ChatGPT

CodeAct Module (DSPy) – Technical & Strategic Analysis

1. Technical Documentation & Implementation

Overview & API: CodeAct is a module in the DSPy framework that enables **LLM-driven code generation combined with tool execution** [1]. It exposes a Python API via `dspy.CodeAct(signature, tools=[...], max_iters=5, interpreter=None)`, where you provide a **signature** (describing inputs/outputs) and a list of tool functions [2] [3]. Under the hood, CodeAct inherits from DSPy's ReAct and ProgramOfThought modules [4], effectively blending *chain-of-thought reasoning* with *code writing/execution*.

How It Works: At runtime, CodeAct operates iteratively: the LLM first receives the user query (and available tools) and **generates a Python code snippet** intended to solve the task [5]. This code is executed in a **secure Python sandbox** (powered by Deno + Pyodide) [6] [7]. The module captures the code's output and checks if the goal is achieved; if not, the LLM can incorporate the results or errors into its "trajectory" and **refine its code in subsequent iterations** (up to `max_iters`, default 5) [8] [9]. Internally, CodeAct constructs a specialized prompt signature that includes the problem instructions, tool specifications, a running `trajectory` of past actions, and fields for `generated_code` and a `finished` flag [9]. This guides the LLM to output both a code solution and an indicator of completion on each step. A secondary "extractor" chain-of-thought may then be used to produce the final answer in the specified output format using the results of the code execution [10].

Architecture & Execution Environment: By building on DSPy's `ProgramOfThought`, CodeAct leverages a **persistent sandbox interpreter** (`dspy.PythonInterpreter`) for stateful execution [11]. The sandbox ensures that generated code runs with restricted permissions (no arbitrary disk or network access unless explicitly enabled) [12] [13]. **Tool functions** provided to CodeAct are wrapped in DSPy's Tool abstraction and injected into the sandbox, allowing the LLM-generated code to call them directly [14]. This design effectively turns the LLM into a "mini-programmer" that writes code to orchestrate tool calls and Python logic, rather than returning answers in plain text. The approach is analogous to OpenAI's Code Interpreter or "chat with Python" paradigms – in fact, the documentation even notes that *"CodeAct…utilizes the Code Interpreter and predefined tools to solve the problem."* [4]. All language model calls made by CodeAct use the **globally configured LLM** (set via `dspy.configure(lm=...)`), so developers can plug in any supported model (GPT-4, Claude, local LLMs, etc.) as the code-generating brain.

Under-the-Hood Prompting: While the exact prompt template is abstracted away by DSPy, CodeAct essentially provides the LLM with: (1) a description of the task and the I/O signature, (2) documentation of each tool (function name, signature, docstring) for reference, and (3) the history of previous code attempts and their outcomes ("trajectory"). This context allows the model to write code that calls the given tools (and standard library functions) to derive the answer. If a code execution yields an error or partial result, that feedback is appended to the trajectory, and the LLM can attempt a corrected approach in the next iteration. This *think-code-test* loop continues until the model signals completion by setting `finished=True` [15] or until `max_iters` is reached. Finally, CodeAct can take the final program output and map it to the

requested output fields – for example, if the signature's output is an `answer` string or a numeric result, that value (perhaps computed or printed by the code) is returned in the `Prediction` object. In summary, **CodeAct turns the problem into a programming task for the LLM**, harnessing Python as a robust "action space" for reasoning. This design has been shown to improve success rates on complex tasks, as discussed later (since the LLM can use loops, conditionals, and tool combinations more freely than in rigid schemas) [16].

Source Code & SDK: CodeAct is part of the open-source DSPy library (MIT-licensed). Developers can inspect its implementation in the `dspy/predict/code_act.py` module [17]. Key components include the `CodeAct` class (subclassing `Module`) and the integration with DSPy's `Tool` and `PythonInterpreter` primitives. An important aspect is that **Deno** must be installed on the system to provide the sandbox JavaScript runtime for Pyodide [18] [7]. DSPy's tooling abstracts most of this setup – e.g. by default, `CodeAct` will instantiate a sandbox interpreter automatically [11] – but developers should ensure the environment is prepared (Deno installed, etc.). The DSPy SDK does not require separate paid services for CodeAct; it runs wherever you run DSPy (locally or on a server), using your chosen LLM's API under the hood.

2. Use Cases for CodeAct

Complex, Multi-step Tasks: CodeAct shines in scenarios where a query requires **several computational steps or conditional logic** that would be cumbersome for an LLM to juggle in pure natural language. For example, if asked *"Calculate the sum of the primes below 10,000 and then format the result as hex"*, a CodeAct-powered agent could write a short Python program to perform the calculation (looping over numbers, checking primality, summing them, then converting to hex). In a traditional ReAct agent, the model would have to either do math internally (error-prone) or call a "calculator" tool repeatedly – CodeAct can handle it in one coherent code snippet. Researchers note that **text-based tool calls lack flexibility** (each action is isolated and cannot easily reuse prior results or apply control flow) [19]. CodeAct's approach addresses this by letting the agent **carry out arbitrary computation** between tool invocations.

Repetitive or Looping Operations: One highlighted advantage is handling tasks that involve repetition or batch operations. *"Imagine giving an AI assistant a task to process 100 files with the same operation. Traditional approaches would require 100 separate commands. CodeAct? Just a simple for loop."* [20] In real terms, this could apply to use cases like batch-processing a dataset, iterating over multiple API calls, or monitoring a set of stock prices and making decisions – all in one code run. CodeAct enables the LLM to **compose multiple tool uses in one go**, which is especially useful for such batch tasks or when combining results from several sources.

Tool-Using Agents & API Orchestration: CodeAct is effective in agent scenarios where the AI has to integrate results from different tools or data sources. For instance, an AI assistant might need to query a database (via a provided `query_db` function), then call a calculation service, then format an output. Rather than prompt the LLM to sequentially call `query_db`, get an answer, call the next tool, etc., CodeAct can generate a small script that **calls all necessary tools and glues their outputs together** programmatically. This is both more efficient and less error-prone for the LLM. It closely resembles writing a simple program to solve a problem – a paradigm that is easier to verify and debug compared to interpreting a chain of vague natural-language steps.

Data Analysis and Math: Borrowing from the success of "code interpreter" style interactions, CodeAct can handle tasks like analyzing a CSV file, performing arithmetic or statistical computations, or transforming data formats. For example, if the tools include a `load_csv()` and some analysis functions, the LLM can write code to load data, compute summary statistics, and output a result. These are tasks where LLMs alone might struggle with accuracy, but by generating code to use reliable library functions (or custom tools), the accuracy improves. Empirically, consolidating actions into executable code can yield **higher success rates (~20% improvement)** on complex benchmarks [21]. This makes CodeAct suitable for use cases in **finance (e.g. running calculations on stock data)**, **scientific computing (using provided math utilities)**, or any domain where multi-step calculations are needed.

When CodeAct is Most Effective: Generally, you'll prefer CodeAct whenever the solution benefits from **clear procedural logic**. If an answer involves multiple conditional checks, looping through items, or combining outputs from several functions, having the model generate code will likely be more effective than prompting it to explain or do each step in English. It is particularly useful in *LLM agent* architectures – for example, the AI assistant Manus.im reportedly uses CodeAct as its core for handling user tasks [16]. In summary, CodeAct is a powerful choice for **LLM-augmented programming tasks**: it turns the problem into code, executes that code, and thus can solve problems that plain language prompting or single-function calling would struggle with.

3. Limitations & Known Issues

Despite its strengths, CodeAct comes with several important limitations (some inherent to its design, others current shortcomings of the implementation):

- **Tools Must Be Pure Functions:** CodeAct **only accepts standalone function objects as tools**, not bound methods or class instances [22]. For example, passing a class with a `__call__` (callable object) will raise an error [23]. The expectation is that each tool has no hidden state and produces deterministic outputs from inputs. This means you cannot directly use complex objects or stateful instances as tools – you may need to refactor them into pure functions. Additionally, functions with closures or free variables are problematic unless all dependencies are themselves included as tools (see next point).

- **No External Libraries in Tool Code: Third-party libraries are disallowed inside CodeAct's generated code** (and even in the tool definitions) unless they're part of Python's standard library [24]. The sandbox environment is initially limited to the standard library; if the LLM tries to `import numpy` or use an unavailable package, it will fail [25]. In practice, this means any specialized computation must be provided via the `tools` parameter. There are experimental ways to load pure Python packages into Pyodide (e.g. using `micropip` as seen in a community discussion), but heavy libraries (pandas, NumPy, etc.) can be too large or unsupported in the WebAssembly environment. This limitation affects tasks requiring complex data processing – developers might need to implement or wrap needed functionality in a custom tool beforehand.

- **All Dependencies Must Be Provided:** Every function or helper that your tool functions depend on **must be passed into CodeAct as part of the tool list** [26]. CodeAct cannot automatically resolve references that were not given. For instance, if `tool_A()` internally calls another function `helper_B()`, then `helper_B` should also be included in `tools=[…]` when constructing

CodeAct. Failing to do so will lead to runtime errors in the sandbox (since `helper_B` won't be defined there) [27]. Essentially, CodeAct's environment is isolated – it only knows the functions you explicitly supply. A related point is that **global variables or external state** used inside tools will not be present in the sandbox unless explicitly recreated there.

- **Sandbox Performance & Resource Limits:** The use of Pyodide (Python-on-Wasm) means that code execution is **significantly slower** than native Python for computation-heavy tasks. CodeAct is intended for relatively short, high-level code (leveraging vectorized library calls or simple logic), not long-running number-crunching. Additionally, the sandbox might have memory and time limits. In DSPy's default configuration, the sandbox runs inside Deno; extremely large outputs or infinite loops might either be terminated or hang the agent. The `max_iters` parameter (default 5) is a safeguard to prevent infinite refine loops, but developers should also ensure that the tasks are scoped appropriately to avoid runaway code generation.

- **No Built-in Internet Access:** By default, the sandbox has no network access (for security) [13]. So, CodeAct can't fetch external URLs or call online APIs unless you explicitly enable network domains in the PythonInterpreter settings. This is not so much a bug as a safety feature, but it does limit certain use cases (for example, you'd need to provide a tool function if you want the agent to perform web queries or similar). Enabling network or file system access is possible via the `PythonInterpreter` configuration (e.g. `enable_network_access` or `enable_read_paths`), but that must be done carefully to avoid security risks.

- **Error Handling and Debugging:** While CodeAct can attempt to correct its code when errors occur (since the traceback is captured and given to the LLM in the trajectory), it is not foolproof. Complex errors or subtle bugs might not be resolved within the allowed iterations. In some cases, the agent might stop with an exception or a wrong answer if it exhausts `max_iters`. As a developer, you might need to inspect the `act.history` or use DSPy's debugging callbacks to see what went wrong. This iterative debugging capability is powerful, but it also means an error could consume multiple LLM calls before resolution. In time-sensitive or cost-sensitive scenarios, this could be a limitation.

- **Environment Setup:** From an engineering standpoint, a minor limitation is that CodeAct requires an extra runtime (Deno) to be installed and configured. This adds a dependency to your deployment environment. If Deno is not available or the server lacks proper permissions, the CodeAct module will fail to initialize [18]. Ensuring the sandbox can start (and has access to necessary system resources) is an additional step compared to simpler agents that don't execute code.

It's worth noting that some of these limitations are active research areas. The concept of using code as an action space is fairly new, and solutions to incorporate more libraries or to formally verify the generated code are likely to evolve. As of DSPy's current version, the above constraints are the key points to remember when deciding if CodeAct fits your use case.

4. Best Practices for Using CodeAct

To get the most out of CodeAct, developers should follow several best practices:

- **Design Clear Signatures:** Start by precisely defining the input/output schema (Signature) for your task. This signature is used to auto-generate prompt instructions for the LLM, so include helpful descriptions. A well-defined signature (for example, `"text -> sentiment: bool"` or `"numbers -> sorted_numbers"`) guides the model and makes it easier to integrate the module's output into your application [3]. Clear expected outputs also reduce ambiguity when the model writes code to produce those outputs.

- **Provide Focused, Well-Documented Tools:** The tools you pass in should be **relevant to the task and thoroughly documented** (via function names and docstrings). CodeAct will include tool names and possibly their docstrings in the prompt it builds [9]. Thus, a descriptive name (`def find_max()` vs. `def f1()`) and a concise docstring improve the model's understanding of how to use the tool. Avoid dumping too many tools "just in case" – extraneous tools can confuse the LLM's decision-making. Instead, include only the tools that make sense for the task at hand. If you need the agent to use a particular capability (e.g. date parsing), implement it as a tool function and pass it in, rather than hoping the model will code it from scratch.

- **Ensure Tools Are Deterministic & Side-Effect Free:** Since CodeAct may call tools multiple times across iterations, it's best if the tools always return the same output for the same input and don't rely on external side effects. For example, a tool that calls a live API or a random number generator could produce variability that confuses the LLM's subsequent steps. If you must use such tools, consider caching their results or wrapping them to make them more predictable. This aligns with the requirement of pure functions [22] and will lead to more stable agent behavior.

- **Leverage the Sandbox Controls:** Use the `PythonInterpreter` options to your advantage. For instance, if your generated code needs to read or write files (perhaps as part of the task), pre-authorize specific paths with `enable_read_paths` / `enable_write_paths` when instantiating the interpreter [28]. Similarly, if network calls are needed (say your tools include an HTTP fetcher), whitelist the required domains via `enable_network_access` [13]. By configuring these at start, you prevent the model's code from hitting unnecessary security blocks. Always open the interpreter with a context manager (`with PythonInterpreter() as interp:`) so that it properly initializes and cleans up after execution [29].

- **Limit Iterations and Monitor Trajectory:** The default `max_iters=5` is a reasonable safeguard, but depending on your use case, you might tune it. For straightforward tasks, you can lower it (fewer attempts, saving cost/time). For complex tasks where the model might need several refinement loops, you could raise it slightly – but do so with caution to avoid runaway costs. It's a good practice to **log or print the** `act.history` **or trajectory after each call** during development to see how the agent is reasoning. This transparency can help you identify if the model is going down a wrong path or using tools sub-optimally, and then adjust your tool design or instructions accordingly.

- **Error Messages as Feedback:** If your module often struggles on first attempt (e.g., always off-by-one or minor syntax mistakes in code), consider capturing and surfacing error messages. CodeAct

automatically feeds Python exceptions back into the LLM's context for the next iteration – make sure your tool errors are also understandable. For example, if a tool raises a custom exception, ensure the exception message is clear, as it will appear in the trajectory. The model will use that info to debug. In essence, treat the LLM like a junior developer: give it meaningful feedback when something goes wrong.

- **Community Tips:** According to the DSPy community, one workaround for using external libraries is to preload them in the interpreter. For advanced users, **you can execute setup code in the sandbox before running CodeAct**, such as installing a pure Python package via `micropip`. For example, one might do:

```
interp = PythonInterpreter()
interp.execute("import micropip; await micropip.install('sympy')")
```

  to make a library available. This can enable certain libraries in CodeAct's environment, but it's experimental and may not work for large packages (and will add overhead). Always test such approaches thoroughly.

- **Security Considerations:** If you're deploying an app where end-users might influence the prompt or tools, remember that CodeAct is essentially executing code derived from an LLM – guard against prompt injection that could produce malicious code. Use the sandbox's restrictions (no network/file access by default) as a safety net. You might also implement timeouts or watchdogs for the code execution. In critical applications, consider reviewing the generated code (if possible) or running the agent in a firewalled environment.

By following these practices, you can harness CodeAct's power while avoiding common pitfalls. In short: **carefully curate what you give the agent (tools, instructions) and keep an eye on what it does (logging and limits)** for optimal results.

5. Comparison to Other Code/Agent Tools

CodeAct represents a **"code as action"** approach for LLM agents. Here we compare it to several related tools and paradigms:

- **VS. Traditional ReAct Agents:** *ReAct* (Reason+Act, e.g. LangChain or DSPy's own ReAct module) uses the LLM to produce natural-language reasoning and tool calls step by step. CodeAct, on the other hand, has the LLM output a **piece of executable code** that can perform multiple steps at once. This gives CodeAct a flexibility and efficiency edge on complex tasks – studies found it achieves up to *20% higher success rate than text or JSON-based actions* on benchmarks [21]. ReAct agents might require many back-and-forth turns (one per tool use), whereas CodeAct can solve it in fewer iterations by batching logic in code [20]. However, ReAct is conceptually simpler and may be easier to debug in sequential tasks (each step is readable). CodeAct's "program" might be harder to interpret at a glance, though the trajectory helps. In essence, **ReAct is like guiding the AI step-by-step, while CodeAct lets the AI script its own multi-step solution**. Many real-world agent systems (e.g.

OpenAI's function-calling) currently use a ReAct-style schema, but CodeAct demonstrates a more general and often more powerful paradigm [30] [16].

- **VS. OpenAI Code Interpreter (ChatGPT):** OpenAI's Code Interpreter (now part of ChatGPT as "Advanced Data Analysis") similarly allows an AI to execute Python code in a sandbox. The philosophy is the same – using code to extend the AI's capabilities. One key difference is **availability and integration**: CodeAct is an open-source module you can integrate into your own Python apps, whereas OpenAI's Code Interpreter is a closed service (limited to the ChatGPT UI and not directly via API as of now). Code Interpreter comes with many libraries pre-installed (pandas, NumPy, etc.) and handles file uploads, which is convenient for data tasks. CodeAct's environment starts more bare-bones (only standard library and provided tools), requiring you to anticipate needs via tools or micropip installs. On the flip side, CodeAct gives you fine-grained control: you decide which tools/functions the AI can use and can embed it into custom agent workflows, whereas OpenAI's solution is a more general black box. In terms of **strategic use**, if you need an AI to do code-based reasoning inside your product or backend, CodeAct (or similar frameworks) is the way to go, since OpenAI's offering cannot be self-hosted or customized. Both approaches underscore the benefit of code execution for complex queries, but CodeAct is **LLM-agnostic and developer-extensible**, while Code Interpreter is proprietary.

- **VS. GitHub Copilot:** GitHub Copilot is an AI pair-programmer, not an autonomous agent. It suggests code completions as you write code. In contrast to CodeAct, Copilot *does not execute code or make decisions* – it's guided by the human developer in an IDE. There is a conceptual link: CodeAct essentially uses the LLM to write code like Copilot would, but then it immediately runs that code to affect the world (get an answer or perform an action). Copilot doesn't have an execution loop; it's meant for coding assistance rather than letting an AI solve a task end-to-end. If we compare them as tools: CodeAct is for **runtime automation** (the AI acts autonomously by coding), whereas Copilot is for **developer productivity** (AI helps a human code). They serve different audiences. That said, the quality of code generation from Copilot/GPT-4 is what makes CodeAct possible – it's leveraging the same underlying capability (LLMs writing code), but applied in an agentic loop. One could imagine a future Copilot plugin that uses CodeAct internally to test or run suggested code, but out-of-the-box Copilot doesn't close that loop.

- **VS. Auto-GPT / BabyAGI and similar agents:** Auto-GPT-like agents attempt to achieve goals by dynamically generating plans, code, or tool calls, and can spawn new threads or tasks. Some versions do employ code execution as one of the steps (e.g., they might write to a file and then execute it). CodeAct can be seen as a **focused implementation of the "execute code" step** in an agent pipeline. It doesn't on its own decide high-level goals or do long-term planning – it is typically one module within a larger agent system (for example, you might have a high-level planner that decides when to invoke a CodeAct module for a subtask). Compared to generic Auto-GPT, CodeAct is *more constrained but arguably more robust* in its domain: it sticks to Python code execution with provided tools. Auto-GPT agents have been known to be brittle or inefficient, partly because they lack a structured way to use tools. CodeAct offers that structure by unifying what could be many separate actions into one coherent program [31]. In practice, CodeAct (via DSPy) can be combined with other modules – e.g., one could use a ChainOfThought or planning module to decide a strategy, then use CodeAct to execute the complicated part of the strategy (this pattern is hinted by the "Think -> Code -> Observe" loop in the literature). Auto-GPT systems might incorporate CodeAct-like

execution to improve their effectiveness. In summary, CodeAct is **complementary** to such agents: it can be the engine that handles coding tasks within a larger autonomous agent framework.

- **VS. LlamaIndex Code Action Agent:** Notably, the idea of CodeAct has been adopted elsewhere – LlamaIndex offers a *"prebuilt CodeAct Agent"* inspired by the same concept [32]. The LlamaIndex variant similarly lets you provide functions to an agent and have it generate code to call them. One advantage cited is that you don't need to pre-enumerate every possible tool invocation in prompts, because the agent can just write whatever sequence of function calls it needs [33]. The LlamaIndex implementation emphasizes the need for proper sandboxing as well [34], echoing the safety considerations of DSPy's approach. In feature terms, LlamaIndex's CodeActAgent and DSPy's CodeAct both illustrate a move beyond plain JSON-based function calling. The key differences would lie in integration: if you're already using LlamaIndex for document queries, their agent might slot in easily. If you need a more general/powerful AI programming framework, DSPy's CodeAct offers integration with DSPy's optimization, training, and modular pipeline features. Both underscore a trend: **code-as-action is emerging as a powerful pattern**, so much so that multiple frameworks are converging on it.

Summary of Comparison: CodeAct (DSPy) is at the cutting edge of LLM agent design, pushing beyond the limitations of single-step tool use. Compared to mainstream tools like ChatGPT's function calling or Copilot, it offers a more **programmatic, self-directed problem solving** method. It requires more setup and careful use, but rewards with potentially higher success on complex tasks. Strategically, if an AI engineer's goal is to build **reliable, modular AI workflows** (where the AI can reason, act, and even write code to use tools), CodeAct is a compelling option, and a differentiator from simpler prompt-based methods.

6. Integration Considerations

When integrating CodeAct into a real-world project, keep in mind the following:

- **System Requirements & Deployment:** Ensure that the host environment has **Deno installed**, since the Python sandbox depends on it [7]. In a Docker deployment, you'd include Deno in the image. The CodeAct module is part of the `dspy` Python package, so a simple `pip install dspy` (latest version) brings it in [35]. Check the DSPy version compatibility – as of 2025, DSPy is actively developed (nearly 30k stars on GitHub), so watch for version updates that might change the API. It's wise to pin a version or thoroughly test when upgrading. Because CodeAct is relatively advanced, you may find that **cutting-edge releases introduce changes**; reading the release notes or Discord community updates can be helpful. The good news is DSPy is open-source and well-maintained by the Stanford NLP group, with an active Discord for support [36] [37].

- **API Reliability & Usage Costs:** CodeAct itself doesn't have an external API call (it runs locally), but it does invoke the LLM for each iteration. Thus, reliability mostly hinges on your chosen LLM provider. If using OpenAI API, you'll want error handling for API timeouts or rate limits. Each iteration might consume a prompt containing the tools and trajectory – which can grow in token length. Monitor the token usage; DSPy provides usage tracking for modules (you can retrieve token counts from predictions) which can help estimate cost [38]. A complex CodeAct run with many iterations can be token-expensive. To mitigate surprise costs, consider setting a budget: e.g., limit `max_iters`, or even implement logic to abort if the trajectory grows too large. In terms of throughput, executing code (especially with Pyodide overhead) adds latency to each iteration – expect CodeAct to be slower

than a pure LLM call. If building a user-facing app, you might use DSPy's streaming or callback capabilities to inform the user of progress (e.g., print intermediate results or a "thinking..." status) to improve UX.

- **Integration Pattern:** CodeAct can be used as a stand-alone solution for a single task, but often you'll integrate it as part of a broader pipeline. For instance, you might have a conversational agent that, upon recognizing a certain intent (like "data analysis" or "math problem"), routes to a CodeAct module. In such cases, ensure you properly **initialize the DSPy module once and reuse it**. Creating a CodeAct object is not extremely heavy, but it does spin up a PythonInterpreter; doing this for every user query could be inefficient. A typical pattern is to instantiate `act = CodeAct(...)` at app startup or first use, and then call `act(...)` for each relevant request, rather than recreating it each time. DSPy modules are designed to be reusable; they even support serialization (e.g., `act.save()` to save state) [39] [40], though for CodeAct there isn't much state to save beyond history and maybe some learned prompt (in contrast, a few-shot module could retain learned examples).

- **Scaling Considerations:** If you need to handle multiple queries in parallel, be cautious with the sandbox. The `PythonInterpreter` isn't inherently thread-safe for concurrent access. One approach is to give each worker/thread its own interpreter instance. Another is to use DSPy's `Parallel` module or batch functionality to process multiple inputs in parallel if they are independent [41] [42]. Also consider the memory impact – each Pyodide instance can consume tens of MBs of memory. If you expect high concurrency, test how many simultaneous interpreters your server can handle. There might be an opportunity to pool interpreters or reuse them across tasks, but currently DSPy doesn't provide an out-of-the-box pool for PythonInterpreter.

- **Pricing:** The DSPy library is free to use. The main costs will come from the LLM inference (e.g., OpenAI API fees or infrastructure costs for running a model) and possibly compute overhead for the sandbox. If using a paid API, every iteration in CodeAct is essentially a prompt + completion. Fortunately, CodeAct's design often reduces the total number of calls needed by allowing more work per call (one code generation can replace several tool-call prompts). Still, budgeting for worst-case (e.g., max_iters=5 with a large prompt each time) is prudent. If using an open-source model (like CodeActAgent-Mistral-7B from the research paper [43]), you avoid API costs but must host the model – DSPy can integrate with local model backends as shown in documentation (e.g., via `dspy.LM("openai/gpt-4", api_key=...)` or local Ollama, etc. [44] [45]). Ensure whichever model you use is proficient in code generation for best results (GPT-4, Claude 2, or specialized CodeAct-tuned models would perform better than a generic 7B model at coding).

- **Developer Support & Community:** As an open project, DSPy has community resources. The official GitHub (stanfordnlp/dspy) is active – issues and discussions there often address advanced usage of modules like CodeAct (for example, questions about using external libraries in CodeAct have been discussed) [46]. The maintainers (including researchers from Stanford) are responsive in addressing bugs or feature requests (CodeAct itself is an evolving feature, and community feedback is likely to shape it). There's also a DSPy Discord server [36] where AI engineers share tips; one can find advice from others who have integrated DSPy into production. If you encounter a bug (say, the sandbox hanging or a tool not functioning as expected), it's worth checking if the issue is known or fixed in a later release. Since CodeAct combines a lot of moving parts (LLM prompting, code execution, etc.), debugging might require looking at logs from both the LLM side and the sandbox side. Logging the

LLM's generated code and any exceptions is crucial in a production integration – you'll want that data if something fails, to either improve your prompts or report a bug upstream.

- **Future-Proofing:** AI tooling is rapidly advancing. CodeAct's approach is at the frontier, and we can expect future improvements. Keep an eye on newer versions of DSPy – for instance, there may be performance optimizations (the DSPy team or others might switch to alternative sandbox tech if faster), or expanded support for libraries. The research community (e.g., the CodeAct ICML paper and others) are working on making LLMs better at code actions, so newer LLMs fine-tuned for this could be swapped in to immediately boost performance with the same DSPy interface. Because DSPy is declarative, **swapping the underlying model is easy** (`dspy.configure(lm=new_model)`), meaning you can benefit from advances without rewriting your agent logic. This decoupling is a strategic advantage of using DSPy/CodeAct now – your investment in designing tool functions and workflows is not tied to a single provider's LLM.

In conclusion, integrating CodeAct requires some careful setup and understanding of its runtime, but it offers a powerful capability to your AI system. By planning for its requirements (sandbox and model) and constraints (token usage, iteration limits), you can deploy an agent that leverages cutting-edge "code as action" techniques. The end result is an AI component that is more adaptable and capable, potentially handling tasks that would otherwise require writing a lot of static code or orchestrating dozens of individual prompt calls. With proper integration, CodeAct can be a **force multiplier** for LLM applications, combining the creativity of AI with the precision of code execution [16].

Sources:

- Official DSPy Documentation – *CodeAct Module* [1] [5] [22] [24]
- DSPy GitHub (stanfordnlp/dspy) – CodeAct source and README [4] [8] [9]
- *Executable Code Actions Elicit Better LLM Agents* (Wang et al., 2024) – CodeAct research paper results [21]
- Medium article by heavendai – *"CodeAct: When LLMs Execute Code Instead of Text"* [20] [16]
- DSPy PythonInterpreter Docs – sandbox requirements and usage [7] [28] [13]
- DSPy Cheatsheet & Tutorials – usage examples and patterns [47] [18]

---

[1] [2] [3] [4] [5] [6] [8] [9] [10] [11] [14] [15] [17] [22] [23] [24] [25] [26] [27] [38] [39] [40] [41] [42] CodeAct - DSPy
https://dspy.ai/api/modules/CodeAct/

[7] [12] [13] [28] [29] PythonInterpreter - DSPy
https://dspy.ai/api/tools/PythonInterpreter/

[16] [30] [31] CodeAct: The Engine Behind Manus— How LLMs Are Learning to Code Their Way to Action | by ArXiv In-depth Analysis | Towards Dev
https://towardsdev.com/codeact-the-engine-behind-manus-how-llms-are-learning-to-code-their-way-to-action-17c6c0fe1068?gi=59ebaa652694

[18] ProgramOfThought - DSPy
https://dspy.ai/api/modules/ProgramOfThought/

19 20 CodeAct: When LLMs Execute Code Instead of Text — A Game-Changing Approach to AI Agents | by heavendai | Oct, 2025 | Medium
https://medium.com/@mingyang.heaven/codeact-when-llms-execute-code-instead-of-text-a-game-changing-approach-to-ai-agents-daea79f5a72c

21 43 GitHub - xingyaoww/code-act: Official Repo for ICML 2024 paper "Executable Code Actions Elicit Better LLM Agents" by Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, Heng Ji.
https://github.com/xingyaoww/code-act

32 33 34 Prebuilt CodeAct Agent w/ LlamaIndex | LlamaIndex Python Documentation
https://developers.llamaindex.ai/python/examples/agent/code_act_agent/

35 36 37 GitHub - stanfordnlp/dspy: DSPy: The framework for programming—not prompting—language models
https://github.com/stanfordnlp/dspy

44 45 DSPy
https://dspy.ai/

46 Use Tool functions that require external libaries in CodeAct · Issue #8839 · stanfordnlp/dspy · GitHub
https://github.com/stanfordnlp/dspy/issues/8839

47 Cheatsheet - DSPy
https://dspy.ai/cheatsheet/