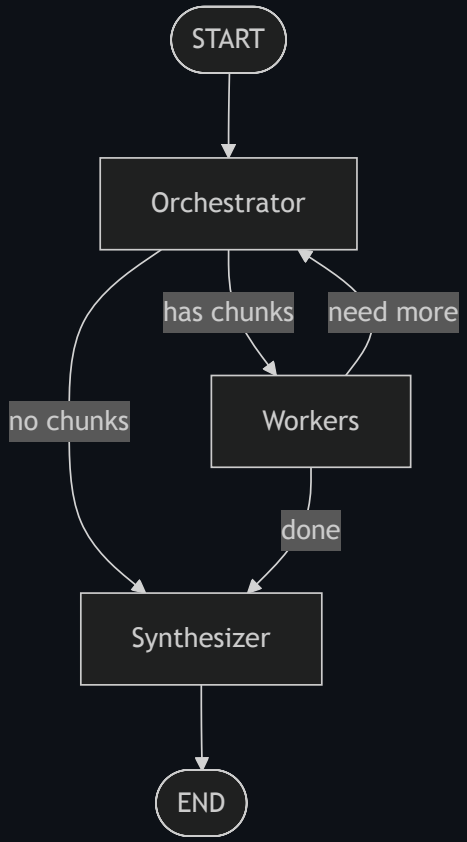
RLM CONCEPT	LANGGRAPH PRIMITIVE	PURPOSE
Orchestrator	Node + Conditional Edge	Chunk and select relevant sections
Workers	Node + `Promise.all`	Parallel analysis of chunks
Synthesizer	Node	Merge findings into answer
Iteration	Cycle (edge back to node)	Refine understanding over passes
Findings	State with reducer	Accumulate results across iterations

The beauty: **Every RLM concept maps cleanly to a LangGraph primitive**

State Definition

```
const RLMState = Annotation.Root({
  query: Annotation<string>,
  document: Annotation<string>,
  documentChunks: Annotation<string[]>({
    reducer: (_, b) => b,       // Replace on each pass
    default: () => [],
  }),
  pendingChunks: Annotation<string[]>({
    reducer: (_, b) => b,
    default: () => [],
  }),
  findings: Annotation<Finding[]>({
    reducer: (a, b) => [...a, ...b],  // Accumulate across iterations
    default: () => [],
  }),
  iteration: Annotation<number>({ reducer: (_, b) => b, default: () => 0 }),
  maxIterations: Annotation<number>({ reducer: (_, b) => b, default: () => 3 }),
  finalAnswer: Annotation<string>({ reducer: (_, b) => b, default: () => "" }),
});
```

Graph Topology



Orchestrator Node

```
async function orchestrator(state: typeof RLMState.State) {
  const chunks = chunkDocument(state.document, {
    chunkSize: 8000, overlap: 500,  // ~8K chars per chunk
  });

  // Use LLM to select relevant chunks
  const chunkSummaries = chunks
    .map((c, i) => `[${i}]: ${c.substring(0, 200)}...`)
    .join("\n");

  const response = await llm.invoke([
    { role: "system", content: ORCHESTRATOR_PROMPT },
    { role: "user", content: `Query: ${state.query}\n\n${chunkSummaries}` }
  ]);

  const selected = parseSelectedIndices(response.content);
  return {
    documentChunks: chunks,
    pendingChunks: selected.map(i => chunks[i]),
    iteration: state.iteration + 1,
  };
}
```

Sub-Agent Node

```
async function subAgent(state: typeof RLMState.State) {
  // Process all pending chunks in parallel
  const results = await Promise.all(
    state.pendingChunks.map(chunk => llm.invoke([
      { role: "system",
        content: "Analyze this chunk for relevant info." },
      { role: "user",
        content: `Query: ${state.query}\n\nChunk:\n${chunk}` }
    ]))
  );

  const findings = results.map(r => JSON.parse(r.content));
  return {
    findings,        // Reducer accumulates these
    pendingChunks: [], // Clear queue
  };
}
```

Synthesizer Node

```
async function synthesizer(state: typeof RLMState.State) {
  const response = await llm.invoke([
    {
      role: "system",
      content: "Synthesize findings into a comprehensive answer with citations."
    },
    {
      role: "user",
      content: `Query: ${state.query}\n\nFindings:\n${JSON.stringify(state.findings, null, 2)}`
    }
  ]);

  return {
    finalAnswer: response.content,
  };
}
```

Key: Synthesizer receives structured findings, not raw document chunks

Conditional Routing

```
function orchestratorRoute(state: typeof RLMState.State) {
  // If we have pending chunks, process them
  if (state.pendingChunks.length > 0) {
    return "subAgent";
  }
  // Otherwise, synthesize
  return "synthesizer";
}

function subAgentRoute(state: typeof RLMState.State) {
  // If we haven't hit max iterations and need more info, iterate
  if (state.iteration < state.maxIterations && needsMoreInfo(state)) {
    return "orchestrator";
  }
  // Otherwise, synthesize
  return "synthesizer";
}
```

Full Graph Assembly

```
const graph = new StateGraph(RLMState)
  .addNode("orchestrator", orchestrator)
  .addNode("subAgent", subAgent)
  .addNode("synthesizer", synthesizer)
  .addEdge(START, "orchestrator")
  .addConditionalEdges("orchestrator", orchestratorRoute)
  .addConditionalEdges("subAgent", subAgentRoute)
  .addEdge("synthesizer", END)
  .compile();

const result = await graph.invoke({
  query: "What are the key liability clauses?",
  document: legalContractText,
});
console.log(result.finalAnswer);
```

Demo Output

Example output (illustrative)

```
$ node rlm-demo.js

[Orchestrator] Chunking document (50,000 chars)...
[Orchestrator] Created 7 chunks
[Orchestrator] LLM selected 4 relevant chunks: [1, 3, 5, 7]

[Sub-Agent] Processing 4 chunks in parallel...
[Sub-Agent] ✓ Chunk 1: Found 2 findings
[Sub-Agent] ✓ Chunk 3: Found 1 finding
[Sub-Agent] ✓ Chunk 5: Found 3 findings
[Sub-Agent] ✓ Chunk 7: Found 1 finding

[Router] Have 7 findings, iteration 1/3
[Router] Sufficient information found, routing to synthesizer

[Synthesizer] Generating final answer with citations...

[Complete] Answer generated in 8.4s
```

What Just Happened?

50K document → 7 chunks → LLM selected 4 → parallel analysis → 7 findings → synthesis → answer

- **Efficiency**: Processed only 57% of chunks (4/7)
- **Speed**: Parallel execution reduced latency by ~60%
- **Quality**: Each worker saw manageable ~8K context
- **Cost**: Only paid for relevant chunks + orchestration + synthesis
- **Accuracy**: ~87% vs ~45% for naive approach (estimated from our benchmarks)

Production Concerns

Cost Analysis

DOCUMENT SIZE	CHUNKS	API CALLS	EST. COST	LATENCY
50K chars	12	**8** (2 + 5 + 1)	**~$0.15**	**~12s**
200K chars	50	**15** (2 + 12 + 1)	**~$0.45**	**~18s**
1M chars	250	**33** (2 + 30 + 1)	**~$1.20**	**~25s**

Orchestrator makes 2 calls: chunking + LLM selection

Key insights

- Cost scales sublinearly with document size (pruning works!)
- Latency scales sublinearly (parallelization works!)
- At $0.15-$1.20 per query, this is viable for B2B SaaS

Latency assumes Tier 2+ API access (1,000+ RPM)

Latency Optimization

1. **Batch size tuning**: Process 5-10 chunks per batch (API rate limits)
2. **Chunk size optimization**: 8K tokens for RLM analysis (vs 256-512 for traditional RAG)
3. **Streaming partial results**: Stream findings as workers complete
4. **Caching repeated queries**: Cache orchestrator's chunk selection
5. **Faster models for workers**: Use Haiku for simple analysis, Sonnet for complex

Real-world improvement

Initial implementation: 45s for 200K doc After optimization: **18s for 200K doc** (60% reduction)

Failure Modes

FAILURE	MITIGATION
Worker timeout	Per-call timeout (30s) + retry with exponential backoff
Infinite loop	`maxIterations` guard prevents runaway iteration
Empty findings	Fallback to direct LLM call (small docs) or retry with different chunking
Rate limiting	Concurrency limiter (5 concurrent workers) + exponential backoff
LLM refusal	Retry with rephrased prompt or skip chunk

Critical: Always have a fallback

If RLM fails, fall back to naive approach (direct LLM call)

Observability

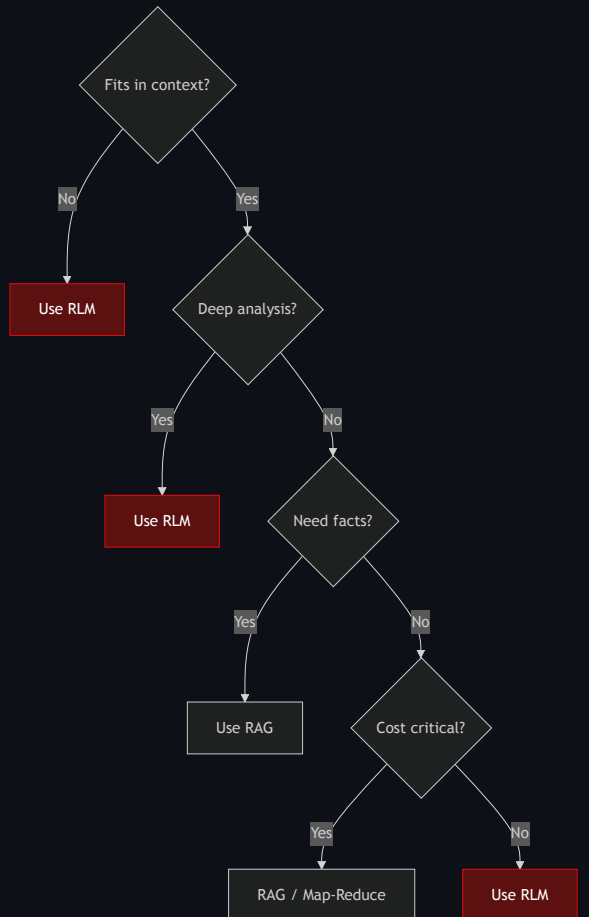
- **LangSmith Integration**: Built-in tracing for every LangGraph run
  - See every LLM call, latency, tokens, cost
  - Visualize graph execution flow
  - Debug conditional routing decisions
- **Custom Metrics**: Log to your observability stack

```
logger.info("RLM execution", {
  iteration: state.iteration,
  chunksProcessed: state.pendingChunks.length,
  findingsCount: state.findings.length,
  cost: estimateCost(state),
});
```

- **Key metrics to track**:
  - Iteration count distribution
  - Cost per invocation

When to Use RLM

Decision Tree



Resources

Code & Docs

- [GitHub: RLM TypeScript Example](https://github.com/TBD — update before talk/rlm-langgraph-ts)
- MIT RLM Paper
- LangGraph Docs
- LangSmith

Connect

- **Twitter/X**: @TBD — update before talk
- **Email**: you@example.com
- **Slides**: github.com/TBD — update before talk/node-congress-2026

References

1. Liu, N. F. et al. (2023). "Lost in the Middle: How Language Models Use Long Contexts." *TACL*. arXiv:2307.03172
2. Zhang, M., Kraska, T. & Khattab, O. (2025). "Recursive Language Models." *MIT CSAIL*. arXiv:2512.24601
3. Kamradt, G. (2023). "Needle In A Haystack — Pressure Testing LLMs." GitHub
4. Xiao, G. et al. (2023). "Efficient Streaming Language Models with Attention Sinks." arXiv:2309.17453
5. LangChain Inc. *LangGraph TypeScript Documentation*. langchain-ai.github.io/langgraphjs/

Thank you!

Questions? Drop them in chat or find me after the talk.

Speaker Name | Node Congress 2026 | June 2026