ChatGPT

Plan to Integrate Darwin Gödel Machine (DGM) and ITRS Features into LaTeXify

DGM Core Components and Mechanisms

- **Self-Modifying Agent Codebase:** At the heart of DGM is a coding agent that can **read and modify its own Python code** using a large language model (LLM) [1]. The agent's code (initially a baseline version) is fed to a frozen code-oriented LLM which proposes improvements (code changes) to optimize performance [2]. This enables the system to **rewrite any part of its logic**, effectively **improving its ability to improve itself** over time [3].

- **Tool Integration for Code Execution:** The agent is equipped with tools to interact with its environment, such as executing code, running tests, reading/writing files, and issuing shell commands [4]. By leveraging these tools, the agent can **validate its code changes** (e.g. run unit tests on updated code) and gather feedback. Using a container or sandbox (e.g. Docker) to run model-generated code is critical for safety [5], ensuring that untrusted code modifications do not harm the host system.

- **Empirical Performance Evaluation:** After each self-modification, the new code is **evaluated on benchmark tasks** to measure performance [6]. In DGM's case, this involves coding benchmarks like SWE-Bench and Polyglot – e.g. solving coding challenges or fixing bugs and checking if tests pass [7]. The agent's **fitness score** is the fraction of tasks solved. If a code change **improves the success rate**, it is considered a beneficial mutation [8]. This empirical validation loop answers: *Did the self-edit make the agent better at its task?* [6].

- **Archive of Agents (Open-Ended Evolution):** DGM maintains an **archive of all agent versions** instead of just the latest one [9]. Each iteration, an agent from the archive is selected as a "parent" to produce a modified "child" variant. Higher-performing agents are more likely to be picked, but even lower-performing ones occasionally get chosen to preserve diversity [10]. This archive acts like a **gene pool**, enabling **parallel exploration** of multiple design trajectories rather than a single hill-climbing path [11] [12]. The system **does not discard weaker agents outright**, since even a suboptimal version might contain a novel idea (a "stepping stone") that later leads to a breakthrough when combined with other changes [13].

- **Iterative Self-Improvement Loop:** Putting it together, the DGM runs a loop: **Select an agent from the archive, have the LLM propose a code patch for it, apply the patch, then evaluate the new agent on tasks** [14] [7]. If the new agent's performance is as good or better than its parent (or offers a novel capability), add it to the archive [15]. Repeat this process for many generations. Over many iterations, agents accumulate significant improvements, and the **best agents in the archive show dramatically higher performance** than the seed agent [16] [17]. This loop embodies Gödelian self-reference (the agent modifies itself) combined with Darwinian selection (better variants are retained) [18].

Key Self-Improvement Features from DGM

- **Patch Validation Step:** The agent automatically **validates proposed code fixes by running tests before considering a task solved** [19]. (DGM discovered this on its own: after generating a code solution, it executes the project's test suite to verify the fix, rather than assuming the solution is correct.) Implementing this **"test-before-commit" step** ensures that each code change is vetted, reducing the chance of introducing regressions [19].

- **Enhanced Code Editing Tools:** Initially, the agent could only replace entire files when modifying code. DGM evolved finer-grained editing capabilities – e.g. the ability to **search and replace specific functions or strings within a file** [20]. In our plan, we will implement such **surgical editing tools** up front: the agent should be able to open its source files, locate specific lines or symbols (perhaps via regex or AST parsing), and apply targeted changes. These finer editing operations allow more precise self-modifications, which DGM found to be beneficial [20].

- **Multiple Solution Generation and Ranking:** Instead of making a single guess at a solution, the agent can generate **multiple candidate solutions and then evaluate or rank them to pick the best** [21]. We will incorporate this by allowing the LLM to propose several alternative patches or answers for a given problem, execute each (or analyze them), and then select the highest-scoring one (based on tests passed or an LLM-estimated reliability). This feature increases the chance of success on tough tasks by exploring different approaches in parallel [22].

- **Peer Review and Self-Reflection:** DGM also introduced an idea of one agent reviewing another's code before execution [21]. In our implementation, we can simulate this by using the LLM in a verification mode – e.g. after a patch is generated, ask a second reasoning pass (or another model) to critique the patch or predict possible errors. This "AI code review" can catch obvious mistakes before running the code.

- **History Tracking of Attempts:** The system **maintains a history of what changes have been tried and why they failed** [23]. We will implement a logging mechanism where each iteration records the attempted modification, the rationale (e.g. which weakness it aimed to fix), and the outcome (tests passed or failure reasons). The agent's LLM can consult this history to avoid repeating past failed strategies. This aligns with DGM's emergent behavior of using past context to inform future tries [23].

- **Long-Context Handling:** As an additional consideration, DGM improved at managing long files or descriptions (e.g. splitting code or focusing on relevant sections) [24]. For LaTeXify's purposes, if lengthy documents or complex tasks are involved, plan to incorporate strategies for handling long contexts (chunking input, summarizing intermediate state, etc.) so that the LLM can cope with large content.

ITRS Features to Adapt for LaTeXify

- **LLM-Driven Strategy Selection (Zero Heuristics):** In ITRS, **all decision-making is delegated to the LLM** – there are *no fixed if-then heuristics* for choosing what to do next [25]. We will adopt this by prompting the LLM to decide the agent's next action each iteration. Instead of hardcoding a schedule or threshold for when to explore vs. exploit, the agent's prompt will include the current state (performance, recent changes, history) and ask the LLM to determine which refinement strategy to pursue. This enables **adaptive, context-sensitive strategy shifts** guided by the model's emergent intelligence [25] .

- **Refinement Strategies:** Incorporate ITRS's six reasoning strategies as modes the agent can operate in [26] :

- *TARGETED:* Focus on a specific known issue or weakness. (E.g. "Improve error handling in module X because it's causing test failures.")
- *EXPLORATORY:* Try out a novel or random change for potential gain, even if no specific problem is targeted.
- *SYNTHESIS:* Combine ideas from multiple past agent versions or multiple proposed patches into a new solution.
- *VALIDATION:* Emphasize verifying current solutions – possibly trigger actions like re-running tests, double-checking logic, or tightening constraints (this overlaps with the patch validation step we already plan to implement) [27] .
- *CREATIVE:* Encourage out-of-the-box changes that add entirely new capabilities or redesign parts of the system.
- *CRITICAL:* Analyze the agent's own reasoning and code critically to find flaws or contradictions, leading to fixes (akin to an introspection or debug mode).

We will design the agent to support all these strategies. The LLM controller can suggest one of them each iteration (or a combination), and the agent will adjust its behavior accordingly. This might be implemented by having different prompt templates or toolsets for each strategy. For example, a **Targeted** iteration might begin by identifying a failing test or limitation to address, whereas a **Creative** iteration might ask the model for any entirely new feature that could boost performance.

- **Persistent Thought Document (with Versioning):** The agent will maintain a **persistent "thought log" document** that records its reasoning process, decisions, and findings across iterations [28] . This could be as simple as a markdown or text file that the agent appends to each cycle, or a more structured in-memory object. It will have sections (e.g. current hypotheses, known issues, plan for next step, outcomes) that get updated. We will include a version or timestamp for each iteration's entry to trace the evolution of thought. This persistent document provides transparency and helps the LLM recall context over long runs (since it can be fed back into prompts). It implements ITRS's idea of a structured memory that the reasoning iteratively refines, complete with **semantic version tracking of changes** [28] .

- **Knowledge Graph Integration:** Introduce a dynamic **knowledge graph** to represent relationships discovered during reasoning [29] . In practice, for the coding agent, nodes in the graph could represent things like: features or patches introduced, specific functions or files in the code, benchmark tasks or test cases, and outcomes (errors, successes). Edges can denote relationships such as "patch X affects test Y" or "feature A depends on tool B". As the agent runs, it will update this graph (e.g. when a test fails, link it to the code component that caused it; when a new tool is added, link it to capabilities it enables). The graph provides a **symbolic memory** that the LLM can query (we could allow the LLM to ask for relations like "what changes impacted module Z?") to inform its decisions. This helps with transparency and reasoning about the agent's own structure.

- **Semantic Vector Memory (Contradiction Detection):** Alongside the graph, maintain a **vector-based memory** of the agent's knowledge to catch contradictions or regressions [29]. For example, we can embed the content of the thought document or key state descriptions at each iteration into a vector store. The agent can then retrieve similar past states to see if it's contradicting itself. In a reasoning context, this flags inconsistent answers; in our coding context, it could detect if a new change causes a previously solved problem to fail (a regression). The system can then trigger a CRITICAL refinement if a contradiction/regression is found. This semantic memory also allows the agent to recall distant context that might have fallen out of the immediate LLM context window by searching for semantically similar situations from the past.

- **Dynamic Parameter Adjustment:** Allow the LLM to not only choose actions but also tweak parameters (like temperature, number of solutions to generate, time budget per iteration, etc.) based on the scenario [30]. For instance, the LLM might decide to increase creativity (higher temperature) if stuck in a local optimum, or reduce it to focus when near a solution. We will expose certain meta-parameters to the LLM's control by including adjustable settings in the prompt context (the LLM can output a desired change which the code then applies in the next iteration). This implements ITRS's notion of the system dynamically optimizing itself without hardcoded rules.

- **Transparency and Visualization:** All the above (thought logs, knowledge graph updates, performance metrics per generation) will be recorded so that the reasoning process is **completely transparent and auditable** [26]. In a live setting, one might visualize the reasoning steps or graph; for our plan, the final LaTeX report (see below) will serve to present this information clearly.

Implementation Tasks for Integrating DGM+ITRS into LaTeXify

1. **Environment and Model Setup:** Prepare the development environment with necessary libraries and ensure GPU utilization. Install PyTorch (and optionally TensorFlow) to leverage the NVIDIA 5090 GPU (32GB VRAM) for running large models. Set up a suitable **foundation model** for code and text generation – for example, a code-specialized LLM (like Code Llama or StarCoder) for proposing code improvements, and possibly a general LLM for reasoning steps. Make the model loading abstract so you could swap in either a PyTorch-based model (preferred for HuggingFace Transformers) or a TensorFlow model if needed. Verify that the GPU is accessible and configured (this may involve setting proper device IDs or using `cuda` in PyTorch). ***(Result:* Environment ready with a loaded code-generation model on GPU.)_

2. **Define Agent Class and Codebase Handling:** Create an `Agent` class (or similar) that encapsulates the agent's state and behaviors:

3. Include methods for **loading its own source code** into memory (perhaps reading the Python files of LaTeXify's codebase or the parts it is allowed to modify).
4. Provide an interface to the LLM: e.g., a method `propose_patch()` that takes the agent's current code (or relevant subset) plus context (performance stats, history) and returns an LLM-generated code diff or code snippet improvement [2].
5. Integrate **tools** the agent can use: e.g., `run_tests()` to execute a test suite or tasks, `execute_code()` to run arbitrary code (with sandboxing), `edit_file()` to apply a patch to the

code files, and logging functions. Ensure these tools run in isolation (using a subprocess or Docker container as needed) [5] .

6. Maintain internal data structures for the agent's **performance metrics** (e.g., tasks attempted/ solved), a link to its **knowledge graph** representation (if using an external store or an internal graph structure), and references to the **persistent thought log**.

7. Include an identifier or version for the agent (to distinguish archive members).
**(*Result:* A flexible Agent class capable of self-inspection, code modification, and tool use.)_

8. **Benchmark Task Suite & Evaluation Function:** Determine the set of tasks or benchmarks that reflect LaTeXify's goals, and implement a way to evaluate the agent on them:

9. If LaTeXify's domain is *document generation*, tasks might be high-level (e.g., "produce a LaTeX document section given some content requirements") or smaller coding-style challenges related to LaTeX generation (like functions that transform inputs to LaTeX). If it's extending the original LaTeXify library, tasks might be converting certain Python expressions to correct LaTeX strings and verifying the output.
10. For each task, define a **ground truth or test**. For coding tasks, this could be unit tests or expected outputs (similar to SWE-Bench issues and Polyglot problems) [31] . For document-generation tasks, it could be comparing the output document to a reference or checking that generated LaTeX compiles without error.
11. Implement an `evaluate_agent(agent)` function that runs the agent on all benchmark tasks (using the agent's tools to solve them) and returns a performance score (e.g., percentage of tasks solved) [7] . This will likely involve prompting the agent's LLM to solve each task or having pre-coded solution methods that use the agent's code.
12. Make sure this evaluation uses the agent's current codebase – i.e., after each self-modification, when we call `evaluate_agent`, it's executing the modified code. Leverage the GPU if any ML models are used during task solving.

13. The evaluation step should also collect details like which tasks failed and error messages, so that the agent's next iteration can target those issues (pass this info into the thought log and LLM context).
**(*Result:* A suite of tasks and an evaluation routine that returns quantitative performance metrics and feedback for any agent version.)_

14. **Initial Agent Bootstrapping:** Instantiate the initial baseline agent. This would be the current state of the LaTeXify project (or a simplified version of it) which we will allow the DGM process to improve. For example, use the existing LaTeXify code as the starting agent's codebase. Confirm that the initial agent can solve some tasks (likely at a low performance baseline, similar to DGM's 20% on SWE-Bench) [16] . Log its baseline performance by running `evaluate_agent` and store the results (this forms generation 0 in the archive, with its score).

15. **Archive Structure and Diversity Selection:** Create a data structure to maintain an **archive of agents** (e.g., a list or database of agent instances or their saved states). Alongside, implement the **selection mechanism** for choosing a parent agent for the next iteration:

16. Use a probabilistic selection favoring higher performing agents (e.g., roulette wheel selection where probability ~ agent score), but ensure a non-zero chance for any agent to be picked [10] . This could be tuned via a temperature parameter on the score distribution.

17. If using quality-diversity ideas, optionally incorporate a novelty measure: e.g., cluster agents by some feature vector (maybe the set of features they have or their performance profile) and sometimes pick an agent that covers an underexplored niche.
18. When an agent is selected to spawn a child, **clone its code and state** to a new agent instance that will be modified. Assign the child a new version ID.

19. Keep track of lineage if desired (e.g., store parent->child relationships) for later analysis or visualization (this is not essential to functionality but helps in transparency).
**_(Result:_ Archive list and a `select_parent()` function that yields an agent from the archive based on fitness-proportional and random diversity.)_

20. **LLM Prompting for Self-Improvement:** Implement the logic for generating a self-improvement proposal using the LLM:

21. Construct a prompt that provides the LLM with **the agent's current code (or a diff from original), the agent's recent performance results, and the history of changes attempted**. Include specific sections: e.g., "Here is the agent's code (or relevant snippet)…, here are known issues or failures…, here is the change history… **Propose a code change to improve performance.**"
22. To incorporate ITRS's zero-heuristic approach, **embed the strategy selection into the prompt**: The prompt can explicitly ask "Decide on a refinement strategy (Targeted, Exploratory, Synthesis, Validation, Creative, Critical) and explain your plan, then provide the code changes." The LLM's response should ideally state something like "Strategy: TARGETED – I will fix X bug… Patch: [code diff]".
23. Parse the LLM's response. You may have it output a diff format (which we then apply using a patch tool), or just new code for a specific function. It might be useful to have the LLM provide a rationale as well; this rationale can be appended to the thought document for record-keeping.
24. Ensure the prompt and parsing can handle multi-solution suggestions: for instance, ask the LLM to propose up to 3 candidate patches if in Exploratory/Creative mode. Then we can evaluate each candidate (perhaps run a quick test subset) and choose the best. This ties in with the multi-solution feature [22] .

25. Incorporate constraints or reminders in the prompt from the knowledge graph and history (e.g., "Do not repeat a change that was tried and failed: [list of past failed ideas]."). This will help prevent cycling on the same mistakes [23] . **_(Result:_ A `generate_patch(parent_agent)` function that uses the LLM to produce a proposed improvement patch (and associated strategy rationale) for the parent agent's code.)_

26. **Applying Code Patches:** Once a patch is received from the LLM:

27. Implement functionality to **apply the code diff** to the agent's codebase. If the LLM provided a unified diff, use a patch application library or custom code to modify the files. If it provided code snippets with identifiers of where changes occur, implement logic to locate those places in the file and replace/add accordingly.
28. Back up the parent agent's code in case the patch needs to be reverted or for record. Apply the patch to the child agent's code (the child was cloned in step 5).
29. After applying, run a quick syntax check or compilation to ensure the code is not breaking basic syntax.

30. Add an entry to the **thought log** like "Iteration N: Applied patch X (Strategy=Targeted) to function Y to address Z." Include the diff summary and any rationale excerpt from LLM.

31. If using a knowledge graph, update it: e.g., add nodes/edges for "Patch X modifies component Y" and possibly link "Patch X intended to fix issue Z". ***(Result:* The child agent's code is updated with the LLM-proposed changes, and changes are documented.)_

32. **Validation & Testing Phase (Patch Validation):** Before fully committing the new agent to the archive, perform the **patch validation step** [19] :

33. Run the agent's test suite or evaluation on at least the tasks it was supposed to improve (e.g., if the patch targeted a specific failing test, run that first).
34. Optionally, run a quick smoke test on a subset of all tasks to see if anything else broke (regression check). Use the previously embedded vector memory to recall if any task that used to pass now fails (contradiction detection).
35. If the patch causes immediate errors or worsens performance drastically, you might decide to reject this child (not add to archive). In DGM, even some performance-neutral or slightly negative changes were kept if "interesting" [15] , but you can set criteria such as "must not reduce overall score by more than X unless it adds a new capability."
36. If the patch passes its focused tests, proceed to do a full `evaluate_agent(child)` on all tasks to get its new performance score [7] . Record the results.

37. As the patch validation was an emergent improvement in DGM, implementing it from the start ensures our agent always verifies its solutions with tests, mirroring that beneficial behavior [19] . ***(Result:* The new child agent is tested; we obtain its performance metrics and confirm it didn't introduce fatal flaws before archival.)_

38. **Archive Update and Open-Ended Selection:** Decide whether to add the new agent to the archive:

39. If the child's performance is higher than or equal to the parent's, definitely add it (and consider pruning the archive if it grows too large, perhaps keeping top N performers plus some diverse ones).
40. If performance is slightly lower but the agent has a novel feature (e.g., it solved a previously unsolved type of task or introduced a useful new tool), you may still add it as a potential stepping stone [15] . This judgment can be LLM-assisted: include in the LLM's output a recommendation "Keep or Discard" along with justification to mimic DGM's openness to novelty.
41. Log this decision in the thought document (e.g., "Agent v7 added to archive with score 45%. Introduced feature: better file parser. Slight drop on easy tasks but improved on hard tasks – kept for diversity.").
42. Update the knowledge graph with any new relationships observed (perhaps link "Agent7 inherits from Agent3", "Agent7 solved TaskX that parent couldn't", etc.).

43. If not adding the child (e.g., patch was harmful), document why, and possibly still store the attempt somewhere for record (could mark it as pruned). Then you may allow another attempt from the same parent or move on. ***(Result:* Archive is updated with the new agent (if accepted), and ready for the next iteration. The archive now contains multiple evolving agents enabling open-ended exploration.)_

44. **Iterative Loop Orchestration:** Write a top-level loop to orchestrate the above steps:

    - for generation = 1 to N (or until some stopping criteria):
    - Select a parent from archive (using step 5).
    - Generate a patch via LLM for that parent (step 6).
    - Apply the patch to create a child agent (step 7).
    - Validate and evaluate the child (step 8).
    - Update archive with child if appropriate (step 9).
    - Update the persistent thought log and knowledge graph with outcomes (the thought log can be saved to disk periodically, and graph maybe exported).
    - Possibly check for convergence or stopping: The loop can stop when a target performance is reached, or no improvement seen for X iterations, or a manual stop. However, per ITRS, we can also let the LLM decide when to stop. For example, after each iteration, ask the LLM "Do you consider further improvements likely? If not, we will terminate." This aligns with zero-heuristic termination criteria [25] .
    - Include periodic saving of the state (so the process can resume if interrupted).
    - Monitor resource usage (the tasks mention potentially heavy computation; ensure GPU memory is freed appropriately between model calls, etc.). ***(Result:* A driving function `run_evolution()` that carries out the DGM iterative self-improvement loop, integrating the custom logic we added. The system will continually produce new agent variants and accumulate improvements.)_

45. **Integrate ITRS Reasoning Aids:** Augment the iteration loop with ITRS-inspired reasoning support:

    - **Strategy Guidance:** Before generating a patch each iteration, feed the LLM a summary of the agent's situation and ask it to explicitly choose a strategy (Targeted/Exploratory/etc.) [26] . Parse this and adjust the patch prompt or agent behavior accordingly. E.g., if *Targeted*, identify a specific failing test or bottleneck for the LLM to focus on; if *Exploratory*, allow a more free-form patch possibly introducing new functionality.
    - **Thought Document Maintenance:** After each iteration, update the persistent thought document with: strategy used, rationale given by LLM, code changes made, outcome (scores and any insights). Use semantic versioning or simply an index for each iteration's entry [28] . This document can be fed (in truncated form if large) into subsequent prompts to provide memory. Ensure this log is human-readable as well for transparency.
    - **Knowledge Graph Updates:** At key points (post-evaluation), call functions to update the knowledge graph with new info. For instance, if Task 5 was solved by Agent8, add edge Agent8 -> Task5 "solved_by". If Agent8 introduced ToolA, add Agent8 -> ToolA. If a contradiction is detected (e.g., Agent8 fails a task that Agent7 passed), flag that on the graph and log it. Keep the graph data structure in memory; optionally, use a visualization library or output a Graphviz dot file for later review.
    - **Contradiction Check:** After evaluation, use the vector store to find if any new failures contradict previous successes. Also compare the LLM's stated rationale to outcomes (if it said it would fix X but didn't, that's a kind of inconsistency). If found, mark it in the log and possibly trigger a *Critical* strategy next iteration to address it.
    - **No Hardcoded Decisions:** Avoid coding rigid rules like "stop after 100 iterations" or "if score > 90%, stop". Instead, occasionally ask the LLM in the loop (maybe every few iterations or if improvement stalls) whether to continue or adjust approach. This implements the zero-heuristic ethos where the LLM can suggest "I think we've converged, you can stop now" or

"Switch to a new strategy now" [25] . **(Result:* The self-improvement loop now has an embedded reasoning layer that uses the LLM's strengths not just for coding but for high-level control, making the process more adaptive and transparent.)_

46. **Final LaTeX Report Generation:** Once the iterative run is finished (or on demand), generate a comprehensive **LaTeX document** summarizing the entire process:

    - Include an introduction explaining the experiment (automatically written or use a template).
    - Present the **performance improvements** over time (e.g., a table or chart of generation vs. score). If possible, use matplotlib to plot the performance curve and include it as an image in the LaTeX (this would require saving a plot and using `\includegraphics`).
    - Document the **key innovations and features added** by the agent through its evolution. This can be pulled from the thought log: list out notable patches (e.g., "Gen 5: Added patch validation module" [32] , "Gen 9: Introduced multi-solution ranking" [33] , etc.).
    - If available, include an **evolutionary tree or lineage** diagram of agents. This could be generated from the parent-child records in the archive (perhaps using Graphviz). If making the diagram is complex, at least list each agent version with its parent and score.
    - Provide sections for each ITRS component used: e.g., a subsection on "Knowledge Graph of Agent's Knowledge" – possibly render a small part of the graph (or describe it), and "Reasoning Log Excerpt" showing a portion of the thought document to illustrate transparency.
    - Ensure all this is formatted in LaTeX. You might use a Python LaTeX library (like `pylatex`) or simply output a .tex file as a template filled with the data. After generating the .tex, optionally auto-compile it to PDF if a LaTeX engine is available.
    - This final document serves as both a report for humans and a form of **transparent audit of the self-improvement process** [26] . It aligns with the ITRS emphasis on explainability and DGM's practice of recording experiment logs for analysis. **(Result:* A file `EvolutionReport.tex` (and PDF) containing the narrative of how the LaTeXify agent self-improved, including performance stats and the features it gained. This fulfills the "end-of-pipeline document generation" objective.)_

47. **Testing and Refinement:** Execute the full system on a smaller scale test to ensure all parts work together:

    - Run a few iterations and monitor the outputs (check that patches are applied correctly, tests run, the log and graph update, etc.).
    - Debug any issues (e.g., LLM might produce an unparseable diff – add formatting constraints in the prompt if so, or adjust parsing).
    - Once stable, run the system for the desired number of generations or until convergence. Keep an eye on resource usage (the GPU memory, time per iteration) and adjust batch sizes or model parameters if needed to fit the hardware limits.
    - Verify that improvements are indeed being made (if not, analyze whether the LLM needs better prompt tuning or if the tasks are too hard without more shots, etc.). Potentially employ the multi-solution approach more heavily if stuck.
    - Finally, generate the LaTeX report and review it. Ensure it accurately reflects the process. This step closes the loop by confirming that LaTeXify with DGM+ITRS features achieves open-ended self-improvement tailored to its domain. **(Result:* A fully implemented self-evolving LaTeXify agent, tested and ready. It uses DGM's core loop of recursive code enhancement [1]

and ITRS's advanced reasoning features [26] to continually refine its performance, with all outcomes transparently documented.)_

Each of these tasks builds towards a system where **LaTeXify can iteratively improve itself** in a safe, transparent, and open-ended manner. By combining DGM's evolutionary code self-modification [11] with ITRS's transparent reasoning and strategy framework [26], the agent will not only get better at its tasks but will also clearly explain and record how it got there. This detailed plan should guide the implementation such that an advanced coding model can generate the required components in one go, and the resulting system will leverage the full power of the available GPU hardware for efficient execution.

---

[1] [6] [9] [11] [32] [33] The Darwin Gödel Machine: AI that improves itself by rewriting its own code
https://sakana.ai/dgm/

[2] [3] [4] [7] [8] [10] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [31] The Darwin Gödel Machine: Open-Ended Improvement via Recursive Code Mutation and Empirical Fitness | by Adnan Masood, PhD. | Medium
https://medium.com/@adnanmasood/the-darwin-g%C3%B6del-machine-open-ended-improvement-via-recursive-code-mutation-and-empirical-fitness-a777681d73e4

[5] GitHub - jennyzzt/dgm: Darwin Gödel Machine: Open-Ended Evolution of Self-Improving Agents
https://github.com/jennyzzt/dgm

[25] [26] [27] [28] [29] [30] GitHub - thom-heinrich/itrs: Iterative Transparent Reasoning System by chonkyDB combining reasoning, graph and vector for trustworthy, explainable and smart LLMs want to explore? Visit chonkyDB.com
https://github.com/thom-heinrich/itrs