ChatGPT

Plan for an LLM-Driven Iterative Essay Writing App (AlphaEvolve-Inspired)

We propose an **evolutionary writing pipeline** that mirrors DeepMind's AlphaEvolve: use LLMs to iteratively improve essay drafts. AlphaEvolve "orchestrates an autonomous pipeline of LLMs" that propose code edits and uses evaluators to score them [1]. In our case, the LLMs propose textual edits. In fact, an open-source "AlphaEvolve Writing" system already demonstrates this concept: it generates, evaluates, and evolves creative or general writing pieces through iterative AI competitions [2]. Our app will follow a similar loop: generate drafts, have LLMs suggest revisions, score the results, and repeat.

Evolutionary Pipeline: Our system's core modules include a prompt sampler, an LLM ensemble, and evaluators – much like AlphaEvolve's loop [3]. We start by creating an initial batch of essay drafts (e.g. 5–10) from the user's prompt. We then enter the loop: the LLMs receive prompts (possibly including the current best draft and a revision instruction) and output *diffs* or edits. These edits are applied to produce new essay variants. Next, automated evaluators score each variant on criteria such as coherence, grammar, style or factual accuracy. Promising essays (the "top performers") are added to a draft database. This mirrors AlphaEvolve's process where LLM-generated code diffs are "scored by Evaluators" and the best solutions drive further refinement [3]. Over multiple generations, the essays should steadily improve.

- **Initial Draft Generation:** Use one or more LLMs (e.g. GPT-4, Claude) to generate multiple complete essays from the prompt. This can be done with a "prompt sampler" that constructs varied instructions or contexts. The AlphaEvolveWriting project does exactly this: it "creates multiple stories from your prompt" using configurable AI models [4].
- **Evaluation and Ranking:** Define an evaluation rubric (clarity, correctness, style, etc.) and have each draft scored. We could use LLM-based judges (prompt the model to rate or compare essays) or external tools (grammar checkers, plagiarism detectors, readability metrics). For example, AlphaEvolveWriting runs pairwise AI-judged "ELO tournaments" to rank stories [5]; a similar tournament could compare drafts in pairs to update Elo scores. At minimum, compute a score for each essay on the rubric and rank them.
- **Selection and Variation:** Select the top K essays (e.g. top 3) as "parents." For each, prompt the LLM to produce one or more *variants*. Instead of rewriting from scratch, you can instruct the model to output edits or a diff (e.g. "Improve this paragraph's structure" or "Replace passive voice with active voice"). AlphaEvolve's code agent specifically asks the model for diffs which are then applied to the program [3]. We adapt this: ask the LLM to suggest specific changes to the text. Apply those changes to get child drafts.
- **Repeat the Loop:** Add the new drafts to the pool and repeat the evaluation-selection cycle. Continue for a fixed number of iterations or until performance plateaus. As AlphaEvolve's authors note, this "iterative process evolves increasingly better writing over time" [6]. We should track and store versions of drafts so we can review how the essay improved.

Modular Modes and Settings

The system will be **highly modular**. We can support multiple modes by changing prompts, models, and evaluation rules. For example, include a *creative mode* (focus on narrative flair) versus a *formal mode* (focus on argument clarity). In the AlphaEvolveWriting code, there are exactly two modes: "creative

storytelling" and "general writing" [7]. In creative mode, prompts encourage literary techniques (e.g. "explore complex emotional landscapes" or emulate an author's style [8]), while in formal mode the prompts emphasize structure and evidence ("clear thesis, logical flow" [9]). We can even add a third mode (e.g. summarization or technical writing) by defining new prompt templates and rubrics.

We should also use a **multi-model ensemble** for variety. AlphaEvolve uses one fast model for breadth and one powerful model for depth [10]. Similarly, we might use a smaller/cheaper model (for quick drafts or many edits) and a larger model (for deep editing suggestions). The open-source implementation allows configuring models via JSON [11] [3]. Our app will read a config file (or CLI flags) to choose models and parameters, making it easy to switch between modes or experiment.

Incorporating an MCP Server

We will integrate a **Model Context Protocol (MCP) server** to extend the LLM's capabilities. MCP is an open protocol that standardizes how LLMs access external tools and data [12] [13]. In practice, we run a local or cloud server that exposes specialized "tools" the LLM can call via JSON. For our writing app, a useful MCP server use-case is **external knowledge retrieval**.

For instance, we could run an **Academic Search MCP server** (like the one for Claude) that connects to Semantic Scholar or CrossRef. This server offers tools such as `search_papers` or `fetch_paper_details`. When the essay draft mentions a topic, the LLM can query this MCP tool to get real paper titles, abstracts, and metadata [14]. For example, asking the MCP server "search_papers("climate change mitigation", 5)" might return key recent papers. The LLM can then incorporate accurate references or facts into the essay. The referenced Academic Search server explicitly "provides LLMs with real-time academic paper search, paper metadata and abstracts, [and] ability to retrieve full-text content" [15]. Embedding this into our pipeline means each draft can be fact-checked or enriched by on-demand scholarship.

Another MCP use-case is **live web search**. An MCP like the "Brave Search" server lets the LLM call an API to perform web searches [16]. This would be useful to ground the essay in up-to-date information (e.g. current events or statistics) that the LLM's training data might miss. In general, an MCP server "adds context and capabilities" by exposing functions to the model [13]. We would configure our LLM with the MCP so that, for example, if the draft includes a statement like "According to X," the LLM might invoke a search tool to fetch a source.

Beyond search, we could create custom MCP servers for grammar checking or synonym lookup, but existing LLM prompts might suffice for those. The key learning benefit is to practice building an MCP server. By picking a concrete example (e.g. academic papers or web search), we make the integration meaningful: the LLM learns to use tool calls to improve content.

Example Workflow

A high-level implementation plan is:

1. **Setup:** Write a Python (or Node) service for the MCP server (using an MCP SDK [12]). For example, use the academic-search MCP repository as a template. Configure environment variables for any API keys (Google Scholar API, etc.).
2. **Initial Drafts:** Use the OpenAI (or Anthropic) API to generate N essays on a given prompt. Store them with initial scores (e.g. Elo=1500).
3. **Iterative Loop (configurable "max_generations"):**

4. Use an LLM to run evaluation prompts on all drafts (or pairwise compare them) and update their scores [5].
5. Select the top K drafts. For each, call the LLM again with an "edit" prompt (the prompt can say e.g. "Improve the introduction of this essay" or include a rubric) to create M variants. (Encourage outputting specific edits or a rewrite.)
6. Score the new variants. Keep them and possibly the originals in the pool (as AlphaEvolveWriting does if desired [17]).
7. **Modes and Customization:** Have a config switch (e.g. `--creative` or `--general`) that changes the LLM prompts and evaluation criteria. For example, creative mode might give the LLM style guidelines (e.g. "write like Virginia Woolf" [8]), while general mode might emphasize thesis structure [9].
8. **MCP Integration:** During generation or evaluation, allow the LLM to call the MCP server. For example, in the prompt you can instruct the model to use the tool (depending on model support for function calls). The MCP server then returns data (e.g. a list of papers or search results) that the LLM can incorporate.
9. **Modularity:** Structure the code so that each component (prompt generator, LLM caller, evaluator, MCP client) is separate. Use a configuration file (JSON) as in AlphaEvolveWriting [18] to specify models, number of generations, etc. This makes the system extensible and testable.

Throughout, we would continuously test with simple prompts. For example, try writing a historical essay: generate drafts, then see how the MCP-backed model adds citations. Adjust the evaluation rubric or model prompts based on results.

Conclusion

This plan combines AlphaEvolve's iterative LLM pipeline [1] [3] with practical writing goals. By structuring the application into modular stages (generation, evaluation, revision) and supporting multiple modes, we maintain flexibility. Integrating an MCP server gives our LLM real-time tools (like academic search [14] or web queries [16]) to enhance factual accuracy. Over several evolution cycles, the system should yield increasingly polished essays – for example, improving each draft by applying LLM-suggested edits and scoring them with rich rubrics [4] [3]. In summary, we implement an *evolutionary essay writer* with configurable settings and a built-in MCP knowledge tool to learn all parts of the pipeline.

Sources: We build on the AlphaEvolve framework [1] [3] and its adaptation to writing [2] [4]. The MCP use-case follows the Model Context Protocol documentation [12] [13] and examples of academic search servers [14] [16].

---

[1] [3] storage.googleapis.com
https://storage.googleapis.com/deepmind-media/DeepMind.com/Blog/alphaevolve-a-gemini-powered-coding-agent-for-designing-advanced-algorithms/AlphaEvolve.pdf

[2] [4] [5] [6] [7] [8] [9] [11] [17] [18] GitHub - tamassimonds/AlphaEvolveWriting
https://github.com/tamassimonds/AlphaEvolveWriting

[10] AlphaEvolve: A Gemini-powered coding agent for designing advanced algorithms - Google DeepMind
https://deepmind.google/discover/blog/alphaevolve-a-gemini-powered-coding-agent-for-designing-advanced-algorithms/

[12] Introduction - Model Context Protocol
https://modelcontextprotocol.io/docs/getting-started/intro

13 What Is the Model Context Protocol (MCP) and How It Works
https://www.descope.com/learn/post/mcp

14 15 GitHub - afrise/academic-search-mcp-server: Academic Paper Search MCP Server for Claude Desktop integration. Allows Claude to access data from Semantic Scholar and Crossref.
https://github.com/afrise/academic-search-mcp-server

16 Example Servers - Model Context Protocol
https://modelcontextprotocol.io/examples