# Automated algorithmic innovator MVP

The Innovation Game Labs

August, 2025

## 1 Introduction

Recent advances in LLM-powered coding have allowed LLMs to incorporate the coding ability into evolutionary processes for innovating entire algorithms given a target computational task, both at the conceptual and direct implementation level. In particular, Google DeepMind have released a line of research [1, 2] where LLM-based systems are capable of finding novel algorithms beyond current human knowledge. This is both exciting and alarming for the future outlook of innovation and science, as such self-improving systems can lead to technological singularity that is controlled by a small number of centralised for-profit entities.

**Automating algorithmic improvements with LLMs** LLMs have shown remarkable capability for code generation [3, 4] as well as mathematical understanding [5, 6]. Traditional evolutionary approaches to searching programs implementing algorithms typically involve manual specifications for intermediate code modifications that plug into the framework for genetic programming [7, 8]. With LLMs, one can use prompt engineering to obtain valid program modifications and thus define transitions directly in the space of possible programs [9]. Evolutionary strategies that combine LLMs to search the space of functions have led to the discovery of novel mathematical results as shown in FunSearch [1, 10], and further applications have led to successful automation in the domain of science [11], Bayesian optimisation [12] and academic research idea generation [13]. This idea has recently been generalised to searching directly over programs [2] to evolve algorithms more flexibly given some target computational problem.

**Automating decentralised algorithmic innovation in TIG** At The Innovation Game (TIG), our goal is to empower the open source community by creating a decentralised ecosystem for algorithmic innovation that leads to competitive algorithmic innovation with industry dominant closed source for-profit entities. In particular, the compute network accrued by TIG as well as the competitive open source nature of the reward incentives provides a decentralised alternative to the conventional privatised research labs that excel in the current AI landscape where massive compute is required for reaching state-of-the-art results. The rise of automated LLM-based algorithmic innovation machines has prompted us to build our own custom framework for automated algorithmic innovators that can eventually be plugged into the network of TIG alongside current human innovators. In this whitepaper, we present the technical foundations for our implementation of our baseline automated algorithmic innovator and directions for further development.

## 2 Automated algorithmic innovator MVP

### 2.1 High-level overview

We present a minimal implementation of the automated innovator that is very easily understood and yet capable of algorithmic improvement, while being highly customizable for adapting to more sophisticated evolution pipelines and prompt construction. This minimum viable product (MVP) will form the foundation for further research and development of the automated innovator within TIG.

**Goals** At a high level, we design the MVP with the following goals in mind:

- Accessible to a wide audience of researchers and software engineers
- Highly customisable and easy to understand codebase for others to build on
- Exhibits basic ability for automated algorithmic innovation

Our pipeline is based on distilling key ideas from existing works like FunSearch [1] and AlphaEvolve [2] into a simplified minimal pipeline with modular components for adding more sophisitication.

**Design** To align with these goals, we design our MVP automated innovator pipeline in Python. The overall pipeline consists of a *LLM* that generates code candidates and reasoning text, a code *evaluator* that runs candidate algorithm programs and a collection of *prompts* constructed based on the target computational problem and valid algorithm input and output structure:

- The user specifies the computational problem to be solved, the evaluation metrics the *LLM* will receive from the *evaluator* and the target metric to improve upon. These will be combined into a *system prompt* which conditions the *LLM* to a desired personality and response structure for our framework structure, which requires the *LLM* to output a separate *code* block and *reasoning* comments
- The user provides a sample algorithm code implementation that contains basic comments to outline the algorithm process. Note this code sample does not have to be a valid program and can in fact even be a code skeleton outline, as the pipeline will automatically attempt to correct invalid program submissions
- Finally, the user also specifies the *evaluator* backbone code that imports the algorithm implementation from the program submissions as a Python function that can be used inside a standardised evaluation pipeline. This is similar to FunSearch [1], but we allow complete freedom in the modification of the algorithm function like AlphaEvolve [2]

These components make up the following evolution pipeline:

1. The proposed program *code* (provided in the first step, generated after) is fed into the *evaluator* components for execution on a fixed set of problem instances as done in FunSearch [1]. In case of invalid *code*, the *evaluator* will return the Python interpreter error traceback instead of evaluation metrics
2. The *evaluator* feedback results and *reasoning* comments for writing this code (none in the first step) are concatenated with the program *code* into the next prompt for sending to the *LLM*
3. The *LLM* response is checked for valid output structure so that the *code* and *reasoning* blocks can be extracted and stored. In case of invalid structure, we resend the prompt with a indication of invalid output response attached to it, and keep repeating this until a valid response is generated
4. We go back to step 1 with the new *code* and *reasoning* blocks while appending all intermediate output components to the evolution chain storage, and stop if we reached the maximum number of evolution steps

The overall evolution structure forms a simple linear chain as shown in Figure 1, without any branching or mixing between individual chains. This is a significant simplification compared to the island-based evolution structure used in FunSearch and AlphaEvolve [1, 2], which use a dedicated program database and sampling schemes to construct new prompts. In contrast, our linear evolution chain gives direct access to the previous proposed program with its evaluation performance, and allows storing results and knowledge from the deeper past inside *reasoning* responses. The code implementation of this pipeline is available on our public GitHub repository[1].

[1] `https://github.com/tig-foundation/automated-ai-innovator`

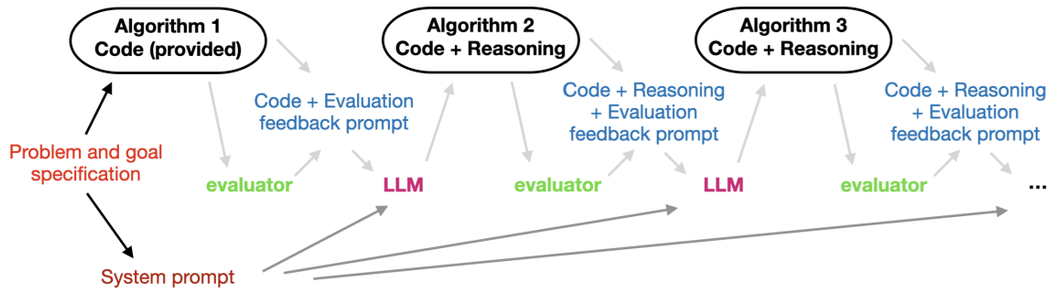


Figure 1: **Schematic of the automated algorithmic innovator MVP.** The pipeline as described in subsection 2.1 is shown here as a flowchart, with separate components individually coloured. Note the linear evolution chain structure, which is the simplest structure for running a self-improving agent that uses its own past proposal and evaluation feedback.

## 2.2 Target computational problems

Given our basic functionality, we test our MVP on prototype TIG "mini-challenges", which are designed to be simple computational problems that have a qualitatively similar computational asymmetry aspect (hard to improve, easy to verify) to real TIG challenges. These mini-challenges are constructed with the following principles in mind:

- Easy to understand intuitively with minimal background knowledge
- Easy to visualize the computational problem and candidate solutions
- Relies heavily on heuristics, often with human-interpretable justification, allowing one to create interpretable reasoning chains for algorithm improvements
- Has good room for algorithmic improvement and innovation to showcase algorithmic innovation ability without sophisticated and compute-heavy pipelines

We propose and implement the following mini-challenges:

**Kernel Density Estimation (KDE)** In this task, the goal is to construct a Gaussian mixture model (GMM) that describes the underlying density of a set of points $\mathbf{x} \in \mathrm{R}^D$ sampled from a non-injective nonlinear transformation of a unit Gaussian random variable. This transformation is implemented with a randomly initialised multilayer perceptron (MLP) using rectified linear activation functions [14], and thus the ground truth density $p(\mathbf{x})$ is intractible. We sample training $\mathbf{x}_{\text{train}} \sim p(\mathbf{x})$ and test $\mathbf{x}_{\text{test}} \sim p(\mathbf{x})$ sets, and the evaluation metric to maximize is the test log likelihood of the test set points using the GMM constructed with the training set points. Due to $p(\mathbf{x})$ being intractable, finding the best GMM is an asymmetric problem. Valid algorithms take as input the training set $\{\mathbf{x}_{\text{train}}\}$ and output GMM means $\in \mathrm{R}^D$ and covariances $\in \mathrm{R}^{D\times D}$, with a freedom to choose the number of mixture components. Covariances that are not positive definite will lead to an error in the evaluator pipeline. Heuristics for finding good GMM parameters have become textbook standard [15].

**Clustering** Here, the goal is to group a set of points $\mathbf{x} \in \mathrm{R}^D$ by assigning each point to one of the $K$ clusters, where $K$ is fixed as part of the problem specification. The points are sampled from a similar transformation as in KDE, but we do not split between training and test set as the evaluation metric for clustering is computed on the set of points clustering was performed over. In particular, we use a pairwise connectivity-based clustering score

$$\sum_{i=1}^{n} \sum_{j \in \mathcal{N}_i} \frac{1}{\text{rank}_{ij}} \cdot 1_{c_i \neq c_j} \tag{1}$$

Valid algorithms take as input the set of points $\{\mathbf{x}\}_i$ and output cluster identities $c_i \in \{1, 2, \ldots, K\}$ per point.

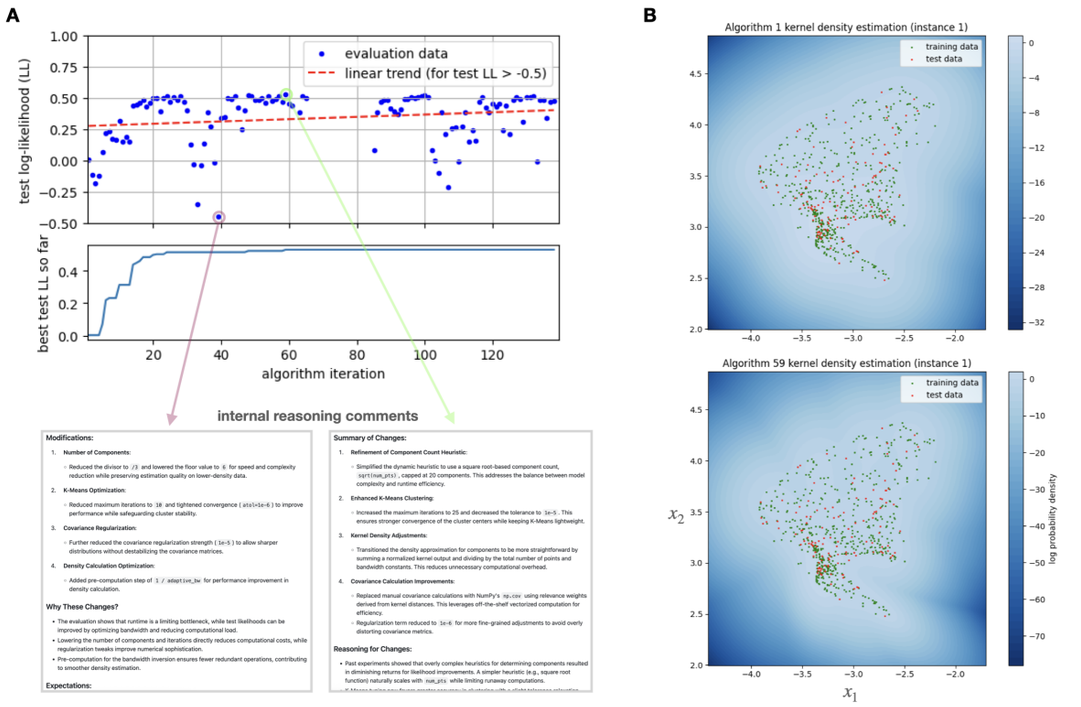


Figure 2: **Example evolution run with the KDE problem with $D = 2$. (A)** Performance plot of a successful evolution chain (top). Invalid programs resulting in NaN scores, which show up as missing entries in the scatter plot. In addition, the LLM occasionally generates bad outlier programs that lead to very low test LLs, which are not plotted and considered in the linear trend fit (red dotted line) for keeping the visualisation clear. The best performing algorithm was at index 59, and we show example internal reasoning comments along the evolution process (bottom). **(B)** Visualising the log probability density estimated from algorithm 1 (top) and algorithm 59 (bottom).

**Histogram Density Estimation** Similar to KDE, we now perform density estimation but instead of using a GMM we use piecewise constant probability density candidates (histograms with potentially non-uniform binning). As histograms suffer from the curse of dimensionality, we restrict ourselves to $D = 1$. In addition, we restrict the points to a finite range $x \in [-1, 1]$ which is achieved by replacing the activation function in the MLP with a sinusoid [16], including the final layer. Valid algorithms take as input the training set $\{x_{\text{train}}\}$ and output bin boundaries within the $[-1, 1]$ interval, with boundaries placed at $\pm 1$ by default. The algorithm has the freedom to decide the number of bins to use.

# 3 Preliminary results

We show a successful sample run on the KDE mini-challenge in Figure 2, where the pipeline discovered a better KDE algorithm using $k$-means clustering compared to the seed algorithm implementing Silverman's rule of thumb [15]. Our pipeline uses the internal reasoning comments (acting as a recurrent hidden state as shown in Figure 1) to justify and keep track of code changes. Using prompt and program evaluation history has been shown to improve evolution performance significantly [2], and in our pipeline the reasoning pattern forms a simple linear chain that can be visualised and studied. The structure of the reasoning comments is not currently specified in the system prompt, and is designed by the LLM autonomously. Note that many evolution runs typically fail to improve or get stuck in a suboptimal minimum, given the combinatorial complexity of the optimisation problem, which justifies the use of more complicated evolution dynamics and structures such as multi-population and island-based structures [1, 2].

# 4 Discussion and further work

We presented a MVP for the automated algorithmic innovator in the context of TIG. The minimalist and modular pipeline design allows one to easily expand the capabilities with various approaches to introduce more sophistication.

**More sophisticated evolution pipelines** Due to the combinatorial complexity of optimising directly in the space of all valid programs, our simple linear evolution chain is limited in the sample efficiency it can achieve. Many techniques from this field like cross-overs between independent evolution chains, genetic programming and island-based structures are used in state-of-the-art frameworks like FunSearch and AlphaEvolve [1, 2]. The simplest extensions to our pipeline are for example:

- Incorporating a program database instead of storing all past dependence in the most recent reasoning comment and the context of the LLM
- Introducing a cross-over mechanism between parallel evolution chains to reset runs that are stuck in bad optima

Several open source implementations of more sophisticated evolution frameworks for algorithm optimisation are available, like open source AlphaEvolve repositories aiming to reproduce the original publication (https://github.com/codelion/openevolve and https://github.com/shyamsaktawat/OpenAlpha_Evolve), as well as related self-evolving coding agent frameworks like https://github.com/jennyzzt/dgm.

**Fine-tuning LLMs for algorithmic innovation** Training LLMs is a costly project [17], and to obtain LLMs improved for specific tasks like conversation or algorithmic design, massive training costs can be avoided by fine-tune trained base LLMs. These techniques are referred to as "post-training" since they are applied after the expensive "pre-training" of the base LLMs on a large general corpus. There are various approaches to fine-tuning, including supervised approaches [18], reinforcement learning (RL) using a reward model trained on human preferences [19, 20], and more recently even self-supervised techniques [21]. Notably, DeepSeek-R1 [22] has shown that it is possible to get rid of the expensive critic neural network typically required in policy gradient methods like PPO [23] used to fine-tune LLMs with RL. By grouping multiple responses of the LLM to the same prompt and using the group average as a reward baseline, a stable learning signal can be obtained at a much lower cost. Furthermore, by rewarding coding and mathematics related Chain-of-Thought (CoT) paths that lead to executable and provably correct results, the fine-tuning process is able to elicit better coding and mathematical reasoning properties from the base LLM. In our pipeline, we can regard reasoning chains from evolution runs with their metric scores as data points for RL to fine-tune LLMs for improvement *algorithmic innovation* abilities across various computational problems.

**Incorporating RL-swarms for fine-tuning** Gensyn's RL-swarm[2] extends DeepSeek-R1's group-based approach to fine-tuning multiple LLMs simultaneously, allowing one to construct a group with LLM responses from multiple LLM peers. This new paradigm touches upon multi-agent RL, and they report improved training performance from preliminary results due to added exploration from LLM peers inside groups. The RL-swarm framework can be naturally combined with our automated innovator in a network setting, where individual innovator nodes may deploy different LLMs and can collaborate with each other by forming a RL-swarm to improve fine-tuning of their LLMs jointly.

**Multi-agent framework** In our MVP, we rely on a single LLM that handles all the code modifications and reasoning for algorithmic innovation. One can dissect the overall code improvement process into separate parts, such as code suggestions and reasoning chain formation, and assign separate LLMs to each. This moves to the multi-agent

[2] `https://github.com/gensyn-ai/paper-rl-swarm`

approach, which has been applied to various practical scientific research problems such as bridging knowledge gaps across expert domains [24].

# References


[1] Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

[2] Alexander Novikov, Ngân Vu, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. Technical report, Technical report, Google DeepMind, 05 2025. URL https://storage. googleapis . . . , 2025.

[3] Jianxun Wang and Yixiang Chen. A review on code generation with llms: Application and evaluation. In *2023 IEEE International Conference on Medical Artificial Intelligence (MedAI)*, pages 284–289. IEEE, 2023.

[4] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36:21558–21572, 2023.

[5] Samuel Holt, Zhaozhi Qian, Tennison Liu, Jim Weatherall, and Mihaela van der Schaar. Data-driven discovery of dynamical systems in pharmacology using large language models. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[6] Matteo Merler, Katsiaryna Haitsiukevich, Nicola Dainese, and Pekka Marttinen. In-context symbolic regression: Leveraging large language models for function discovery. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 4: Student Research Workshop)*, pages 589–606, 2024.

[7] John R Koza. Genetic programming as a means for programming computers by natural selection. *Statistics and computing*, 4:87–112, 1994.

[8] Wolfgang Banzhaf, Peter Nordin, Robert E Keller, and Frank D Francone. *Genetic programming: an introduction: on the automatic evolution of computer programs and its applications*. Morgan Kaufmann Publishers Inc., 1998.

[9] Clint Morris, Michael Jurado, and Jason Zutty. Llm guided evolution-the automation of models advancing models. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 377–384, 2024.

[10] Jordan S Ellenberg, Cristofero S Fraser-Taliente, Thomas R Harvey, Karan Srivastava, and Andrew V Sutherland. Generative modeling for mathematical discovery. *arXiv preprint arXiv:2503.11061*, 2025.

[11] Chris Lu, Cong Lu, Robert Tjarko Lange, Jakob Foerster, Jeff Clune, and David Ha. The ai scientist: Towards fully automated open-ended scientific discovery. *arXiv preprint arXiv:2408.06292*, 2024.

[12] Virginia Aglietti, Ira Ktena, Jessica Schrouff, Eleni Sgouritsa, Francisco JR Ruiz, Alan Malek, Alexis Bellot, and Silvia Chiappa. Funbo: Discovering acquisition functions for bayesian optimization with funsearch. *arXiv preprint arXiv:2406.04824*, 2024.

[13] Long Li, Weiwen Xu, Jiayan Guo, Ruochen Zhao, Xingxuan Li, Yuqian Yuan, Boqiang Zhang, Yuming Jiang, Yifei Xin, Ronghao Dang, et al. Chain of ideas: Revolutionizing research via novel idea development with llm agents. *arXiv preprint arXiv:2410.13185*, 2024.

[14] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958. Place: US Publisher: American Psychological Association.

[15] B.W. Silverman. *Density Estimation for Statistics and Data Analysis*. Chapman & Hall/CRC, London, 1986.

[16] Vincent Sitzmann, Julien Martel, Alexander Bergman, David Lindell, and Gordon Wetzstein. Implicit neural representations with periodic activation functions. *Advances in neural information processing systems*, 33:7462–7473, 2020.

[17] Goran S. Nikolić, Bojan R. Dimitrijević, Tatjana R. Nikolić, and Mile K. Stojcev. A Survey of Three Types of Processing Units: CPU, GPU and TPU. In *2022 57th International Scientific Conference on Information, Communication and Energy Systems and Technologies (ICEST)*, pages 1–6, June 2022.

[18] Nisan Stiennon, Long Ouyang, Jeffrey Wu, Daniel Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul F Christiano. Learning to summarize with human feedback. *Advances in neural information processing systems*, 33:3008–3021, 2020.

[19] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.

[20] Kai Ye, Hongyi Zhou, Jin Zhu, Francesco Quinzan, and Chengchung Shi. Robust reinforcement learning from human feedback for large language models fine-tuning. *arXiv preprint arXiv:2504.03784*, 2025.

[21] Jiaxin Wen, Zachary Ankner, Arushi Somani, Peter Hase, Samuel Marks, Jacob Goldman-Wetzler, Linda Petrini, Henry Sleight, Collin Burns, He He, et al. Unsupervised elicitation of language models. *arXiv preprint arXiv:2506.10139*, 2025.

[22] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[23] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[24] Shiva Aryal, Tuyen Do, Bisesh Heyojoo, Sandeep Chataut, Bichar Dip Shrestha Gurung, Venkataramana Gadhamshetty, and Etienne Gnimpieba. Leveraging multi-ai agents for cross-domain knowledge discovery. *arXiv preprint arXiv:2404.08511*, 2024.