ChatGPT

Integrating Recursive Language Models (RLM) into LLMC: Phase 1 HLD

Recursive Language Models (RLM) offer a new inference strategy for handling extremely large contexts by treating the prompt or data **as external state** (like a variable in a REPL) rather than packing everything into the model's token window [1]. The model is empowered to **write code that "peeks" into or filters this external context, and to recursively call sub-models** on selected snippets [1]. This approach has been shown to handle inputs on the order of **millions of tokens** and outperform base LLMs or naive summarization/retrieval scaffolds on long-context tasks [2]. The core idea is to move heavy-lift processing out of the LLM's direct context and into external tools/code, allowing the LLM to **focus its attention budget** on the relevant bits of information. Below, we propose how to integrate the RLM architecture into the LLMC codebase to improve extended-context reasoning and repository comprehension, delivering a high-level design (HLD) for Phase 1 of this integration.

Integration Points in LLMC Architecture

LLMC already provides a solid foundation for RLM-like behavior, with distinct subsystems for retrieval, agent orchestration, and tool execution [3]. The integration will primarily target two areas:

- **LLMC Agent Loop (`llmc_agent/*`)** – This is the "brain" orchestrating prompts, Retrieval-Augmented Generation (RAG), and tool usage [4]. To support RLM, the agent needs to handle *recursive reasoning* calls and dynamic sub-querying. Integration here means extending the agent's prompt construction and decision loop to allow the model to **output special code or commands** (instead of a final answer) which trigger sub-tools or sub-LLM calls. Concretely, the agent can enter a *"recursive mode"* where the user's query is not answered directly; instead, the model receives a system prompt enabling code execution (see below) and access to a *context variable* representing the large input (e.g. the entire repository or a large document). The agent will detect when the model's output is code (or a structured plan) and execute it, rather than treating it as normal conversational text. This likely involves intercepting the agent's generation loop to look for markers (e.g. a special code block or a call to a `FINAL()` function as in the RLM paper [5]) that signal the end of reasoning vs. continuation. Key integration points in `agent.py` would be around where it formats the prompt with context and tools, and where it handles the model's output to decide on next actions.

- **LLMC MCP Server (`llmc_mcp/*`)** – The Model Context Protocol server in LLMC provides a *tool execution environment with isolation and guards* [6]. For RLM, the MCP needs to support a **persistent code execution environment** (a REPL) where the large context can reside as a variable and where the model's generated code can call sub-tools (including invoking the LLM recursively). LLMC already has a "code execution" mode that consolidates many tools into a Python sandbox. We will extend this. Specifically, we introduce an RLM-specific **tool/API** in the MCP: for example, a function `llm_query(chunk: str) -> str` that the model can call from Python code to spawn a *sub-LLM* query on a snippet of text. Also, we ensure the code execution environment can maintain state across multiple calls in one session (so that `context` or other variables persist through the recursive process). Integration points include the MCP's tool registry and the sandbox manager – we may add an **RLM Tooling Module** (under `llmc_mcp/tools/`) that defines `llm_query` and manages context state. We will leverage the existing `code_exec` mechanism, which was inspired by Anthropic's approach of letting the model write code to use tools efficiently. In fact, the "Code Execution Mode" in LLMC already replaces numerous direct tools with a few bootstrap tools and a Python exec environment, so the RLM integration largely means *configuring code_exec mode to load the initial* `context` *variable and adding our recursive call hooks*. In summary, **the agent will interface with the MCP to run model-generated code**, and the MCP will expose APIs (`llm_query`, etc.) for recursive calls and context inspection.

By integrating at these points, we transform LLMC's existing RAG + tools framework into an **out-of-core context navigation runtime** [7] – essentially what RLM formalizes.

Recursive Tool Abstractions for the LLM

To enable the LLM to **self-decompose queries and manage extended context**, we introduce a suite of intelligent tools/abstractions that the model can invoke (potentially recursively) via the code execution sandbox. These tools provide functionality like summarization, routing to relevant chunks, inspecting intermediate reasoning, and navigating code structures. Below are the key tool abstractions we propose, along with references to similar open-source implementations or research for guidance:

- **Summarization Tool** – *Function:* Condense a text chunk to a shorter summary, preserving essential info. This helps the model compress parts of the context that it has deemed relevant, freeing up memory for reasoning. *Design:* Implement as a function `summarize(text: str, max_len: int) -> str` accessible in the sandbox. The LLM can call `summarize()` on long sections of `context` to produce abstracts that fit in its token window. We can leverage existing hierarchical summarization techniques – e.g. LlamaIndex's "Tree Summarize" mode performs **hierarchical summarization** by recursively summarizing chunks and then summarizing the summaries [8]. The tool might internally call a smaller LLM (or the same model with a narrower prompt) to generate the summary. **Open-source precedent:** The idea of *recursive summarization* has been demonstrated to extend LLM context by iteratively condensing information [9] [8]. We can cite the approach in which an LLM summarizes sections of a long document, then summarizes the summaries, etc., as a known method to handle very long inputs. This Summarization Tool will be critical for **context windows beyond the base LLM's capacity**, allowing the model to decide what details to keep or throw away.

- **Chunk Routing / Decomposition Tool** – *Function:* Dynamically choose which part of the large context to focus on next, possibly by splitting the query or context into sub-parts. In practice, this could be a function `route_query(query: str, context_index: IndexType) -> List[Snippet]` that, given a sub-question, retrieves or selects the most relevant chunks from the repository or document index. While LLMC's RAG already does static retrieval before prompting, the **chunk router tool lets the model actively guide retrieval in multiple passes**. For example, the model might first call `route_query` with a broad query to get high-level sections, then ask follow-up sub-queries on specific sections. This resembles a *sub-question query engine* or multi-step RAG approach [10]. **Open-source precedent:** Multi-step retrieval planning appears in frameworks like

LlamaIndex (which supports query routers and sub-question engines [10]) and in research on graph-augmented retrieval. In fact, the community has found that augmenting RAG with a knowledge graph (GraphRAG) can stabilize outputs on large-scale docs by guiding the selection of relevant pieces [9]. Our chunk router tool can draw on these ideas – e.g. using the repository's file structure or a call graph as a "map" to decide where to look next. This is essentially a form of **model-driven recursive retrieval**, analogous to how ReDel's recursive agents decompose tasks into sub-tasks for specialized handling [11] (in our case, the sub-task is "find info on X in the context").

- **Chain-of-Thought Evaluator (Self-Reflective Tool)** – *Function:* Provide a mechanism for the LLM to **evaluate or debug its own intermediate reasoning**. RLM allows the model to improve itself by iterative refinement; this tool formalizes that by letting the model explicitly analyze its chain-of-thought or a candidate answer. Implementation could be as simple as a function `reflect(thoughts: str) -> str` that returns a critique or list of errors in a reasoning trace. The model might call `reflect()` on its current plan or answer, prompting it (as a sub-query) to find flaws or missing pieces, which it can then address in a revised approach. **Open-source precedent:** The *Reflexion* framework (Shinn et al. 2023) introduced exactly this idea of an agent reflecting on mistakes and refining its approach [12]. In practice, Reflexion has been used to let agents correct errors in code or logic by analyzing their own outputs [12]. Another example is the *self-critiquing* mode of some agents which produce a "critic" analysis as an intermediate step. By incorporating a `reflect` tool, the LLMC agent can, for instance, generate a solution, then call `reflect(solution)` (which internally may call an LLM prompt like "Here is my solution, what might be wrong or could be improved?"). The result helps the agent adjust its chain-of-thought, leading to a self-improving loop. This tool will be especially useful for complex tasks (e.g. code generation or multi-hop reasoning) where verifying intermediate results can prevent compounding errors.

- **REPL State Inspection Tool** – *Function:* Allow the model to query the state of variables or the environment in the recursive REPL. For example, after loading `context` (which might be a huge string or a parsed object) and doing some filtering, the model might want to see metadata about `context` or preview a slice of it. We propose a utility like `inspect(var_name: str, slice: Optional[str] = None) -> str` that the model can call to get information about a given variable in the Python exec environment. If `slice` is provided, it could specify an aspect (e.g. `length`, `head`, `tail`, or a key if the context is a dict). *Rationale:* This is akin to a programmer printing variables while debugging. It gives the LLM transparency into what data it's holding out-of-context. For instance, `inspect("context", "length")` might return the number of tokens or characters in the context, so the model knows how large the data is. Or `inspect("filtered_lines")` could show a few sample lines that were extracted. **Open-source precedent:** While not a standalone library, this concept fits into the pattern of giving LLMs tools to manage long contexts. The RLM paper itself treats the prompt as an external variable and allows the model to programmatically manipulate it [1]. Implementing an inspection tool is straightforward (essentially reading a Python variable and converting to a safe string representation). We just must ensure proper sanitization to avoid flooding the LLM with too much data if it mistakenly tries to print a huge variable. This tool, combined with summarizers, helps the model "sense-check" the results of any filtering or chunking code it wrote.

- **Code Symbol Graph Navigator** – *Function:* Specialized for repository comprehension, this tool lets the model query a **graph of code symbols and their relationships** (functions, classes, imports, call hierarchy, etc.). LLMC's indexing already builds a knowledge graph of the repo (GraphRAG in `llmc/graph/*`), so we expose that via a tool. For example, `find_symbol_references(name: str) -> List[Location]` or `query_graph(query: str) -> str` where the query can be natural language. This way, the LLM can ask questions like "Which functions call `UserService.create_user`?" or "Show classes that implement interface X." and get structured answers. **Open-source precedent:** There are emerging tools exactly in this vein. For instance, the *Code Graph RAG* project provides an MCP server with tools such as `query_code_graph` to answer questions about code relationships [13]. A call to `query_code_graph("What functions call UserService.create_user?")` would return the list of functions and locations in the code [14]. By integrating a similar tool, the LLMC agent can perform *on-the-fly static analysis* of the repository. The agent might recursively use this: e.g. break a question "Explain how module A interacts with module B" into sub-tasks of retrieving the call graph between A and B, then summarizing those findings. The tool can interface with LLMC's existing graph DB (or an external graph database like Memgraph as in Code Graph RAG) to retrieve the needed info. This greatly enhances the LLM's ability to **comprehend large codebases**, since it's not blindly reading code files token-by-token, but rather leveraging a pre-computed graph of code semantics.

- **Retrieval Inspector Tool** – *Function:* (Optional but useful) Provide transparency into the RAG process itself. This could output metadata like "what documents/chunks were retrieved and with what scores?". For example, `inspect_retrieval(query: str) -> List[Tuple[DocID, Score, Excerpt]]`. If the agent formulates a sub-query and calls RAG search internally (via `route_query` or even the built-in LLMC search), it might want to verify the results. The retrieval inspector can be a diagnostic tool where the model can double-check whether it has the right context. If the results seem irrelevant (scores low, or content doesn't match query intent), the model might decide to reformulate the query or try a different approach (akin to **self-ask with search**). **Precedent:** This idea aligns with the concept of *interactive retrieval*, where an LLM iteratively searches and verifies. While we don't have a direct open library for "retrieval inspection by the model," the need for it is evidenced in complex QA systems and tools like AgentDSP which let a model evaluate if a retrieved snippet answers the question. Implementing it is straightforward since LLMC's RAG already computes scores; we just expose those to the model in a controlled way (perhaps truncating text). Error handling will ensure the model cannot abuse this to retrieve *everything* (it must specify a query).

All these abstractions will be implemented as callable functions or methods in the sandbox environment that the LLM can invoke when in recursive mode. To minimize prompt tokens, we might not list all tools upfront; instead, we'll use a technique similar to Anthropic's *dynamic tool discovery*. For example, provide a filesystem-like interface (`.llmc/stubs/`) where the LLM can `import` a tool on demand. In code execution mode, Claude was shown a directory of tool stubs and could choose which to import and call [15] [16]. We can follow that pattern for our new tools: the LLM sees, say, a `tools/` package with `summarize.py`, `route.py`, `reflect.py`, etc., each with docstrings describing usage. This way, the model only loads what it needs, keeping token usage lean.

Recursive Reasoning Workflow in LLMC

This section outlines the expected **flow of a recursive task** using the above components, from the user's query through sub-agent calls and back to a final answer. It will clarify how the agent, tools, and RAG context interplay in an RLM-enhanced LLMC:

1. **User Query Initiation:** A user poses a complex query – e.g. "*Summarize the design of the Foobar module and how it interacts with the rest of the system*" – which likely cannot fit entirely in the prompt if the repository is large. The LLMC agent recognizes (via configuration or heuristics) that this query could benefit from RLM mode (perhaps because the repo or context size >> model context length, or a flag in `llmc.toml` enables recursive reasoning for certain queries).

2. **Context Preparation:** The agent gathers initial context for the query. In vanilla LLMC, this would be RAG: retrieving the top *N* relevant chunks from the index. In RLM mode, instead of forming a single flat prompt, the agent will **load the entire relevant dataset into the external environment**. For example, it may load *all* repository files (or a very large subset like all code in `Foobar/` directory) into a Python variable `context`. (If it's extremely large, an alternative is storing references and providing functions to fetch pieces on demand – but for Phase 1, we might assume it's at least chunkable into a few big parts that can be loaded one at a time or so.) The agent starts the MCP sandbox, and executes something like `context = "<all relevant text or a pointer to it>"` so that the variable is ready for the model's code. It also provides metadata, e.g., `context_summary = {"total_tokens": X, "num_files": Y, ...}` for the model's convenience, and perhaps pre-imports our tool functions.

3. **Initial Prompt to the LLM (Root Agent):** The agent now crafts a system prompt indicating that the model can use the Python environment to solve the task. This prompt might say (in natural language) instructions akin to: *"You have access to a Python environment with the full repository text stored in the variable* `context`*. You can write Python code to analyze this context. You have tools like* `summarize`*,* `llm_query`*, etc., available to help. Your goal is to produce the final answer and print it or return it via* `FINAL_VAR` *when done."* We also include *few-shot examples* if available, demonstrating how to call a sub-LLM via `llm_query` and how to mark the final answer (the RLM paper uses `FINAL()` or returning a special variable [17] [5]). This step is crucial: it sets the stage for the model to behave like a *controller that can write code* rather than directly output text. The agent then yields control to the LLM (calls the model API) to get its response.

4. **Model's Recursive Reasoning (Code Generation):** The LLM (now the "root" in RLM terms) receives the prompt and will ideally begin generating a plan in the form of code. For example, it might output Python pseudocode like:

```python
# Plan: Identify Foobar module files, summarize each, then find relations
import json
files = [f for f in context.split("FILE:") if "Foobar" in f]  #
hypothetical splitting by file marker
summaries = []
for file in files:
    text = get_file_content(file)  # assume context is structured or has a
```

```
helper
    summaries.append(summarize(text, max_len=1000))
combined = "\n".join(summaries)
answer =
llm_query(f"Using the following summaries, explain Foobar's design and its
interactions:\n{combined}")
FINAL_VAR = answer
```

This is a made-up example, but it shows the pattern: the model decomposes the task into manageable pieces – summarizing parts of the context, then asking a sub-LLM (via `llm_query`) a high-level question on those summaries. During this process, it may also use other tools: e.g., if it needed to find which files relate to Foobar, it could call `route_query("Foobar module architecture?", context_index)`. Or it might use the code graph: `graph_results = query_code_graph("Foobar <-> other modules")` to get interactions, then incorporate that into its reasoning. The key is that the *model decides the sequence of operations*, writing code that invokes our abstractions. This code is output token-by-token to the LLMC agent.

5. **Tool Execution via MCP:** As the agent receives the model's code output, it executes it step by step in the MCP sandbox. This is where the isolation and guardrails of MCP come in [6] – we ensure the code is safe (no unauthorized operations, timeouts on long loops, etc.). Execution yields results: e.g., the list of `summaries` gets populated. If the model calls `llm_query(...)`, that function will trigger a *nested call* to an LLM backend (possibly the same model or a smaller one specialized for summarization). The MCP can handle this by pausing the root's execution, passing the prompt string to the agent or directly to a model API, and then returning the result back into the sandbox as the function's return value. This essentially creates a **sub-agent** on the fly. The integration here must handle multiple nested `llm_query` calls serially (concurrency can be considered later). Each sub-query might itself use RAG (e.g., if the snippet triggers another retrieval) but ideally the snippet is small enough for a direct LLM answer. During execution, any errors are caught and returned to the model as exceptions or error messages (so the model can adjust if, say, it tried to call a tool incorrectly). This forms a feedback loop: if the code fails (perhaps the model assumed a variable that doesn't exist or tried to open a file incorrectly), the exception is captured as a Python error, and the agent can feed that back into the LLM's context (the model can then try a different approach in the next iteration of code, akin to self-healing codegen).

6. **Recursive Refinement:** The root LLM may go through several cycles of generating code, seeing the execution outcome, and continuing. For instance, after getting summaries, it did a sub-query and stored an `answer`. The agent might see that the model hasn't explicitly finalized output yet (maybe it didn't set `FINAL_VAR`). The agent would then feed back something like: *"Your code executed successfully. You can continue coding or output the final result. (Result of llm_query stored in* `answer` *.)"* – effectively prompting the model to decide if it's done. The model could then choose to, say, inspect or post-process the answer. Perhaps it calls `reflect(answer)` to double-check correctness, and if the reflection says "some details are missing," the model might loop: maybe it finds a missed piece in `context` and calls another `llm_query` or does a different summarization. This step could iterate a few times (we need to allow loop, but also have safeguards to avoid infinite loops).

7. **Final Answer Output:** Eventually, the model should indicate it's ready to present the answer. In the RLM scheme, they used either printing the answer or assigning a special variable, which we'll detect. Suppose the model does `FINAL_VAR = answer` or calls a provided `FINAL(answer)`. The MCP captures that, and the agent knows the task is complete. The final answer (a concise explanation of the Foobar module's design in our example) is then returned to the user as the assistant's message. The agent may also include some provenance info if needed (since the answer was derived via external context, perhaps we attach source references if the chain-of-thought tracked them – this could integrate with LLMC's citation mechanism, though that is an extension).

Throughout this flow, **the RAG context is used on-demand rather than in one big prompt**. The extended context (repository) lives in `context` and is navigated with tools, which is exactly the point of RLM: *process data in code, not in prompt*. As one practitioner put it, *"the insight is to let the AI write code to fetch data, process it locally (outside the model's context), and return only a summary"*, reducing millions of context tokens to just thousands [18]. Our design follows that insight closely.

Tool Interfaces and Error Handling

Designing clear I/O schemas for the tools and robust error handling is essential for a smooth recursive agent operation. We outline how each tool's interface will look and how errors are handled:

- **Tool I/O Schema:** We will define each tool's function signature with JSON-serializable inputs/outputs where possible. For example, `llm_query(prompt: str) -> str` will take a prompt string and return the model's text completion. Tools like `query_code_graph` might return a Python dict or list (which the LLM can then pretty-print or further process). To keep things consistent and safe for the LLM, we might have the tools always return *JSON strings* (so the model gets a string it can parse with Python's `json.loads` if needed). This ensures the LLM isn't confused by Python object representations. For instance, `query_code_graph("...")` could return a JSON string like `'[{"function": "Foo", "calls": ["Bar","Baz"]}, ...]'`. The LLM then can `json.loads()` it and reason about it. We will document these schemas in the tool docstrings that the model sees.

- **Error Handling:** When the model's code triggers errors, the MCP should catch exceptions and present them in a controlled manner. Likely, we will catch any Python exception and convert it to a result like `error: RuntimeError("X")` that the LLM can read. One approach (used in some code execution agents) is to capture stderr/stdout and provide them to the model [19]. For example, if a tool is called with wrong arguments, we can have the tool function return an error message instead of throwing (since the LLM won't see a Python traceback unless we feed it back). Alternatively, we actually allow the exception to halt execution and then wrap the error in a special object that the agent includes in the next model prompt. For Phase 1, a simpler path: if an error occurs, stop execution and show the model a console output like "Error: NameError – `get_file_content` is not defined". The model, seeing that, can adjust (maybe it meant to use a different method). We'll include few-shot examples in the prompt of how to handle errors (perhaps an example where the LLM tries something, gets an error, then fixes its code).

- **Guards:** The MCP already provides isolation (e.g., no internet, limited filesystem) [6]. We will add RLM-specific guards such as preventing the model from dumping the entire `context` variable to output (which would circumvent the context window limit by just printing everything – we must forbid that). If the model tries to print a huge string, we can truncate it and warn. We might also enforce that `llm_query` calls have a prompt length below a threshold, to avoid the model recursively sending massive prompts (which could be expensive or hit API limits). Essentially, each recursive call should itself deal with a manageable summary or piece.

- **Distinguishing Final Answers vs. Thoughts:** One noted challenge from the RLM paper is ensuring the model cleanly indicates which output is the final answer versus intermediate reasoning [5]. We will address this by our interface: requiring the model to call a finalization function or variable. By construction, when the model does that, no further code will be executed. The agent will treat any print to stdout before finalization as intermediate (and likely not show it to the end user, unless for logging). Only the content passed to `FINAL_VAR` or printed after a `FINAL()` marker would be relayed as answer. This clear contract avoids the ambiguity of the model mixing "thoughts" with the answer. In case the model fails to use the protocol (e.g. it answers directly in natural language instead of writing code), the agent can fall back: it would interpret a direct answer as just a normal answer (maybe the task was simple enough that no recursion was needed after all). But ideally, with the system prompt guidelines, the model will know it's supposed to produce code first.

Overall, the tool interfaces are designed to be as **model-friendly** as possible: simple function signatures, mostly string I/O, and behavior that the LLM can predict from the documentation (many of these tools mirror things the model might conceptually know – e.g., summarization or search).

Compatibility with Target Models

We plan our RLM integration to be *model-agnostic*, but practical adjustments will ensure it works optimally across different LLM backends used in LLMC (LLaMA-derived, OpenAI, Anthropic, etc.). Below we discuss compatibility considerations for each target model family:

- **LLaMA (GLM 4.5, 4.7)** – These appear to be local models (possibly fine-tuned generative language models in the LLMC context). They likely have moderate context windows (4k-16k tokens) and some coding ability. For LLaMA-based models, a key point is that they might not follow complex instructions as reliably as GPT-4/Claude, especially in multi-step tool use. We may need to employ few-shot examples demonstrating the RLM style. If GLM 4.5/4.7 have been instruction-tuned on code, they could perform well at writing Python to use our tools. We should test with smaller tasks first. We also might limit recursion depth for these models to avoid confusion. Performance-wise, local models might be slower or have smaller context, so the **Summarization Tool** is crucial to condense context before making subcalls. In compatibility terms, we ensure our sandbox and `llm_query` can call out to whichever local model or API GLM corresponds to (possibly via an Ollama or FastAPI if they are served locally). The design does not assume any proprietary features, so LLaMA should handle it given proper prompting. If needed, we can introduce a special prompt template for LLaMA focusing more on step-by-step guidance.

- **MiniMax 2.1** – MiniMax is likely an API-driven model service (possibly similar to OpenAI's API, but from a different provider). We will treat it akin to OpenAI GPT-3.5 in terms of capabilities. Compatibility considerations: it may not have the code execution pattern "built-in", so prompting will be critical. We'll use the same system message approach describing the environment and tools. If MiniMax supports function calling or a tool plugin system, we might leverage that instead of pure code (but given our architecture, we prefer the model generates code to call `llm_query` and others, which is model-agnostic). We may need to watch out for output length limits on MiniMax's side for sub-queries. Also, if MiniMax has any specific tokenization quirks (especially if multilingual), we ensure the `context` variable is encoded in a safe format (UTF-8 text). We'll test with their 2.1 version to see if any adjustments to style (e.g., more direct instructions vs. open-ended) are needed. The Phase 1 design keeps things general, so presumably MiniMax can comply with "you are an agent with a Python REPL" instructions similarly to others.

- **OpenAI (GPT-4/GPT-3.5)** – These models are very capable with instructions and code, but they have some differences: OpenAI's models have *function calling* features and guardrails. GPT-4 can certainly produce Python code to use tools (OpenAI's own evaluators do something similar). One challenge: OpenAI APIs do not *natively* allow the model to execute code – that's why we have our own MCP. But GPT-4 can be guided to output well-structured code which we then execute. We must be mindful of the token limits (GPT-4 currently ~8k or 32k context). If our `context` is extremely large, even listing filenames could approach limits. So for OpenAI, we might rely more on pre-filtering outside the model. For instance, rather than dumping raw `context` into the variable, perhaps list the file names and let the model `open_file(name)` as needed via a tool (we could implement `get_file_content(name)` safely). This way GPT-4 doesn't see giant strings until it explicitly requests a snippet. Additionally, OpenAI's content filters might react to code that looks like it's reading sensitive data; we should ensure the system prompt clarifies this is a controlled environment. We'll test the code execution prompt thoroughly. The design should work on GPT-4, as it's been demonstrated that GPT-4 can effectively use a code tool approach (e.g., similar ideas are used in OpenAI Functions and in tools like "AutoGen" by Microsoft for multi-step reasoning). One more note: If using GPT-3.5 (which is weaker in following complex instructions), we may restrict RLM mode to GPT-4 and above for reliability, or use GPT-3.5 only for simple summarization subcalls under supervision of a more capable model.

- **Google Gemini** – While details on Gemini (especially specific versions like "Gemini 2.0 flash-thinking exp" as seen in some references [20] [21]) are scarce, it's expected to be a cutting-edge model from Google with strong reasoning and possibly tool-use abilities. Gemini will likely support **multi-modal and extended context** better. For our design, we assume it can handle code execution instructions similarly. If Gemini has an API with *tool specification* (perhaps via Google's PaLM API functions), we might in future integrate directly. But Phase 1 will treat it like any other: we give it the sandbox prompt. Given Google's focus, Gemini might excel at reading code and could perhaps ingest more of the repository without summarizing as aggressively. Still, using RLM with it would be beneficial for structuring the problem. We should ensure compatibility by testing Python code generation; Gemini should be good at code (Google's models like PaLM 2 are, and Gemini is expected to continue that). One consideration: if Gemini has larger context (say 100k), one might question if RLM is needed – but as RLM paper shows, beyond just context length, it's about **structured reasoning**. We'll use RLM mode if the task complexity demands it, even if Gemini could theoretically read it all, because dividing the task can lead to better quality results.

- **Anthropic Claude (Sonnet 2, etc.)** – Anthropic's Claude models (Claude 2, and anecdotally "Claude Sonnet" versions) are actually the inspiration for the code execution approach. Claude's "Code Mode" (as in Claude 2's ability to write and execute code via an internal tool) aligns perfectly with our plan. In fact, LLMC's code execution support was likely built with Claude in mind. So Sonnet (Claude 2.1 or 3.0 possibly by 2026) should be our easiest target. It likely already *expects* that if connected to an MCP, it might get to write code. As a result, we might give Claude a slightly different prompt: more concise, since it might already know the pattern. Anthropic's article on Code-Execution-with-MCP highlights up to **98% token reduction** by letting the model use code instead of seeing tool outputs directly – Claude will take advantage of that. We just need to ensure our tool stubs in `.llmc/stubs` are accessible, as Claude might try to browse them (Anthropic's pattern was to have the model browse a virtual file system for tools [22] [23]). Our integration can simply list available tools in a stub directory and let Claude import. For other models we might explicitly list tool APIs in the prompt instead. So the difference is mostly in prompt formatting; the architecture (agent + MCP) remains the same. We will, of course, test with Claude to refine any prompt or formatting issues (Claude might be more flexible with slight deviations).

In summary, the HLD is built to be model-agnostic: the *RLM capabilities are provided by LLMC's infrastructure (agent, tools, MCP)* rather than any proprietary feature of a specific model. The models just need to follow instructions to utilize those capabilities. Where models differ (e.g., coding skill, context size), we adjust the prompt and maybe the degree of autonomy given. A less capable model might need more step-by-step guidance ("First, call `route_query`... then summarize...") or even a fallback to a simpler RAG if it fails. A more capable model can be allowed to make bigger recursive plans by itself.

Architectural Summary and Next Steps

In this Phase 1 High-Level Design, we described **how RLM can weave into LLMC's existing subsystems** to enable recursive reasoning over extended contexts. To recap:

- We will augment the **LLMC agent** to recognize when to enter RLM mode and to parse/execute model-generated code for recursion.
- We will extend the **LLMC MCP** with a persistent REPL environment and new tools (`llm_query`, `summarize`, etc.) that the model can call for handling large texts and complex tasks.
- We proposed a set of **tool abstractions** (summarizer, chunk router, self-reflector, context inspector, code graph navigator, etc.) inspired by open implementations like hierarchical summarizers [8], graph-based retrieval systems [9], and self-refinement agents [12]. These tools empower the LLM to break down tasks and query context in a smart way, effectively acting as its own sub-agents.
- We walked through an **example flow** of a recursive query, illustrating how the agent and LLM interact through code execution and sub-LLM calls to gradually build an answer. The design emphasizes minimal context usage by offloading work to code (consistent with Anthropic's 98% token reduction via this method [15] [18]).
- We specified **tool I/O schemas** (leaning on JSON and simple strings) and error-handling strategies to ensure the system is robust. The model will be guided to clearly signal final answers vs. intermediate steps, avoiding confusion.
- We addressed **multi-model compatibility**, noting how each target model (LLaMA variants, MiniMax, OpenAI GPT, Google Gemini, Anthropic Claude) will be accommodated without changing the core logic – mainly through prompt tuning and leveraging known strengths (e.g., Claude's familiarity with code tools, or using summarization more heavily for smaller local models).

This high-level design sets the stage for implementation. **Next steps (Phase 2)** would include creating detailed specifications for each new component (e.g., exact function signatures for tools, modifications to `agent.py` state machine), prototyping with one model (likely Claude or GPT-4 for initial ease), and writing tests on long documents to ensure the RLM loop works as expected. We would also consult LLMC's `DOCS/`

(design docs) and possibly implement a small example in the `tests/` suite (like a dummy "large text question" that the new system can answer by chunking and sub-calling). By doing so incrementally, we can validate the approach and measure improvements (e.g., able to handle 500-page repo docs that were previously infeasible). Over time, this RLM integration will make LLMC far more adept at extended context reasoning – *letting it self-improve by decomposing big problems into solvable pieces*, just as the RLM paper envisioned [1] [11].

Sources:

- LLMC Repository & Notes – *Architecture and existing capabilities* [3] [4]
- "Recursive Language Models Can Self-Improve" – *RLM concept and observations* [1] [2]
- Anthropic Code Execution (MCP) – *Efficient tool use via code; inspiration for our design* [18]
- Open Implementation References – *Hierarchical summarization and graph-based RAG for long contexts* [8] [9]
- Agent Self-Reflection Research – *Reflexion framework for iterative self-improvement* [12]
- Code Graph Tools – *Example of querying code relationships via an MCP tool* [14]
- Recursive Agents (ReDel) – *Academic toolkit for task decomposition with sub-agents* [11]

---

[1] [2] [3] [5] [7] [17] /mnt/data/RLM_paper_notes_2512.24601v1.md
file://file_00000000b7ac7206b189f44c78fbdb38

[4] [6] /mnt/data/LLMC_repo_intake_2026-01-24.md
file://file_000000007d847209a8abe437bef85d2e

[8] [10] LlamaIndex | RAGBible
https://ragbible.com/projects/llamaindex/

[9] RAG is broken without graphs. GraphRAG + Claude Code CLI is the future. Change my mind ! : r/ClaudeAI
https://www.reddit.com/r/ClaudeAI/comments/1n7e3rx/rag_is_broken_without_graphs_graphrag_claude_code/

[11] ReDel: A Toolkit for LLM-Powered Recursive Multi-Agent Systems
https://arxiv.org/html/2408.02248v1

[12] Comprehensive Guide to Chain-of-Thought Prompting
https://www.mercity.ai/blog-post/guide-to-chain-of-thought-prompting

[13] [14] [20] [21] GitHub - vitali87/code-graph-rag: The ultimate RAG for your monorepo. Query, understand, and edit multi-language codebases with the power of AI and knowledge graphs
https://github.com/vitali87/code-graph-rag

[15] [16] [18] [22] [23] Dramatically Reducing AI Agent Token Usage with MCP Code Execution | by Shamsul Arefin | Medium
https://medium.com/@shamsul.arefin/building-an-ai-agent-with-mcp-code-execution-from-confusion-to-clarity-6b13fccc8c4b

[19] GitHub - shams858/code-execution-with-mcp: Langchain agent that generates and executes Python code to interact with Airtable using the MCP code execution pattern from [Anthropic's blog](https://www.anthropic.com/engineering/code-execution-with-mcp).
https://github.com/shams858/code-execution-with-mcp