ChatGPT

Multi-Agent Collaboration with OpenAI Codex CLI

Overview: Multiple Codex Agents and Emergent Collaboration

Using **multiple AI "assistant" agents** in tandem can significantly enhance coding projects. Recent research on large-language-model (LLM) teams shows that a coordinated group of specialist agents can solve complex software tasks far better than any single agent alone [1] [2]. For example, a new framework called MAGIS (Multi-Agent GitHub Issue Solver) had four AI agents collaborate (with roles like Manager, Developer, etc.) and achieved an **eight-fold improvement** in resolving real GitHub issues compared to GPT-4 working solo [3]. Such multi-agent setups also display **emergent behaviors** – unexpected but productive dynamics that arise from agent interactions [1]. In other words, a well-orchestrated AI team can become *greater than the sum of its parts* [4], uncovering creative solutions or catching errors that a lone assistant might miss.

Agent "archetypes" (roles) are key to this synergy. By giving each Codex agent a distinct role or personality, you mimic a human software team with diverse expertise. One agent might act as a **Planner/Manager** (breaking down tasks and coordinating others), while another is a **Coder** (writing and refactoring code), and a third serves as a **Tester/QA** (reviewing code and running tests) [5]. This mirrors proven workflows – for instance, MAGIS's agents included a *Manager* to devise a plan and assign work, *Developer* agents to implement code, and a *QA* agent to verify the changes [5]. Likewise, the MetaGPT project takes a **"software company"** approach: it assigns GPT-based agents to roles like product manager, architect, project manager, and engineer, each following Standard Operating Procedures to collaborate on tasks [6]. By simulating such archetypes, you encourage each AI to contribute unique insights and **collaboratively evolve** the codebase. Over multiple iterations, their interactions can produce increasingly refined and innovative outcomes – the hallmark of emergent behavior.

Setting Up Codex CLI on an Ubuntu VM

To experiment with multi-agent orchestration, first ensure you have the **OpenAI Codex CLI** installed and configured on your Ubuntu environment. (Codex CLI supports Ubuntu 20.04+ or Debian Linux, as well as macOS, with Node.js 18+ or 20+ installed [7].) Install the CLI via npm if you haven't already: for example, `npm install -g @openai/codex` will globally add the `codex` command [8]. You'll need to set your OpenAI API key as an environment variable (e.g. `export OPENAI_API_KEY="sk-..."`) so the CLI can access the Codex model [9]. Once set up, you can invoke Codex in the terminal either interactively (`codex` opens a REPL chat) or by providing a one-shot prompt/command. For instance, running:

```
codex --approval-mode full-auto "create the fanciest todo-list app"
```

will instruct the agent to start building a project (with full autonomy to execute code) [10]. In general, you'll want to use **"Full Auto" mode for multi-agent runs**, so the agents can act without pausing for your approval on each step. Full-auto mode lets Codex read/write files and run shell commands autonomously (within a sandbox), instead of constantly asking permission [11]. This is crucial for letting multiple AI agents

work continuously on the codebase. (Of course, only enable full autonomy once you're confident in the safety of the environment, since it gives the AI broad latitude to modify files and execute code in the VM [11].)

Project context: Prepare a code workspace (e.g. a Git repository or project folder) that all agents will share. Populate it with any starter code or an empty scaffold for them to build on. It's wise to initialize version control (git) so you can track changes each agent makes and roll back if needed. OpenAI's Codex is *repo-aware* – it understands it's operating within a codebase and will leverage existing files and tests in context [12]. By working in a shared directory, agents can communicate implicitly through the code itself: one agent's output (new code, documentation, test results, etc.) becomes the next agent's input.

Defining Agent Archetypes with `AGENTS.md`

Codex CLI provides a mechanism to guide and customize the AI's behavior using special markdown files called `AGENTS.md` [13]. These files act like project or persona notes for the AI. When Codex runs, it automatically looks for any `AGENTS.md` in your environment and **merges them into its prompt context** (with a hierarchy: a global `~/.codex/AGENTS.md` for personal guidance, plus any `AGENTS.md` in the repo root or current subfolder) [13]. According to OpenAI, you can use these files to tell the Codex agent *how to navigate your codebase, which commands to use for building or testing, and the conventions or standards to follow* [14]. Essentially, `AGENTS.md` is where you encode the "knowledge" and rules of your project for the AI to follow.

You can leverage this feature to define multiple agent archetypes. For example, create separate `AGENTS.md` profiles for each role: one for the **Architect/Manager** agent that contains high-level guidelines (overall project goals, architectural principles, and a prompt to always create a plan or design document), another for the **Developer** agent with instructions to focus on writing correct code and perhaps notes about the coding style or specific module details, and a third for the **Tester/QA** agent that explains how to run tests and what quality criteria to enforce. These files might reside in different subdirectories or be swapped in when running each agent. Since Codex merges guidance from the repo root and current folder, you could, for instance, have a main `AGENTS.md` with common info and supplemental ones in subfolders like `agents/manager/AGENTS.md`, `agents/dev/AGENTS.md`, etc. When running a given agent, you simply launch Codex from the directory that contains that agent's guidance file (so that it picks up those specific notes). Codex will then bias its behavior according to the role description you provided. *For instance, your "Manager" agent's file might say:*

> **Role:** *Project Planner.* You analyze feature requests or bug reports and break them into development tasks. You do **not** write code directly; instead, you produce a clear project plan, pseudo-code, or TODO list for developers. Always consider the overall architecture and mention any design patterns or modules affected.

Meanwhile, the "Developer" agent's `AGENTS.md` might include:

> **Role:** *Software Engineer.* You implement features and fixes in code. Given a project plan or issue description, write high-quality code following our style guide. Ensure the project runs and passes all tests. You can create new files or modify existing ones as needed, and you will

run `npm test` (or appropriate commands) to validate your changes, iterating until tests pass.

By embedding such instructions, each Codex instance gains a pseudo-personality aligned with its archetype. When you later automate their interaction, these profiles help each agent stay in its lane (e.g. planner vs coder) while working toward the common goal. *(Note:* The content and format of `AGENTS.md` are flexible – they can include any text or checklists you find useful. They behave like extended system prompts for the Codex model.)

Orchestrating Agent Communication and Collaboration

With multiple Codex agents configured, the next challenge is **getting them to talk to each other and cooperate**. Since the Codex CLI wasn't explicitly built for multi-agent dialogs, you'll need to set up an orchestration logic externally (for example, a **bash or Python script** acting as a moderator). The basic idea is to run the agents in a sequence (or in parallel with some synchronization) such that each one picks up where the last left off, gradually converging on a solution. All agents share the same **workspace (code repository)**, so **their communication can be through the code and files** they modify, as well as any notes they leave for each other.

A straightforward approach is **sequential turn-taking**: one agent runs, produces some output or changes, then the next agent runs taking those changes into account, and so on in rounds. For example:

1. **Planning round (Agent 1 – "Architect"):** Use the Manager/Planner agent to analyze the project goal or issue. Launch Codex CLI with the manager's profile, prompting it with something like: *"Plan: Analyze the current project and outline a step-by-step plan to implement feature X (or fix issue Y). Provide a list of tasks or modules to change."* The planner agent will read the repository (Codex can read all files in the repo in its context) and generate a plan or TODO list. You might direct it to save this plan to a file (e.g. `PLAN.md`) for others to read. Because Codex can execute shell commands, you could even have it echo its plan into a file – but simplest is to capture its text output from the CLI. The orchestration script can save the assistant's written plan into `PLAN.md`.

2. **Development round (Agent 2 – "Coder"):** Next, invoke the Developer agent. This time the prompt could be: *"Implement the plan in PLAN.md. Follow the steps and write the necessary code. Ensure all tests pass."* Since `PLAN.md` is now part of the workspace, Codex will see it (and you also have the developer's `AGENTS.md` guiding it to follow plans and write code). The developer agent will proceed to create or modify code files accordingly. In **full-auto mode**, it can run build commands or tests as needed to verify its work [11]. You'll see it propose changes (diffs) or commit them if auto-approved. Let it run to completion – ideally it should indicate "All tasks done" or simply finish after making changes. The script can monitor the codex process output to know when it's done, or set a time limit per round.

3. **Testing/Review round (Agent 3 – "QA"):** After code is written, trigger the QA agent. Prompt example: *"Review the codebase for any errors or stylistic issues. Run all tests and report any failures. If tests fail or the plan wasn't fully addressed, list what needs fixing."* This QA agent might run `npm test` or other commands (Codex will do so in the sandbox) and gather results. If tests pass, it could approve the changes and maybe even deploy or conclude. If tests fail or something looks off, the QA

agent can write feedback – perhaps updating a `REPORT.md` or leaving comments in the code or plan about what went wrong.

4. **Iterate further rounds as needed:** You can loop back to the Developer to fix issues identified by QA, or back to Planner if new subtasks emerged. For instance, if QA says "Test XYZ is failing due to an edge case," your script could prompt the Developer agent again: *"Fix the bug that QA identified regarding XYZ failing."* This cycle can repeat until QA approves or a set number of iterations is reached.

Throughout this process, the **shared artifacts (code files, PLAN.md, REPORT.md, test results)** are the medium of communication. Each agent indirectly "speaks" to the others by altering the state of the project. To make communication clearer, you can also have agents write explicit notes. For example, the Planner could write its plan prefaced with "# Plan for Feature X" and detailed steps, which the Developer will read. The Developer might insert comments in code like "// TODO: handle edge case for Y (from Planner's step 3)" if it can't finish a part, which the QA or Planner can notice and address.

Another technique is using **commit messages or a version control branch per agent**. You could let each agent commit its changes with a message (Codex will output diffs and can commit if instructed). The commit message could serve as the agent's summary of work done or any assumptions. Then the next agent can be prompted to `git log -1` to read the last commit message and code diff, simulating how team members review each other's commits. However, this might be more complex; a simpler file-based communication is often sufficient in a controlled VM environment.

To automate running Codex CLI for each agent, you can use a script with the `codex exec` command. The CLI supports non-interactive usage suitable for scripting. For example, in a CI pipeline one might do:

```
codex exec --full-auto "update CHANGELOG for next release"
```

which runs a one-shot instruction in full-auto mode [15]. Similarly, your script can call something like:

```
codex exec --full-auto "PLAN.md exists. As a Project Manager AI, read it and
refine the development plan for the team."
```

then later:

```
codex exec --full-auto "Implement the tasks in PLAN.md. (You are a Developer AI
collaborating on this project.)"
```

and so forth. Each `codex exec` invocation spins up the Codex agent anew with the current files, plus whatever guidance `AGENTS.md` it finds in the working directory [13]. Since the state isn't persisted between runs (aside from file changes), the orchestrator (your script) is responsible for feeding the right context each time (via files and prompt text). It's a bit like prompting a series of ChatGPT instances in a

coordinated way, except here each instance can actually execute code and make persistent changes to the environment.

Tip: Be sure to run all agents in the **same project directory** (or ensure they each have access to the latest project state). If you instead cloned multiple separate copies of the repo for each agent role, you'd then need to merge their outputs, which is complicated. It's usually easier to have them operate sequentially on one workspace so you don't get diverging code. The trade-off is that they work one after the other rather than truly concurrently – but given that Codex tasks can utilize parallelism internally (it can run tests in parallel, or you can open multiple terminal sessions if needed), sequential turns are fine and often mimic a reasonable dev workflow (plan → code → test → refine).

Simulating Archetypes and Encouraging Emergent Behavior

By carefully crafting each agent's role and interactions, you set the stage for **emergent problem-solving**. Each archetype brings a different "skillset" or viewpoint, and when these outputs feed into each other, new creative solutions can emerge that wouldn't have been produced by a single monolithic AI. For instance, a **"Creative Designer"** agent might suggest an unconventional UI or refactoring that a purely analytical coder agent wouldn't think of – but the analytical **"Engineer"** agent can then refine those ideas into robust code. Conversely, an Engineer agent might pinpoint technical constraints, prompting the creative agent to adjust the design. This back-and-forth can lead to superior outcomes. In human teams, diversity of thought leads to innovation; similarly, in AI teams, having archetypes with varied approaches yields richer exploration of the solution space.

Notably, studies like *AgentVerse* find that multi-agent groups, when well-orchestrated, exhibit **specific collaborative behaviors** that improve efficiency – akin to division of labor or bouncing ideas off one another [16]. You may observe your Codex agents spontaneously dividing tasks or validating each other's work. For example, your QA agent might notice something the Developer overlooked and directly adjust a minor detail, even if you didn't explicitly prompt it to (Codex might decide to "just fix a small typo" during the QA phase). Similarly, the Developer agent might partially anticipate QA by writing extra tests or explaining its code in comments, going beyond its basic mandate because it "knows" a QA check is coming. These are signs of emergent synergy.

To further simulate archetypal differences, you can **vary the model or parameters per agent**. The Codex CLI allows choosing different underlying models or providers via configuration profiles [17] [18]. For instance, you might run one agent with GPT-4 (more thorough, but slower) and another with GPT-3.5 (faster, maybe more varied output). Or even use a community model via the CLI's provider plugin (like an Ollama local model or OpenRouter API) to inject diversity [19] [20]. A faster, "impatient" model could simulate a quick, reactive agent, while a more powerful model plays the thoughtful strategist. Differing model outputs could spark emergent ideas when combined. However, keep in mind consistency and compatibility of the code suggestions – mixing models might require more oversight if one produces lower-quality code.

Practical Example Workflow

Let's tie this together with a concrete mini-scenario: **Suppose you want the agents to collaborate on adding a new feature to a web app.** The feature involves backend logic and some UI changes. Here's how a cycle might look:

- **Agent 1 (Planner)** – Launch in the repo root (with Planner `AGENTS.md`). Prompt: *"Our goal is to implement Feature X (describe it). Please analyze the repository and list the files/modules that need changes, and outline a plan in steps."*
  Outcome: Agent 1 produces a markdown list in `PLAN.md` enumerating steps, e.g., "1. Update `api/routes.js` for new endpoint, 2. Modify `frontend/App.jsx` for UI, 3. Write unit tests for …". It commits or saves this plan.

- **Agent 2 (Backend Developer)** – Launch in project subdir or with profile focusing on backend (with Dev `AGENTS.md`). Prompt: *"Implement backend parts of Feature X as per PLAN.md (steps 1 and 3). Focus only on server-side changes and tests."*
  Outcome: Agent 2 reads the plan, edits `api/routes.js`, perhaps creates `api/services/featureX.js`, and writes tests in `api/__tests__/featureX.test.js`. It runs tests; suppose backend tests pass. It might skip UI parts since instructed to focus server-side. It commits changes with message "Backend for Feature X done, tests passing".

- **Agent 3 (Frontend Developer)** – Prompt: *"Implement the frontend/UI for Feature X (step 2 in PLAN.md). The backend is done (see commit or code). Ensure UI is integrated with the new API and add any needed frontend tests."*
  Outcome: Agent 3 updates `frontend/App.jsx` or relevant files, possibly adjusts CSS, etc. Runs `npm run build` or frontend tests if any. Fixes any issues, then commits "Frontend for Feature X done."

- **Agent 4 (Integrator/QA)** – Prompt: *"Verify Feature X end-to-end. Run all tests, and do a quick code review. If something is missing or failing, report it or fix if trivial."*
  Outcome: Agent 4 runs the full test suite. Imagine one test fails due to an integration bug. It identifies that maybe the frontend expected an API response slightly differently. Agent 4 could either directly fix the minor discrepancy or at least log it in a `REPORT.md`: e.g., "Test `FeatureX.test.js` failing – likely need to adjust API response format." It leaves the repo with test results.

- **Iteration:** The orchestrator sees the QA report of a bug. It prompts *Agent 2 or 3* again accordingly (whichever is responsible, here maybe backend) to fix the issue. They do so, commit, tests pass, QA re-runs and confirms all good.

- **Completion:** All agents conclude the feature is implemented and verified. You, as the human overseer, can now review the git history of changes (each commit from an agent) and ensure everything looks acceptable, then deploy or merge the feature.

Throughout this, notice how the agents essentially **communicated via artifacts**: the plan, the code, tests, and the QA report. No direct chat messages were exchanged, but the *effect* is that of a conversation mediated by the project state. The planner "told" the developers what to do via the plan; the developers

"told" QA what they did by writing code and tests (and could include comments or commit messages); QA "told" the developers what was wrong by writing a report or leaving failing tests, and so on. This is a form of machine-to-machine communication tailored to coding tasks. In multi-agent literature, it's common to use a shared memory or blackboard for agents to write to and read from – here, the code repository is that shared memory.

Considerations and Tips

Orchestrating multi-agent Codex sessions is bleeding-edge and may require some trial and error. Here are a few tips and caveats:

- **Start Small:** Begin with just two agents (e.g. a Coder and a Reviewer) to get a feel for the interaction. Even a simple loop where one agent writes code and another critiques or tests it can add value. You can gradually add more specialized agents as needed.

- **Prompt Clarity:** Be very clear in each agent's prompt about what its task is *and what context to use*. Since each `codex exec` call is stateless except for files, explicitly mention filenames or goals. For example: "File `PLAN.md` contains the plan. Implement it." or "The previous agent's output is in `REPORT.md`. Address those points." This ensures the agent knows where to look. Also, remind them of their role ("As a QA agent, you should..."), although your `AGENTS.md` likely covers that mindset.

- **Synchronize Access:** If you attempt to run agents truly in parallel threads, be careful with them writing to the same files concurrently. You might end up with merge conflicts or overwritten work. Using git can help flag conflicts, but easiest is to run them one at a time or have them work on separate areas (e.g., one agent per distinct submodule) and then integrate. Given Codex tasks can take anywhere from seconds to minutes [21], sequential operation is usually fine. If you do want parallelism (for speed on independent subtasks), consider splitting the project into parts where agents won't collide (different directories), or spin up separate sandbox directories and later merge the results manually or via an integration agent.

- **Quality Control:** Emergent behavior can be double-edged – sometimes agents might go off on tangents or make surprising but incorrect changes. Always review the final code. Codex generally provides *cited justifications* for changes (in the ChatGPT UI it shows logs/tests) [22], but in CLI you'll see diffs and test outputs. Verify that the changes make sense and meet your requirements. The multi-agent setup reduces the chance of glaring mistakes (since agents check each other), but it's not foolproof. The human in the loop is still important as the ultimate QA and product owner.

- **Leverage Tests and Tools:** If your project has a test suite or linters, use them in the loop (automatically via the QA agent). Codex is good at improving code when it can run tests and see failures [23] [24] – that's a form of feedback. Each agent should ideally have some way to evaluate progress: the Developer agent uses tests passing as success criteria, the QA agent could use static analysis or performance checks, etc. This keeps the agents grounded and less likely to hallucinate irrelevant output.

- **Agent Coordination Logic:** For more complex scenarios, you might implement a **meta-controller agent** or simply a more advanced script. For instance, an "Executive" agent could decide which

specialized agent to call next based on the state (like a failure triggers calling the Developer again). However, this can also be hard-coded in your script with simple rules (as we described: if tests fail, call dev agent with fix prompt). Some frameworks (e.g. OpenAI's experimental *Agents SDK* or third-party libraries) are emerging to handle such decision-making. But with Codex CLI, a custom script is the way to go for now.

In summary, **yes – you can absolutely have multiple Codex-based agents interact and collaborate on a project**. Using the Codex CLI on an Ubuntu VM, combined with thoughtful role assignment and a bit of orchestration glue, you can simulate an "AI pair programming team" or even a whole mini software team. By utilizing `AGENTS.md` profiles for guidance and a shared repository as the communication medium, each assistant contributes according to its archetype. This multi-agent approach aligns with how human teams tackle complex problems (divide and conquer by expertise), and it has been shown to yield more coherent and effective results on coding tasks [3]. Moreover, it's just fascinating to watch – you may start to see the agents surprise you with creative strategies or solutions (a taste of emergent intelligence).

Keep in mind that this is a cutting-edge, experimental practice. Expect some debugging of the process itself. But with careful setup, you'll have a powerful system where AI assistants "talk" to each other through code – **collaboratively evolving** your project at a pace and scale that mirror a highly efficient dev team, each member an archetype of a different strength. The end result is code that benefits from multiple perspectives and iterative refinement, potentially arriving at solutions more elegantly (and faster) than a single AI working in isolation. Good luck, and enjoy the emergence!

Sources: The concept of using `AGENTS.md` to steer Codex comes from OpenAI's documentation [14]. Multi-agent role specialization is supported by research like MAGIS (LLM agents for GitHub issues) [5] [3], MetaGPT's role-based "AI company" framework [6], and AgentVerse's findings on collaborative emergent behavior [1]. These show that coordinated agents can outperform single models and develop collaborative strategies beyond what we explicitly program – an exciting frontier for AI-assisted development.

---

[1] [4] [16] AgentVerse: Facilitating Multi-Agent Collaboration and Exploring Emergent Behaviors | OpenReview
https://openreview.net/forum?id=EHg5GDnyq1

[2] [3] [5] 2024-04-01: Coding Agents Tackle Github Issues
https://www.emergentbehavior.co/p/coding-agents-tackle-github-issues

[6] GitHub - FoundationAgents/MetaGPT: The Multi-Agent Framework: First AI Software Company, Towards Natural Language Programming
https://github.com/FoundationAgents/MetaGPT

[7] [8] [9] [10] [11] [12] OpenAI Codex CLI: an Open Source Coding Agent in the Terminal
https://apidog.com/blog/openai-codex-cli/

[13] [15] [17] [18] [19] [20] GitHub - openai/codex: Lightweight coding agent that runs in your terminal
https://github.com/openai/codex

[14] [21] [22] [23] [24] Introducing Codex | OpenAI
https://openai.com/index/introducing-codex/