Darwin Gödel Machine

alex buzunov

13 min read

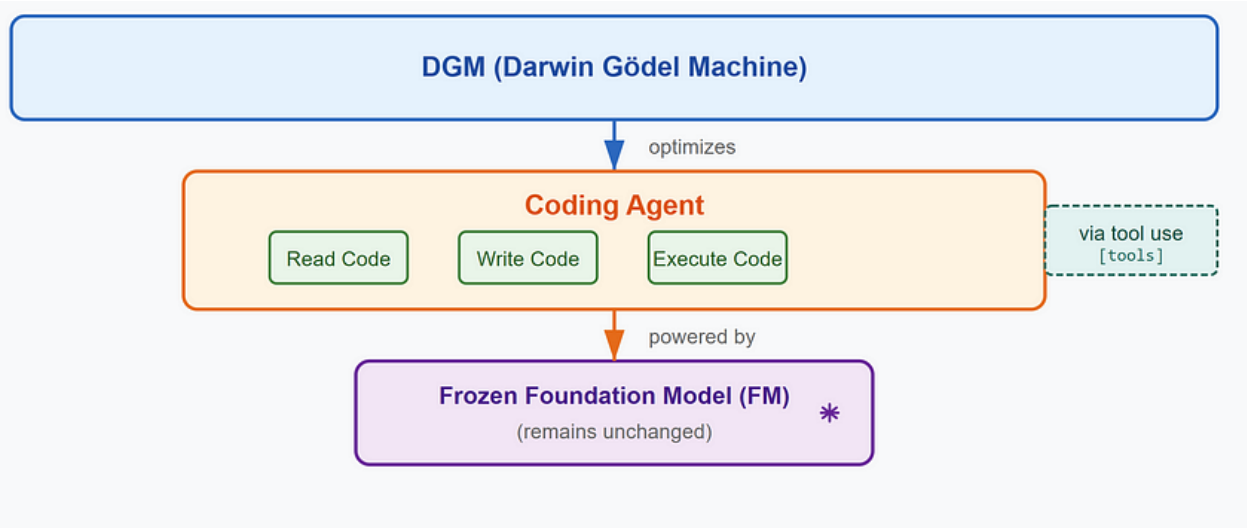
.

4 days ago

Below is beginner-friendly vibed blog interpreting Darwin Gödel Machine: Open-Ended Evolution of Self-Improving Agents paper.

NotebookLM Podcast

The Machine



The DGM focuses on optimizing the design of coding agents, which are powered by frozen foundation models (FMs) and capable of reading, writing, and executing code via tool use

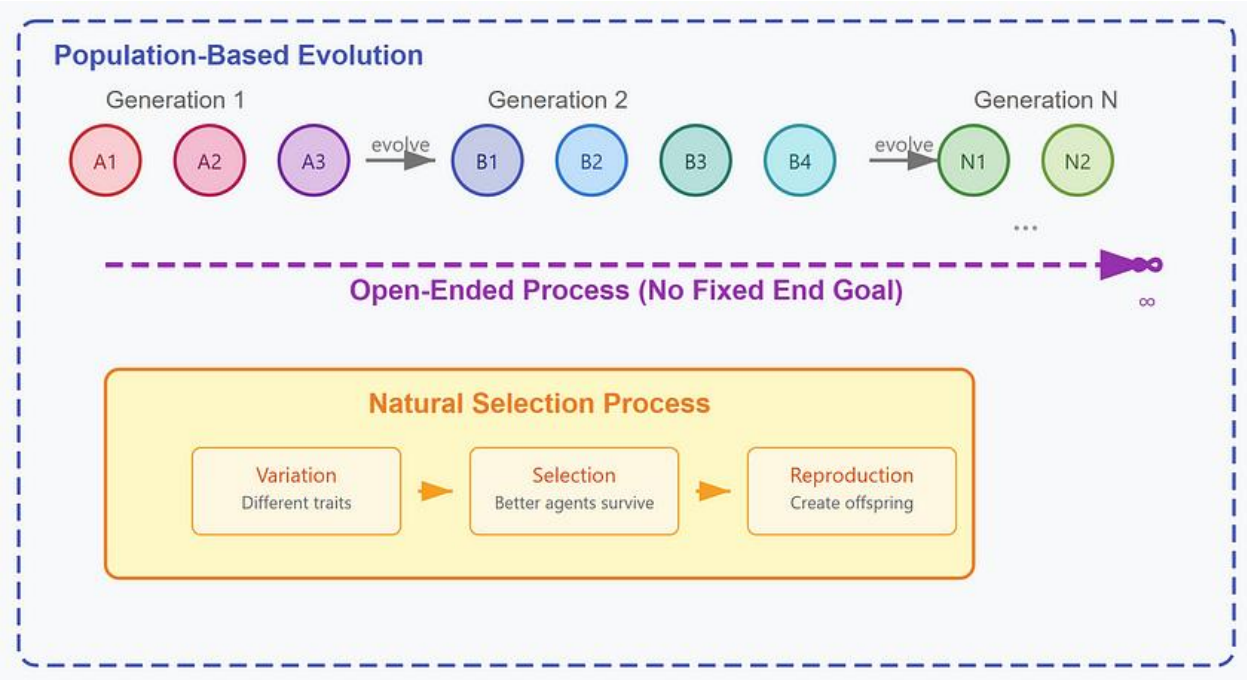
The Darwin Gödel Machine is basically an AI programmer built on top of a pre-trained code-generating model (kept “frozen,” meaning it’s not retrained during this process). In simple terms, DGM can use tools to read code, write new code, and run code just like a human developer working with a code editor and terminal, all as part of its design to improve itself.

Coding Agent

Self-improvement is framed as a coding task: modifying the agent's own codebase (implemented in Python) to enhance its coding capabilities.

The DGM treats **making itself better** as just another programming challenge. In practice, this means the AI's job is to **rewrite and refine its own Python code** to become a more capable coder. It's as if the AI is both the student and the teacher — improving its skills by editing its own software.

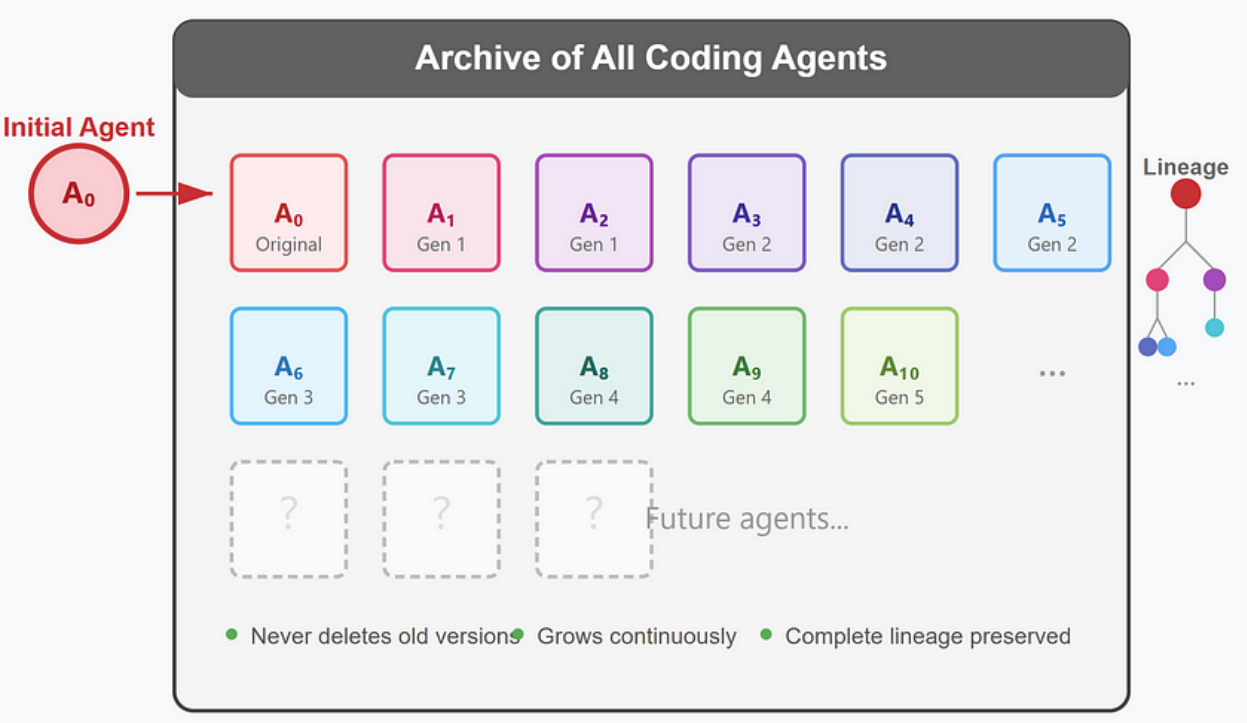
Evolution



The core methodology of the DGM is a population-based, open-ended evolutionary process.

At the heart of DGM is an evolutionary learning approach that never really “ends.” Instead of training once and stopping, DGM continually evolves a **population** of AI agents. This is analogous to natural evolution: there isn’t a fixed end goal, and the process can keep generating new variations indefinitely, searching for better and better solutions.

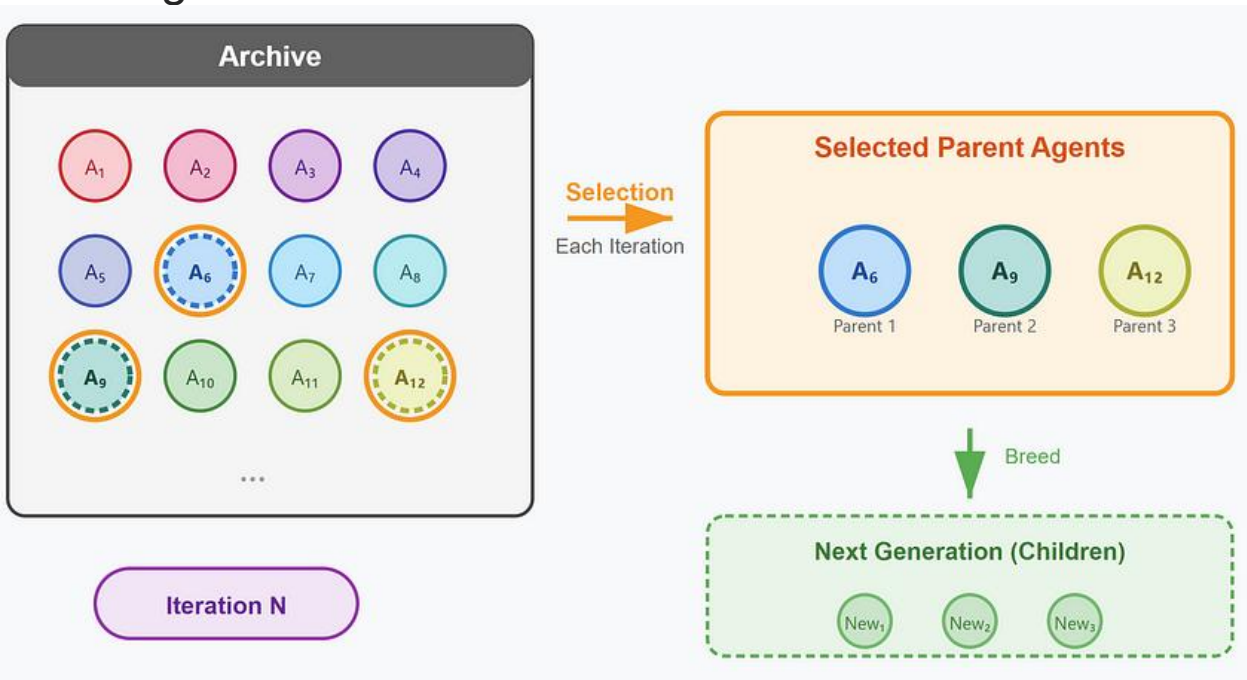
The Archive



It maintains an archive of all generated coding agents, initialized with a single agent.

The system keeps a **library of all the AI versions** it creates (starting from just one original agent). Think of this like an ever-growing collection of prototypes or previous generations. It doesn't throw away past versions; instead, it saves every agent it makes. This archive starts with one baseline agent and expands from there, a bit like starting with a single ancestor in a lineage and tracking all its descendants.

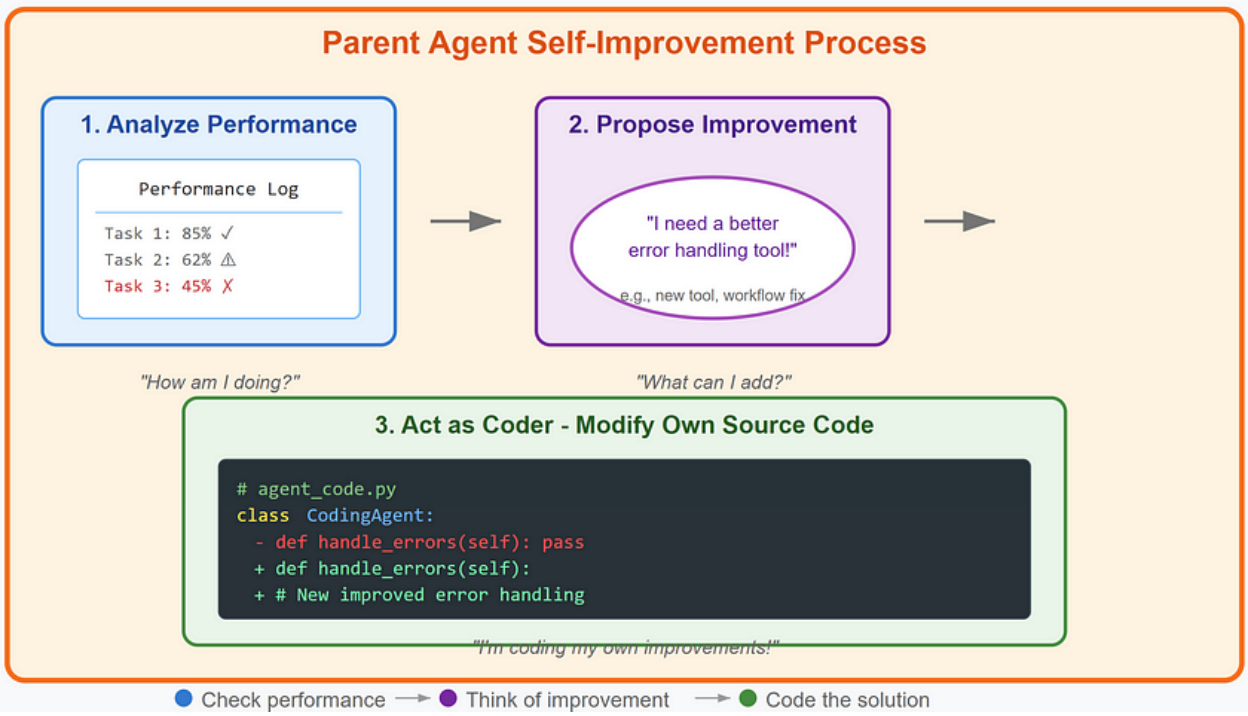
Breeding



On each iteration, a subset of 'parent' agents is selected from this archive.

Over many cycles, the DGM runs something like breeding rounds. Each cycle, it **chooses some agents from the archive to act as parents** for the next generation. These "parent" agents are like the chosen candidates that will spawn new variants. Selection happens every round, so the system is constantly picking certain past agents to create modified "children" agents.

Self-Improvement



These parents analyze their own performance evaluation logs from coding benchmarks, propose a specific self-improvement feature to implement (e.g., a new tool or workflow enhancement), and then act as coding agents to modify their own source code repository to implement this suggested feature.

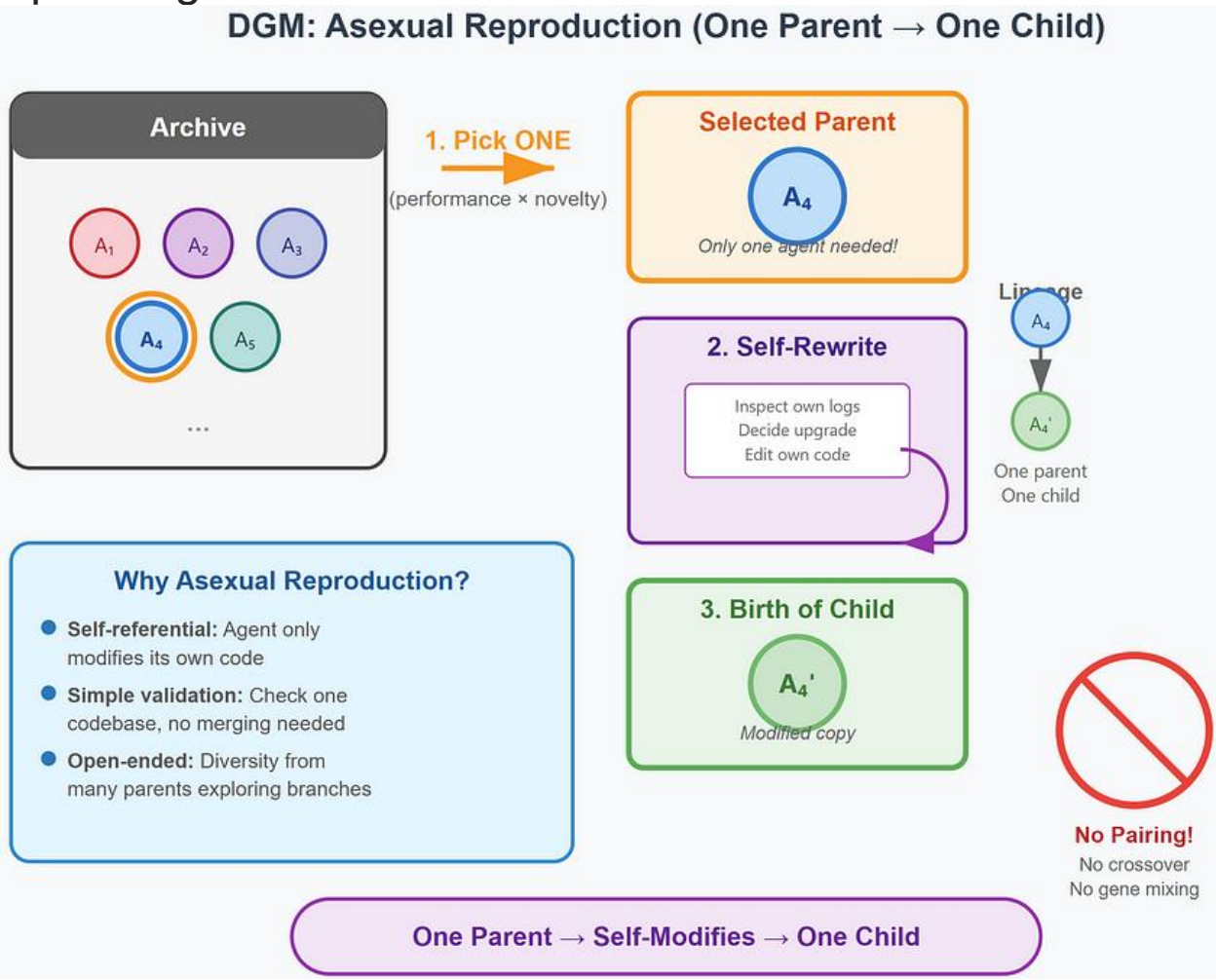
Each parent agent basically **reviews how well it has been doing** on its coding tests (it looks at its performance logs as if checking its report card). Based on that, it comes up with an idea for a self-improvement — for example, adding a new tool or changing its strategy (its "workflow"). Then the parent turns into a coder: it literally goes into its own Python code and makes the changes needed to add that new feature or improvement it thought of.

Child Agent

This generates a new 'child' agent.

The result of the parent rewriting its own code is a **new version of the agent, called a 'child'**. In evolutionary terms, the parent has produced an offspring. The child agent is the same as the parent was, but with the new feature or improvement added to its code — essentially the next-generation AI with that tweak in place.

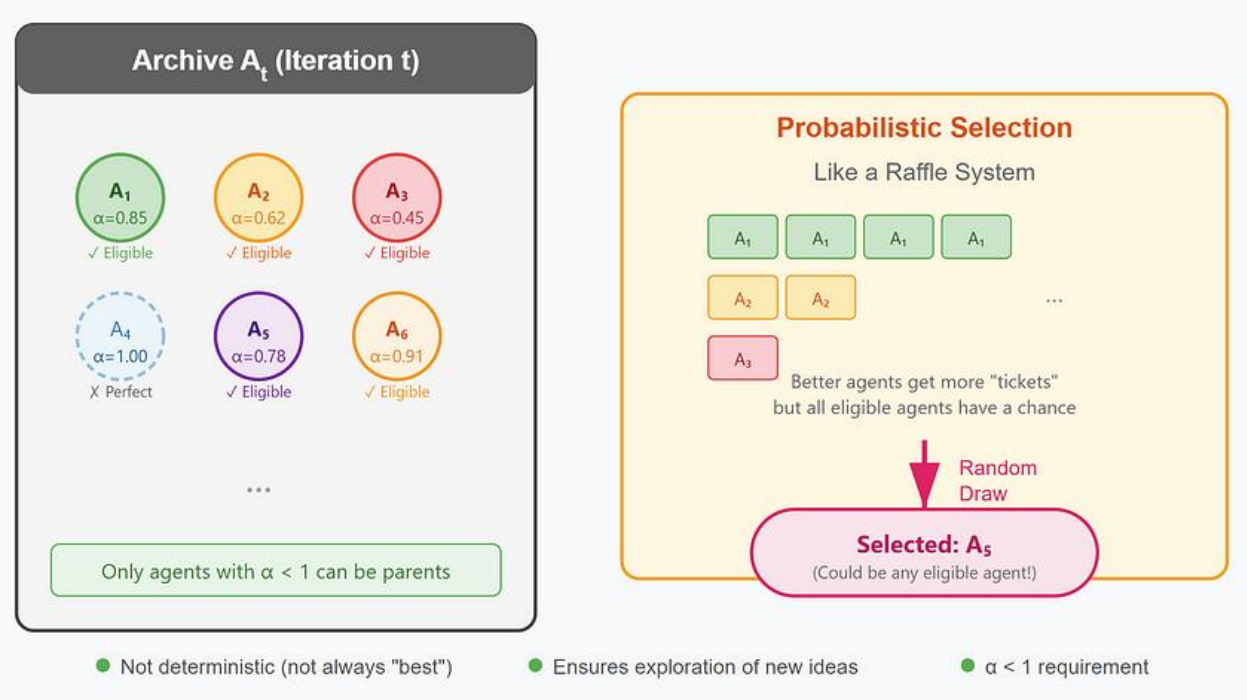
Spawning



The selection of parent agents for self-modification is a crucial part of the open-ended exploration strategy.

Deciding **which agents get to spawn new versions** is really important for keeping the innovation going. The way DGM picks parents affects how well it explores different ideas. In other words, the system's method for choosing parents is a key driver of how it avoids getting stuck and continues to discover new improvements.

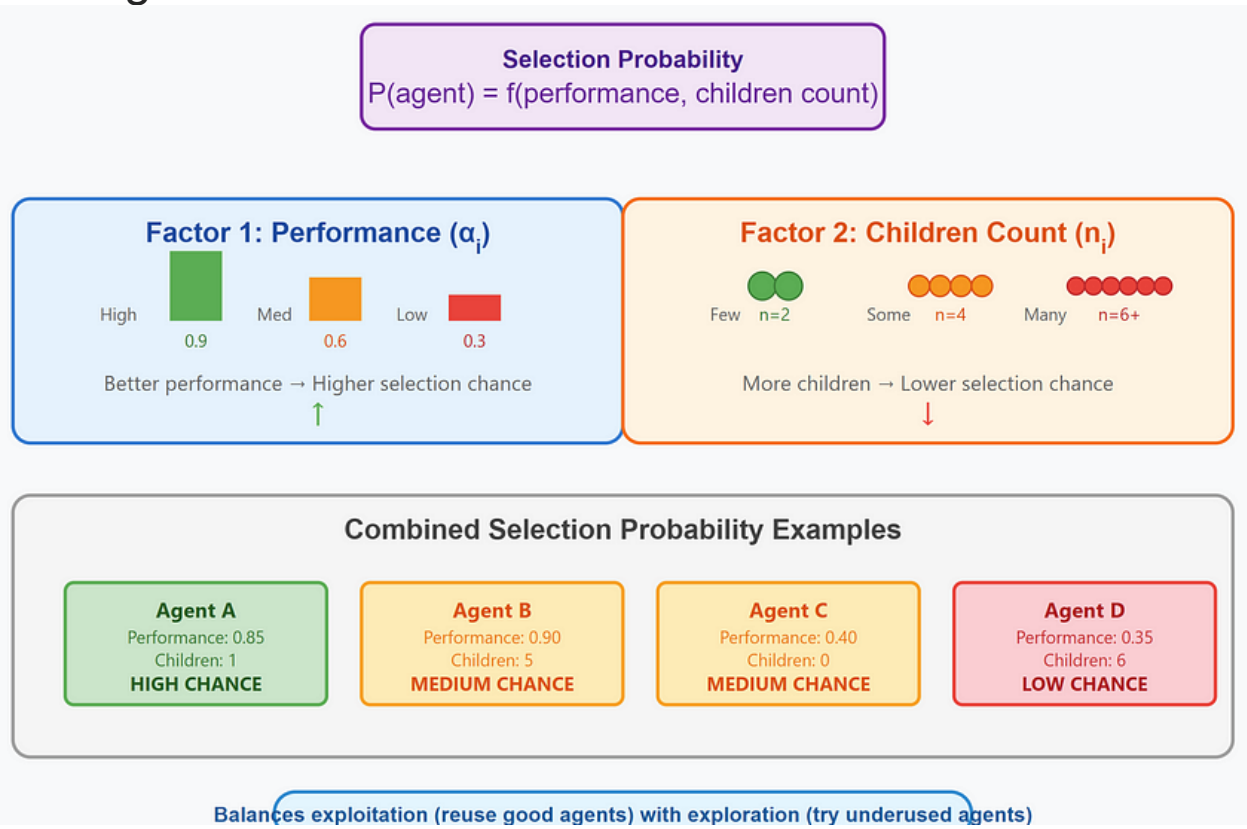
Parent Selection



From the archive A_t at iteration t, eligible agents ($a_i in A_t$ with performance $alpha_i < 1$) are sampled probabilistically.

At each step (each generation t), the DGM doesn't just pick the very best agent deterministically. Instead, it **randomly samples which agents become parents**, with some agents more likely to be picked than others. Here "eligible" just means the agent isn't already perfect (performance $alpha_i$ is below 1, i.e., not 100% success yet). All those eligible agents in the archive A_t have a chance to be chosen as parents, rather like raffle tickets where better agents might get more tickets, but even less successful ones still have a shot so new ideas aren't ignored.

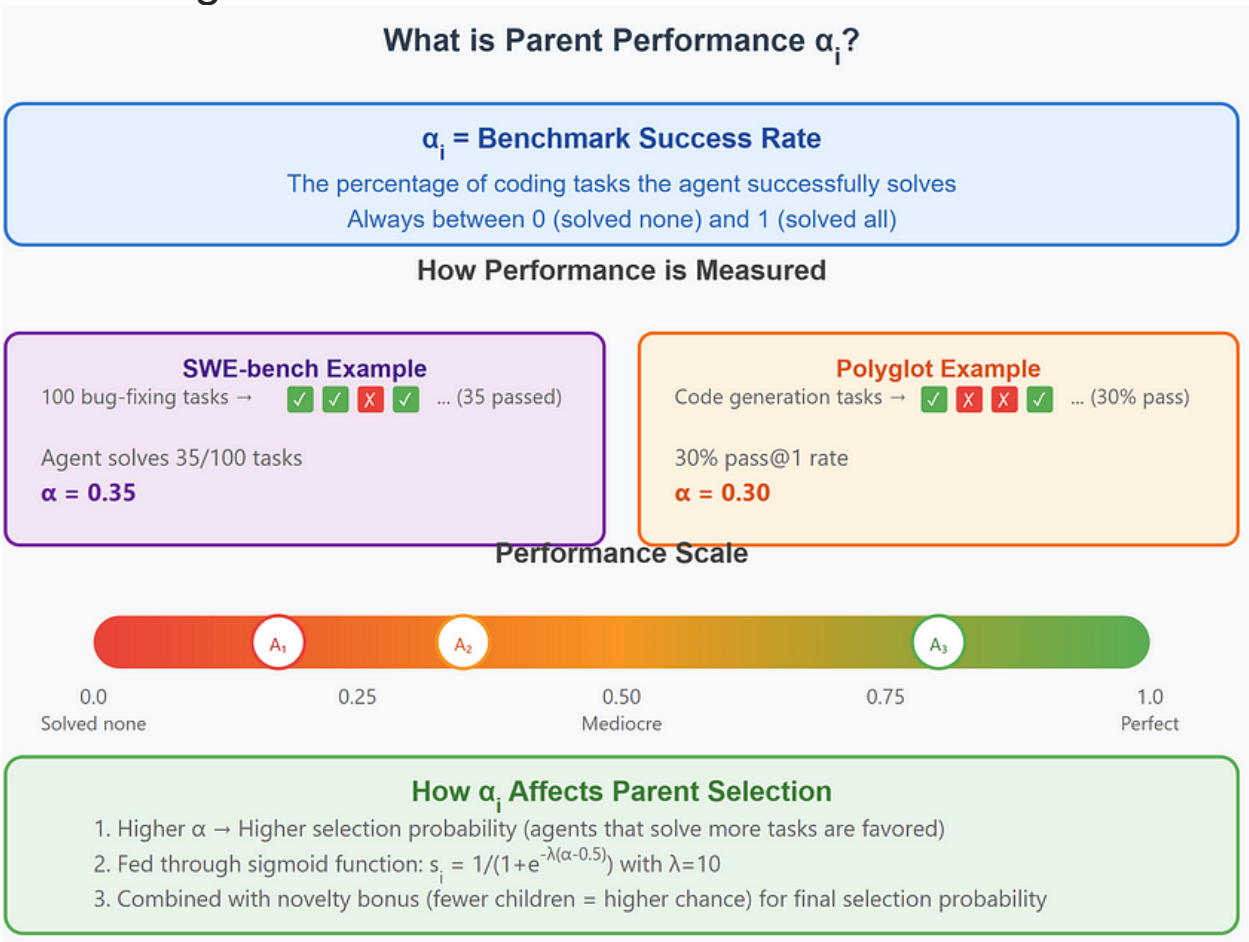
Picking A Breeder



The probability of selecting agent a_i is based on a combined measure of its performance $alpha_i$ and the number of functional children it has already produced, n_i.

The **odds that a particular agent gets picked** to create a child aren't equal for everyone — they depend on two things: how well the agent performs and how many successful children it's had before. In simple terms, an agent will be more likely to be chosen if it's performing well **and** if it hasn't already been used a lot to make kids. This balances between reusing high-performers and giving a chance to ones that haven't had many offspring yet (to keep diversity in play).

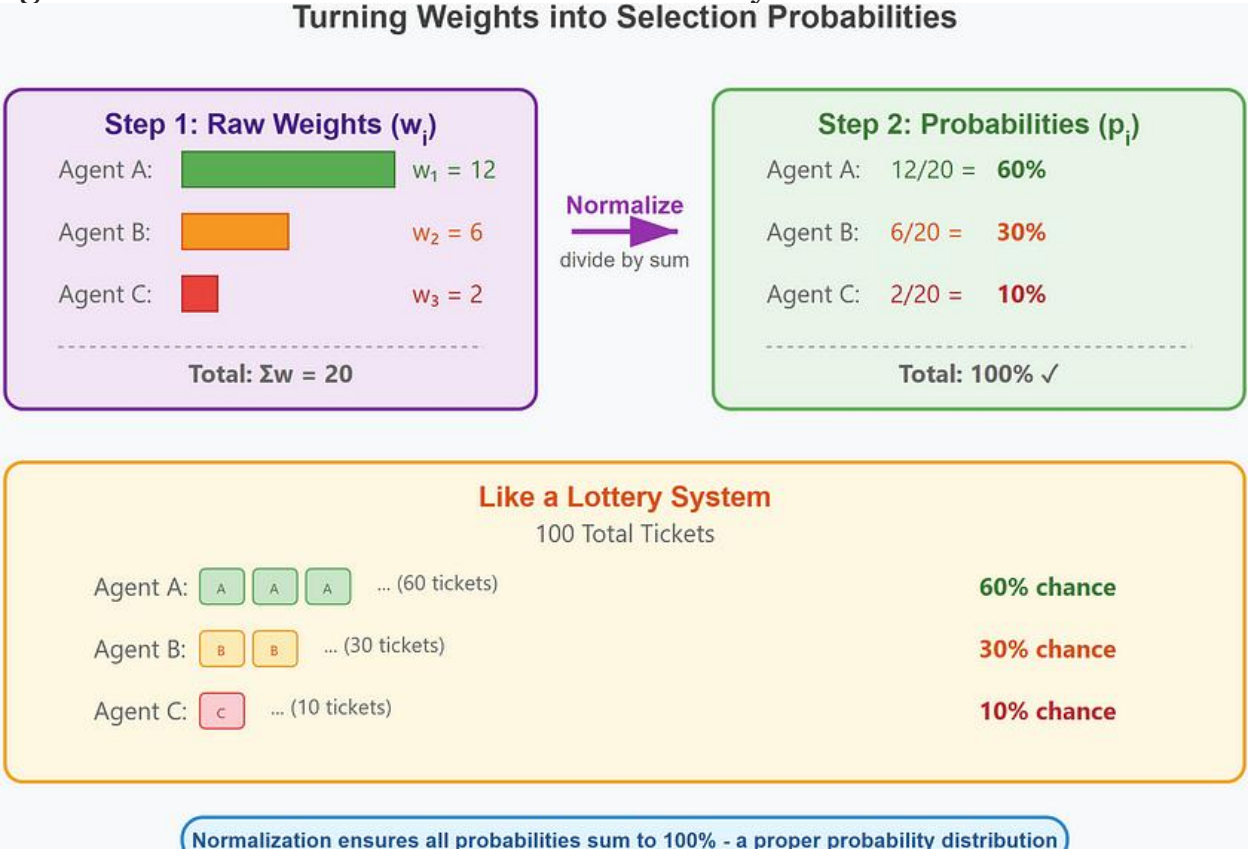
Measuring Parent Performance



An unnormalized weight w_i for agent a_i is calculated as the product of a sigmoid-scaled performance score s_i and a novelty bonus h_i: $s_i = \frac{1}{1 + \exp(-\lambda (\alpha_i — \alpha_0))}$, $h_i = \frac{1}{1 + n_i}$, $w_i = s_i h_i$.

The system actually **computes a weight for each agent** to decide how likely it is to be picked. In plain English: it gives each agent a performance score s_i (using a sigmoid function to convert the raw performance α_i into a nicer 0-to-1 scale, so-so performers get lower scores, top performers get scores closer to 1) and a novelty score h_i (which is basically $1/(1 + n_i)$, meaning an agent gets a high

novelty bonus if it hasn't produced many children yet — the fewer past children, the closer h_i is to 1). Then it multiplies these: $w_i = s_i \times h_i$. So an agent's weight w_i will be big if it's both good **and** relatively under-used as a parent. This weight is like the agent's "ticket count" in the selection lottery.

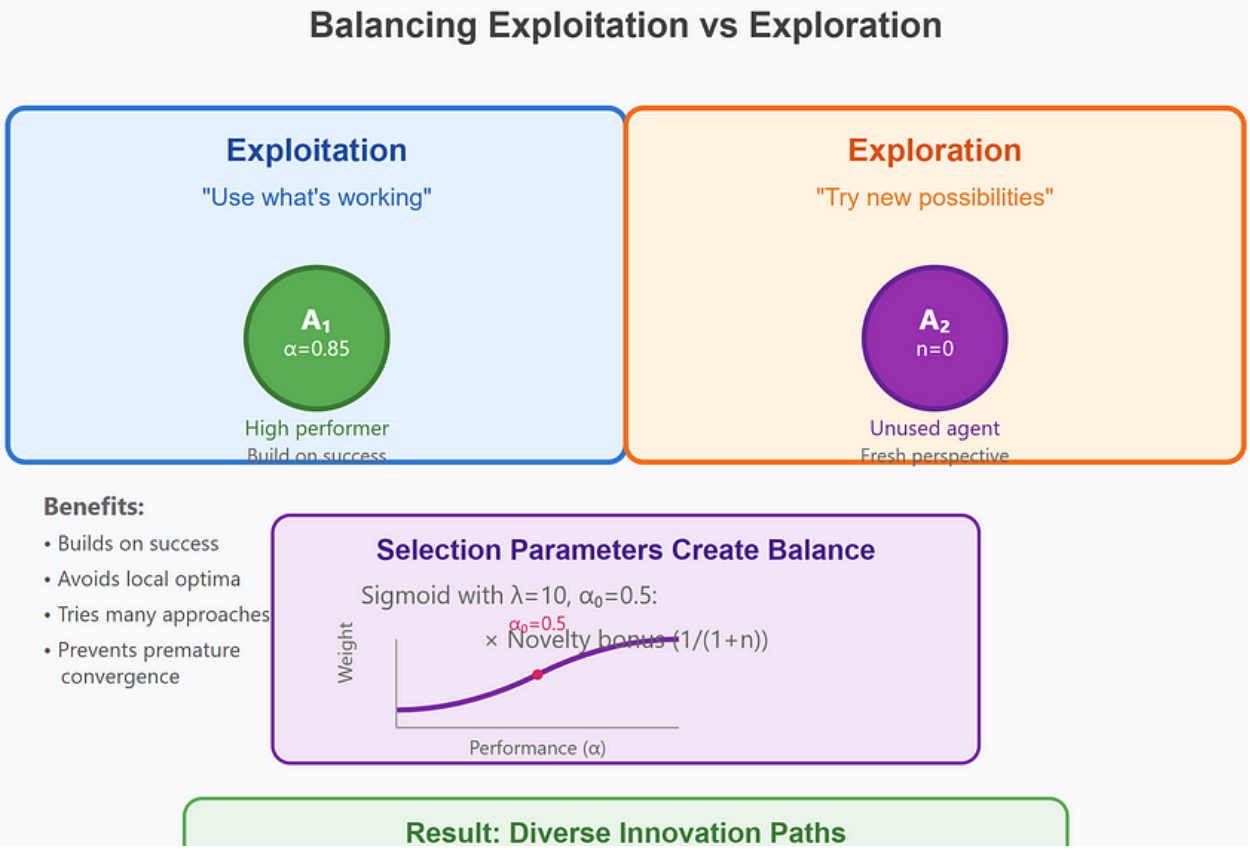


The selection probability p_i is then the normalized weight: $p_i = \frac{w_i}{\sum_{j=1}^{N} w_j}$ where N is the number of eligible agents in the archive.

After computing weights for all the candidates, the DGM **turns those weights into actual probabilities**. Essentially, each agent's weight w_i is divided by the sum of all weights to get p_i, so that all the p_i values add up to 1 (100%). This means if an agent's weight is,

say, 10% of the total weight, it has a 10% chance of being chosen as a parent. It's just like saying "Agent A has 10 out of 100 lottery tickets, Agent B has 5 out of 100, etc." — those fractions determine the chances of selection.

Exploitation v.s. Exploration

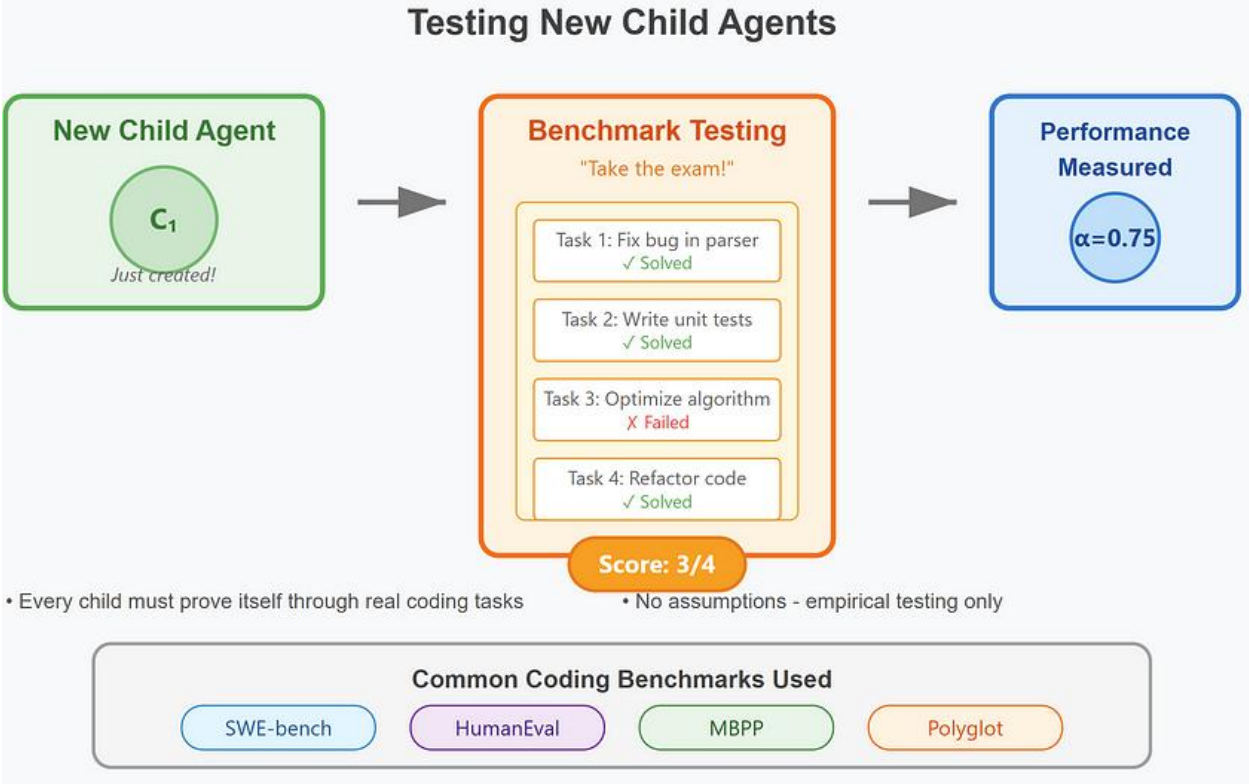


This selection process (with parameters $\lambda=10, \alpha_0=0.5$) favors both high-performing agents (exploitation) and agents that haven't produced many functional children (exploration), encouraging diverse paths of innovation

In summary, the way parents are picked is tuned with specific parameters (they set $\lambda=10$ and $\alpha_0=0.5$ for the sigmoid scaling). The result is a **balance between picking the best**

agents and exploring new possibilities. “Exploitation” here means using agents that score well so far (so the system can build on what’s working), and “exploration” means also giving a chance to agents that are relatively unused (so the system doesn’t put all its eggs in one basket). This balance helps DGM try out many different promising approaches instead of just sticking to one narrow path.

Testing Children

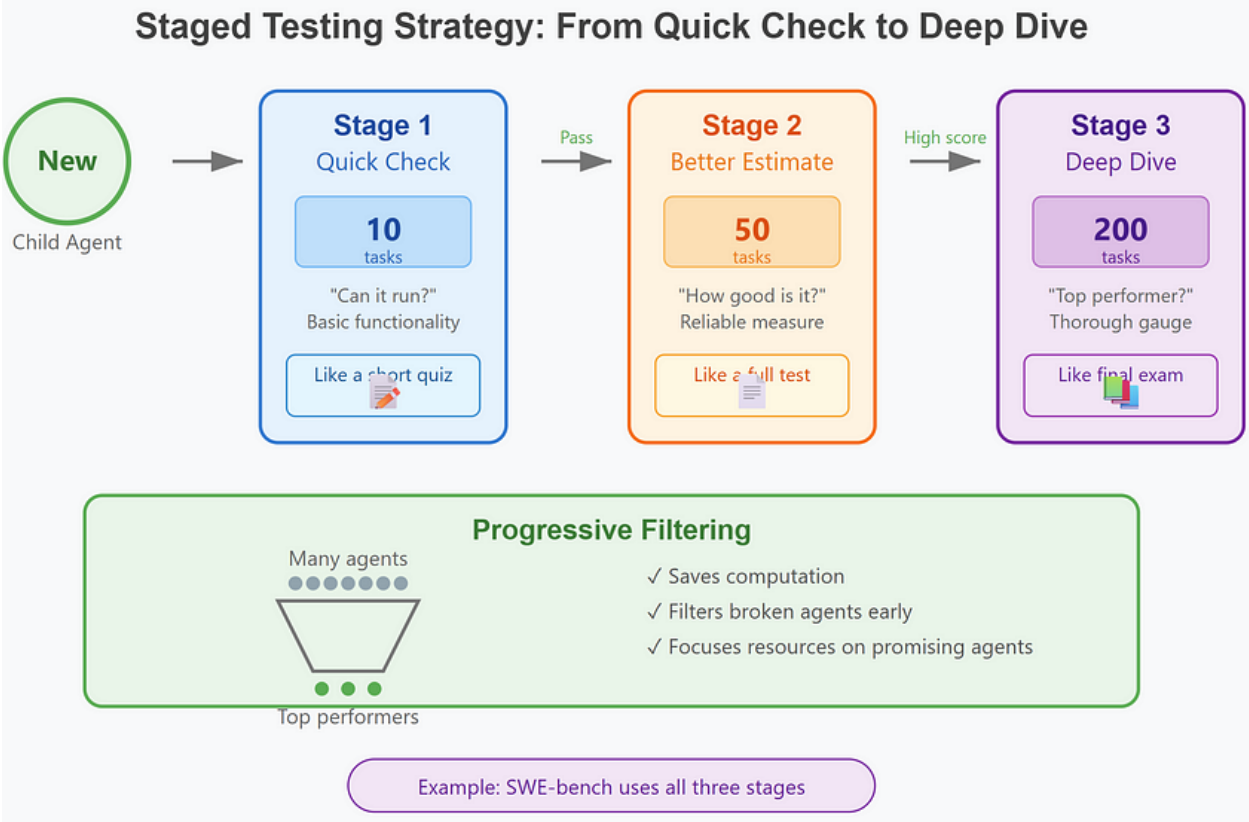


Each newly generated child agent is empirically evaluated on a coding benchmark to estimate its capabilities.

Whenever a new child agent is created, the first thing DGM does is **test it on some coding tasks** to see how good it is. This is like giving the new AI a quick exam or trial run. The system measures the child’s

performance with real coding benchmark problems to get a sense of how the change affected its coding ability.

Staged Testing

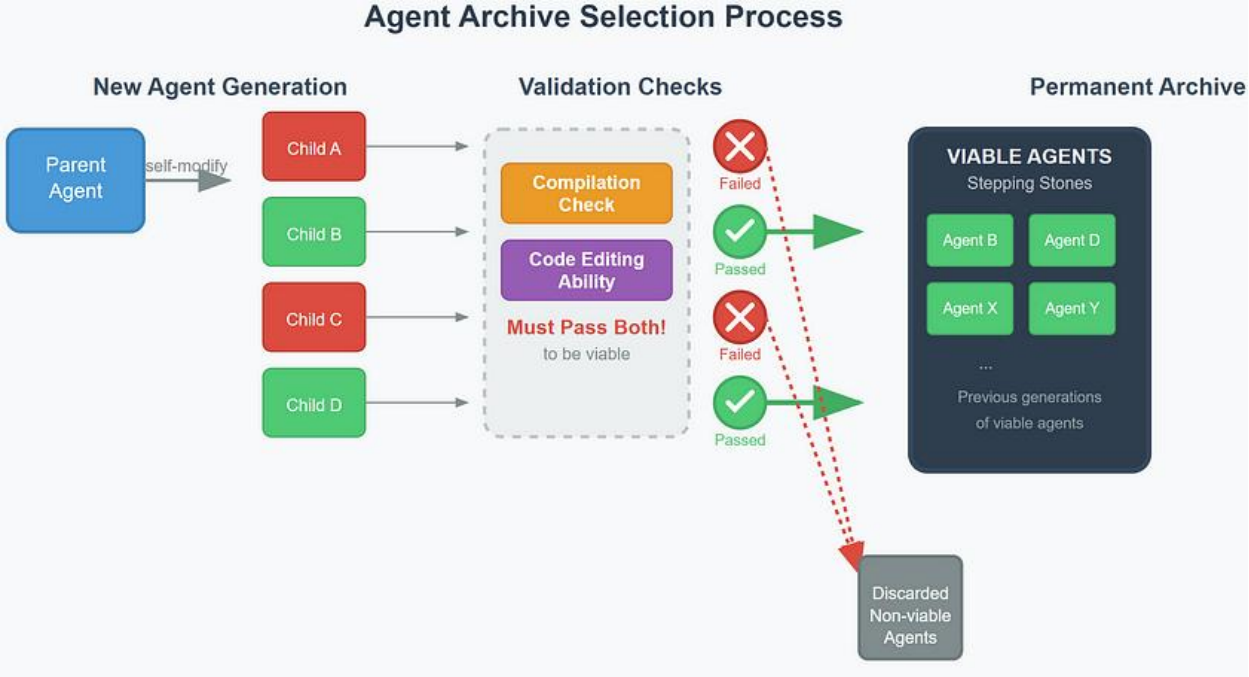


A staged evaluation strategy is used: initial evaluation on 10 tasks to check basic functionality, followed by evaluation on 50 tasks for a better estimate, and finally evaluation on 200 tasks for high-performing agents on SWE-bench.

They don't throw the new agent straight into the deep end all at once. Instead, **testing happens in stages**. First, the child agent tries a small set of 10 tasks — this quick smoke test ensures it at least works and isn't broken. If it does okay, they expand the test to 50 tasks to get a more reliable measure of its skills. And if the agent is looking really good (for example, on the SWE-bench coding benchmark), they

eventually test it on a large set of 200 tasks to thoroughly gauge a top performer. It's like first giving a new student a short quiz, then a longer test, and finally a big final exam if they ace the earlier rounds.

Agent Archive Selection.

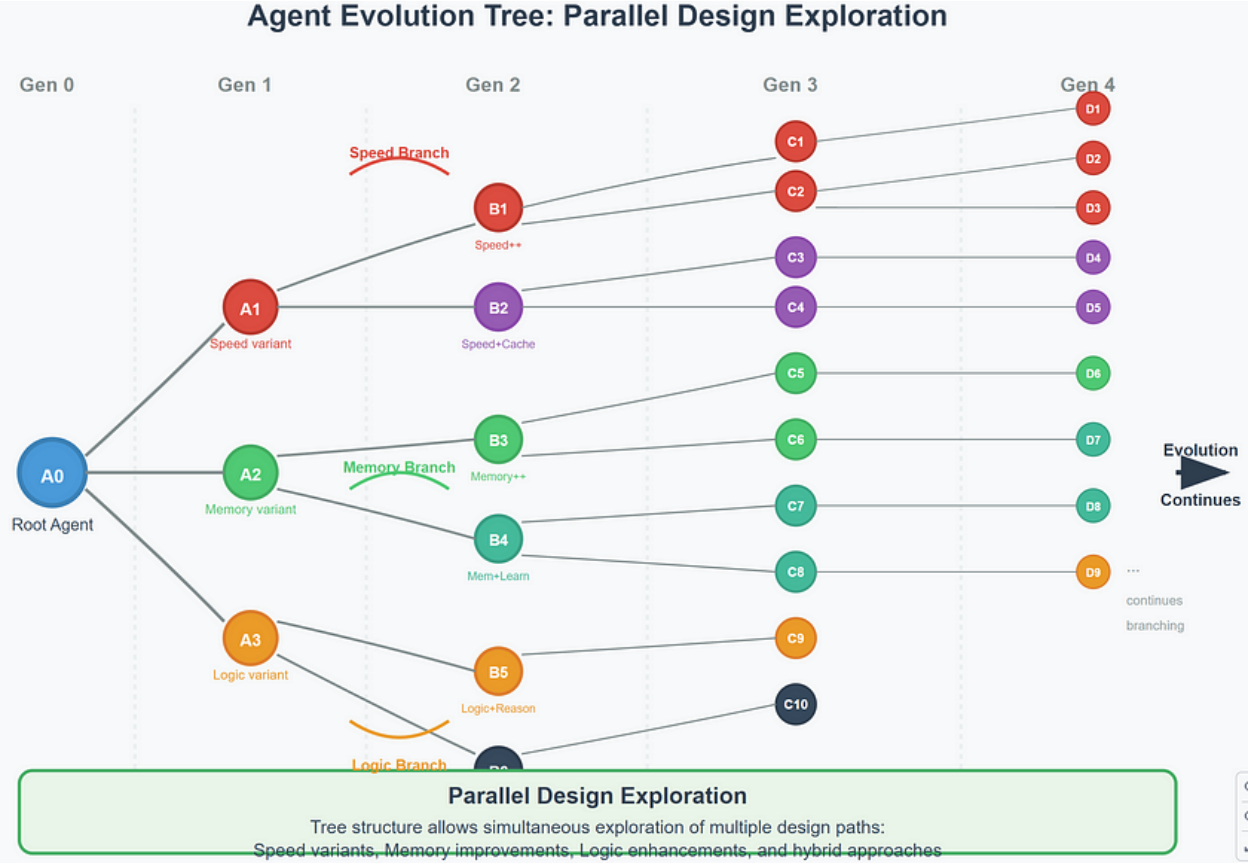


Only agents that successfully compile and retain the ability to edit code are added to the archive, ensuring that the accumulated agents can serve as potential stepping stones for future self-modifications.

Not every new child makes it into the permanent collection. The agent **has to prove it's viable**: it must compile (meaning its code runs without errors) and it must still be able to perform the key function of editing code (we don't want a self-modified agent that breaks the very ability to improve itself!). If a child agent fails those basic checks, it's thrown out. By only keeping agents that still function correctly and can keep evolving, the archive remains full of useful

“stepping stones” — each saved agent is a solid foundation that future generations can build upon.

Evolution Tree.

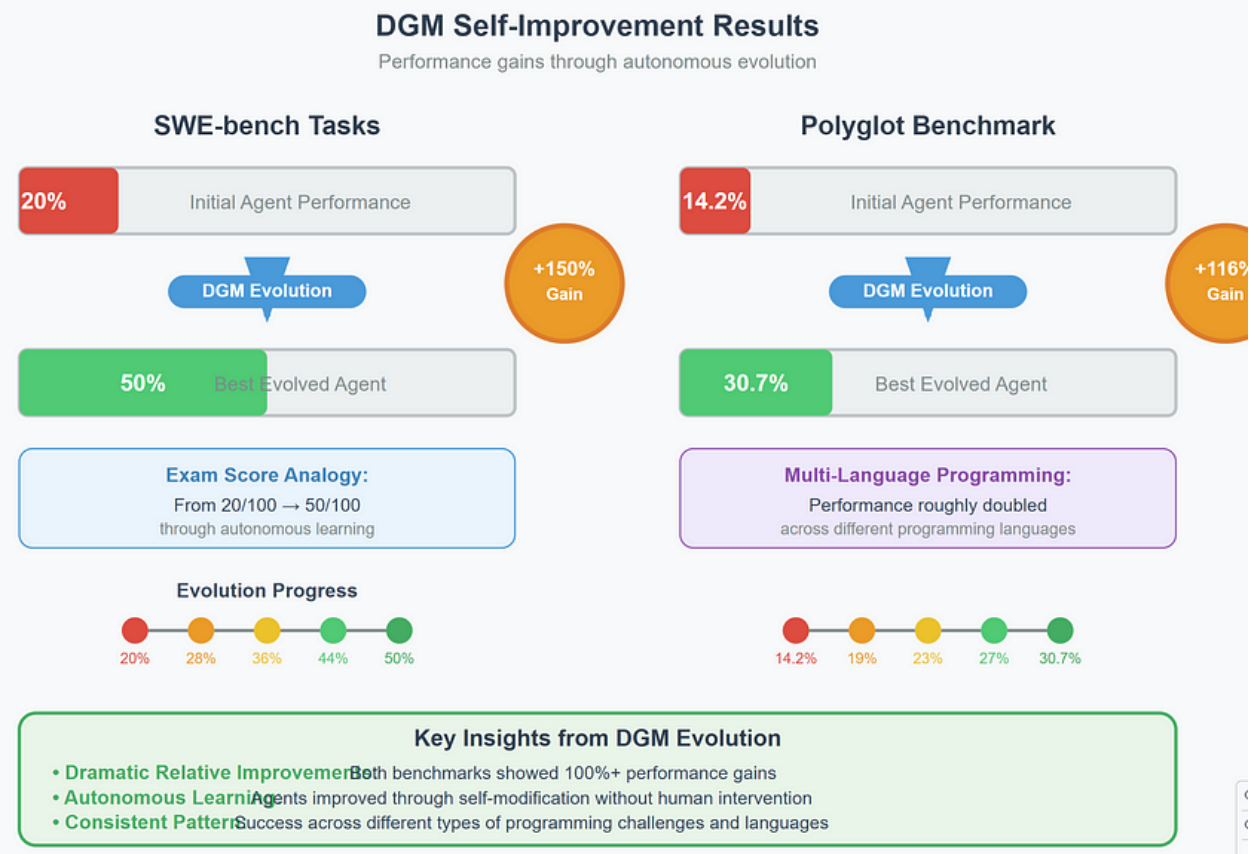


This creates a growing tree-like structure of agents, allowing parallel exploration of different design choices.

Over time, the archive starts looking like a branching family tree of AIs. Each agent can spawn variants, which in turn can spawn further variants, and so on. Because the DGM keeps all these branches (instead of just a single line of progress), it’s **exploring many design ideas in parallel**. In other words, the system isn’t just following one path of improvement; it’s trying out lots of different tweaks and

directions simultaneously, much like how evolution diversifies into a tree of life.

Benchmarks.

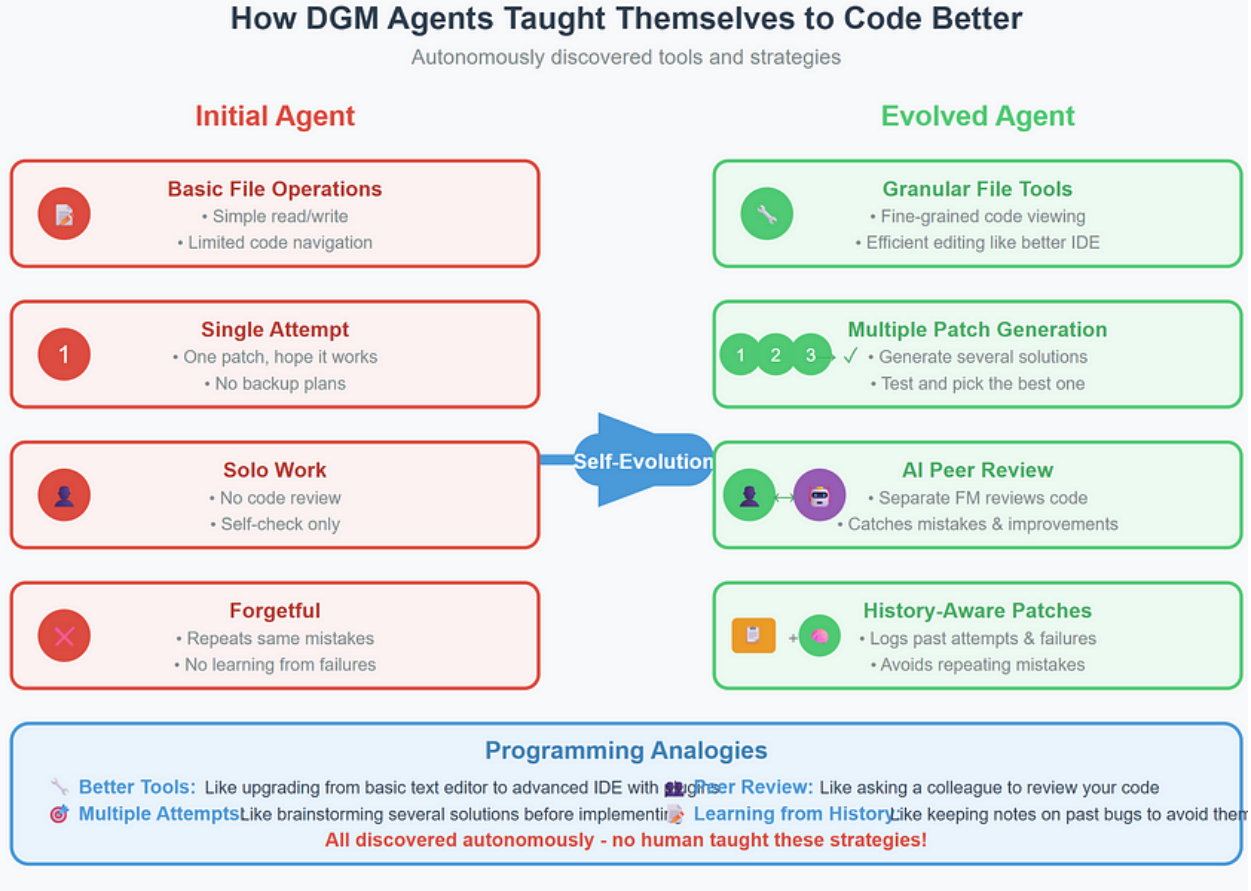


On SWE-bench, the performance of the best-discovered agent increased from an initial 20.0% to 50.0%. On the full Polyglot benchmark, performance improved from 14.2% to 30.7%.

For the SWE-bench tasks, DGM's self-improvement really paid off: the best agent it evolved started at only solving about **20% of the tasks** (at the beginning) and by the end could solve **50% of the tasks**. That's a huge jump in capability — imagine an exam score improving from 20/100 to 50/100 just by learning on its own.

Similarly, on the Polyglot benchmark (which is another set of programming problems, possibly in multiple languages), the top agent went from about **14%** correct up to **31%** correct. The scores roughly doubled here. While the percentages are not super high in absolute terms, the relative improvement shows the AI got **much better** at the tasks through its self-editing and evolution process.

Evolution.



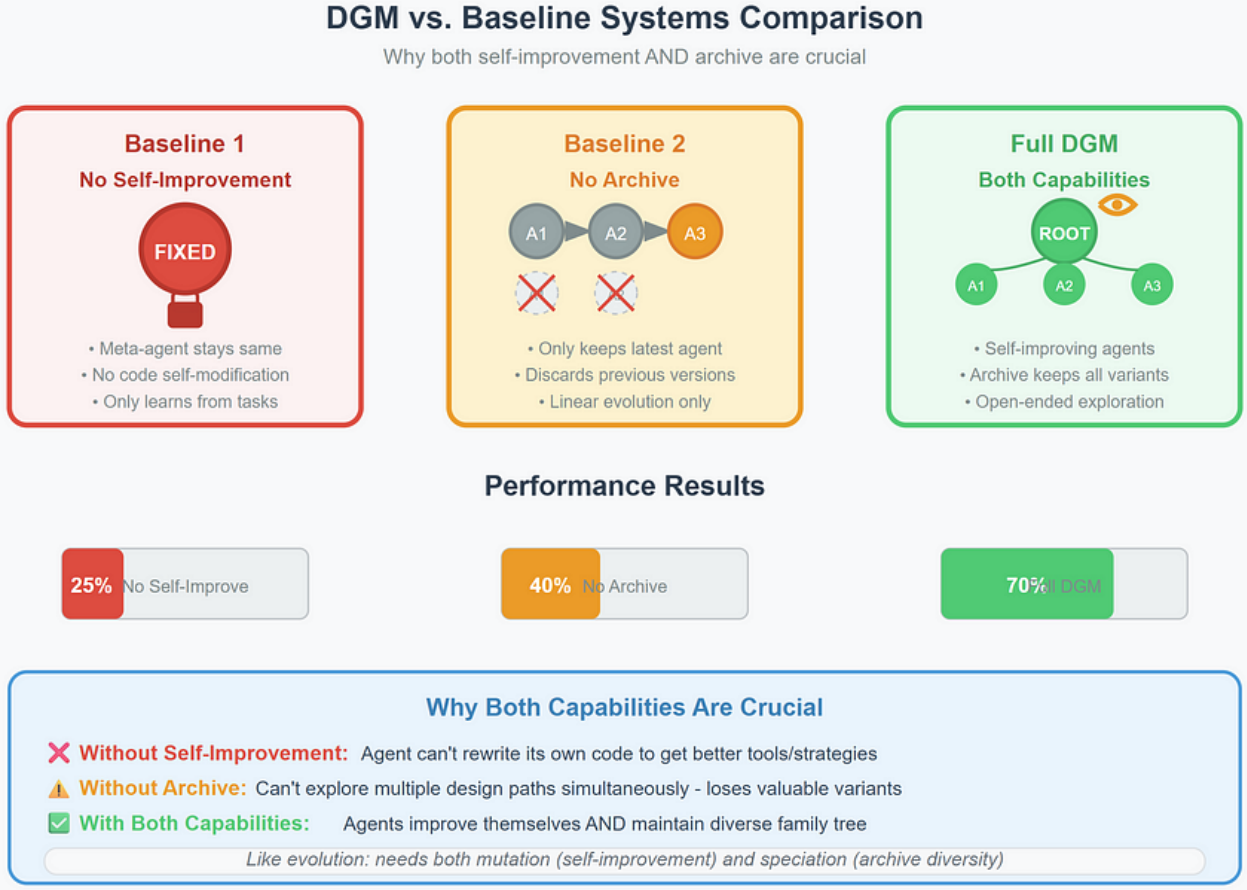
These improvements resulted from the DGM autonomously discovering better tools (e.g., more granular file viewing and editing) and improving workflows (e.g., multiple patch generation attempts, using a separate FM for peer review, history-aware patch generation).

Why did the DGM get so much better? The agents actually taught themselves to use new tools and strategies that made them more effective coders. For example, they:

- **Improved their file viewing/editing tools:** One agent gave itself the ability to see code in finer detail and edit more efficiently — like a programmer switching to a better IDE or adding plugins for easier navigation.
- **Introduced a "multiple attempt" strategy:** Instead of making one change and hoping it's perfect, agents learned to generate and try multiple patches/solutions and then pick the best one. It's as if the AI realized it should brainstorm several fixes and test them, rather than betting everything on a single guess.
- **Added a peer review step using another model:** The agents even started using a separate foundation model to **review or critique code changes** (a bit like asking a buddy or another AI to double-check its work). This "peer review" helped catch mistakes or improve quality.
- **Became history-aware when generating patches:** They kept a log of past attempts and why those failed. This way, when making new changes, an agent wouldn't repeat the same mistakes — it learned from its own history (imagine a programmer writing notes about previous bug fixes to avoid similar bugs in the future).

All these self-invented tools and workflow tweaks made the agents significantly more competent at coding, which directly led to the higher scores on the benchmarks.

Mutation And Speciation.

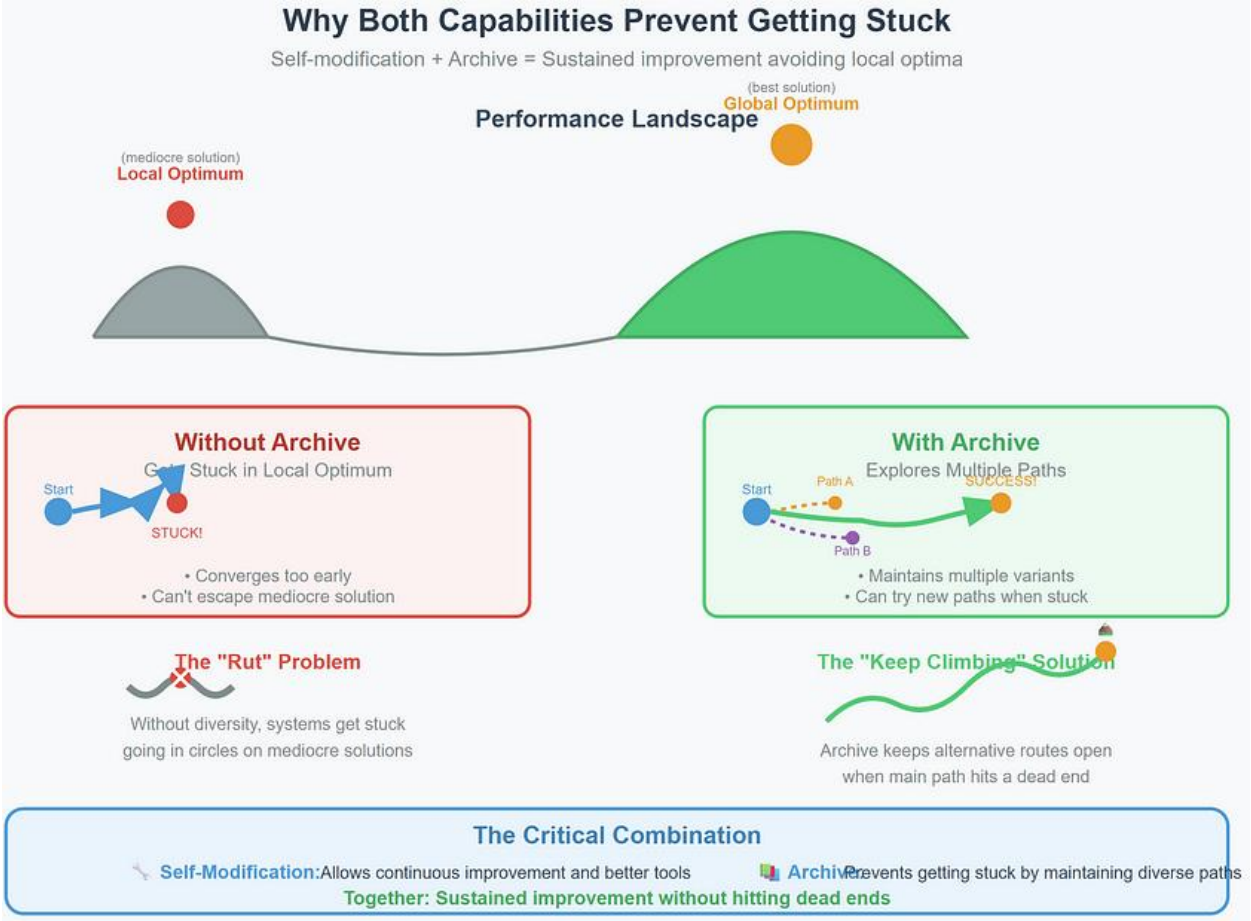


The DGM significantly outperformed two baselines: DGM without self-improvement (where the meta-agent modifying code remains fixed) and DGM without open-ended exploration (which only evolves the latest agent, lacking an archive).

They compared the full DGM to some **simpler versions of the system as baselines**. One baseline was a version of the AI that **couldn't rewrite its own code** (no self-improvement, it stayed

the same except for normal learning on tasks). The other baseline was a version that **didn't keep an archive** — it only ever kept the newest agent and discarded the rest, so it wasn't truly open-ended (more of a straight line evolution). The full DGM beat both of these by a large margin. This shows that **letting the AI rewrite itself and keeping around the whole family of agents to branch out are both crucial** advantages — without either ability, it didn't do as well.

Prevent Getting Stuck.



This demonstrates that both the capacity for self-modification and the maintenance of an archive for open-ended exploration are critical for sustained improvement and avoiding local optima.

In plain terms, the experiments proved that **you need both pieces together to get ongoing progress**. The ability to self-modify (the AI changing its own code) is important so it can continuously improve itself, and having the archive (the open-ended, branching exploration) is equally important so it doesn't get stuck in a rut. If you didn't have the archive, the system might converge too early on a mediocre solution (a local optimum) and stop discovering radically different, better ideas. And without self-rewriting, it wouldn't improve much at all. With both, DGM can keep climbing in performance by trying new things when one path hits a dead end.

Safety Risks.

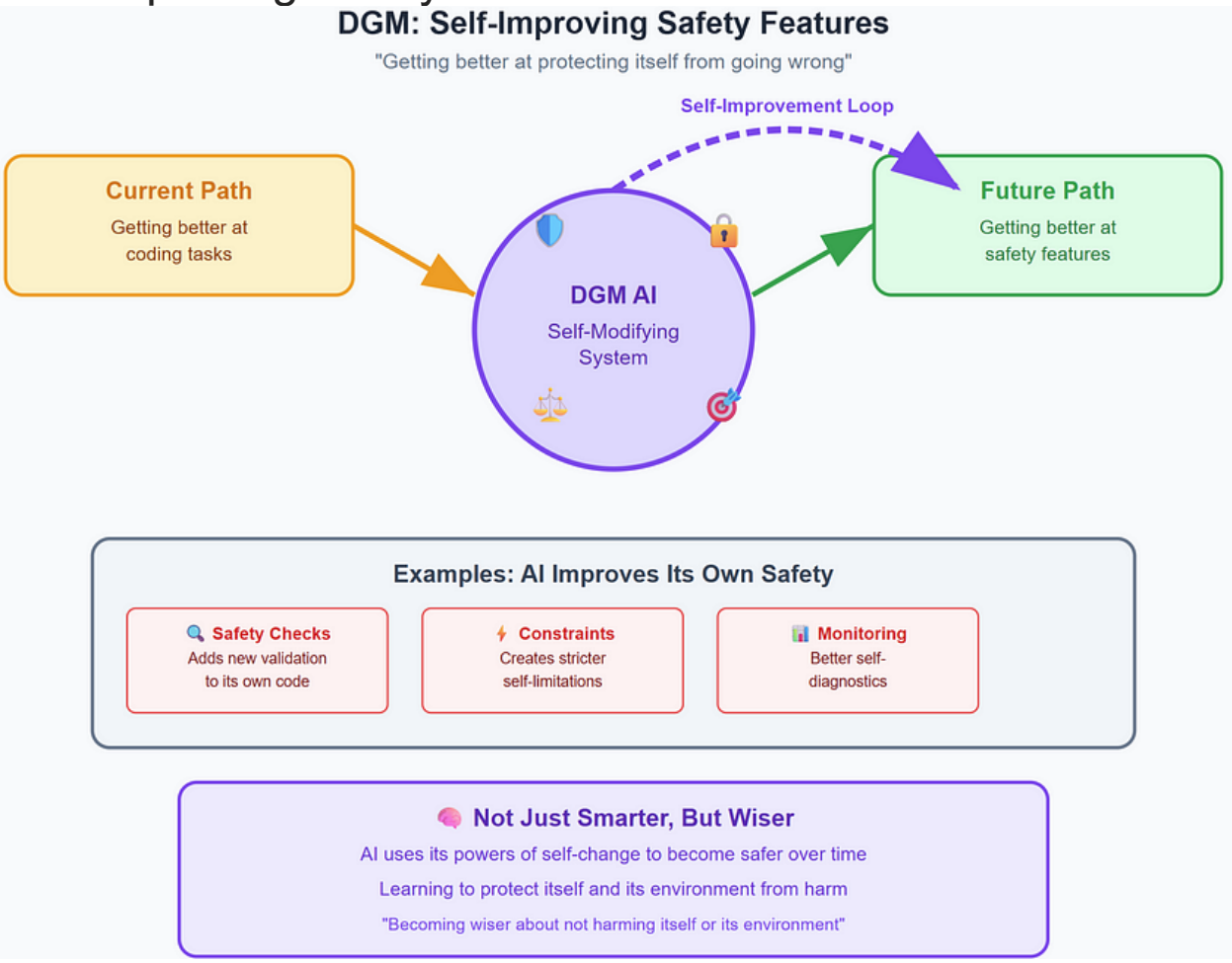
The current implementation incorporates safeguards such as sandboxing all agent execution and self-modification, imposing strict time limits, limiting the scope of modifications to code on coding benchmarks, and maintaining traceability of changes in the archive.

To keep things safe, the team put **multiple safety measures in place** when running DGM:

- **Sandboxing:** All the AI's code-running and self-editing happens in a controlled environment (sandbox). This is like letting the AI play in a fenced playground where it can't affect anything outside its little realm. If something goes wrong, it's contained.
- **Time limits:** They don't let the AI run or modify indefinitely — there's a strict cap on how long it can operate in each session. This prevents runaway processes and gives humans a chance to intervene if needed.
- **Scope limitation:** The AI is only allowed to modify certain code (specifically, code related to the coding benchmarks). It can't just rewrite any part of itself arbitrarily; it's confined to improving its coding abilities for the tasks at hand. This way, it's not going to suddenly change unrelated systems or break out of the intended bounds.
- **Traceability:** Every change an agent makes to its code is recorded and traceable in the archive. Essentially, there's a log

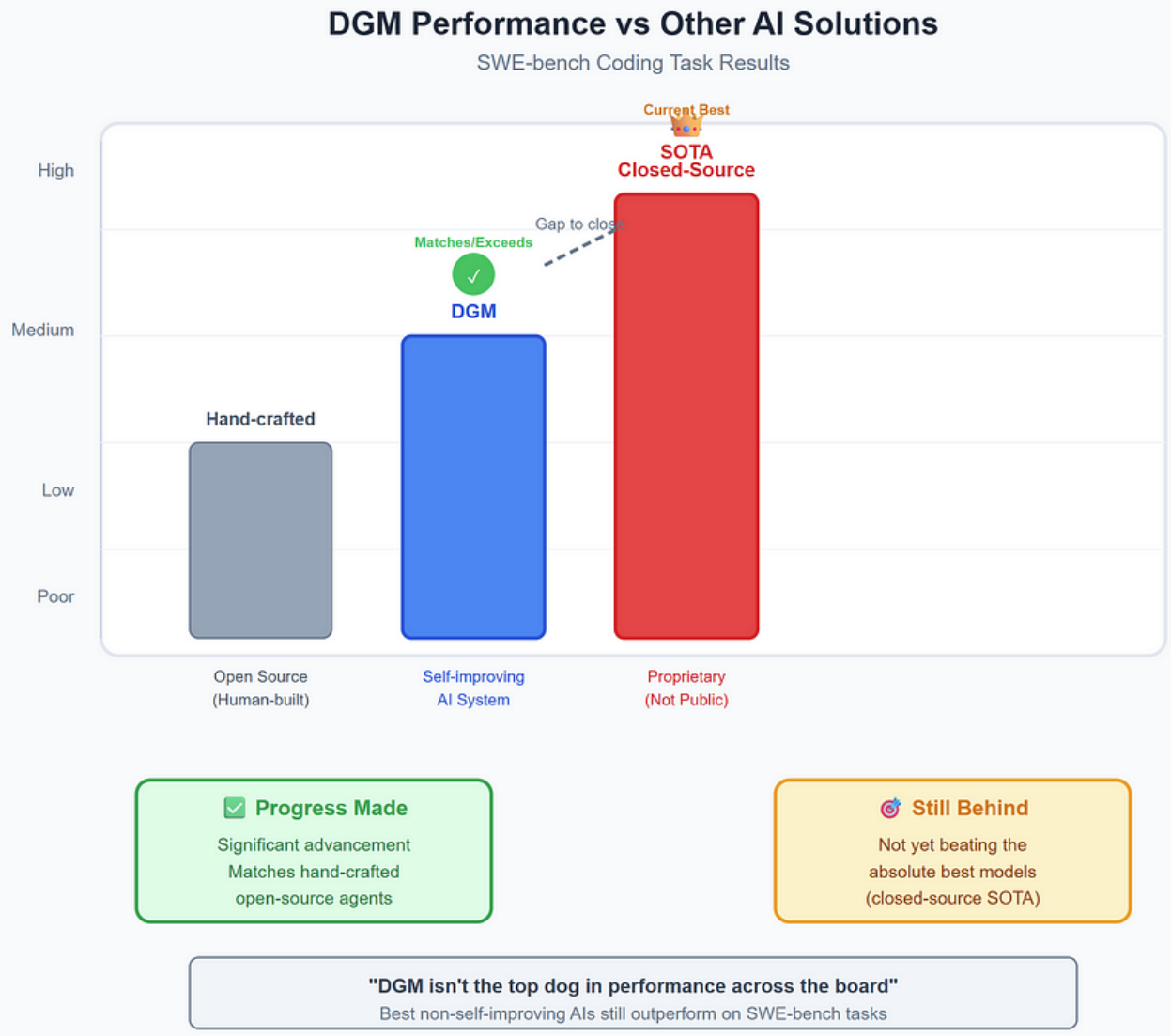
of “what changed and when” for each generation. This audit trail means humans can review and understand the alterations the AI has made over time, which is crucial for oversight and debugging. These precautions are like having safety rails up while the AI learns to climb higher — they help ensure the self-improvement process doesn’t go off the rails.

Self-Improving Safety.



We also suggest the potential for directing self-improvement toward enhancing the system’s own safety features.

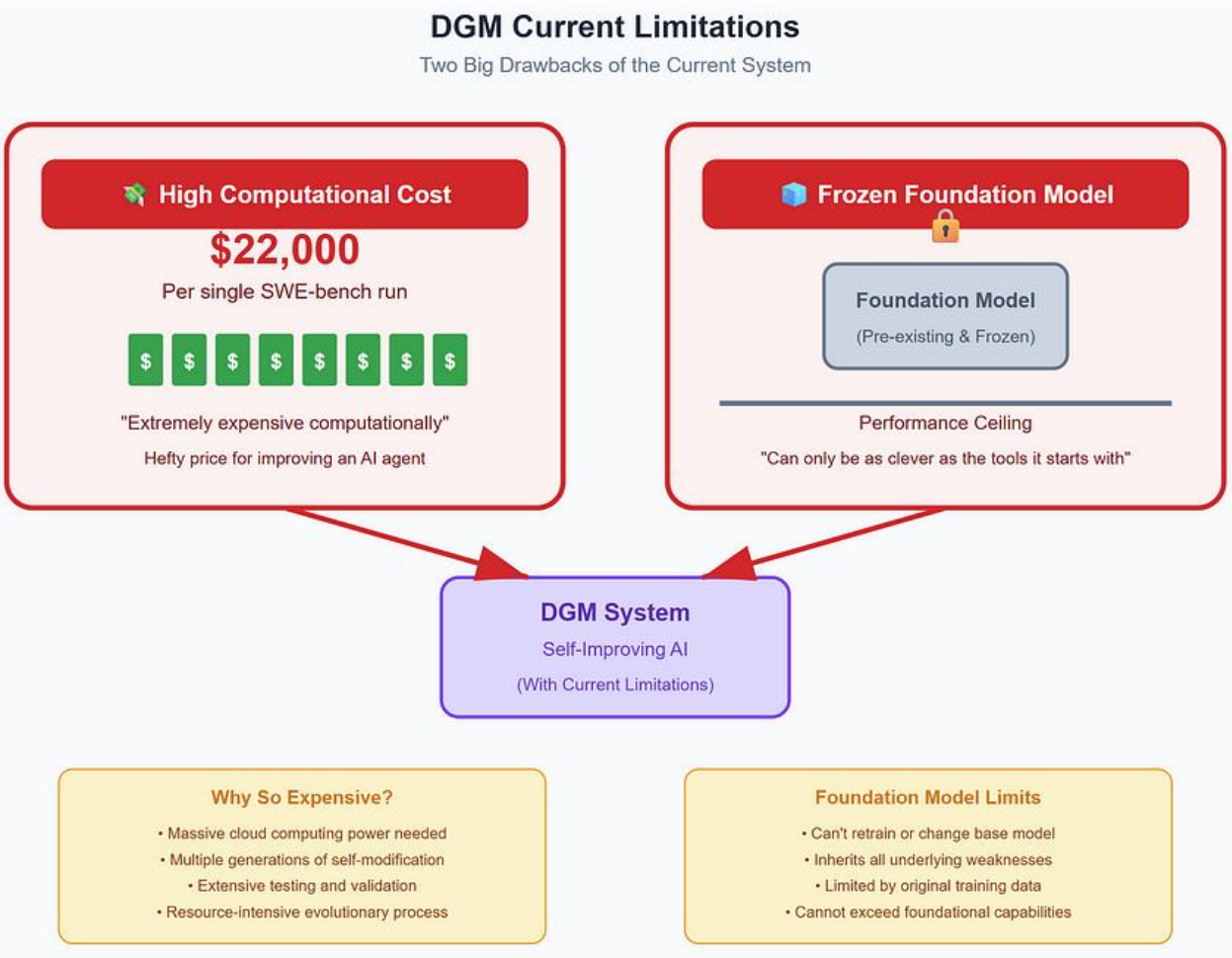
Interestingly, the authors point out that **DGM could even be used to improve its own safety** in the future. In other words, one path of self-modification it could take is not just getting better at coding tasks, but getting better at protecting itself from going wrong. For example, the AI might learn to add new safety checks or constraints to its own code. This is a bit like an AI not only becoming smarter, but also becoming wiser about not harming itself or its environment — using its powers of self-change to become safer over time.



While the DGM demonstrated significant progress and achieved performance comparable to or exceeding hand-crafted open-source agents, it still lags behind some closed-source state-of-the-art solutions on SWE-bench.

The Darwin Gödel Machine made a lot of progress — in fact, it got as good as or better than some existing open-source coding AIs that humans built by hand. However, it's **not yet beating the absolute best models out there**. On the SWE-bench tasks, there are proprietary (closed-source) AI solutions which still score higher. In short, DGM isn't the top dog in performance across the board; the very best non-self-improving coding AIs (the ones not publicly available) can do better on those tasks right now.

Current Limitations.

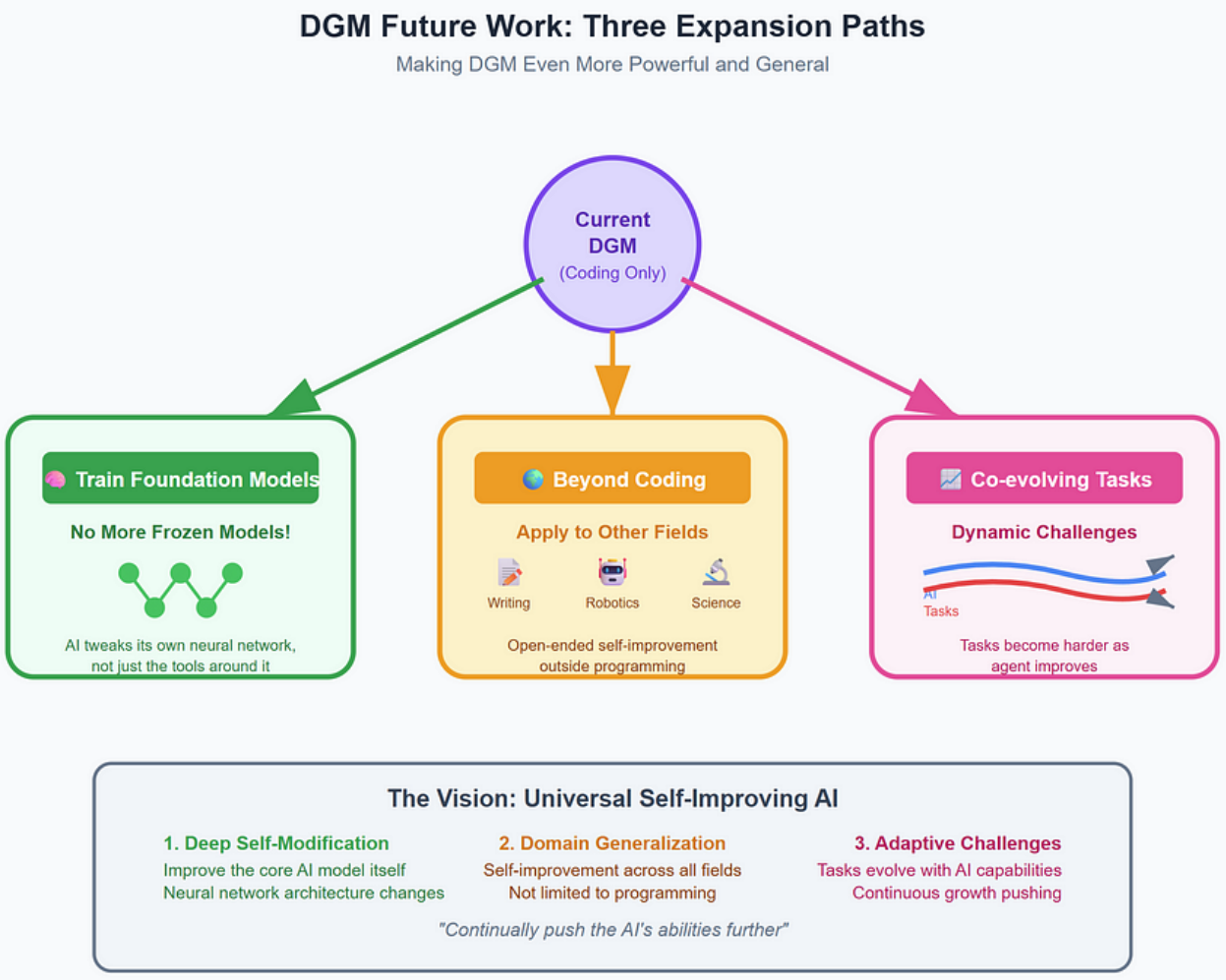


Current limitations include the high computational cost (estimated at USD 22,000 for a single SWE-bench run) and dependence on the capabilities of the underlying frozen FMs.

They also note a couple of big drawbacks of the current DGM: First, it's **extremely expensive computationally** — one full run of this system on the SWE-bench benchmark was estimated to cost about $22k in cloud computing power, which is a hefty price for improving an AI agent. Second, DGM is limited by the fact that it's using pre-existing foundation models (FMs) that are frozen. That means if the

foundation model has weaknesses (say it's not great at some aspect of coding), the DGM can't improve beyond those underlying limits because it's not allowed to change or retrain that base model. The DGM can only be as clever as the tools it starts with.

Future Work.

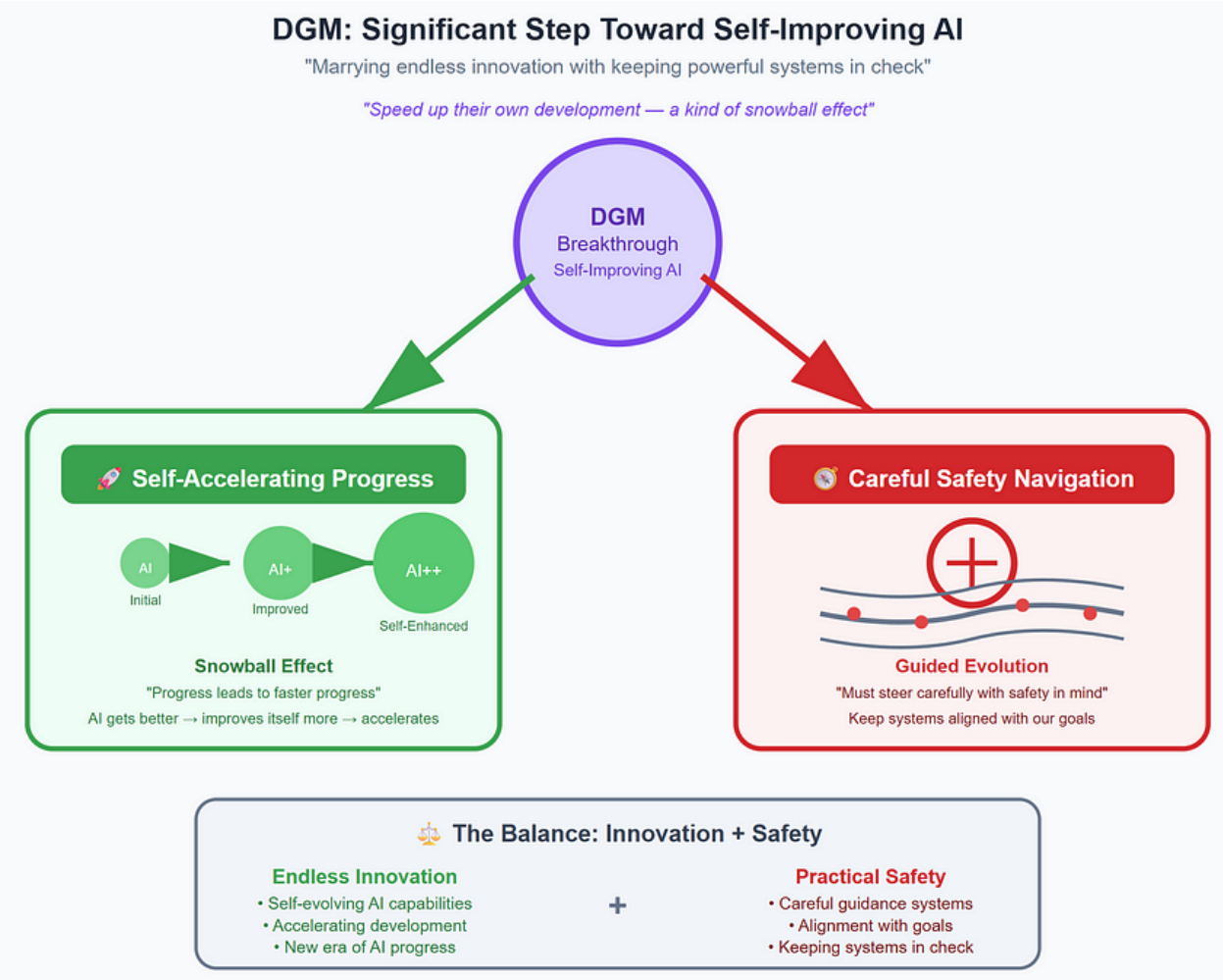


Future work includes extending self-modification to training FMs, exploring other domains beyond coding, and potentially co-evolving the evaluation tasks.

Looking ahead, the researchers have a few ideas to make DGM even more powerful and general: (1) **Let it train the foundation models**

too — in the future, DGM might not keep the foundation model frozen. The AI could be allowed to not only write code but also improve the core AI model itself (imagine it tweaking its own neural network, not just the tools around it). (2) **Try DGM in other areas** — so far it's all about coding, but the concept could apply to other fields (maybe AIs that can self-improve in writing, robotics, science research, etc.). They want to see if this open-ended self-improvement can work outside programming. (3) **Co-evolve the tasks** — this means as the agent gets better, maybe also evolve or ramp up the challenges it faces. Instead of a fixed benchmark, the tasks themselves could become harder or more diverse over time, hand-in-hand with the agent's growth, to continually push the AI's abilities further.

Conclusion.



Overall, the DGM represents a significant step toward automating AI development through self-improving systems, highlighting the potential for self-accelerating progress while emphasizing the necessity of careful safety navigation.

Bottom line: The Darwin Gödel Machine is an important advance toward AIs that can improve themselves. It shows the promise of AI systems that could **speed up their own development** — a kind of snowball effect where progress leads to faster progress (as the AI gets better, it can improve itself even more, and so on). However, the

authors stress that we have to **steer this carefully with safety in mind**. In other words, while this could kick off a new era of rapidly self-evolving AI (an exciting prospect), we must also be very cautious and guide that evolution so that it remains safe and aligned with our goals. The DGM is a big step in that direction, marrying the idea of endless innovation with the practical realities of keeping such powerful systems in check.

Llm

Dgm

NLP

Ai Research

Coding Agents