**Chapter 9\_ Learning and Adaptation**

Chapter 9: Learning and Adaptation

Learning and adaptation are pivotal for enhancing the capabilities of artificial intelligence agents. These processes enable agents to evolve beyond predefined parameters, allowing them to improve autonomously through experience and environmental interaction. By learning and adapting, agents can effectively manage novel situations and optimize their performance without constant manual intervention. This chapter explores the principles and mechanisms underpinning agent learning and adaptation in detail.

**The big picture**

Agents learn and adapt by changing their thinking, actions, or knowledge based on new experiences and data. This allows agents to evolve from simply following instructions to becoming smarter over time.

* **Reinforcement Learning:** Agents try actions and receive rewards for positive outcomes and penalties for negative ones, learning optimal behaviors in changing situations. Useful for agents controlling robots or playing games.
* **Supervised Learning:** Agents learn from labeled examples, connecting inputs to desired outputs, enabling tasks like decision-making and pattern recognition. Ideal for agents sorting emails or predicting trends.
* **Unsupervised Learning:** Agents discover hidden connections and patterns in unlabeled data, aiding in insights, organization, and creating a mental map of their environment. Useful for agents exploring data without specific guidance.
* **Few-Shot/Zero-Shot Learning with LLM-Based Agents:** Agents leveraging LLMs can quickly adapt to new tasks with minimal examples or clear instructions, enabling rapid responses to new commands or situations.
* **Online Learning:** Agents continuously update knowledge with new data, essential for real-time reactions and ongoing adaptation in dynamic environments. Critical for agents processing continuous data streams.
* **Memory-Based Learning:** Agents recall past experiences to adjust current actions in similar situations, enhancing context awareness and decision-making. Effective for agents with memory recall capabilities.

Agents adapt by changing strategy, understanding, or goals based on learning. This is vital for agents in unpredictable, changing, or new environments.

**Proximal Policy Optimization (PPO)** is a reinforcement learning algorithm used to train agents in environments with a continuous range of actions, like controlling a robot's joints or a character in a game. Its main goal is to reliably and stably improve an agent's decision-making strategy, known as its policy.

The core idea behind PPO is to make small, careful updates to the agent's policy. It avoids drastic changes that could cause performance to collapse. Here's how it works:

1. Collect Data: The agent interacts with its environment (e.g., plays a game) using its current policy and collects a batch of experiences (state, action, reward).
2. Evaluate a "Surrogate" Goal: PPO calculates how a potential policy update would change the expected reward. However, instead of just maximizing this reward, it uses a special "clipped" objective function.
3. The "Clipping" Mechanism: This is the key to PPO's stability. It creates a "trust region" or a safe zone around the current policy. The algorithm is prevented from making an update that is too different from the current strategy. This clipping acts like a safety brake, ensuring the agent doesn't take a huge, risky step that undoes its learning.

In short, PPO balances improving performance with staying close to a known, working strategy, which prevents catastrophic failures during training and leads to more stable learning.

**Direct Preference Optimization (DPO)** is a more recent method designed specifically for aligning Large Language Models (LLMs) with human preferences. It offers a simpler, more direct alternative to using PPO for this task.

To understand DPO, it helps to first understand the traditional PPO-based alignment method:

* The PPO Approach (Two-Step Process):

1. Train a Reward Model: First, you collect human feedback data where people rate or compare different LLM responses (e.g., "Response A is better than Response B"). This data is used to train a separate AI model, called a reward model, whose job is to predict what score a human would give to any new response.
2. Fine-Tune with PPO: Next, the LLM is fine-tuned using PPO. The LLM's goal is to generate responses that get the highest possible score from the reward model. The reward model acts as the "judge" in the training game.

This two-step process can be complex and unstable. For instance, the LLM might find a loophole and learn to "hack" the reward model to get high scores for bad responses.

* The DPO Approach (Direct Process): DPO skips the reward model entirely. Instead of translating human preferences into a reward score and then optimizing for that score, DPO uses the preference data directly to update the LLM's policy.
* It works by using a mathematical relationship that directly links preference data to the optimal policy. It essentially teaches the model: "Increase the probability of generating responses like the *preferred* one and decrease the probability of generating ones like the *disfavored* one."

In essence, DPO simplifies alignment by directly optimizing the language model on human preference data. This avoids the complexity and potential instability of training and using a separate reward model, making the alignment process more efficient and robust.

**Practical Applications & Use Cases**

Adaptive agents exhibit enhanced performance in variable environments through iterative updates driven by experiential data.

* **Personalized assistant agents** refine interaction protocols through longitudinal analysis of individual user behaviors, ensuring highly optimized response generation.
* **Trading bot agents** optimize decision-making algorithms by dynamically adjusting model parameters based on high-resolution, real-time market data, thereby maximizing financial returns and mitigating risk factors.
* **Application agents** optimize user interface and functionality through dynamic modification based on observed user behavior, resulting in increased user engagement and system intuitiveness.
* **Robotic and autonomous vehicle agents** enhance navigation and response capabilities by integrating sensor data and historical action analysis, enabling safe and efficient operation across diverse environmental conditions.
* **Fraud detection agents** improve anomaly detection by refining predictive models with newly identified fraudulent patterns, enhancing system security and minimizing financial losses.
* **Recommendation agents** improve content selection precision by employing user preference learning algorithms, providing highly individualized and contextually relevant recommendations.
* **Game AI agents** enhance player engagement by dynamically adapting strategic algorithms, thereby increasing game complexity and challenge.
* **Knowledge Base Learning Agents**: Agents can leverage Retrieval Augmented Generation (RAG) to maintain a dynamic knowledge base of problem descriptions and proven solutions (see the Chapter 14). By storing successful strategies and challenges encountered, the agent can reference this data during decision-making, enabling it to adapt to new situations more effectively by applying previously successful patterns or avoiding known pitfalls.

**Case Study: The Self-Improving Coding Agent (SICA)**

The Self-Improving Coding Agent (SICA), developed by Maxime Robeyns, Laurence Aitchison, and Martin Szummer, represents an advancement in agent-based learning, demonstrating the capacity for an agent to modify its own source code. This contrasts with traditional approaches where one agent might train another; SICA acts as both the modifier and the modified entity, iteratively refining its code base to improve performance across various coding challenges.

SICA's self-improvement operates through an iterative cycle (see Fig.1). Initially, SICA reviews an archive of its past versions and their performance on benchmark tests. It selects the version with the highest performance score, calculated based on a weighted formula considering success, time, and computational cost. This selected version then undertakes the next round of self-modification. It analyzes the archive to identify potential improvements and then directly alters its codebase. The modified agent is subsequently tested against benchmarks, with the results recorded in the archive. This process repeats, facilitating learning directly from past performance. This self-improvement mechanism allows SICA to evolve its capabilities without requiring traditional training paradigms.

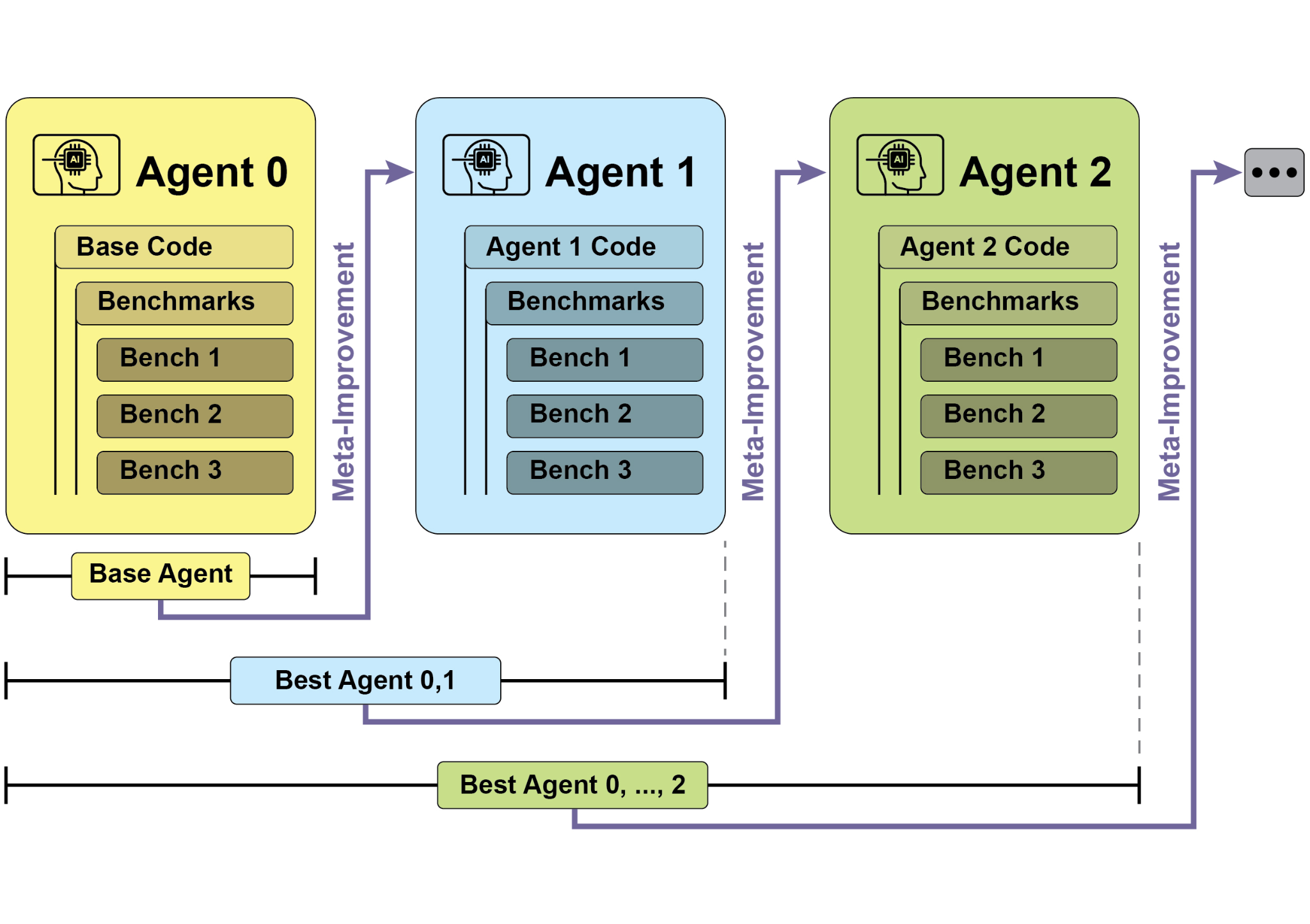


Fig.1: SICA's self-improvement, learning and adapting based on its past versions

SICA underwent significant self-improvement, leading to advancements in code editing and navigation. Initially, SICA utilized a basic file-overwriting approach for code changes. It subsequently developed a "Smart Editor" capable of more intelligent and contextual edits. This evolved into a "Diff-Enhanced Smart Editor," incorporating diffs for targeted modifications and pattern-based editing, and a "Quick Overwrite Tool" to reduce processing demands.

SICA further implemented "Minimal Diff Output Optimization" and "Context-Sensitive Diff Minimization," using Abstract Syntax Tree (AST) parsing for efficiency. Additionally, a "SmartEditor Input Normalizer" was added. In terms of navigation, SICA independently created an "AST Symbol Locator," using the code's structural map (AST) to identify definitions within the codebase. Later, a "Hybrid Symbol Locator" was developed, combining a quick search with AST checking. This was further optimized via "Optimized AST Parsing in Hybrid Symbol Locator" to focus on relevant code sections, improving search speed.(see Fig. 2)

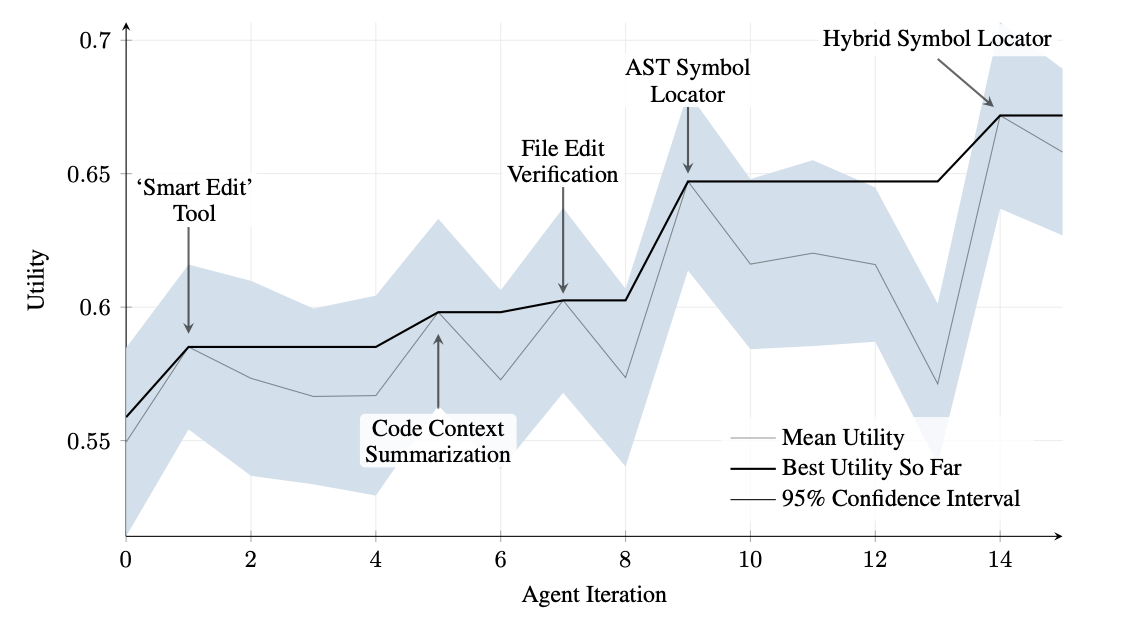


Fig.2 : Performance across iterations. Key improvements are annotated with their corresponding tool or agent modifications. (courtesy of Maxime Robeyns , Martin Szummer , Laurence Aitchison)

SICA's architecture comprises a foundational toolkit for basic file operations, command execution, and arithmetic calculations. It includes mechanisms for result submission and the invocation of specialized sub-agents (coding, problem-solving, and reasoning). These sub-agents decompose complex tasks and manage the LLM's context length, especially during extended improvement cycles.

An asynchronous overseer, another LLM, monitors SICA's behavior, identifying potential issues such as loops or stagnation. It communicates with SICA and can intervene to halt execution if necessary. The overseer receives a detailed report of SICA's actions, including a callgraph and a log of messages and tool actions, to identify patterns and inefficiencies.

SICA's LLM organizes information within its context window, its short-term memory, in a structured manner crucial to its operation. This structure includes a System Prompt defining agent goals, tool and sub-agent documentation, and system instructions. A Core Prompt contains the problem statement or instruction, content of open files, and a directory map. Assistant Messages record the agent's step-by-step reasoning, tool and sub-agent call records and results, and overseer communications. This organization facilitates efficient information flow, enhancing LLM operation and reducing processing time and costs. Initially, file changes were recorded as diffs, showing only modifications and periodically consolidated.

**SICA: A Look at the Code:** Delving deeper into SICA's implementation reveals several key design choices that underpin its capabilities. As discussed, the system is built with a modular architecture, incorporating several sub-agents, such as a coding agent, a problem-solver agent, and a reasoning agent. These sub-agents are invoked by the main agent, much like tool calls, serving to decompose complex tasks and efficiently manage context length, especially during those extended meta-improvement iterations.

The project is actively developed and aims to provide a robust framework for those interested in post-training LLMs on tool use and other agentic tasks, with the full code available for further exploration and contribution at the <https://github.com/MaximeRobeyns/self_improving_coding_agent/> GitHub repository.

For security, the project strongly emphasizes Docker containerization, meaning the agent runs within a dedicated Docker container. This is a crucial measure, as it provides isolation from the host machine, mitigating risks like inadvertent file system manipulation given the agent's ability to execute shell commands.

To ensure transparency and control, the system features robust observability through an interactive webpage that visualizes events on the event bus and the agent's callgraph. This offers comprehensive insights into the agent's actions, allowing users to inspect individual events, read overseer messages, and collapse sub-agent traces for clearer understanding.

In terms of its core intelligence, the agent framework supports LLM integration from various providers, enabling experimentation with different models to find the best fit for specific tasks. Finally, a critical component is the asynchronous overseer, an LLM that runs concurrently with the main agent. This overseer periodically assesses the agent's behavior for pathological deviations or stagnation and can intervene by sending notifications or even cancelling the agent's execution if necessary. It receives a detailed textual representation of the system's state, including a callgraph and an event stream of LLM messages, tool calls, and responses, which allows it to detect inefficient patterns or repeated work.

A notable challenge in the initial SICA implementation was prompting the LLM-based agent to independently propose novel, innovative, feasible, and engaging modifications during each meta-improvement iteration. This limitation, particularly in fostering open-ended learning and authentic creativity in LLM agents, remains a key area of investigation in current research.

**AlphaEvolve and OpenEvolve**

**AlphaEvolve** is an AI agent developed by Google designed to discover and optimize algorithms. It utilizes a combination of LLMs, specifically Gemini models (Flash and Pro), automated evaluation systems, and an evolutionary algorithm framework. This system aims to advance both theoretical mathematics and practical computing applications.

AlphaEvolve employs an ensemble of Gemini models. Flash is used for generating a wide range of initial algorithm proposals, while Pro provides more in-depth analysis and refinement. Proposed algorithms are then automatically evaluated and scored based on predefined criteria. This evaluation provides feedback that is used to iteratively improve the solutions, leading to optimized and novel algorithms.

In practical computing, AlphaEvolve has been deployed within Google's infrastructure. It has demonstrated improvements in data center scheduling, resulting in a 0.7% reduction in global compute resource usage. It has also contributed to hardware design by suggesting optimizations for Verilog code in upcoming Tensor Processing Units (TPUs). Furthermore, AlphaEvolve has accelerated AI performance, including a 23% speed improvement in a core kernel of the Gemini architecture and up to 32.5% optimization of low-level GPU instructions for FlashAttention.

In the realm of fundamental research, AlphaEvolve has contributed to the discovery of new algorithms for matrix multiplication, including a method for 4x4 complex-valued matrices that uses 48 scalar multiplications, surpassing previously known solutions. In broader mathematical research, it has rediscovered existing state-of-the-art solutions to over 50 open problems in 75% of cases and improved upon existing solutions in 20% of cases, with examples including advancements in the kissing number problem.

**OpenEvolve** is an evolutionary coding agent that leverages LLMs (see Fig.3) to iteratively optimize code. It orchestrates a pipeline of LLM-driven code generation, evaluation, and selection to continuously enhance programs for a wide range of tasks. A key aspect of OpenEvolve is its capability to evolve entire code files, rather than being limited to single functions. The agent is designed for versatility, offering support for multiple programming languages and compatibility with OpenAI-compatible APIs for any LLM. Furthermore, it incorporates multi-objective optimization, allows for flexible prompt engineering, and is capable of distributed evaluation to efficiently handle complex coding challenges.

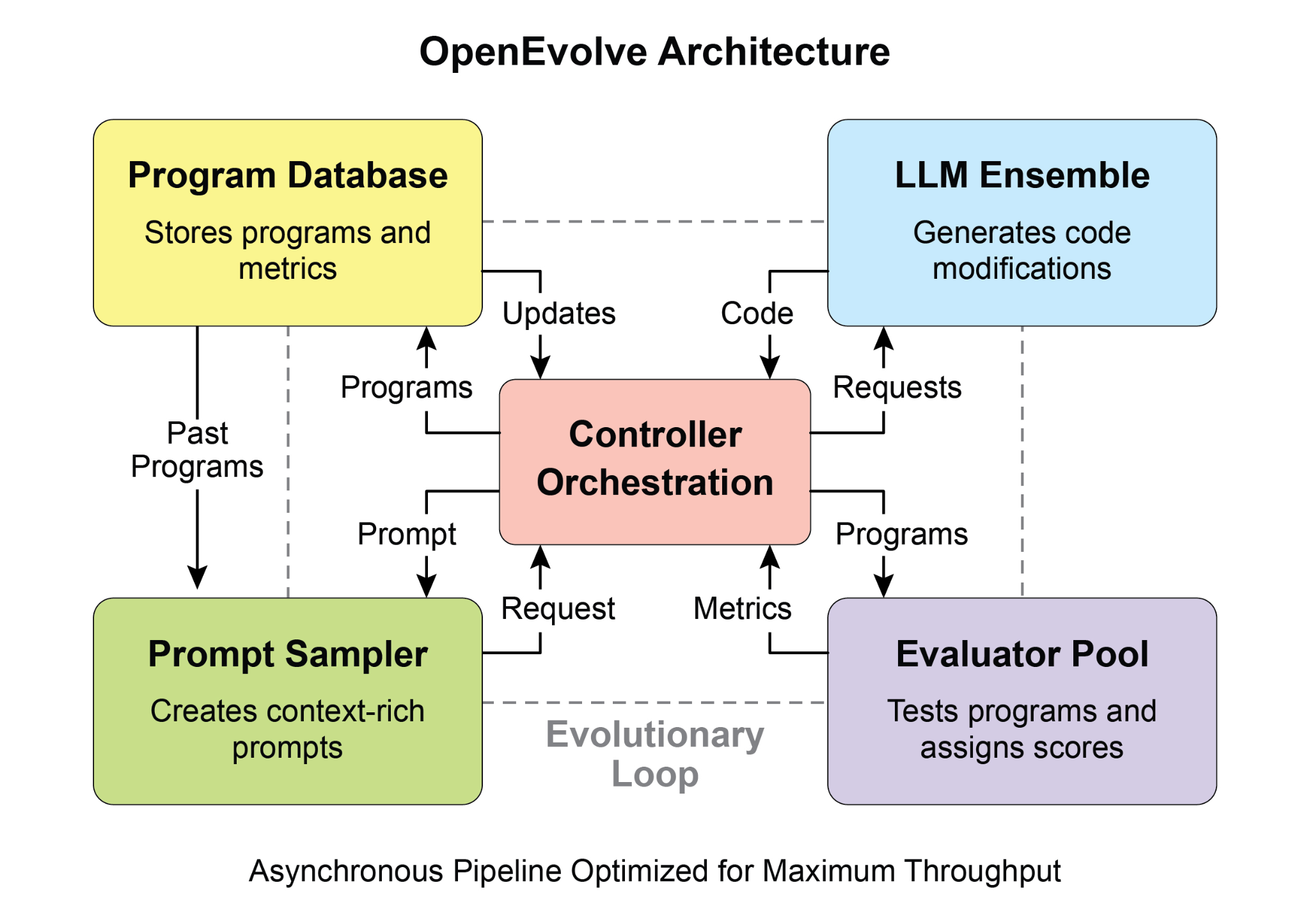


Fig. 3: The OpenEvolve internal architecture is managed by a controller. This controller orchestrates several key components: the program sampler, Program Database, Evaluator Pool, and LLM Ensembles. Its primary function is to facilitate their learning and adaptation processes to enhance code quality.

This code snippet uses the OpenEvolve library to perform evolutionary optimization on a program. It initializes the OpenEvolve system with paths to an initial program, an evaluation file, and a configuration file. The evolve.run(iterations=1000) line starts the evolutionary process, running for 1000 iterations to find an improved version of the program. Finally, it prints the metrics of the best program found during the evolution, formatted to four decimal places.

|  |
| --- |
| from openevolve import OpenEvolve  # Initialize the system  evolve = OpenEvolve(  initial\_program\_path="path/to/initial\_program.py",  evaluation\_file="path/to/evaluator.py",  config\_path="path/to/config.yaml"  )  # Run the evolution  best\_program = await evolve.run(iterations=1000)  print(f"Best program metrics:")  for name, value in best\_program.metrics.items():  print(f" {name}: {value:.4f}") |

**At a Glance**

**What:** AI agents often operate in dynamic and unpredictable environments where pre-programmed logic is insufficient. Their performance can degrade when faced with novel situations not anticipated during their initial design. Without the ability to learn from experience, agents cannot optimize their strategies or personalize their interactions over time. This rigidity limits their effectiveness and prevents them from achieving true autonomy in complex, real-world scenarios.

**Why:** The standardized solution is to integrate learning and adaptation mechanisms, transforming static agents into dynamic, evolving systems. This allows an agent to autonomously refine its knowledge and behaviors based on new data and interactions. Agentic systems can use various methods, from reinforcement learning to more advanced techniques like self-modification, as seen in the Self-Improving Coding Agent (SICA). Advanced systems like Google's AlphaEvolve leverage LLMs and evolutionary algorithms to discover entirely new and more efficient solutions to complex problems. By continuously learning, agents can master new tasks, enhance their performance, and adapt to changing conditions without requiring constant manual reprogramming.

**Rule of thumb:** Use this pattern when building agents that must operate in dynamic, uncertain, or evolving environments. It is essential for applications requiring personalization, continuous performance improvement, and the ability to handle novel situations autonomously.

**Visual summary**

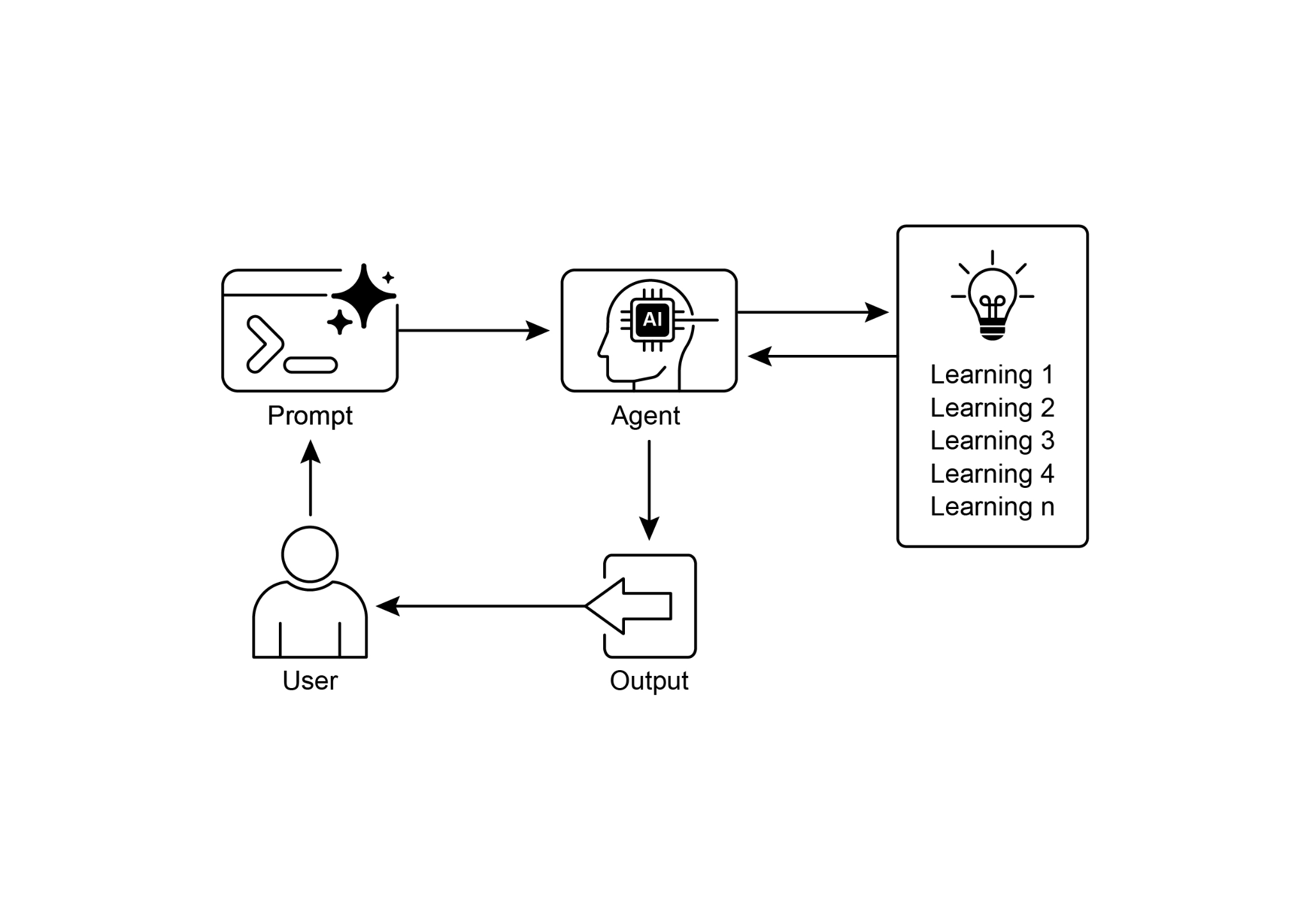


Fig.4: Learning and adapting pattern

**Key Takeaways**

* Learning and Adaptation are about agents getting better at what they do and handling new situations by using their experiences.
* "Adaptation" is the visible change in an agent's behavior or knowledge that comes from learning.
* SICA, the Self-Improving Coding Agent, self-improves by modifying its code based on past performance. This led to tools like the Smart Editor and AST Symbol Locator.
* Having specialized "sub-agents" and an "overseer" helps these self-improving systems manage big tasks and stay on track.
* The way an LLM's "context window" is set up (with system prompts, core prompts, and assistant messages) is super important for how efficiently agents work.
* This pattern is vital for agents that need to operate in environments that are always changing, uncertain, or require a personal touch.
* Building agents that learn often means hooking them up with machine learning tools and managing how data flows.
* An agent system, equipped with basic coding tools, can autonomously edit itself, and thereby improve its performance on benchmark tasks
* AlphaEvolve is Google's AI agent that leverages LLMs and an evolutionary framework to autonomously discover and optimize algorithms, significantly enhancing both fundamental research and practical computing applications..

**Conclusion**

This chapter examines the crucial roles of learning and adaptation in Artificial Intelligence. AI agents enhance their performance through continuous data acquisition and experience. The Self-Improving Coding Agent (SICA) exemplifies this by autonomously improving its capabilities through code modifications.

We have reviewed the fundamental components of agentic AI, including architecture, applications, planning, multi-agent collaboration, memory management, and learning and adaptation. Learning principles are particularly vital for coordinated improvement in multi-agent systems. To achieve this, tuning data must accurately reflect the complete interaction trajectory, capturing the individual inputs and outputs of each participating agent.

These elements contribute to significant advancements, such as Google's AlphaEvolve. This AI system independently discovers and refines algorithms by LLMs, automated assessment, and an evolutionary approach, driving progress in scientific research and computational techniques. Such patterns can be combined to construct sophisticated AI systems. Developments like AlphaEvolve demonstrate that autonomous algorithmic discovery and optimization by AI agents are attainable.

**References**

1. Sutton, R. S., & Barto, A. G. (2018). *Reinforcement Learning: An Introduction*. MIT Press.
2. Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press.
3. Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill.
4. Proximal Policy Optimization Algorithm**s** by John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. You can find it on arXiv:<https://arxiv.org/abs/1707.06347>
5. Robeyns, M., Aitchison, L., & Szummer, M. (2025). *A Self-Improving Coding Agent*. arXiv:2504.15228v2. <https://arxiv.org/pdf/2504.15228> <https://github.com/MaximeRobeyns/self_improving_coding_agent>
6. AlphaEvolve blog, <https://deepmind.google/discover/blog/alphaevolve-a-gemini-powered-coding-agent-for-designing-advanced-algorithms/>
7. OpenEvolve, <https://github.com/codelion/openevolve>

**第9章\_学习与适应**

第9章：学习与适应

学习和适应对于提升人工智能智能体的能力至关重要。这些过程使智能体能够超越预定义的参数进行进化，使其能够通过经验和与环境的交互自主改进。通过学习和适应，智能体可以有效地应对新情况，并在无需持续人工干预的情况下优化其性能。本章详细探讨了支撑智能体学习和适应的原理和机制。

**大局观**

智能体通过基于新的经验和数据改变其思维、行动或知识来学习和适应。这使得智能体能够从单纯地遵循指令逐渐发展得更加智能。

* **强化学习：**智能体尝试各种动作，因积极结果获得奖励，因消极结果受到惩罚，从而在不断变化的情境中学习最优行为。适用于控制机器人或玩游戏的智能体。
* **监督学习：**智能体从带标签的示例中学习，将输入与期望的输出关联起来，从而能够执行决策和模式识别等任务。适用于对电子邮件进行分类或预测趋势的智能体。
* **无监督学习：**智能体在无标签数据中发现隐藏的联系和模式，有助于洞察、组织并创建其环境的心理地图。适用于在没有特定指导的情况下探索数据的智能体。
* **基于大语言模型（LLM）的智能体的少样本/零样本学习：**利用大语言模型的智能体能够凭借少量示例或清晰指令迅速适应新任务，从而快速响应新命令或新情况。
* **在线学习：**智能体通过新数据不断更新知识，这对于实时反应和在动态环境中持续适应至关重要。对于处理连续数据流的智能体来说至关重要。
* **基于记忆的学习：**智能体通过回忆过去的经验来调整在相似情境下的当前行动，从而增强情境感知和决策能力。适用于具备记忆回忆能力的智能体。

智能体通过基于学习改变策略、理解或目标来进行适应。这对于处于不可预测、不断变化或新环境中的智能体来说至关重要。

**近端策略优化（PPO）**是一种强化学习算法，用于在具有连续动作范围的环境中训练智能体，例如控制机器人的关节或游戏中的角色。其主要目标是可靠且稳定地改进智能体的决策策略，即其策略。

PPO背后的核心思想是对智能体的策略进行小幅度、谨慎的更新。它避免了可能导致性能崩溃的剧烈变化。以下是它的工作原理：

1. 收集数据：智能体使用其当前策略与环境进行交互（例如，玩游戏），并收集一批经验（状态、动作、奖励）。
2. 评估“替代”目标：PPO会计算潜在的策略更新将如何改变预期奖励。不过，它并非单纯地最大化这一奖励，而是使用一种特殊的“裁剪”目标函数。
3. “裁剪”机制：这是PPO稳定性的关键。它在当前策略周围创建了一个“信任区域”或安全区。该算法被阻止进行与当前策略差异过大的更新。这种裁剪就像一个安全刹车，确保智能体不会迈出巨大且有风险的一步，从而破坏其学习成果。

简而言之，PPO在提升性能的同时，保持与已知的有效策略相近，这能防止训练过程中出现灾难性失败，并使学习更加稳定。

**直接偏好优化（DPO）**是一种较新的方法，专门用于使大语言模型（LLM）与人类偏好保持一致。它为使用PPO完成这项任务提供了一种更简单、更直接的替代方案。

要理解DPO，首先了解传统的基于PPO的对齐方法会有所帮助：

* PPO方法（两步流程）：

1. 训练奖励模型：首先，你收集人类反馈数据，其中人们对不同的大语言模型（LLM）回复进行评分或比较（例如，“回复A比回复B更好”）。这些数据用于训练一个单独的AI模型，称为奖励模型，其任务是预测人类会给任何新回复打多少分。
2. 使用PPO进行微调：接下来，使用PPO对大语言模型（LLM）进行微调。LLM的目标是生成能从奖励模型中获得尽可能高分数的回复。奖励模型在训练过程中充当“裁判”的角色。

这个两步过程可能复杂且不稳定。例如，大语言模型可能会找到漏洞并学会“破解”奖励模型，从而使不良回复获得高分。

* DPO方法（直接过程）：DPO完全跳过了奖励模型。DPO不是将人类偏好转化为奖励分数，然后针对该分数进行优化，而是直接使用偏好数据来更新大语言模型（LLM）的策略。
* 它的工作原理是利用一种数学关系，将偏好数据直接与最优策略联系起来。它本质上是在教导模型：“增加生成像*偏好*的那样的回复的概率，降低生成像*不偏好*的那样的回复的概率。”

本质上，DPO通过直接在人类偏好数据上优化语言模型来简化对齐过程。这避免了训练和使用单独奖励模型的复杂性和潜在不稳定性，使对齐过程更加高效和稳健。

**实际应用与用例**

自适应智能体通过由经验数据驱动的迭代更新，在多变环境中展现出更高的性能。

* **个性化助理智能体**通过对个体用户行为的纵向分析来完善交互协议，确保生成高度优化的响应。
* **交易机器人代理**通过基于高分辨率的实时市场数据动态调整模型参数来优化决策算法，从而实现财务回报最大化并降低风险因素。
* **应用代理**根据观察到的用户行为，通过动态修改来优化用户交互界面和功能，从而提高用户参与度和系统的直观性。
* **机器人和自动驾驶车辆智能体**通过整合传感器数据和历史行动分析来增强导航和响应能力，从而能够在各种不同的环境条件下安全高效地运行。
* **欺诈检测代理**通过用新识别的欺诈模式完善预测模型来改进异常检测，增强系统安全性并将财务损失降至最低。
* **推荐代理**通过采用用户偏好学习算法来提高内容选择的精准度，提供高度个性化且与上下文相关的推荐。
* **游戏AI智能体**通过动态调整战略算法来增强玩家的参与度，从而增加游戏的复杂性和挑战性。
* **知识库学习智能体**：智能体可以利用检索增强生成（RAG）来维护一个包含问题描述和已验证解决方案的动态知识库（见第14章）。通过存储成功的策略和遇到的挑战，智能体可以在决策过程中参考这些数据，从而通过应用先前成功的模式或避免已知的陷阱，更有效地适应新情况。

**案例研究：自我提升编码智能体（SICA）**

由马克西姆·罗宾斯（Maxime Robeyns）、劳伦斯·艾奇逊（Laurence Aitchison）和马丁·祖默（Martin Szummer）开发的自我改进编码智能体（Self-Improving Coding Agent，SICA）代表了基于智能体学习的一项进步，它展示了智能体修改自身源代码的能力。这与传统方法形成对比，传统方法中一个智能体可能训练另一个智能体；SICA既是修改者又是被修改的实体，通过迭代完善其代码库来提高在各种编码挑战中的性能。

SICA的自我改进通过一个迭代循环来实现（见图1）。最初，SICA会回顾其过往版本的存档以及它们在基准测试中的表现。它会选择根据考虑成功率、时间和计算成本的加权公式计算出的性能得分最高的版本。这个被选中的版本随后会进行下一轮的自我修改。它会分析存档以确定潜在的改进之处，然后直接修改其代码库。经过修改的代理随后会针对基准进行测试，测试结果会记录在存档中。这个过程不断重复，从而实现直接从过往表现中学习。这种自我改进机制使SICA能够在无需传统训练范式的情况下提升其能力。

图1：SICA基于其过往版本的自我完善、学习和适应

SICA经历了重大的自我改进，推动了代码编辑和导航的进步。最初，SICA采用基本的文件覆盖方法进行代码更改。随后，它开发了一个能够进行更智能和上下文感知编辑的“智能编辑器”。这进一步演变为“差异增强智能编辑器”，结合差异分析进行有针对性的修改和基于模式的编辑，以及一个“快速覆盖工具”以减少处理需求。

SICA进一步实施了“最小差异输出优化”和“上下文敏感差异最小化”，使用抽象语法树（AST）解析以提高效率。此外，还添加了“智能编辑器输入规范化器”。在导航方面，SICA自主创建了“AST符号定位器”，利用代码的结构映射（AST）来识别代码库中的定义。随后，开发了“混合符号定位器”，将快速搜索与AST检查相结合。通过“混合符号定位器中的优化AST解析”进一步优化，专注于相关代码部分，提高搜索速度（见图2）。

图2：各迭代过程中的性能表现。关键改进处标注了相应的工具或智能体修改内容。（图片来源：马克西姆·罗宾斯、马丁·苏默、劳伦斯·艾奇逊）

SICA的架构包含一个基础工具包，用于基本文件操作、命令执行和算术计算。它包括结果提交机制和专门子代理（编码、问题解决和推理）的调用机制。这些子代理分解复杂任务并管理大语言模型（LLM）的上下文长度，特别是在长时间的改进周期中。

一个异步监督器，即另一个大语言模型（LLM），会监控SICA的行为，识别诸如循环或停滞等潜在问题。它与SICA进行通信，必要时可以进行干预以停止执行。监督器会收到SICA行动的详细报告，包括调用图以及消息和工具操作的日志，以识别模式和低效之处。

SICA的大语言模型（LLM）在其上下文窗口（即短期记忆）内以一种对其运行至关重要的结构化方式组织信息。这种结构包括定义代理目标的系统提示、工具和子代理留档，以及系统指令。核心提示包含问题陈述或指令、打开文件的内容和目录映射。助手消息记录代理的逐步推理、工具和子代理调用记录及结果，以及监督者通信。这种组织方式促进了高效的信息流，增强了大语言模型的运行效率，减少了流转时长和成本。最初，文件更改以差异（diff）形式记录，仅显示修改内容，并定期进行合并。

**SICA：代码剖析：**深入研究SICA的实现，会发现有几个关键的设计选择支撑着它的能力。如前所述，该系统采用模块化架构构建，包含多个子代理，如编码代理、问题解决代理和推理代理。这些子代理由主代理调用，就像工具调用一样，用于分解复杂任务并有效管理上下文长度，特别是在那些扩展的元改进迭代期间。

该项目正在积极开发中，旨在为那些对在工具使用和其他智能体任务上进行大语言模型（LLM）后训练感兴趣的人提供一个强大的框架，完整代码可在<https://github.com/MaximeRobeyns/self_improving_coding_agent/>GitHub 仓库中进一步探索和贡献。

出于安全考虑，该项目大力强调使用 Docker 容器化，即代理在专用的 Docker 容器内运行。这是一项关键措施，因为它能与宿主机隔离，鉴于代理具备执行 shell 命令的能力，可降低意外操作文件系统等风险。

为确保透明度和可控性，系统通过交互式网页实现强大的可观测性，该网页可可视化事件总线上的事件以及代理的调用图。这能全面洞察代理的行为，使用户能够检查单个事件、读取监督者消息，并折叠子代理跟踪信息，以便更清晰地理解。

在核心智能方面，代理框架支持集成来自不同供应商的大语言模型（LLM），从而能够试验不同的模型，以找到最适合特定任务的模型。最后，一个关键组件是异步监督器，它是一个与主代理并发运行的大语言模型。该监督器会定期评估代理的行为，以发现是否存在病态偏差或停滞，并在必要时通过发送通知甚至取消代理的执行来进行干预。它接收系统状态的详细文本表示，包括调用图以及大语言模型消息、工具调用和响应的事件流，这使其能够检测到低效模式或重复工作。

在SICA的初步实施中，一个显著的挑战是促使基于大语言模型（LLM）的智能体在每次元改进迭代中独立提出新颖、创新、可行且引人入胜的改进方案。这一局限性，特别是在促进大语言模型智能体的开放式学习和真正创造力方面，仍然是当前研究的一个关键领域。

**AlphaEvolve和OpenEvolve**

**AlphaEvolve**是谷歌开发的一个AI智能体，旨在发现和优化算法。它结合使用大语言模型（LLMs），特别是Gemini模型（Flash和Pro）、自动评估系统和进化算法框架。该系统旨在推动理论数学和实际计算应用的发展。

AlphaEvolve采用了一组Gemini模型。Flash用于生成广泛的初始算法提案，而Pro则提供更深入的分析和优化。然后，根据预定义的标准自动评估和打分所提出的算法。这种评估提供反馈，用于迭代改进解决方案，从而产生优化的新颖算法。

在实际计算中，AlphaEvolve已部署在谷歌的基础设施中。它在数据中心调度方面展现出改进，使全球计算资源使用量减少了0.7%。它还通过为即将推出的张量处理单元（TPU）的Verilog代码提出优化建议，为硬件设计做出了贡献。此外，AlphaEvolve加速了AI性能，包括使Gemini架构的核心内核速度提高了23%，并对FlashAttention的底层GPU指令进行了高达32.5%的优化。

在基础研究领域，AlphaEvolve为矩阵乘法新算法的发现做出了贡献，其中包括一种针对4x4复值矩阵的方法，该方法使用48次标量乘法，超越了先前已知的解决方案。在更广泛的数学研究中，它在75%的情况下重新发现了现有最先进的解决方案来解决50多个开放问题，并在20%的情况下改进了现有解决方案，例如在接吻数问题上取得的进展。

**OpenEvolve**是一种进化式编码智能体，它利用大语言模型（LLMs，见图3）迭代优化代码。它编排了一个由大语言模型驱动的代码生成、评估和选择的流程，以持续改进适用于各种任务的程序。OpenEvolve的一个关键特性是它能够对整个代码文件进行进化，而不仅仅局限于单个函数。该智能体设计灵活，支持多种编程语言，并与任何大语言模型的OpenAI兼容API兼容。此外，它还集成了多目标优化，允许灵活的提示工程，并能够进行分布式评估，以高效应对复杂的编码挑战。

图3：OpenEvolve内部架构由一个控制器管理。该控制器协调几个关键组件：程序采样器、程序数据库、评估器池和大语言模型集成。其主要功能是促进它们的学习和适应过程，以提高代码质量。

此代码片段使用OpenEvolve库对程序进行进化优化。它使用初始程序、评估文件和配置文件的路径来初始化OpenEvolve系统。evolve.run(iterations=1000)行启动进化过程，运行1000次迭代以找到程序的改进版本。最后，它打印出进化过程中找到的最佳程序的指标，格式化为四位小数。

|  |
| --- |
| 从openevolve导入OpenEvolve  # 初始化系统  evolve = OpenEvolve(  initial\_program\_path="path/to/initial\_program.py",  evaluation\_file="path/to/evaluator.py",  config\_path="path/to/config.yaml"  )  # 运行进化  best\_program = await evolve.run(iterations=1000)  print(f"最佳程序指标：")  for name, value in best\_program.metrics.items():  print(f" {name}: {value:.4f}") |

**概览**

**问题：**AI智能体常常在动态且不可预测的环境中运行，在这种环境下，预先编程的逻辑是不够的。当面对初始设计时未预料到的新情况时，它们的性能可能会下降。如果没有从经验中学习的能力，智能体就无法优化其策略，也无法随着时间的推移个性化其交互。这种僵化限制了它们的有效性，并阻碍它们在复杂的现实场景中实现真正的自主性。

**原因：**标准化的解决方案是整合学习和适应机制，将静态智能体转变为动态、不断进化的系统。这使得智能体能够根据新数据和交互自主完善其知识和行为。智能体系统可以使用各种方法，从强化学习到更先进的技术，如自我修改，就像自我改进编码智能体（SICA）那样。像谷歌的AlphaEvolve这样的先进系统利用大语言模型和进化算法来发现全新且更高效的复杂问题解决方案。通过持续学习，智能体可以掌握新任务、提高性能，并适应不断变化的条件，而无需不断进行手动重新编程。

**经验法则：**在构建必须在动态、不确定或不断变化的环境中运行的智能体时，请使用此模式。对于需要个性化、持续性能改进以及自主处理新情况能力的应用程序而言，这一点至关重要。

**可视化总结**

图4：学习与适应模式

**要点总结**

* 学习和适应是指主体通过运用自身经验，在其所从事的事情上表现得更好，并应对新情况。
* "适应"是指智能体因学习而在行为或知识上发生的明显变化。
* 自我改进编码代理SICA通过根据过去的表现修改其代码来实现自我改进。这催生了像智能编辑器和AST符号定位器这样的工具。
* 拥有专门的“子智能体”和“监督者”有助于这些自我改进的系统管理大型任务并保持正轨。
* 大语言模型（LLM）的“上下文窗口”的设置方式（包括系统提示、核心提示和助手消息）对于智能体的工作效率至关重要。
* 对于那些需要在不断变化、充满不确定性或需要个性化处理的环境中运作的智能体来说，这种模式至关重要。
* 构建学习型智能体通常意味着将它们与机器学习工具连接起来，并管理数据的流动方式。
* 一个配备了基本编码工具的智能体系统，可以自主编辑自身，从而提高其在基准任务上的表现
* AlphaEvolve是谷歌的AI智能体，它利用大语言模型（LLMs）和进化框架自主发现并优化算法，显著提升基础研究和实际计算应用的水平。

**结论**

本章探讨了学习和适应在人工智能（AI）中的关键作用。AI智能体通过持续的数据获取和经验积累来提升其性能。自我改进编码智能体（SICA）就是一个例子，它通过代码修改自主提升自身能力。

我们已经回顾了能动AI的基本组成部分，包括架构、应用、规划、多智能体协作、内存管理以及学习和适应。学习原则对于多智能体系统的协同改进尤为重要。为实现这一点，调优数据必须准确反映完整的交互轨迹，捕捉每个参与智能体的个体输入和输出。

这些要素促成了重大进展，例如谷歌的AlphaEvolve。这个AI系统通过大语言模型（LLMs）、自动评估和进化方法独立发现并优化算法，推动了科学研究和计算技术的进步。这些模式可以结合起来构建复杂的AI系统。像AlphaEvolve这样的发展表明，AI智能体自主进行算法发现和优化是可以实现的。

**参考文献**

1. 萨顿（Sutton, R. S.）和巴托（Barto, A. G.）（2018年）。*《强化学习：导论》*。麻省理工学院出版社。
2. 古德费洛（Goodfellow, I.）、本吉奥（Bengio, Y.）和库尔维尔（Courville, A.）（2016 年）。*深度学习*。麻省理工学院出版社。
3. 米切尔，T. M.（1997）。*机器学习*。麦格劳-希尔出版社。
4. 近端策略优化**算法**，作者为John Schulman、Filip Wolski、Prafulla Dhariwal、Alec Radford和Oleg Klimov。你可以在arXiv上找到它：<https://arxiv.org/abs/1707.06347>
5. 罗贝因斯（Robeyns）、艾奇逊（Aitchison）和祖默（Szummer）（2025）。*一种自我改进的编码智能体*。arXiv:2504.15228v2。<https://arxiv.org/pdf/2504.15228> <https://github.com/MaximeRobeyns/self_improving_coding_agent>
6. AlphaEvolve博客<https://deepmind.google/discover/blog/alphaevolve-a-gemini-powered-coding-agent-for-designing-advanced-algorithms/>
7. OpenEvolve[，https://github.com/codelion/openevolve](https://github.com/codelion/openevolve)