

PyDayBCN2025



CODE EVOLUTION

Self-Improving Software with LLMs and Python

CAMILO CHACÓN SARTORI

29/NOV/2025



Workshop

It is divided into four parts:

1

Self-Refine

Automating bug fixes.

2

Code Evolution

Discovering new algorithms through genetic algorithms.

3

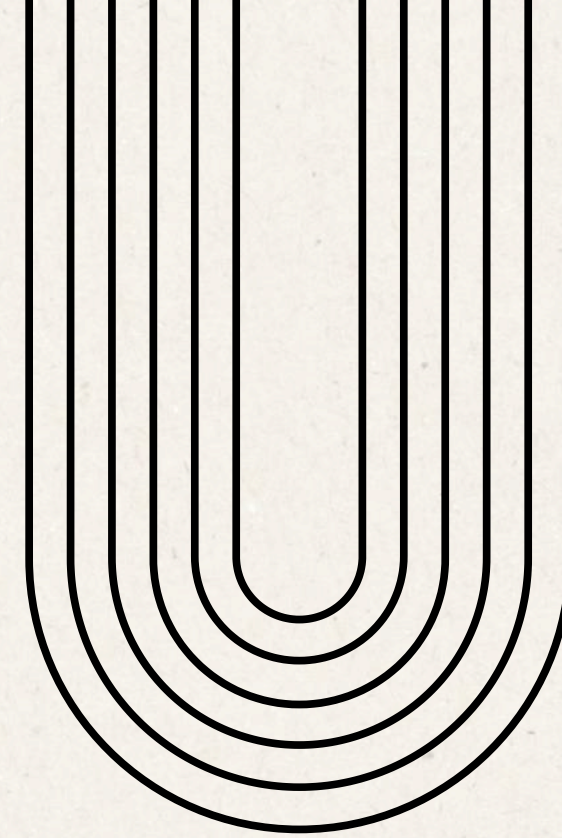
Hot-Swapping

Change the code being executed.

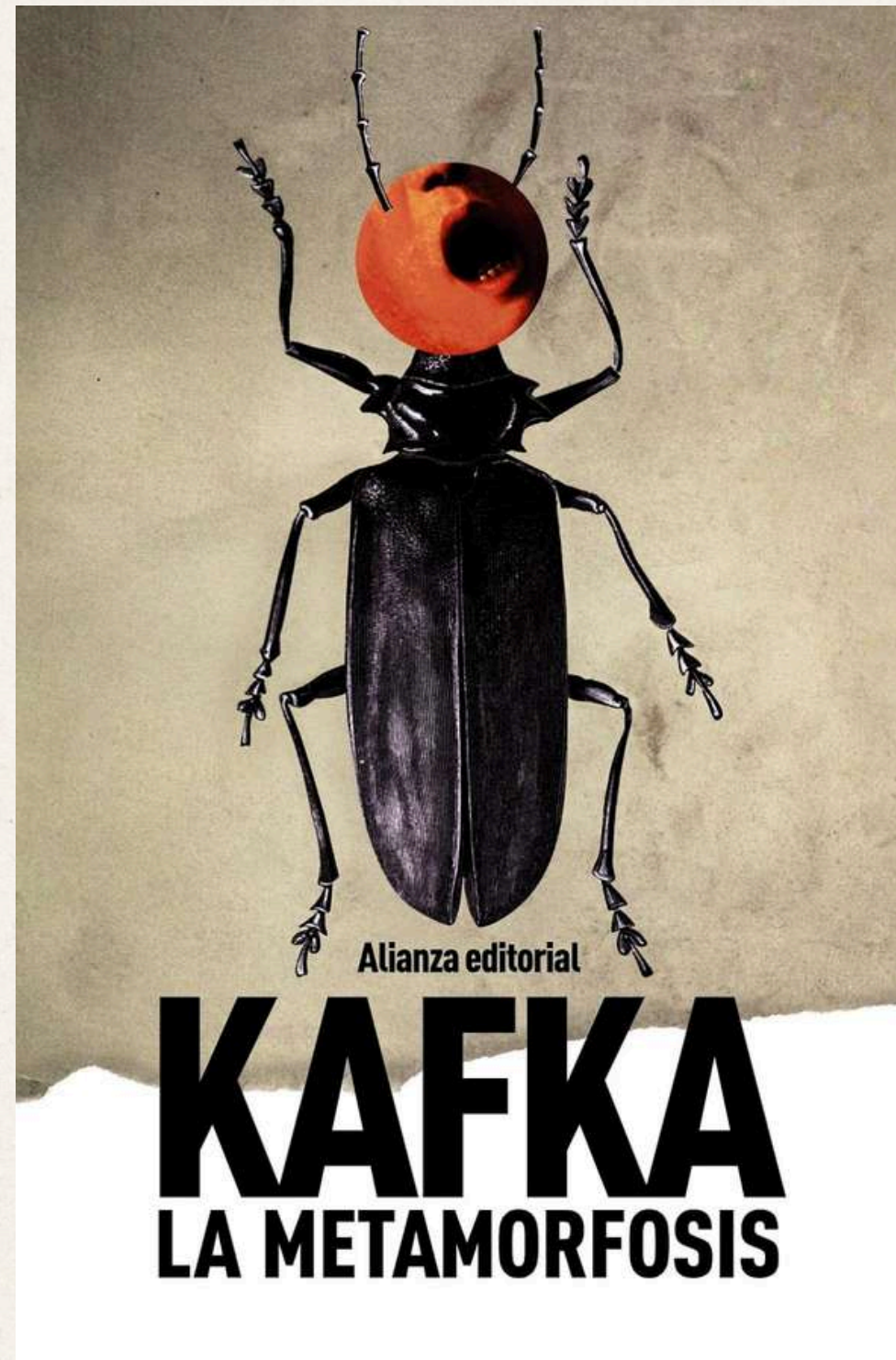
4

Evolving Multi-agent

Agents that develop and improve their own system prompts.

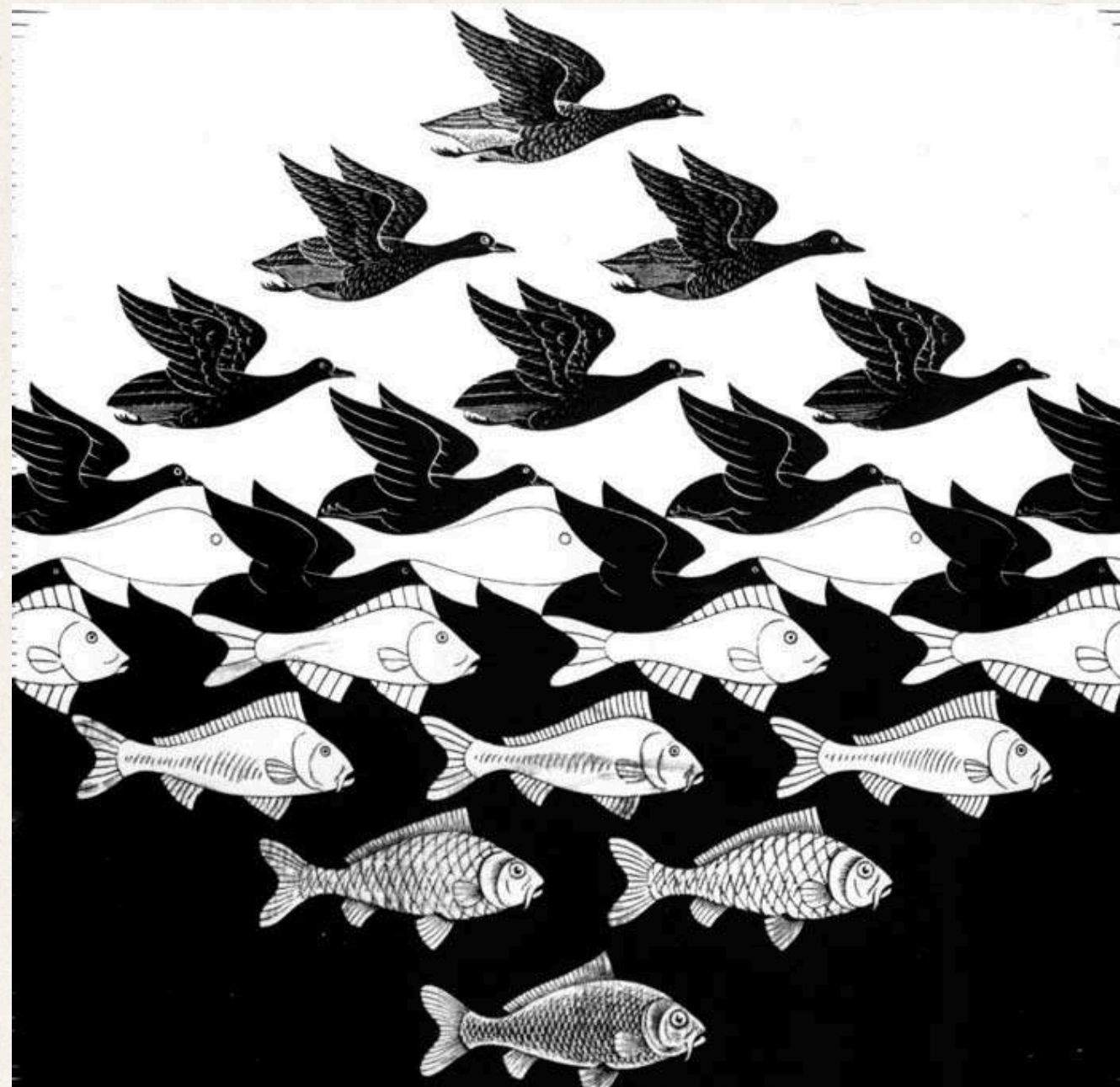


Transformation?



Illusions?

Maurits Cornelis Escher

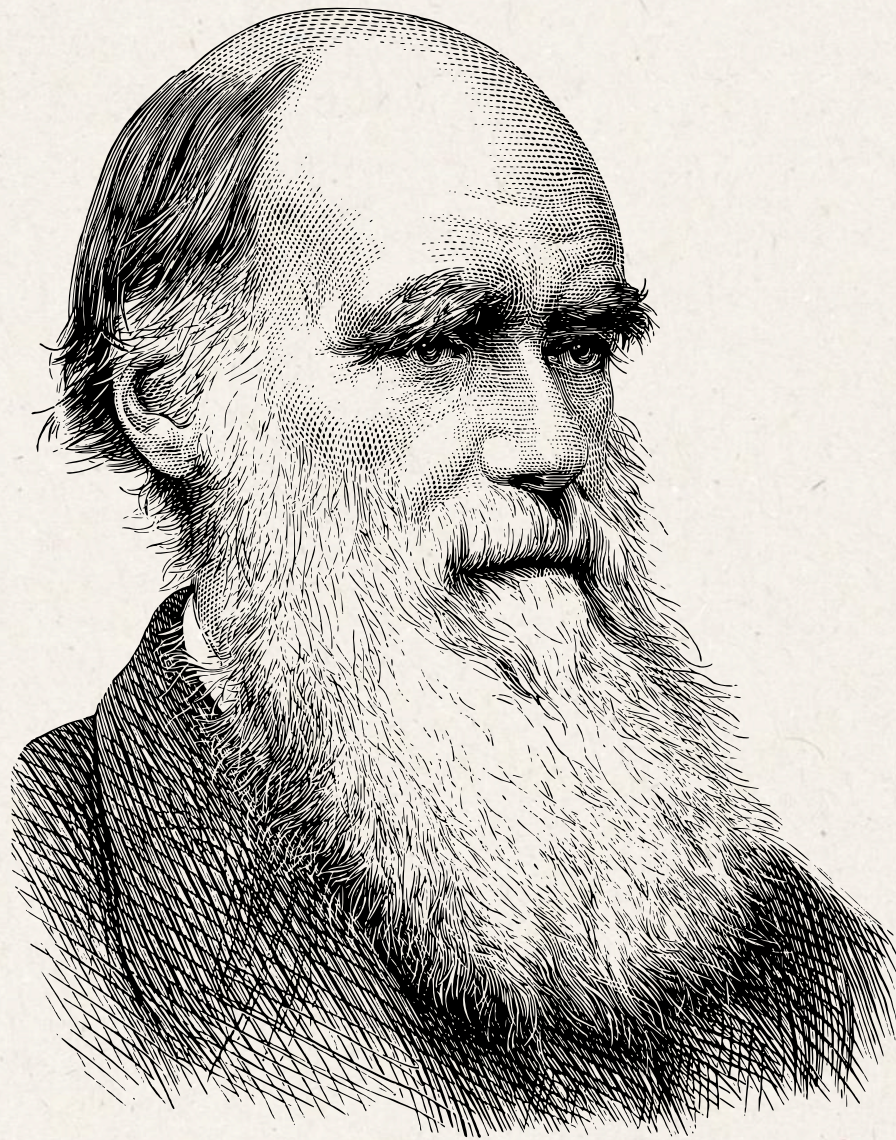


Sky and Water I (1938)



Butterflies (1950)

Evolution?




Charles Darwin





What do transformation (Kafka), illusion (Escher), and evolution (Darwin) have in common?



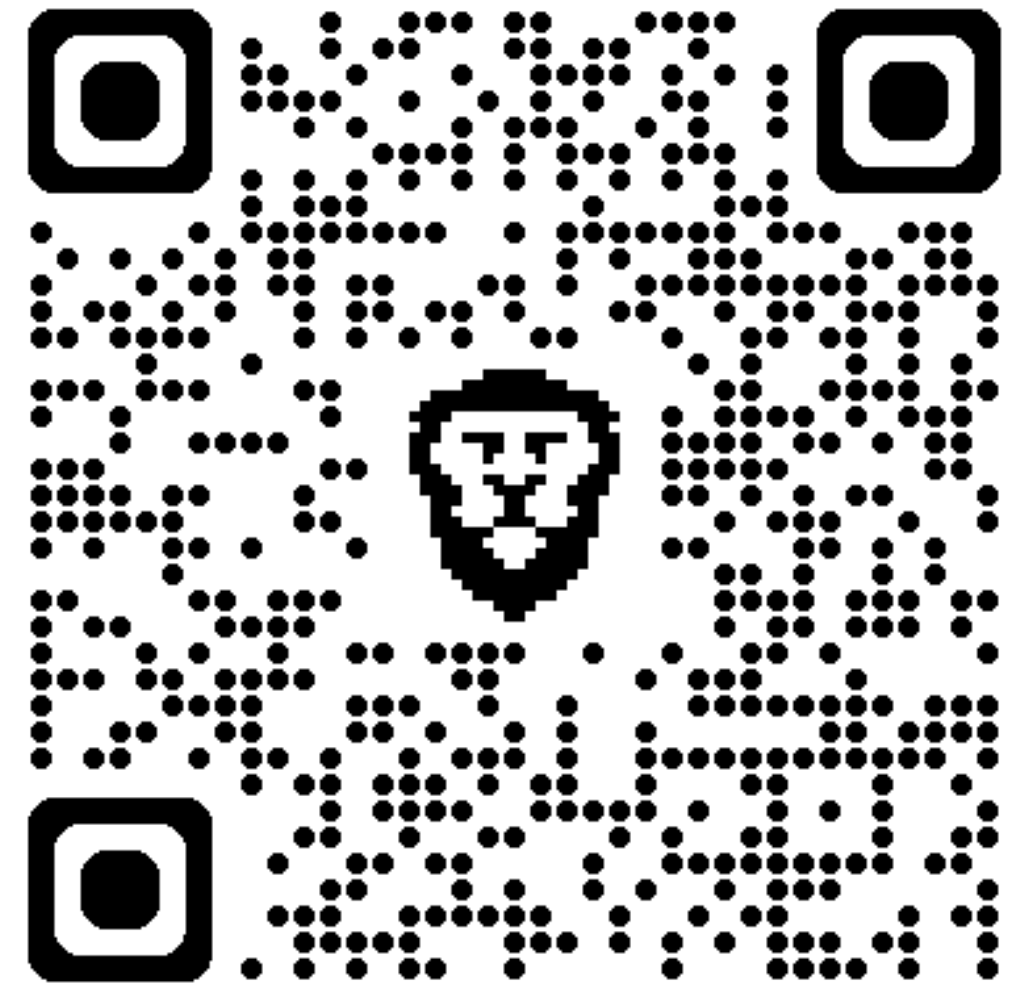
**What do transformation (Kafka), illusion
(Escher), and evolution (Darwin) have in
common?**

han e

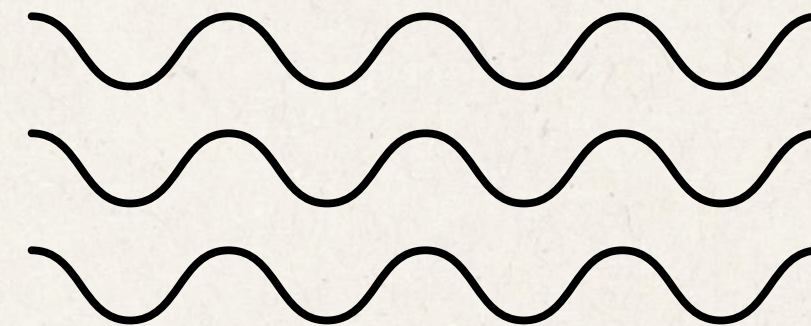
Evolutionary Computation

- **Evolutionary computation** arose in the mid-20th century by turning Darwin's idea of heritable variation under selection into a computational method.
- Early schools, genetic algorithms, evolutionary strategies, and evolutionary programming, evolved solutions through mutation, recombination, and competition.
- The field's power comes from letting populations adapt rather than requiring designers to specify solutions directly.

GitHub Repository



<https://github.com/camilochs/pydaybcn2025-workshop-code-evolution>

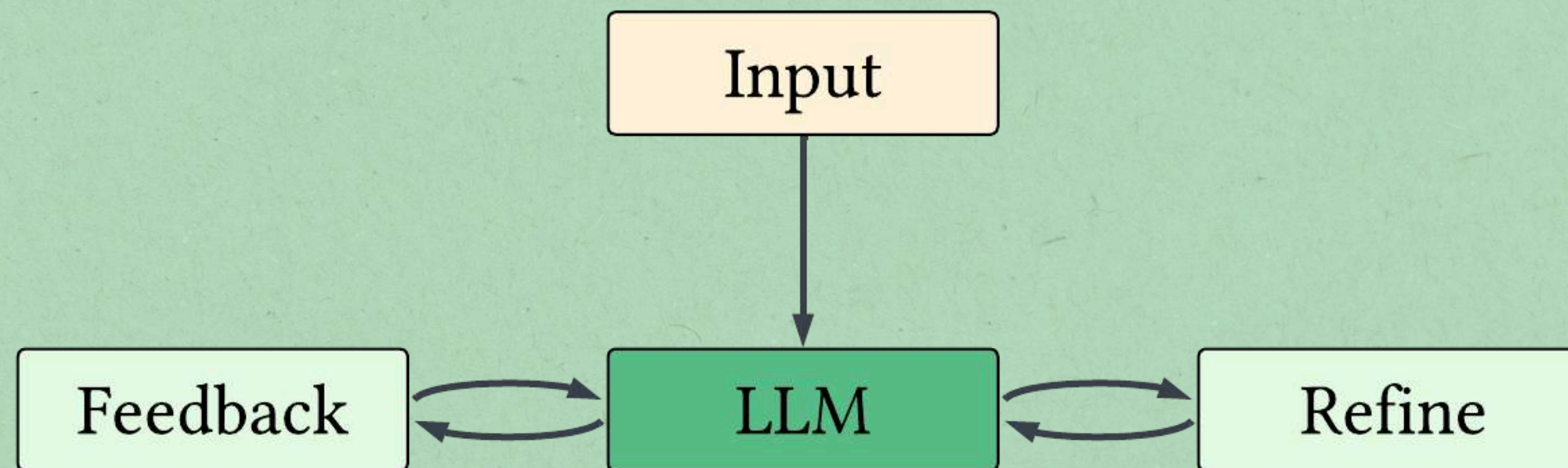
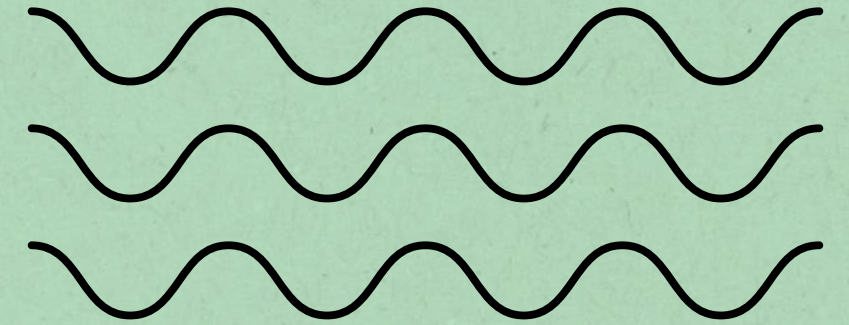


DEMO 01

Automating bug fixes.

The Self-Refine

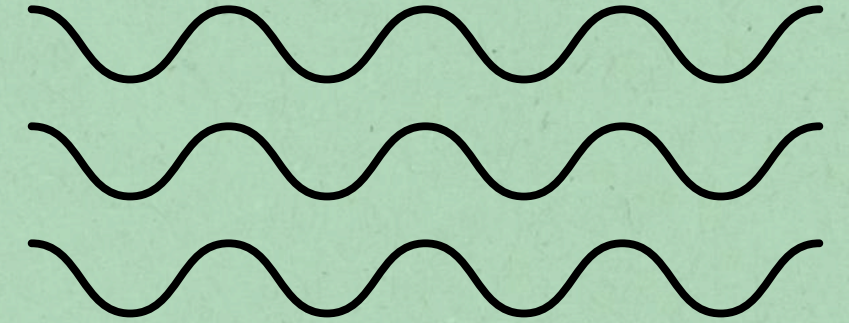
Principle #1



1. **Generate Code:** The LLM writes the initial script or function based on the user's Input.
2. **Debug/Review:** The system generates **Feedback** by either executing the code (catching interpreter errors) or statically analyzing it (finding logic bugs or security flaws).
3. **Refactor:** The LLM reads the error logs or analysis and uses the **Refine** stage to patch the code, adhering to syntax rules or optimization goals.
4. **Verify:** The cycle repeats until the code passes all unit tests or runs without errors

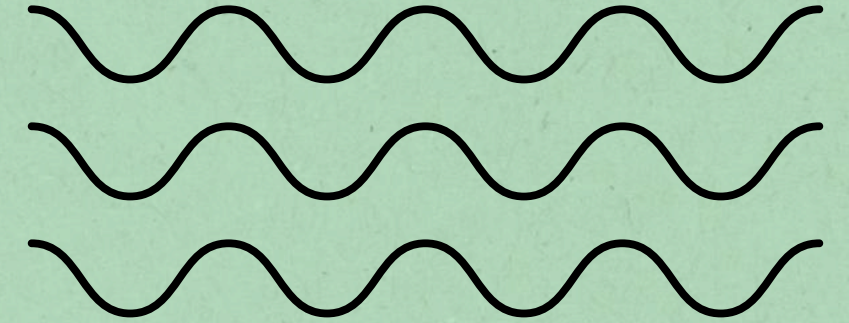
The Self-Refine

Principle #1



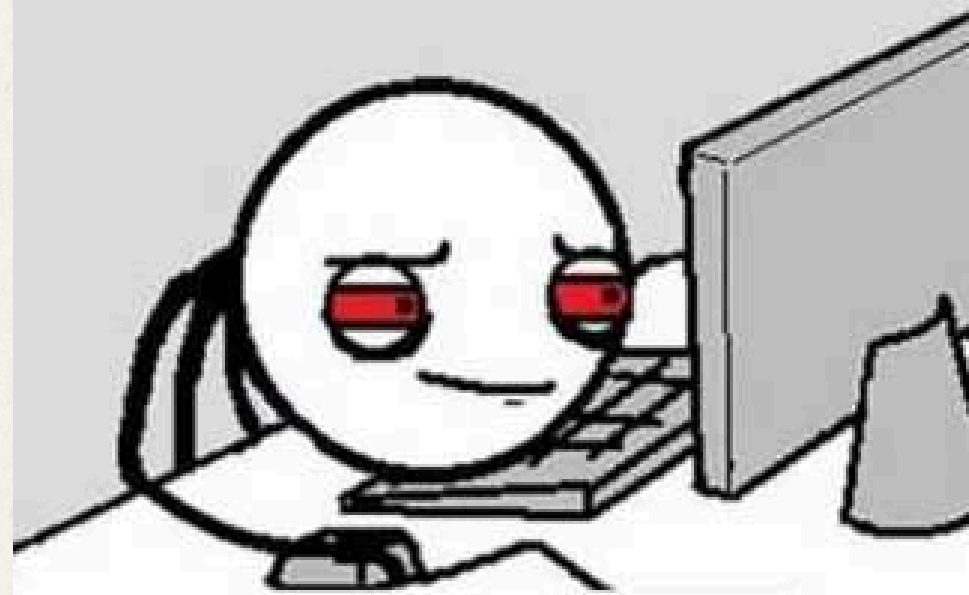
Example Notebook

Key Takeaways - What We Learned

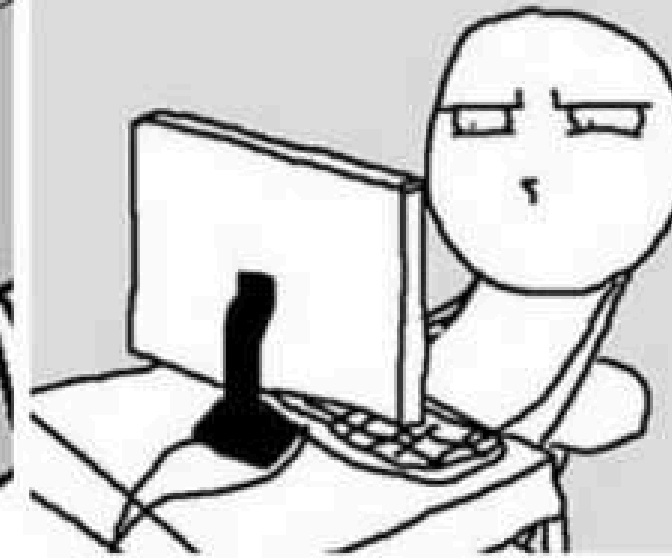


- **The Feedback Loop:** By capturing execution errors and feeding them back to an LLM, we create a simple but powerful self-correction mechanism.
- **stderr as Learning Signal:** The traceback isn't just for humans - it's rich information that tells the LLM exactly what went wrong and where.
- **Dynamic Code Replacement:** Using `exec()` we can replace functions at runtime with their corrected versions.

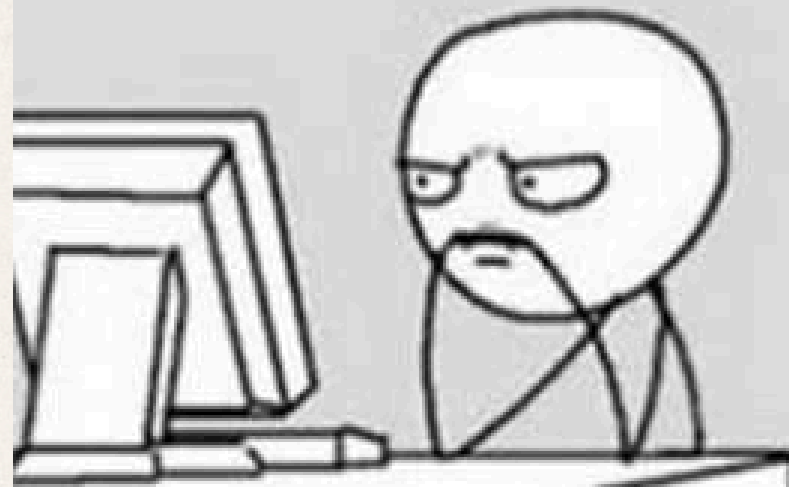
We find the bug



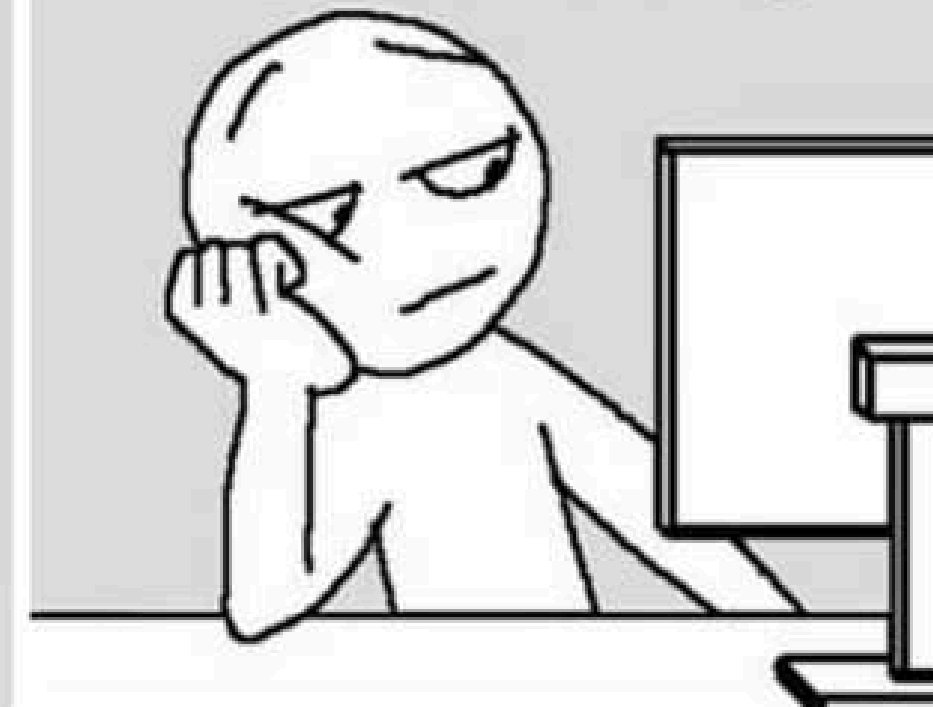
We fix the bug



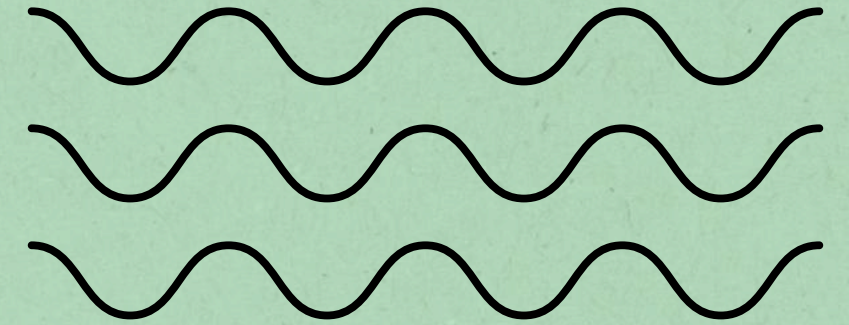
Now we have
Two Bugs



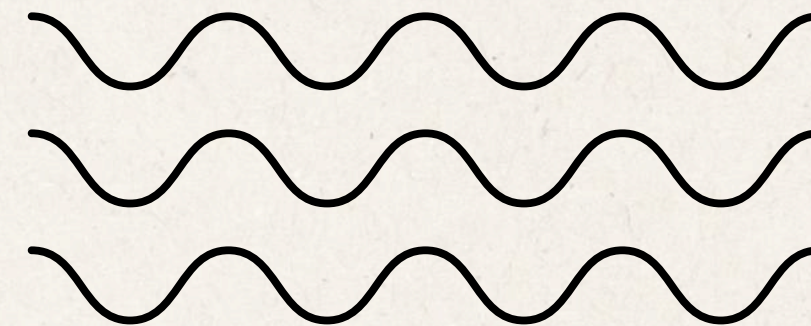
Now We have
Three Bugs



Limitations and Safety Considerations



- Never `exec()` untrusted code - always sandbox LLM-generated code.
- Validate fixes - run comprehensive tests before accepting changes.
- Human review - critical changes should be reviewed before deployment
- Rate limiting - prevent infinite loops of failed fixes.

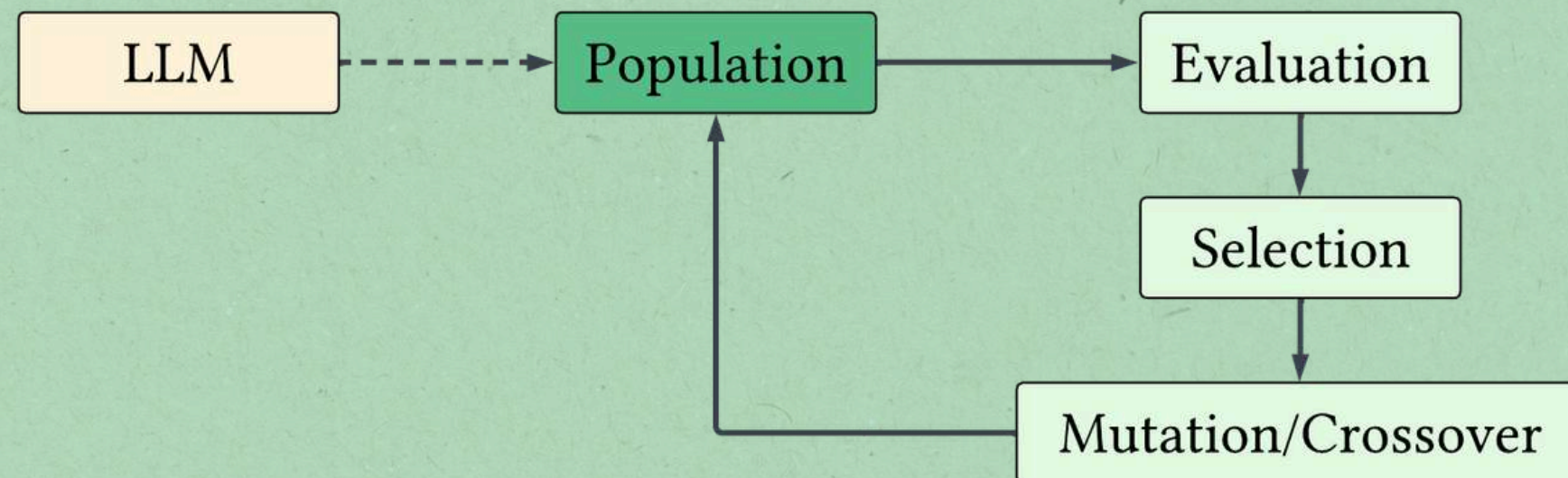
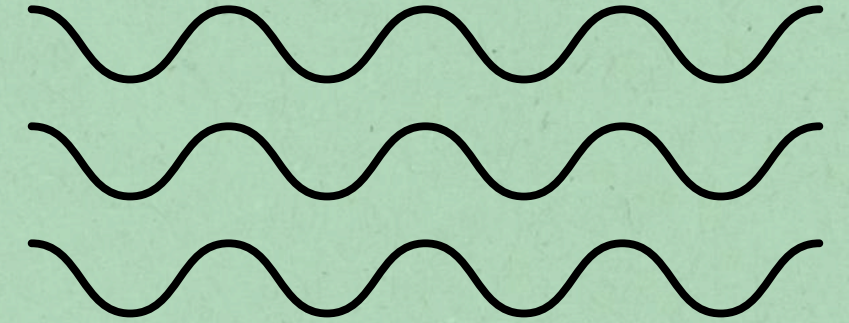


DEMO 02

Code Evolution.

Code Evolution / Genetic Approach

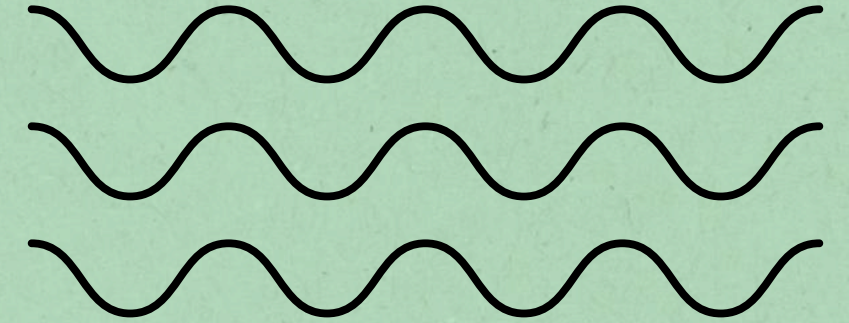
Principle #2



1. **LLM Initialization:** The LLM generates the initial code population (e.g., functional code snippets or algorithms), providing a high-quality starting point rather than pure random seeding.
2. **Evaluation:** Each code segment in the Population is tested (unit tests, performance benchmarks) and assigned a fitness score.
3. **Selection:** Code variants with the highest fitness scores are Selected to act as "parents" for the next generation.
4. **Evolutionary Operators:** The selected code is modified using genetic operators (Mutation/Crossover) to introduce variation and create new candidate solutions.
5. **Iteration:** The newly generated code re-enters the Population to repeat the Evaluation cycle until the optimal code is found.

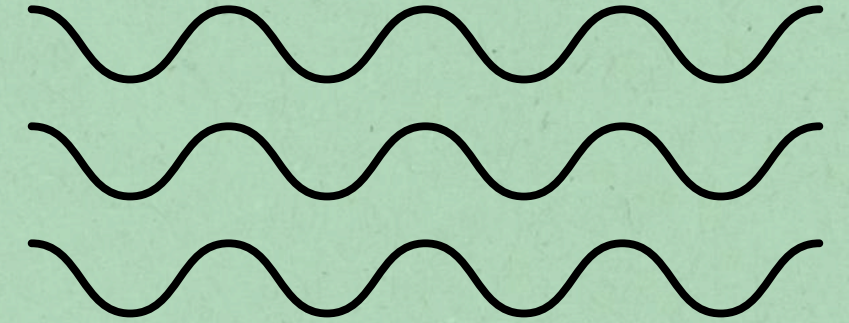
Code Evolution / Genetic Approach

Principle #2



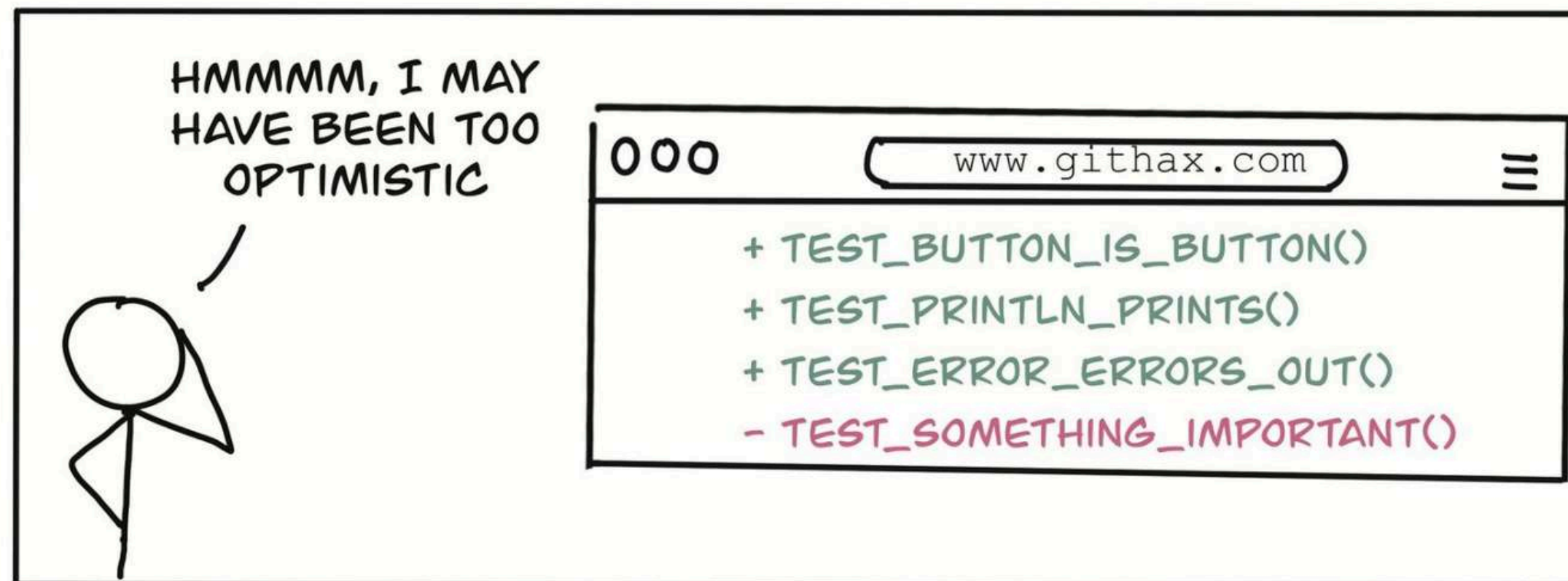
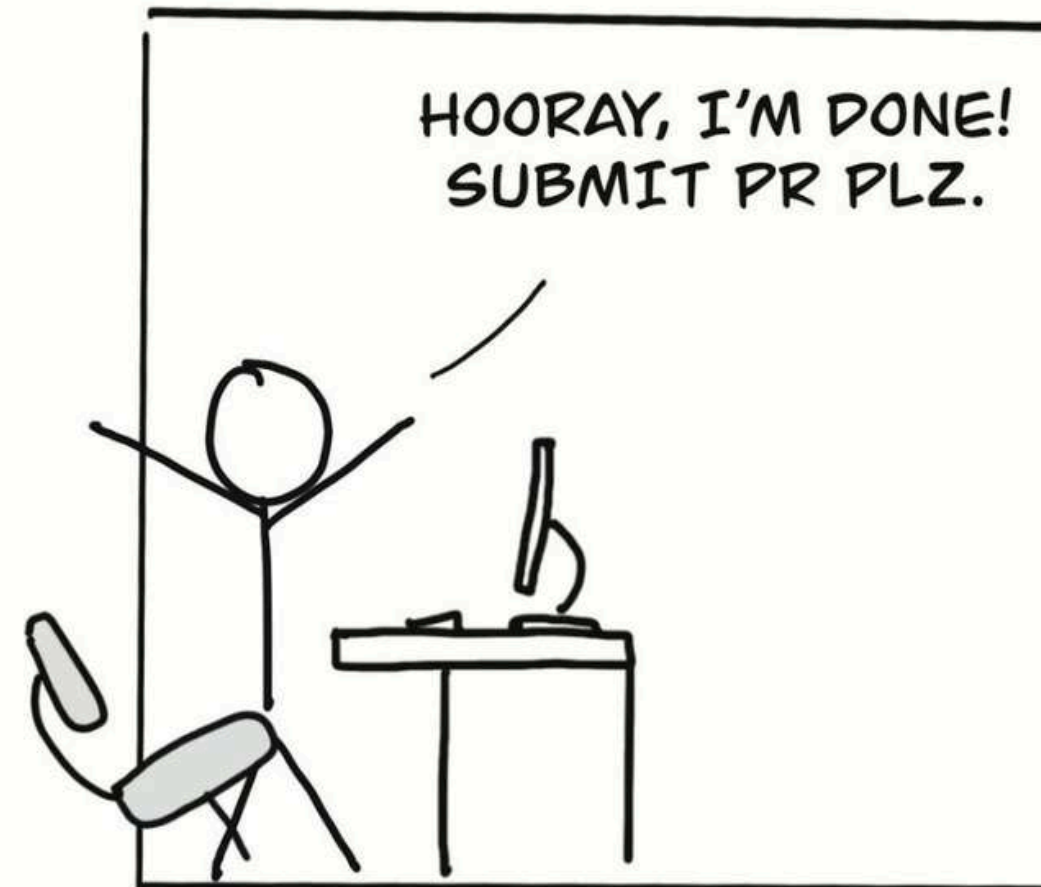
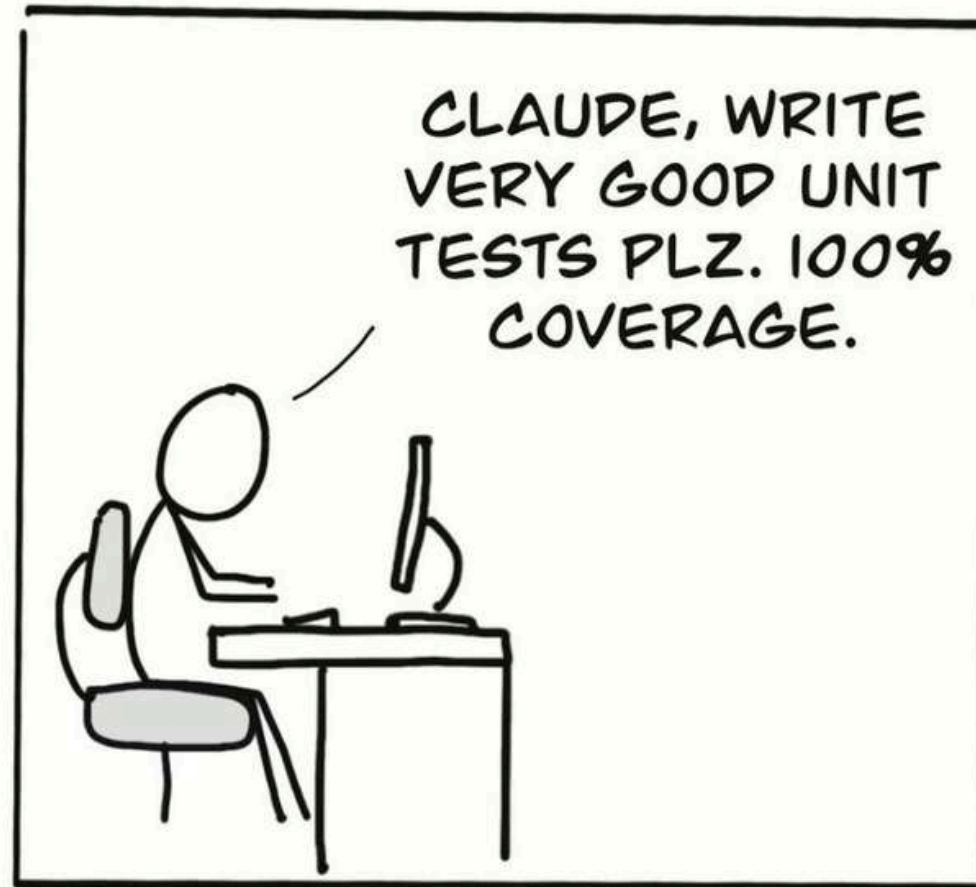
Example Notebook

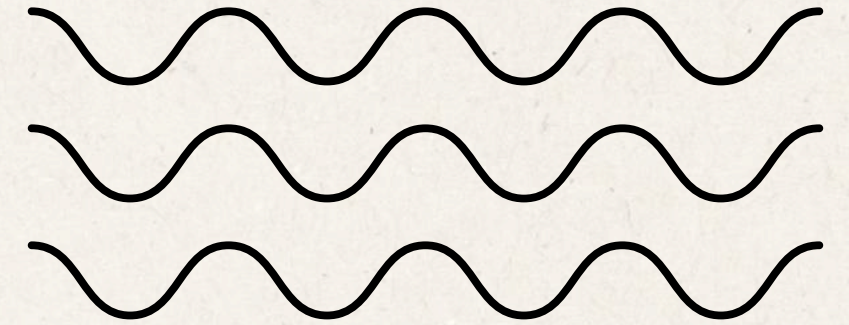
Key Takeaways - What We Learned



- **LLM as Intelligent Operator:** Unlike random mutations, LLMs understand semantics. They can combine the best aspects of two solutions meaningfully.
- **Fitness-Driven Selection:** The fitness function defines what "good" means. Evolution finds solutions that maximize it, even with complex, interacting constraints.
- **Code Evolution:** The same evolutionary principles that work for strings also work for actual code. The LLM understands program structure, not just characters.
- **Exploration vs Exploitation:** High temperature = more diversity (exploration). Lower temperature = more focused improvement (exploitation).

WRITING UNIT TESTS WITH AI



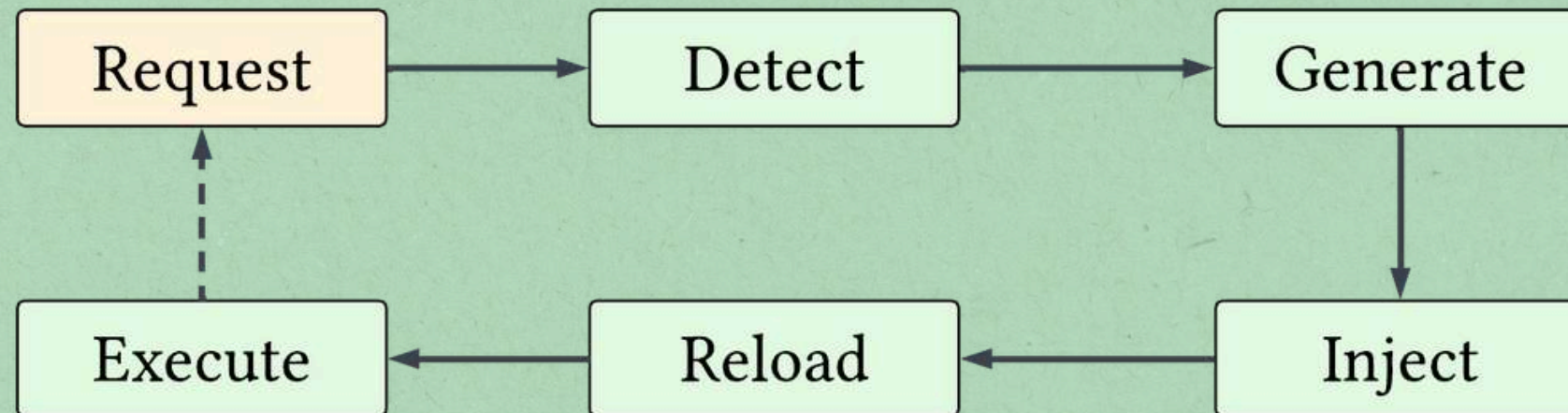
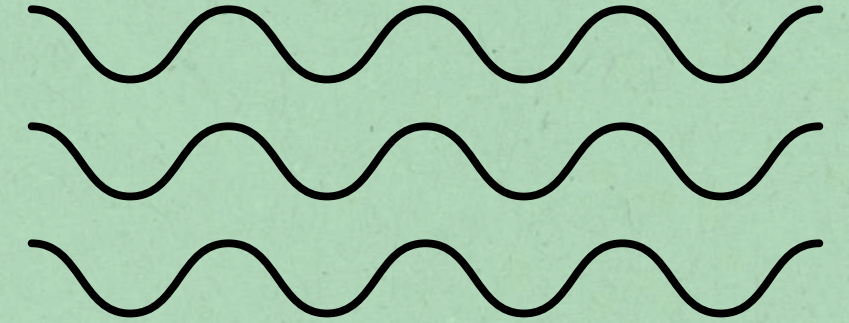


DEMO 03

Change the code being executed.

Hot Swapping with LLMs

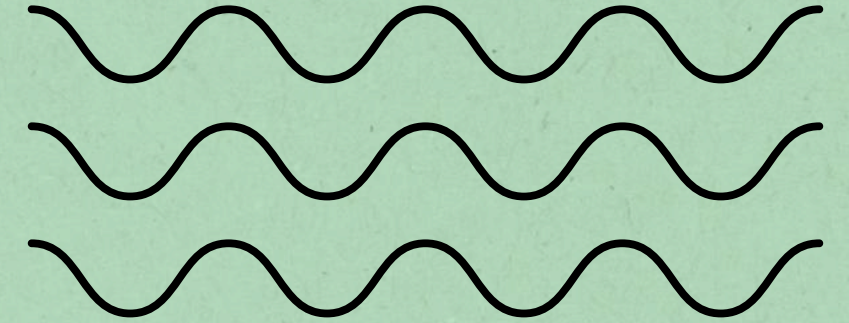
Principle #3



1. **Request:** System receives task requiring missing capability.
2. **Detect:** Identify the gap in current abilities.
3. **Generate:** LLM writes code as string from description.
4. **Persist:** Write code to source file on disk.
5. **Reload:** Refresh module in memory (`importlib.reload`).
6. **Execute:** New capability ready, complete original task.

Hot Swapping with LLMs

Principle #3



Example Notebook

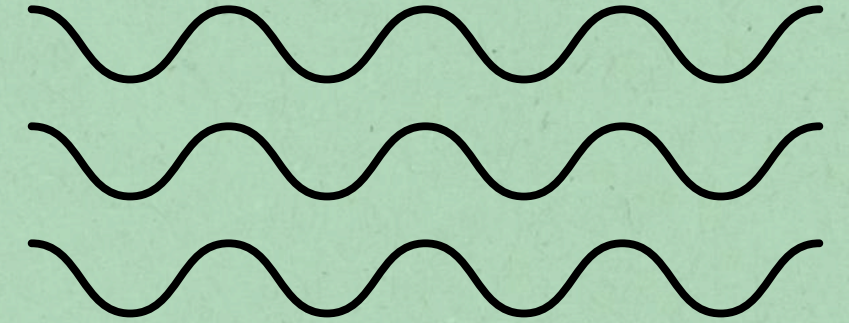
O'REILLY®

Deploy First, Pray Later

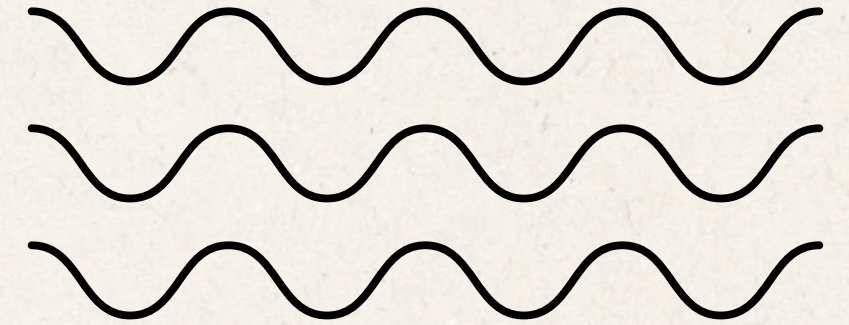
god abandoned this
pipeline long ago.



Safety Considerations



- **Sandbox generated code:** Run in isolated environments (containers, VMs).
- **Validate before execution:** Check for dangerous operations (file deletion, network access, etc.).
- **Human review:** Critical tools should be reviewed before deployment.
- **Audit trail:** Log all self-modifications for debugging and security.
- **Rollback capability:** Keep backups of previous tool versions.

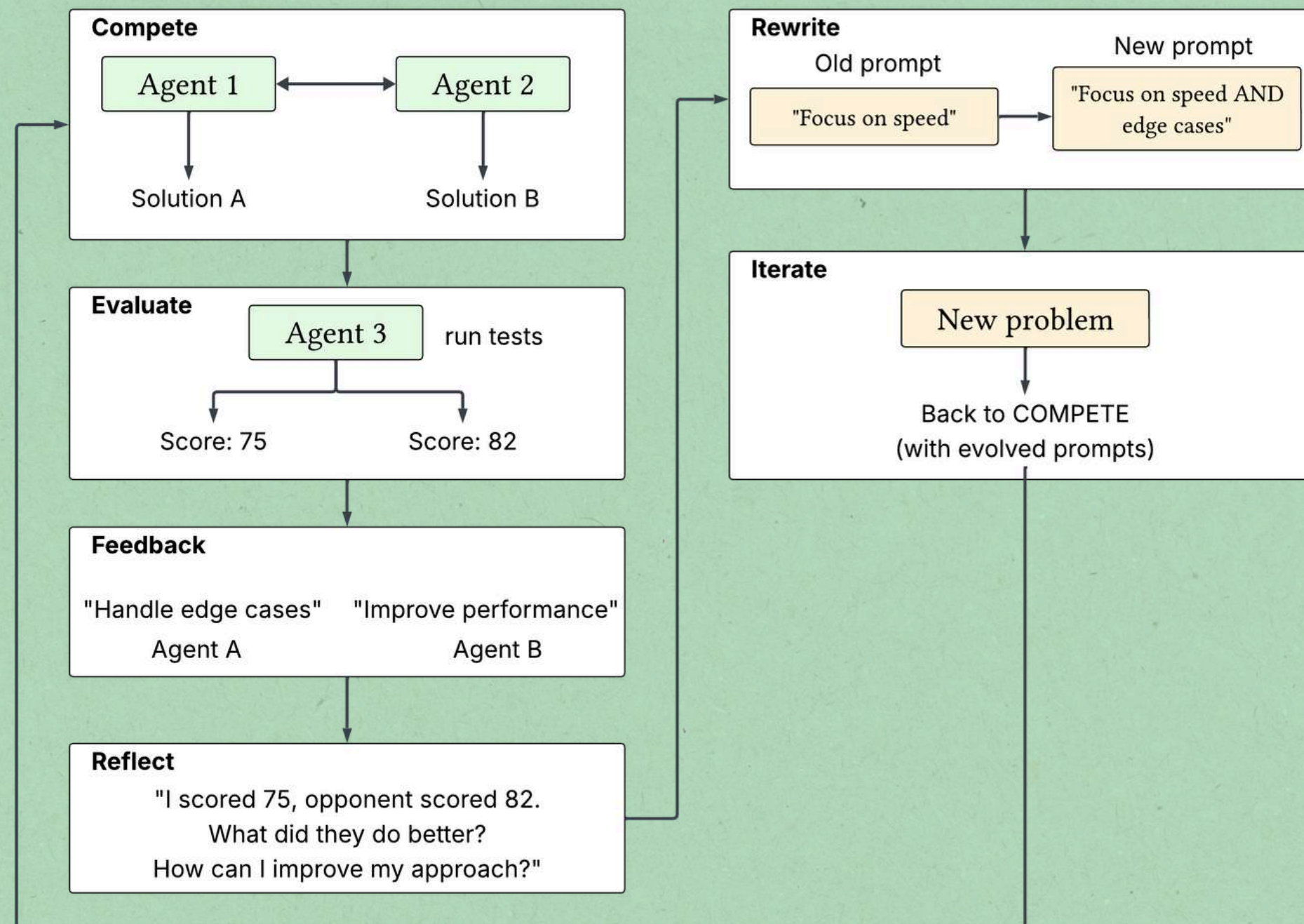
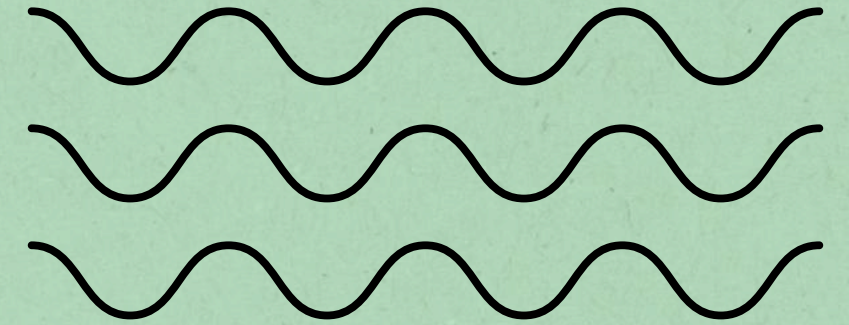


DEMO 04

Agents that develop and improve their own system prompts.

Self-Evolving Prompts

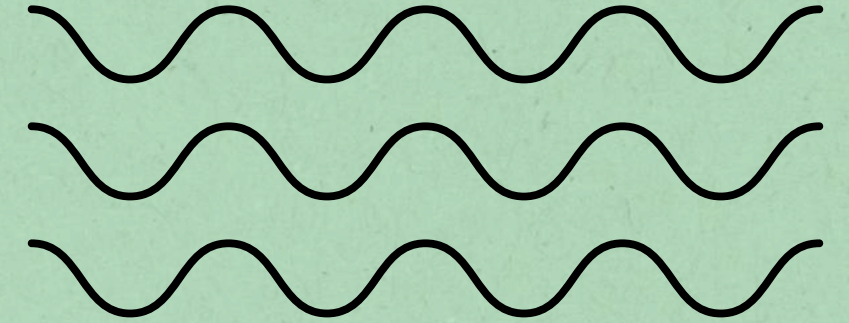
Principle #4



1. **Compete:** Agents solve the same problem with different strategies.
2. **Evaluate:** Tech Lead scores solutions with real tests.
3. **Feedback:** Specific, actionable critique for each agent.
4. **Reflect:** Agent analyzes its performance vs opponent.
5. **Rewrite:** Agent modifies its own system prompt.
6. **Iterate:** New round with evolved prompts.

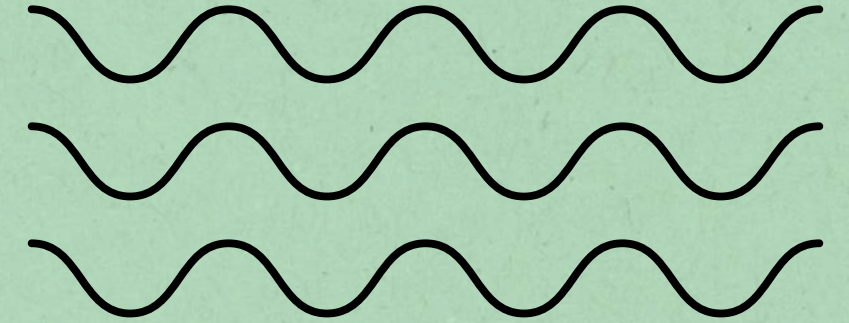
Self-Evolving Prompts

Principle #4



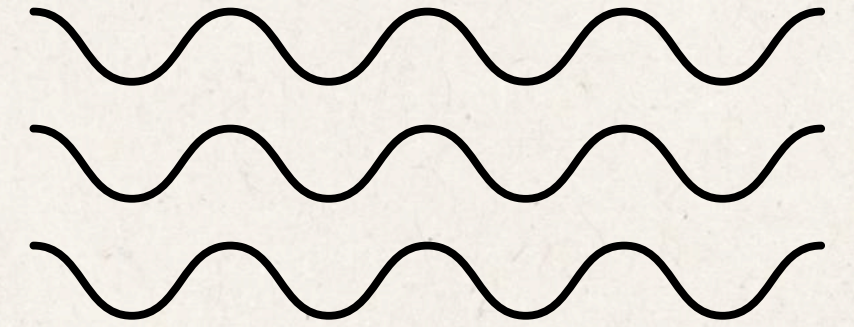
Example Notebook

Key Takeaways - What We Observed



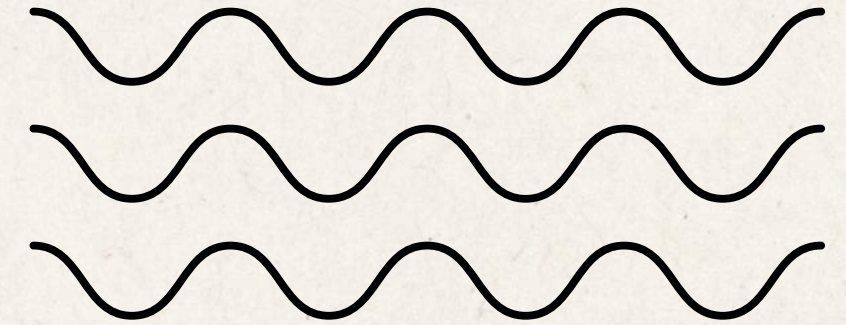
- **Self-Reflection Works:** Agents that analyze their mistakes can improve their own instructions.
- **Competition Drives Improvement:** The losing agent has strong incentive to learn from the winner's approach.
- **Emergent Strategies:** The evolved prompts often contain insights we didn't explicitly teach:
 - a. "Always check edge cases first"
 - b. "Balance elegance with correctness"
 - c. "Add type hints and docstrings"
- **Convergence:** Despite starting with opposite philosophies, agents often converge toward similar best practices.

Further Reading Papers



- Madaan et al. (2023). "Self-Refine: Iterative Refinement with Self-Feedback"
- Shinn et al. (2023). "Reflexion: Language Agents with Verbal Reinforcement Learning"
- Romera-Paredes et al. (2023). "Mathematical discoveries from program search with LLMs" (FunSearch)
- Ma et al. (2023). "Eureka: Human-Level Reward Design via Coding LLMs"
- Wang et al. (2023). "Voyager: An Open-Ended Embodied Agent with LLMs"
- Hong et al. (2023). "MetaGPT: Meta Programming for Multi-Agent Collaboration"
- Yang et al. (2023). "Large Language Models as Optimizers" (OPRO)
- Bai et al. (2022). "Constitutional AI: Harmlessness from AI Feedback"

Code Improving (Author's Research)



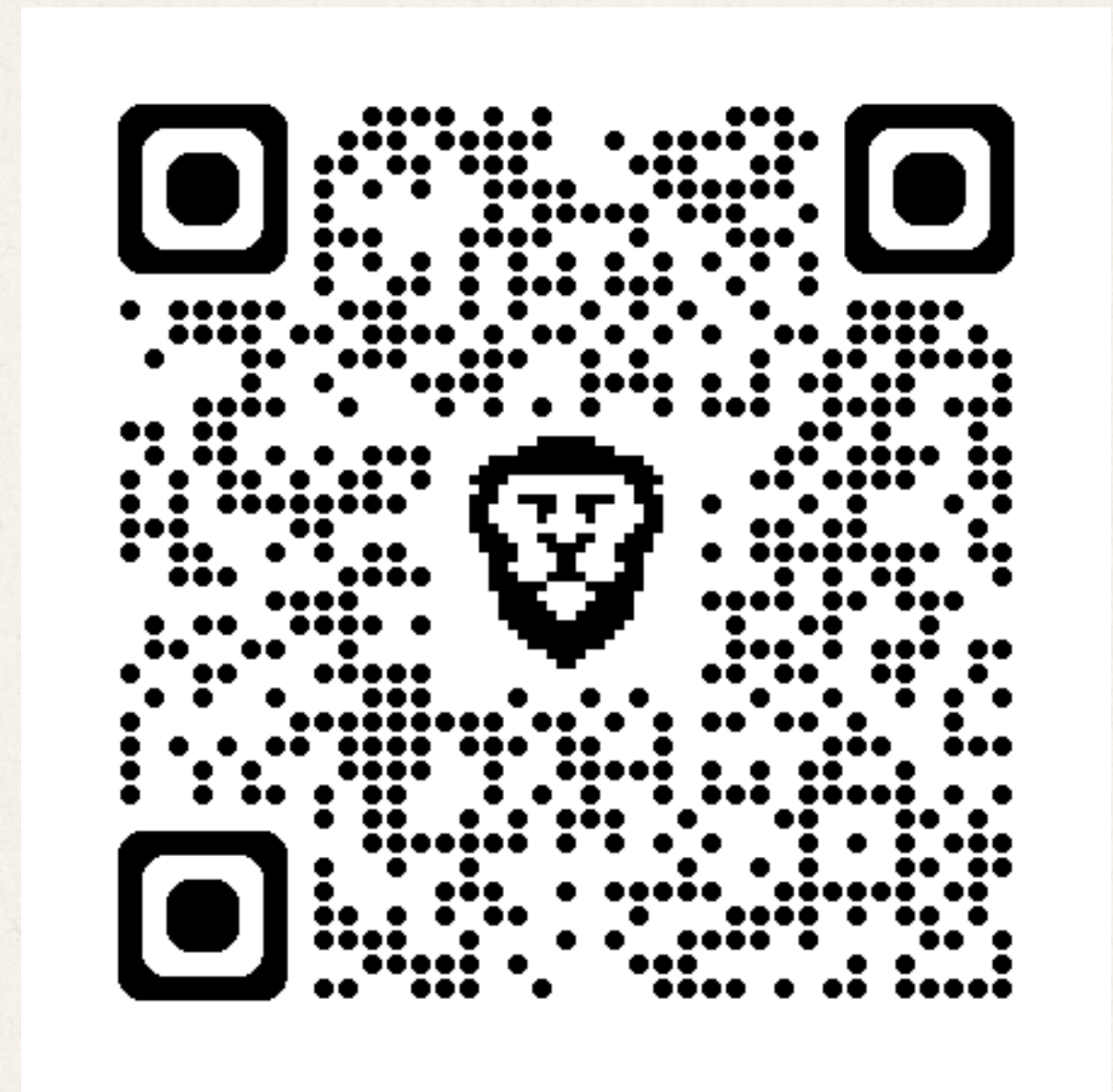
- Chacon Sartori & Blum (2025). "irace-evo: Automatic Algorithm Configuration Extended With LLM-Based Code Evolution"
- Chacon Sartori & Blum (2025). "Optimizing the Optimizer: An Example Showing the Power of LLM Code Generation" (FedCSIS '25)
- Chacon Sartori & Blum (2025). "Combinatorial Optimization for All: Using LLMs to Aid Non-Experts in Improving Optimization Algorithms"
- Chacon Sartori et al. (2024). "Metaheuristics and Large Language Models Join Forces: Toward an Integrated Optimization Approach" (IEEE Access)

Thank you

CONTACT US

E-mail

camilochs@gmail.com



www.camilochacon.com