ChatGPT

Introduction to Autonomous AI Agents and ChatGPT Agent Mode

Autonomous AI agents are AI systems (often powered by large language models, LLMs) that can take goal-driven initiatives: observing their environment, planning actions, and executing tasks with minimal human intervention. Instead of waiting for step-by-step prompts, these agents **decide and act autonomously** – for example by querying APIs, running code, or browsing the web – to fulfill a user-defined objective [1] [2]. Recent advances like OpenAI's *ChatGPT Agent Mode* have popularized this concept. Agent Mode (available in ChatGPT's premium tiers) essentially lets ChatGPT **"execute" requests autonomously**, planning multi-step solutions and using integrated tools (a virtual browser, code interpreter, file I/O, etc.) to deliver results while you oversee the process [3]. This bridges the gap between a traditional chatbot (which only responds when prompted) and an "AI assistant that *does* things" on your behalf [4].

Over the past year, an ecosystem of open-source projects and frameworks – inspired by ideas like ReAct (Reason+Act prompting) and the capabilities of ChatGPT's agent tools – has emerged to implement autonomous agents. Notable examples include **AutoGPT**, **BabyAGI**, **AgentGPT**, **OpenAgents**, **SuperAGI**, and others. These frameworks showcase best practices in prompt design, tool integration, memory management, and decision-making for building effective AI agents. In this guide, we'll explore the state-of-the-art prompting patterns, architectures, and strategies behind autonomous agents, referencing both popular GitHub projects and recent research. Each section provides structured insights – from prompt engineering and workflow design to tool use, memory, and real-world use cases – to help you understand and build AI agents in line with current best practices.

Prompt Engineering Best Practices (for Autonomous Agents)

Designing the right **prompt (or prompts)** is critical for autonomous agent performance. Unlike a simple Q&A prompt, an agent's prompt must establish *the agent's role, goals, constraints, available tools,* and the expected output format. Key best practices include:

- **Define the Agent's Identity and Objective:** Provide a clear *system or role prompt* that explains *who the AI is* and *what its goal is*. For example, AutoGPT prompts the model with a name, a role description, and a list of specific goals given by the user (e.g. "You are *Entrepreneur-GPT*, an AI designed to autonomously develop businesses to increase net worth" with goals X, Y, Z) [5]. This grounds the agent in a persona and purpose from the start.

- **List Constraints and Rules:** Successful agent prompts often include explicit constraints or guidelines the AI must follow. For instance, AutoGPT's prompt defines a set of *"Constraints"* (like the token limit, no self-harm, etc.) and *"Resources"* available (e.g. internet access, long-term memory, etc.) [6]. It also provides *"Performance Evaluation"* rules that instruct the AI to *continuously self-critique and improve its approach*, avoid unnecessary steps, and so on [7]. Clearly stating such rules helps the agent stay on track and make safer decisions.

- **Enumerate Available Tools/Actions:** The agent should be aware of what actions it can take. A common pattern is to enumerate **allowed commands or tools** in the prompt, each with a syntax and description [8] [9]. For example, AutoGPT presents a numbered list of commands like Google search, file operations, code execution, etc., with exact `"command_name"` and argument formats [10] [11]. By listing tools (or using OpenAI's function definitions in code), the agent knows how it can interact with the world. This reduces ambiguity – the model doesn't "invent" random actions, but chooses from provided options.

- **Structure the Agent's Reasoning (ReAct Prompting):** Research shows that prompting the model to explicitly think step-by-step before acting improves performance on complex tasks [12] [13]. Many agent frameworks adopt the **ReAct** style prompt, which means the model's responses are structured into *Thoughts* and *Actions*. For example, a prompt might say: *"When deciding what to do, first think (deliberate in natural language) about the task and possible steps, then output an action command."* In practice, the agent's answer is formatted as:

```
Thought: <agent's reasoning>
Action: <command name> <arguments>
```

(followed by an observation/result and then the loop continues) [14]. By separating *thoughts* from *actions*, the agent can reason internally ("browse for info on X because…") and then perform an action ("Google search X"). This ReAct framework encourages coherent multi-step reasoning and was shown to outperform prompting that skips the thinking step [15].

- **Specify Output Schema for Actions:** It's good practice to have the agent output its chosen action in a *consistent machine-readable format* (often JSON), so the controlling program can easily parse and execute it. For instance, AutoGPT's prompt explicitly instructs: *"You should only respond in JSON format as described… Ensure the response can be parsed by Python's json.loads."* and provides a schema for the JSON [16] [17]. The JSON contains a `"thoughts"` object (with fields like *text*, *reasoning*, *plan*, *criticism*, etc.) and a `"command"` object with the chosen action and args [18]. Requiring a well-structured output makes it much easier to automate the agent's execution loop.

- **Provide Examples or Few-Shot Guidance:** Some frameworks include an example of the format or even a short chain-of-thought demonstration in the prompt. If the agent will loop, providing an *"Example response"* can help. For example, the AutoGPT prompt template shows an example JSON with dummy thoughts and a sample command [19] [20]. This way, the model can mimic the style. In general, *few-shot prompting* (showing the model how an agent should think and act in a few instances) can prime it for better responses.

By combining these strategies – clear role and goal, constraints, a list of tools, step-by-step reasoning format, and a strict output schema – you set up the model to function as an autonomous agent. This **prompt engineering** effectively "programs" the agent's policy in natural language. High-performing prompts from projects like AutoGPT and BabyAGI follow this recipe, and can be adapted to new domains. Always remember to test and tweak prompts iteratively: small changes in wording or order (e.g. listing an important rule higher up) can significantly affect the agent's behavior.

Examples of High-Performing Prompts

To illustrate, let's examine a snippet of an actual agent prompt (from AutoGPT) and how it's structured:

- **Tool Definition Section:**

Example (AutoGPT):

```
Commands:
1. Google Search: "google", args: "input": "<search>"
2. Browse Website: "browse_website", args: "url": "<url>", "question": "<...>"
3. Start GPT Agent: "start_agent", args: "name": "<name>", "task": "<desc>",
"prompt": "<prompt>"
...
20. Task Complete (Shutdown): "task_complete", args: "reason": "<reason>"
```

In the above, the prompt has told the AI that it has 20 possible **commands**, from searching the web to managing files to executing code [21] [22] . Each is numbered and specified with the exact syntax. This means when the agent wants to search Google, it knows to output: `{"command": {"name": "google", "args": {"input": "my query"}}}`.

- **Resource and Self-Evaluation Section:**

Right after listing tools, AutoGPT's prompt lists **Resources** (e.g. internet access, long-term memory, etc.) and a **Performance Evaluation** guide [6] . The performance tips include: *"Continuously review and analyze your actions to ensure you are performing to the best of your abilities"*, *"Constructively self-criticize your big-picture behavior constantly"*, *"Reflect on past decisions to refine your approach"*, and *"Every command has a cost, so be efficient"* [7] . These act like an *inner conscience* for the agent, nudging it to be mindful and not go off-track. Notably, the agent is explicitly told to *self-criticize* and refine – this prompt technique is drawn from research (like the Reflexion method) which showed that agents that reflect on mistakes can improve their reasoning over iterations [23] [24] .

- **Response Format Section:**

Finally, the prompt defines how the AI should format its answer. In AutoGPT, the instruction is to output a JSON with a `"thoughts"` object and a `"command"` object [18] . The `"thoughts"` include subfields for *text (a raw thought)*, *reasoning*, *plan* (often a short bulleted plan of next steps), *criticism* (self-critique), and *speak* (a summarized thought to speak to the user) [18] . By explicitly telling the model that its *plan* should be a bulleted list in JSON, or that it should fill in a *criticism* field, the prompt ensures the agent will articulate those aspects. This prompt structure was *highly effective* – it leads the model to produce outputs like: a rationale, a multi-step plan, and then a chosen command, all neatly organized for the program to parse [19] [25] . Many "autonomous agent" prompts use similar JSON schemas or XML-like formats.

Overall, a **"high-performing" agent prompt is not a single sentence, but a carefully structured template**. It may span many lines of instructions, resembling a mini-manual for the AI. GitHub repositories often include these prompt templates – for instance, the full AutoGPT prompt template (in the

`prompt_settings.yaml` or code) and BabyAGI's task prompts – which can be a great starting point when building your own agent. By studying these, you'll notice how they balance giving the AI freedom to be creative with enough guidance to stay rational and goal-directed.

Agent Architecture and Workflow Patterns

While prompt engineering sets up *what the agent should do*, the **architecture** determines *how it operates over time*. Most autonomous agents follow an **iterative loop** pattern, where the model's outputs are fed back into the next step until the goal is reached. Let's break down some common workflow patterns, using AutoGPT and BabyAGI as exemplars:

- **Sense-Think-Act Loop (AutoGPT style):** AutoGPT uses a **cycle of "plan → act → observe → adapt"** [26] [27] . In each iteration:
- The agent (ChatGPT model) receives a **prompt context** that includes the latest state (including prior thoughts and results, plus relevant memory).
- The model outputs a **JSON** containing its *thoughts* (e.g. reasoning, plan, criticism) and a *command* to execute next [28] .
- The system **parses the command** and executes it in the environment. For example, if the command was `google` with some query, the system performs a web search; if it was `write_file`, it writes to disk. The execution yields a **result** (e.g. search results text or file write confirmation) [27] .
- The result is then **fed back to the agent**. AutoGPT combines the model's previous "thoughts" + the command result into a summary and stores it in memory [29] . This new information is also added (often in truncated form) to the next prompt.
- The agent's **memory system** comes into play: AutoGPT maintains a *short-term memory* (a window of recent interactions) and a *long-term memory* (a vector database of older facts) [30] [31] . Before each new cycle, the system retrieves any relevant long-term memories (via similarity search on embeddings) that might inform the next step. Those relevant snippets from past events are injected into the prompt as *context ("This reminds you of …")* [31] . This way, even if the conversation history was pruned due to token limits, important older info can resurface when needed.
- With the updated context (initial instructions + relevant past memories + last action's result), a **new prompt** is constructed and sent to the LLM for the next iteration [32] . The loop repeats. The agent continues generating thoughts and commands, refining its approach or trying alternative strategies as it goes – ideally guided by its own "criticism" field too.
- The loop ends only when a termination condition is met. In AutoGPT, the agent can explicitly issue a `task_complete` or `shutdown` command when it believes it has achieved the goal [33] . Alternatively, a human operator might stop a continuous agent, or a max iteration limit might be in place to prevent infinite loops.

Diagrammatically, this looks like: Agent "brain" (LLM) -> outputs plan+action -> executor performs action -> environment returns result -> result + memories -> back into LLM… and so on. The AutoGPT architecture has been a reference design for many subsequent agents [34] , demonstrating how to orchestrate LLM, tools, and memory in a loop.

- **Task List Loop (BabyAGI style):** BabyAGI takes a slightly different approach centered on managing a **dynamic task list** [35] [36] . The workflow:
- The user provides a high-level **objective** (e.g. "Find and summarize the latest research on battery technology").

- The system initializes a **task list** with an initial task (often simply to analyze or research the objective).
- **Task Execution:** The agent picks the highest-priority task from the list and executes it using the LLM (with relevant context). For example, Task 1 might be "Search for battery tech research papers" which the agent executes by using a search tool and summarizing results. [37]
- The result of Task 1 is stored (and also saved into a vector store memory). **Task Creation:** Based on that result, the agent (or a dedicated "task creation" sub-agent) generates new follow-up tasks that would move closer to the objective [37]. For instance, after finding some papers, it might create a new task "Extract key findings from paper X".
- **Task Prioritization:** Another component (a "prioritization" agent) reorders the task list according to urgency or logical sequence towards the goal [38]. It ensures the new tasks are placed appropriately (and redundant or done tasks removed).
- Loop continues: The next task in priority is executed, leading to new tasks and re-prioritization, etc. [37].
- Stopping: The loop ends when the task list is empty or a predefined end-condition is reached (e.g., a certain number of iterations or the objective is achieved) [39] [40].

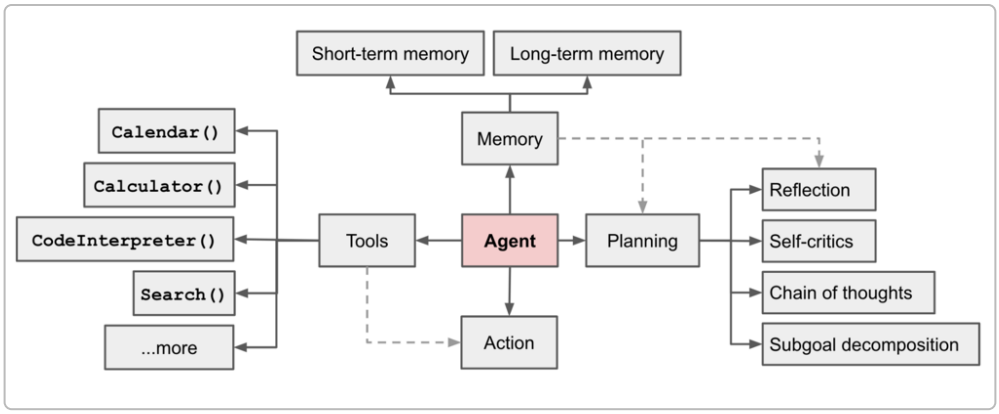
Internally, BabyAGI is implemented with multiple prompts/agents for each stage (execution, creation, prioritization), but it's conceptually simple: a to-do list that keeps updating itself [41] [42]. The use of a vector database is key for memory – each task's result is embedded and stored, so when executing a new task, the agent can pull in relevant snippets from *previous tasks' outputs* to inform the current step [43]. This helps maintain context across long sequences. The original BabyAGI was a compact script using GPT-4, Pinecone (for embeddings), and LangChain to tie it together [44], which made it popular as a learning tool for agent developers.

- **Multi-Agent Collaboration:** Another architectural pattern is to use **multiple LLM agents with different roles** to tackle a problem collaboratively. For example, *MetaGPT* and *CAMEL* are frameworks where you might have an "Engineer AI" and a "Manager AI" (and perhaps more specialist AIs) that communicate with each other to plan and solve tasks [45]. One agent may be assigned to generate ideas or plans, another to criticize or evaluate, etc. They pass messages back and forth (sometimes with a coordinator ensuring they stay on task). This approach can harness specialization – e.g., one agent might be prompted to be a creative brainstormer, another to be a logical critic. OpenAI's function-calling and role system can facilitate this by letting one agent's output feed into another. While multi-agent architectures are still experimental, they have shown promise in complex tasks requiring diverse skills or self-checking (two agents double-checking each other's outputs). Notably, AutoGPT itself had a rudimentary ability to spawn new agents ("start_agent" command) for subtasks [46], though this wasn't commonly used. More explicitly, projects like **CAMEL** had two ChatGPTs (user and assistant roles) working together to write code, and **MetaGPT** created a virtual "software team" of agents (PM, architect, coder, tester) cooperating on a programming project. These multi-agent setups remain an area of active research and development [47], aiming to overcome some limitations of a single agent by introducing dialogue and oversight between agents.

- **Self-Reflection and Retry Loops:** A final pattern worth noting is augmenting the main loop with a **reflection or critic step**. In some frameworks, after a few iterations or after a subtask, the agent is prompted to **reflect**: "What went wrong? What could be improved?" This is akin to an inner loop where the agent reviews its own performance from a meta-level. The *Reflexion* paper introduced this idea of the agent dynamically generating new goals or adjusting its strategy based on past mistakes [23] [24]. Practically, an implementation might detect if the agent is stuck in a loop or failing, and

then have the LLM produce a self-critique and a revised approach (or even chain a second "critic model" that provides feedback to the "actor" model). AutoGPT's prompt already bakes some self-reflection into each step via the "criticism" field [18], but a dedicated reflection phase could be triggered on error. Indeed, AutoGPT *does* include an error handling mechanism: if a tool execution fails or returns an unexpected result, the agent notes the error and revises its plan, a sort of *implicit reflection*. The framework also has a *continuous mode* versus *step-by-step mode* – in continuous mode it will loop by itself, whereas in step mode it will ask the user for permission at each iteration (allowing a human to intervene if it's going astray). In summary, robust agent workflows often incorporate **feedback loops** – either internal (the AI checking itself) or external (human oversight) – to iteratively improve outcomes and avoid failure traps [48].

In practice, many real-world agent systems combine elements of the above patterns. For instance, an agent might maintain a task list (à la BabyAGI) *and* use a ReAct-style loop for each task execution, with some self-reflection if the task fails. The architecture you choose depends on the use case: a research assistant might benefit from creating sub-tasks and searching the web, whereas an coding agent might work best with two LLMs chatting (e.g. one writes code, another reviews). Regardless of pattern, a recurring theme is **orchestration** – i.e. the logic outside the LLM that coordinates prompts, tools, and data flows. Projects like **LangChain** and **Microsoft's AutoGen** provide higher-level abstractions to manage this orchestration, so developers can focus on defining the building blocks (prompts, tools, memory stores) and let the framework handle the loop mechanics.



Overview of a typical LLM-powered autonomous agent system [49] [50]. The LLM (center) serves as the "brain" for planning and decision-making. It interacts with external tools/APIs for enhanced capabilities (left) and uses memory modules (right) to recall past information. Planning components (top) enable task decomposition and self-critique, helping the agent to refine its approach. This diagram highlights how modules like planning, memory, and tool use extend an LLM into a full autonomous agent.

Tool Use, API Integration, and Memory Management

Two of the most critical aspects of an autonomous agent's implementation are **tool use** (integrating external APIs or functions) and **memory management** (handling information beyond the model's context window). Best practices in these areas greatly impact an agent's effectiveness.

Tool Use and API Integration

Equipping agents with tools allows them to overcome the inherent limits of an LLM's knowledge (which might be outdated or static) and capabilities. Tools can include web browsers, search engines, code execution environments, calculators, databases, other AI models, and more. Key points include:

- **Defining Tools Clearly:** As discussed in the prompting section, an agent is usually given a catalog of tools/commands it can invoke. Internally, each tool corresponds to a function or API call in the agent's code. For reliability, it's important to constrain the agent to *only use those tools* and not attempt unauthorized actions. OpenAI's **function calling** feature (available in the ChatGPT API) is a powerful way to enforce this – you define functions (e.g. `search(query)` or `send_email(to, body)`) and the model will output a JSON with a function name & arguments when it decides to use one. This ensures the tool usage is in a parseable format and prevents the model from executing arbitrary text as code. Many modern frameworks support this out of the box.

- **ReAct vs Direct Tool Use:** Earlier agent implementations (AutoGPT, etc.) used the ReAct style where the model outputs an action name in text which is parsed. Newer implementations may use the function calling interface (the principle is the same; the difference is formatting). In both cases, the design pattern is: **LLM outputs an action → some middleware catches it and invokes the corresponding tool → result returned to LLM**. For example, in AutoGPT, if the model outputs the `browse_website` command with a URL, the Python code will perform the HTTP request and then feed the text content back to the model on the next prompt cycle [27]. In OpenAgents' *WebAgent*, a similar approach is used to let the agent navigate web pages autonomously [51] [52].

- **Safety and Permissions:** Tool use can be risky if not sandboxed. Best practice is to **limit the agent's authority** – e.g. filesystem writes only in a specific directory, web access through an allow-listed domain list, and absolutely **no** direct access to dangerous actions (like managing real finances or controlling physical hardware) without explicit user approval. ChatGPT Agent Mode, for instance, will refuse high-risk actions and requires user confirmation for certain operations [53]. When building your own agent, consider implementing a confirmation step or a *"are you sure?"* if the agent tries something potentially destructive (deleting files, spending money, etc.). Some frameworks include a "pause and ask user" capability for this reason.

- **Rich Tool Ecosystems:** Projects like **SuperAGI** and **LangChain** maintain collections of ready-made tool integrations. SuperAGI provides a *Toolkits marketplace*, offering plugins for Twitter, file management, web scraping, Google Calendar, Jira, email, and more [54]. LangChain similarly has many "tools" (like `SerpAPI` search, Wolfram Alpha, PDF loaders, etc.) that can be plugged into an agent. Rather than reinventing the wheel, it's wise to leverage these libraries: they handle API quirks and rate limits, and you can simply invoke them from your agent. For example, if your agent needs internet search, you might use an existing Google search tool wrapper; if it needs to generate images, you could integrate with DALL·E or Stable Diffusion via a plugin. **Integration testing** is important: check that the agent knows how to use the tool correctly. Sometimes giving a usage example in the prompt (one of the few-shot examples) helps the model call the tool with proper arguments.

- **HuggingGPT and Model-as-Tool:** One interesting angle of tool use is using other AI models as tools. The paper *HuggingGPT* (2023) demonstrated an agent that orchestrated multiple AI models on

HuggingFace to solve multi-modal tasks – ChatGPT would decide which specialized model (for vision, audio, etc.) to call and then integrate the results. In practice, this means your agent can have tools like "image_caption(image_url)" which behind the scenes calls a vision model, etc. If building an agent that needs multi-modal understanding or domain-specific ML (say, a medical diagnostic model), consider this model-as-tool approach. The LLM can be the planner that routes sub-tasks to the appropriate model or API.

In summary, effective tool use comes down to **defining a robust interface** between the agent and external functions. Keep the set of tools well-scoped and documented in the prompt, use structured calling formats (to avoid misinterpretation), and guard potentially unsafe operations. With these in place, an autonomous agent can extend far beyond its base LLM abilities – browsing current information, executing code to do math or database queries, controlling third-party services, and so on. This is how agents "act" on the world rather than just chatting.

Memory Management

Memory is the other pillar of autonomy. A standalone LLM has a fixed context window (e.g., 4K or 16K tokens for GPT-4), meaning it can only "remember" that many tokens of conversation. Agents, however, may need to work on tasks that exceed this limit or persist information across sessions. Best practices for memory include:

- **Short-term vs Long-term Memory:** As introduced earlier, many agent architectures divide memory into *short-term* (STM) and *long-term* (LTM). STM is often just the recent interaction history that's kept in the prompt. Long-term memory is an external store (database or file) that can hold information indefinitely. AutoGPT's design, for example, stores the full history of thoughts and results, but only includes the most recent few exchanges in the prompt (to avoid token overflow) [55]. The rest goes into LTM: a vector store indexed by embeddings [56]. When needed, it fetches the top relevant pieces from LTM and injects them as context [57]. This pattern (vector database + similarity search) is widely used because it mimics how a human might only actively recall pertinent memories rather than everything at once.

- **Vector Databases and Embeddings:** The de facto solution for long-term memory in LLM agents is to use embeddings (vector representations of text) to enable semantic search. Tools like **FAISS, Pinecone, Weaviate, Chroma** etc. can store thousands of text chunks (each maybe a summary of an event or a piece of reference data). When the agent needs to recall something, it embeds the query (e.g. the current goal or a reflection of what it needs) and finds similar vectors = related past snippets [58] [57]. For instance, BabyAGI uses Pinecone to store results of each task, so that when a new task is executed, the agent can query Pinecone with the task description to get any past results that might help [43]. This greatly improves coherence in multi-step tasks – the agent isn't relying solely on the conversation tokens to "remember" earlier outcomes; it has an external memory of potentially unlimited size.

- **State Persistence:** For agents that run continuously or need to maintain state between runs, it's important to persist memory to disk or a database. AutoGPT, for example, can save the vector store to a local file. SuperAGI supports multiple vector DBs and likely a way to persist agent memory across sessions [59]. If you shut down an agent and restart it later, you'd want it to load its memory

so it can pick up context. This is more a systems design detail but critical for real applications (you don't want an agent forgetting everything if the process restarts).

- **Summarization for Compression:** Besides embedding-based retrieval, another technique is **summarizing old context** to free up space. An agent might periodically take older dialogue or notes and summarize them into a concise form, and either store that summary in memory or even replace the raw text in the prompt with the summary. This is common in long chat sessions with GPT, but in autonomous agents it can be automated. For example, if an agent has done 10 web searches and gathered a lot of info, it might summarize the combined findings into a paragraph and use that going forward, instead of carrying all 10 raw results (which could be many tokens). The challenge is to ensure the summary doesn't omit critical details – a hybrid approach might keep a summary plus a vector DB of the details if needed later.

- **Forgetfulness and Refreshing:** Interestingly, sometimes it's useful for an agent to *drop* irrelevant memory to avoid confusion. If the agent changed sub-goals, carrying too much stale info can lead to tangents (or even trigger the model's tendency to hallucinate connections). Best practice is to retrieve only the most relevant pieces for the current context [32], not everything. Some frameworks limit long-term memory fetches to e.g. top 5 or top 10 items [57]. Tuning this number is a trade-off: too few and the agent might miss something important, too many and it might waste tokens or reintroduce solved issues. Monitoring an agent's performance with different memory settings is advisable.

- **Memory Index Maintenance:** As the agent works, its memory store grows. It might be worthwhile to occasionally re-index or clean the memory. For example, if the agent has stored every search result it's ever seen, the memory might become noisy. One could implement a heuristic to prune memories that are not useful (maybe those that were never retrieved or very low relevance, or superseded by later results). This is an area that isn't fully solved generally – it often requires domain-specific insight to know what knowledge is no longer needed. Some research explores using the LLM itself to decide what to keep or forget ("memory management via rehearsal"). For now, a pragmatic approach is to set a cap on memory size or age (like only keep the last N items, or last 24 hours of data for a long-running agent) or categorize memories by topic to focus retrieval.

In practice, integrating memory might involve using libraries like **LangChain** or **LlamaIndex (GPT Index)** which provide seamless connectors to vector stores and handle chunking of texts into embeddings. These libraries can abstract a lot of the boilerplate: e.g., you can ask LangChain's memory to `.save_context()` and `.load_memory_variables()`, and behind the scenes it's updating a vector index. The IBM/Watsonx toolkit also emphasizes vector DB integration as a core part of agent orchestration [44].

To summarize, **tool use and memory are what give an agent "agency" beyond a single chat exchange**. Tools let it act on the world; memory lets it accumulate knowledge over time. Combining both effectively is crucial for complex tasks. A failure in tool integration might cause an agent to hallucinate an answer it should have Googled, and a failure in memory might cause it to repeat itself or contradict earlier findings. The best implementations carefully test these components – e.g., verifying that each tool invocation yields the expected observation and that important info is indeed retrievable from memory when needed.

Use Cases and Operational Modes

Autonomous AI agents have been applied (experimentally or in production pilots) to a wide range of tasks. Here we highlight some **common use cases** and how agents are configured (operational modes) to suit them:

- **Internet Research and Reporting:** One popular use case is for an agent to act as a *research analyst*. Given a broad query (e.g. "Analyze the competition in our market" or "Summarize recent developments in quantum computing"), the agent can autonomously search the web, find relevant articles/papers, and synthesize a report. It will break the objective into sub-tasks: search for X, summarize findings, maybe compile into a document. Projects like **AgentGPT** were often used for such demo scenarios. For instance, you could instruct AgentGPT: "Create a comprehensive report on the Nike company," and it will generate a plan, do web searches, gather facts, and produce a summary [60]. Agents shine here because they can follow references recursively – finding one article, then searching something mentioned in it, and so on, without the user's involvement at each step. The output might be a written report or a set of slides (ChatGPT Agent Mode can even directly draft slides or Excel sheets now [3]). However, these research agents require careful prompt scoping to avoid going off-topic, and usually benefit from a final human review due to potential inaccuracies.

- **Business Process Automation:** Companies are exploring agents to automate multi-step business workflows. Examples: an agent that **generates weekly marketing content**, schedules posts, monitors engagement, and compiles analytics. In an Agent Mode example, an agent was tasked with handling a restaurant's weekly marketing: every Monday it proposes themed promotions and drafts social media posts, Tuesday it posts them and monitors feedback, Friday it generates an analytics report and next week's plan – all the owner needs to do is approve the outputs [61] [62]. Similarly, agents can assist with HR (screen resumes and schedule interviews), customer support (automatically handle common inquiries and escalate when needed), and finance (pull data, generate a summary report). The **operational mode** in these scenarios is often *periodic or event-driven*: the agent might wake up on a schedule (daily/weekly) or in response to a trigger (new email or ticket) and then carry out its sequence. Reliability and integration are key – for example, connecting the agent to company databases via secure APIs, and ensuring it doesn't send out communications without approval. In many cases, these agents run in a **"proposed action, human approval" loop**, meaning they prepare outputs (draft emails, draft social posts) but a human presses send. This mitigates the risk while still saving significant time.

- **Software Development (AI Coding Assistants):** Another prominent use case is using autonomous agents in programming. Beyond code-completion (like GitHub Copilot), autonomous agents can take on higher-level tasks: *given a feature request, they can plan the implementation, write code, test it, and even debug.* **AutoGPT** early on demonstrated coding abilities – e.g. it could attempt to write its own code files, run Python scripts to test them, and fix errors iteratively. Specialized variants like **GPT-Engineer** focus on this area: you feed in a spec or prompt, and the agent will generate an entire codebase through a structured plan (writing a project structure, creating files, etc.). In such scenarios, tools like file I/O and code execution are heavily used. The agent might create a file, then run a test, see the output (error or success), then decide to modify code. This is essentially an automated *write-run-debug loop*. Some frameworks use multiple agents here: e.g. one agent writes code, another reviews and suggests improvements (simulating a pair programming scenario). This is the idea behind Microsoft's *Jarvis/Code Interpreter* and related research where the LLM's code output

is executed and results (stack traces, etc.) are fed back in [63]. These agents have to be constrained to not do anything harmful on the system (run code in a sandbox, limit network access, etc.), but they open up the possibility of **hands-free coding** for routine tasks or boilerplate.

- **Data Analysis and AutoML:** Agents also see use in data-related tasks. For example, **OpenAgents' Data Agent** is designed for *data analysis with Python/SQL*: it can autonomously load data, run analysis code, and generate charts or summaries [51] [64]. This could be useful for an analyst who asks, "Agent, analyze our sales data and show key trends," and the agent will write a Python script, execute it, maybe use a plotting library to create graphs, and return insights. In AutoGPT and others, there are often commands to **execute Python code** or query databases, enabling such use cases. Similarly, an agent could perform AutoML (automated machine learning) – e.g., try different models on a dataset to find the best one – by generating code and evaluating results in a loop. These use cases require robust tool integration (for Python, one might use a Jupyter or sandbox environment) and often rely on *long running sessions* since analysis can be time-consuming. ChatGPT Agent Mode's ability to "run a pseudocode on a virtual machine" is a step in this direction [3].

- **Personal Assistants and Agents as a Service:** On the more everyday end, autonomous agents can act as personal digital assistants. Think of an agent that can **handle your emails** (draft responses, sort them), manage your calendar, or even perform shopping tasks (compare products, make a purchase within set limits). For instance, an agent could be told: "Book me a cheapest flight to London next month and email me the itinerary." It would then search flights, perhaps use an API like Skyscanner, pick an option, and even attempt to book it if given payment details. OpenAI's vision is that Agent Mode can function somewhat like this for professionals – automating routine digital chores [65] [66]. Currently, due to trust and safety, fully autonomous personal agents are used cautiously (nobody wants an AI accidentally ordering 100 items from Amazon!). But with constrained domains (like triaging emails or setting up meetings), they can be quite effective. These agents typically operate in **interactive modes**, where they do one task at a time upon user request, but internally they may perform multiple steps to fulfill that request.

- **Continuous vs On-Demand Operation: Operational modes** can be categorized as:

- *On-Demand (Single Objective)* – The user gives an objective and the agent runs through to completion then stops. AgentGPT in the browser works this way: you hit Deploy with a goal, it works step by step and either achieves it or exhausts its attempts. Most of these sessions are ephemeral; memory is not saved afterward (unless you copy it).
- *Continuous Service* – The agent runs as a persistent service (maybe on a server or as a local daemon) and can handle multiple tasks or react to triggers over time. AutoGPT can be run in continuous mode, effectively looping indefinitely until stopped. SuperAGI is geared towards managing *multiple concurrent agents* running as background processes [67], which is useful in a production setting where you might spawn an agent per task or user.
- *Human-in-the-Loop Mode* – The agent pauses for confirmation at key steps. This is often the default for safety. For example, AutoGPT will ask the user "continue (y/n)?" after each action when not in continuous mode. Similarly, a business might run an agent that always sends draft outputs to a human manager for approval before actually executing an irreversible action (like sending an email or transferring money). This mode is slower but much safer, and in many cases the oversight can catch mistakes.

- *Evaluation/Dry-Run Mode* – Sometimes you want the agent to plan and reason but *not* actually execute external actions (or maybe execute them against a test stub). This can be used to evaluate what the agent *would do* without risk. For instance, you might run an agent on a planning task with all tools in a dummy mode that just logs the intended actions. This is useful for debugging prompts and logic before connecting real APIs.

Selecting the right mode depends on the use case's tolerance for errors and need for speed. A spectrum is emerging where fully autonomous, **continuous agents** are on one end (high automation, higher risk) and tightly controlled, **on-demand agents** with frequent human checks are on the other (slower, but reliable). Many current best practices lean toward the latter for practical deployments – use autonomy to draft and do the heavy lifting, but keep a human in final control especially in high-stakes domains [68].

Key Frameworks and Libraries

The autonomous agent landscape is rich and rapidly evolving. Here we highlight some of the **influential frameworks, libraries, and projects** (mostly open-source on GitHub) that embody the strategies we've discussed:

- **AutoGPT** – *GitHub:* 【9†】. The original spark of mainstream autonomous agents, AutoGPT by Significant-Gravitas is a Python application that given a goal will spin up a GPT-4 agent to work continuously on it. It introduced the idea of the agent maintaining **short and long-term memory, and a suite of commands to interact with the world** [30] [69]. AutoGPT is highly extensible – you can add new plugins/commands, swap out the vector store or model provider, etc. – and it's **modular** (recent versions split into a "core" and "platform" for UI). It's best for developers who want fine control and customization of an agent's behavior. With ~180k stars on GitHub, AutoGPT inspired countless derivatives. It's particularly suited for complex, long-running tasks where the ability to recall past context and use many tools is needed [70]. However, it can be resource-intensive (lots of API calls) and sometimes gets stuck without human feedback (so use continuous mode with caution).

- **BabyAGI** – *GitHub:* 【22†】 (the original by Yohei Nakajima, plus many forks). BabyAGI is a **lightweight task management agent**. Its focus is on generating, prioritizing, and executing tasks in a loop to meet an objective [35] [37]. Compared to AutoGPT, BabyAGI's codebase is much simpler (a single file in early versions) and it delegates a lot to **LangChain** for managing prompts and tools [44]. It's great for education and quick prototyping of the task loop concept. Because it's minimal, it may not have as many built-in tools or safety features as AutoGPT, but it's easier to understand and modify. Use BabyAGI if you have a straightforward objective that can be broken into subtasks and you want an agent to iteratively work through them. Many variants exist (e.g. *BabyBeeAGI* which added web browsing, etc.), and IBM's Watsonx has even integrated the concept into their docs [71].

- **AgentGPT (Reworkd)** – *GitHub:* 【14†】. A browser-based agent platform that became famous through an easy web UI. AgentGPT lets you configure an agent's name and goal in your browser and then watches it think and act in real-time [72]. Under the hood it uses **LangChain** and likely functions similarly to AutoGPT, but with an emphasis on *usability* (no coding required to spin up an agent) [73]. It uses a Next.js frontend and a FastAPI backend, making it a full-stack web app [74] [75]. AgentGPT is ideal for demonstrations, quick experiments, or non-developers who want to try autonomous agents (you just plug in your API keys). The trade-off is less customization than something like AutoGPT –

you rely on the preset prompt logic and tools. It's more of a UI layer on existing agent logic. Still, it played a huge role in popularizing autonomous agents by making them accessible in a browser. One can learn from its project how to integrate an agent loop into a web interface and manage state over a client-server setup.

- **OpenAgents (XLang)** – *GitHub:* 【16†】. OpenAgents is an ambitious platform emerging from academic work (see their COLM 2024 paper) to provide an **open-source alternative to ChatGPT's Agent functionalities**. It includes multiple specialized agents:

- *Web Agent* for autonomous web browsing (similar to ChatGPT's browser plugin) [51] ,
- *Data Agent* for data analysis (with tools for coding, visualization) [51] [64] ,
- *Plugins Agent* that has access to 200+ plugins (it essentially can use a wide range of tools) [51] .

OpenAgents provides a unified chat-based web UI where users can interact with these agents, and it emphasizes **real-world usage** and evaluation [76] [77] . It's a full-stack project (with backend, frontend, deployment scripts) and aims to be easy to self-host. If you're looking for a pre-built system that closely mirrors ChatGPT Agent Mode's capabilities, OpenAgents is a top contender – it literally markets itself as *"control your browser as ChatGPT Plus, but with open code"* [78] . Developers can study it to see how to integrate a wide plugin ecosystem and handle the UI/UX of running agents for general users.

- **SuperAGI** – *GitHub:* 【18†】. SuperAGI brands itself as a "dev-first autonomous AI agent framework" focused on *production readiness*. It provides a robust infrastructure to **create, run, and monitor agents at scale** [67] . Key features include:
- Concurrently running multiple agents (with separate goals) – useful for enterprise scenarios [67] .
- A **Graphical User Interface** to manage agents and visualize their workflows [79] .
- A **Marketplace of Toolkits**, so you can easily plug in new capabilities (integration with various services is a highlight) [80] [54] .
- Built-in **Memory storage options** (multiple vector DB support) and token usage optimization controls [81] .
- Telemetry and logging to analyze agent performance [81] .

In essence, SuperAGI is addressing the needs of developers who want to deploy agents reliably and possibly serve them to end-users. It abstracts a lot of complexity – e.g., adding a new tool might be as simple as installing a toolkit package. The project is under active development (it has a cloud offering as well). If your aim is to integrate an autonomous agent in a production app or workflow, SuperAGI is worth a look for its focus on maintainability and scaling. It uses a ReAct-style agent by default and you can write custom logic if needed [82] .

- **LangChain** – *GitHub:* [langchain-ai/langchain]. Not an "agent" per se, but **the premier library** for building LLM applications, including agents. LangChain provides abstractions for:
- LLMs and chat models,
- Tools (it has a standard `Agent` class that can take a list of tools and a LLM and run a ReAct loop),
- Memory (various memory classes to store conversation or vector stores),
- Chains (sequences of prompts/actions).

Many projects (including AgentGPT, BabyAGI variants) internally use LangChain's agent tooling [83] . The reason is it saves time – instead of coding the loop and parsing logic from scratch, you can configure a LangChain Agent (e.g. an `AgentType.OPENAI_FUNCTIONS` agent which uses function calling). LangChain

also offers integration with **evaluation** modules, which can help in testing your agent's outputs. It's widely used and has a large community, which means lots of examples and existing code. If you want to create a custom agent with custom tools and memory, LangChain is a strong choice to accelerate development [84]. Just be mindful that it's a quickly evolving project; ensure version compatibility, and be prepared to dive into docs due to its breadth.

- **Microsoft AutoGen / AgentFramework** – *GitHub:* [85]. This is an open-source framework from Microsoft Research for **multi-agent conversations and workflows**. AutoGen allows you to define multiple agent roles (with their own prompts and tool sets) and orchestrate a conversation between them. It's quite flexible – you can set up an Agent as a "Controller" which can spawn other sub-agents or route tasks. If you are interested in multi-agent systems or integrating agents into a larger Python application, AutoGen provides a structured way to do it (including things like agent factories, message routing, etc.). Microsoft also introduced a .NET based *AgentFramework*, but the Python AutoGen library is likely easier if you're already using Python. It hasn't reached the popularity of LangChain, but it's backed by research (some of the papers presented at NeurIPS 2023 were built on AutoGen). For instance, the *CAMEL* two-agent coding scenario can be implemented with AutoGen pretty straightforwardly.

- **Others:** A few more notable mentions –

- **Hugging Face Transformers Agents:** HuggingFace introduced a concept where you can use an LLM (like StarCoder or GPT-3.5) to control HF's tools (which include other models, e.g. image classifiers, etc.). It's similar to HuggingGPT's approach. This is more experimental but shows the potential of hooking into the vast ML model ecosystem via an agent.
- **GPT-Engineer:** A specialized flow for generating software projects. It's worth noting because it uses a multi-step prompting approach (outline, implement, refine) which could be generalized to other tasks. It's a bit less interactive (it's more one-shot for a given spec), but its prompting strategy can be insightful.
- **MetaGPT:** Mentioned earlier, it's a framework that formulates a "virtual team" of agents to collaboratively build a product (for example, writing code from a spec). It gained a lot of GitHub stars. If your interest is in multi-agent collaboration for complex creative tasks, checking out MetaGPT's prompt structure (how it defines roles like Product Manager, etc.) could be inspiring.
- **Semantic Kernel (Microsoft):** A SDK that can combine LLM skills (functions) and memory, and allows switching between different AI models. It's more of a programming model than an autonomous agent solution, but you can implement agent loops with it. It's relevant if you work in .NET/C# especially.

The field is moving quickly – for instance, OpenAI's own platform might integrate more agent-like capabilities (tools and memory) into its API. We already see that with function-calling and the concept of *"GPTs"* (GPT-4 powered custom agents that OpenAI has talked about for the future). The good news is that many of these frameworks are converging on similar best practices (ReAct prompting, vector memory, function tools), so skills learned on one are transferable to others.

Summary of Key Learnings and Recommendations

Autonomous AI agents represent a significant step toward AI systems that **not only chat, but also act**. By studying projects like AutoGPT, BabyAGI, and others, we've distilled several best practices:

- **Robust Prompt Structures are Essential:** An agent is only as good as its instructions. Successful agents use structured prompts that set clear roles, goals, constraints, and tool interfaces for the model [6] [18]. Investing time in prompt engineering (and iteratively refining prompts based on the agent's output) yields more reliable behavior. Techniques like chain-of-thought prompting (think step-by-step), ReAct formatting, and including self-critique guidance have been empirically shown to improve reasoning [12] [13]. When designing your agent, **simulate its prompt in isolation** with a known test case to see if the format produces the kind of thought/action output you expect.

- **Agent Orchestration & Frameworks:** Don't reinvent the wheel on loop orchestration and parsing if you don't have to. Libraries like LangChain and SuperAGI provide battle-tested patterns for running the thought-action loop, integrating memory, and handling errors. Use these to your advantage. They also offer a lot of **extensibility** (e.g., easy tool integration via a plugin or class) so you can focus on the unique logic of your agent, not the boilerplate. That said, be mindful of the abstraction – when something goes wrong, it helps to understand the underlying process (as we described in the architecture section). It's often useful to log the full prompts and model outputs at each step when debugging an agent [68].

- **Tool Selection and Safety:** Equip your agent with only the tools it truly needs for the task. Each additional tool increases complexity and risk of errant behavior. If your agent's job is, say, to analyze PDF reports and output a summary, it might only need a file-reading tool and perhaps web search for references. It likely doesn't need the ability to execute arbitrary code or send emails – so don't include those. **Least-privilege principle** applies: give the agent just enough access to do its job, and no more. Always consider the implications of a tool – for instance, a "shell command" tool is dangerous, whereas a "run Python (in sandbox)" tool is somewhat safer but still can do a lot. If using function-calling, you can implement server-side checks (e.g., if agent tries to call `send_email` to an unapproved address, block it).

- **Memory Management and Context:** Long-term memory via embeddings is powerful, but ensure the retrieved information stays relevant. Irrelevant memory can confuse the model (leading it to make connections that don't exist). It's good practice to prepend any long-term memory context with a system message like "Previously you learned these facts: ..." or otherwise delineate it, so the model knows it's recall. Also, regularly monitor the memory store for quality – you might find the agent saving trivial or redundant info. In a dynamic environment, *updating* memory is as important as storing – e.g., if an agent's task is updated, old goals in memory should perhaps be marked obsolete. This is an ongoing area of development; tools like LlamaIndex offer features for context management that could be explored.

- **Iterative Development and Testing:** Treat building an AI agent like building any complex software system. Start with small goals and verify the agent's behavior before scaling up. Use unit tests or simulations for critical parts – for example, test that the agent properly invokes each tool given a scenario (you can simulate tool responses to the LLM). Monitor the agent's loop in real time when first running it on a new objective; if it's going in circles or doing something unexpected, you may

need to adjust prompts or add a new rule. The community often shares failures and lessons (like the notorious example of *ChaosGPT*, an AutoGPT fork given a destructive goal – which thankfully mostly just tweeted nonsense). Learn from these – for instance, one common pitfall is **infinite loops** where an agent keeps thinking it needs to redo an action. Setting a reasonable iteration limit or incorporating a reflection after N steps can mitigate this [86].

- **Cost and Efficiency:** Autonomous agents can rack up API calls (each thought is a new call to an LLM). Be mindful of this. Use cheaper models (GPT-3.5 or local models) for less critical thinking steps if possible, and reserve GPT-4 for complex reasoning. Some frameworks allow switching models on the fly (e.g., use GPT-3.5 in the loop, then maybe validate the final answer with GPT-4). Batch operations when you can: if the agent needs to fetch multiple pieces of info, see if you can combine queries. AutoGPT documentation suggests strategies like throttling calls, using shorter prompts, etc., to control costs [87]. Also consider rate limits – if your agent might hit external APIs rapidly, build in pauses or handling of rate limit responses.

- **Human Oversight and Ethical Considerations: Autonomy is not a license to ignore oversight.** Always have a way to audit what the agent did: keep logs of actions, results, and AI decisions [68]. This is especially vital if the agent operates in an environment with real-world impact (financial, customer data, etc.). It's recommended to have a "manual override" – either a person or a monitoring system that can intervene or shut down the agent if it goes off the rails. OpenAI's Agent Mode, for example, emphasizes that it's *"not an autonomous employee"* and requires user supervision for now [88] [53]. From an ethics standpoint, ensure the agent is compliant with privacy and security norms – e.g., if it uses company data, it should not divulge it externally, and if it interacts with users, it should not pretend to be human or do things against user consent. These principles might need to be baked into the prompt as rules ("If the user asks for personal data, refuse") and into the system design (don't give the agent tools that can break privacy, like unrestricted web access to internal databases without checks).

- **Stay Updated with Research:** The field of agentic AI is evolving monthly. Techniques like **Tree-of-Thoughts** (which tries multiple reasoning paths and then selects the best) [89], **Self-Ask** (where the model explicitly asks itself follow-up questions), and advanced planning algorithms are being explored. New papers (some referenced in the NeurIPS 2023 list [90] [91]) are proposing ways to make agents more robust (e.g. better long-horizon planning, avoiding getting stuck, verifying their answers). Keeping an eye on academic and blog publications (like Lilian Weng's *LLM as Agents* blog [92] or other survey papers [93] [94]) can give you ideas to improve your agents. For instance, a recent idea is to use a **"fast" LLM and a "slow" LLM** in tandem (the SwiftSage approach) – the fast one quickly explores options, the slow one deliberates more deeply on promising ones [95]. Such ideas might soon trickle into open-source frameworks.

In conclusion, building autonomous AI agents is a multi-disciplinary craft: part prompt engineering, part software engineering, part UX design, and part safety management. The projects and practices we've discussed provide a **foundation of knowledge** for creating agents that are effective and (hopefully) reliable. With ChatGPT's Agent Mode bringing these concepts to a wider audience, understanding how to orchestrate AI reasoning, tools, and memory is increasingly valuable. By following best practices – clear prompts, structured workflows, prudent tool use, solid memory, and oversight – you can harness the power of autonomous agents in your own applications while mitigating the risks.

As always, **start small, think big**: experiment with toy objectives in a safe setting, learn from each agent run, and gradually scale up the autonomy as your confidence in the agent's behavior grows. Happy building, and may your agents be ever helpful (and only as autonomous as you want them to be)!

Sources:

- Weng, Lilian. "LLM Powered Autonomous Agents." *Lil'Log* (June 2023) – Overview of agent components (planning, memory, tool use) [96] [97] and advanced prompting techniques like ReAct [13] and Reflexion [24].
- Sung, George. "AI Agents: AutoGPT Architecture & Breakdown." *Medium* (Apr 2023) – Step-by-step explanation of AutoGPT's workflow [98] [30], memory mechanism [69], and prompt structure with commands [8].
- OpenAI Developer Community – "Dissecting Auto-GPT's prompt" (2023) – Provides the full AutoGPT prompt template, including tool list and self-evaluation rules [6] [7], and the required JSON output format [18].
- IBM Think Blog. "What is BabyAGI?" (2023) – Describes BabyAGI's task loop and components [35] [37], use of GPT-4 + Pinecone + LangChain [44], and how it prioritizes tasks and stores results in a vector DB [43].
- Built In. "AutoGPT Explained: How to Build Self-Managing AI Agents" (2023) – Discusses AutoGPT's features like error handling and self-criticism loops [99], use cases in code generation [63] and business workflows [100], and compares AutoGPT vs BabyAGI vs AgentGPT [70] [101]. Also provides best-practice tips for cost control [87] and oversight [68].
- Reworkd (GitHub). **AgentGPT** – README description of AgentGPT's purpose and tech stack [72] [75].
- XLang (GitHub). **OpenAgents** – README overview of OpenAgents platform, agents offered and its goals [51] [77].
- TransformerOptimus (GitHub). **SuperAGI** – README highlights of SuperAGI as a dev-first framework with features like concurrent agents, toolkits, GUI, memory, etc. [67] [81].
- Daniel Benson. "ChatGPT Agent Mode: What's New..." *Medium* (Aug 2025) – Explains ChatGPT's Agent Mode capabilities (browser, code, file operations) [3] and its significance as bridging chatbot to autonomous assistant [4]. Provides an example use-case of an autonomous workflow (restaurant marketing) using Agent Mode [61] [62].

---

[1] [45] [47] [93] [94] From Language to Action: A Review of Large Language Models as Autonomous Agents and Tool Users
https://arxiv.org/html/2508.17281v1

[2] [3] [4] [53] [61] [62] [65] [66] ChatGPT Agent Mode: What's New, and Why It Matters for Entrepreneurs & Creators | by Daniel Benson | Medium
https://danielbensonpoe.medium.com/chatgpt-agent-mode-a5f385bac3ac

[5] [8] [9] [10] [11] [26] [27] [28] [29] [30] [31] [32] [33] [34] [55] [56] [57] [58] [69] [98] AI Agents: AutoGPT architecture & breakdown | by George Sung | Medium
https://medium.com/@georgesung/ai-agents-autogpt-architecture-breakdown-ba37d60db944

[6] [7] [16] [17] [18] [19] [20] [21] [22] [25] [46] Dissecting Auto-GPT's prompt - Prompting - OpenAI Developer Community
https://community.openai.com/t/dissecting-auto-gpts-prompt/163892

12 13 14 15 23 24 49 50 89 92 96 97 LLM Powered Autonomous Agents | Lil'Log
https://lilianweng.github.io/posts/2023-06-23-agent/

35 36 37 38 39 40 41 42 43 44 71 What is BabyAGI? | IBM
https://www.ibm.com/think/topics/babyagi

48 63 68 70 73 84 86 87 99 100 101 AutoGPT Explained: How to Build Self-Managing AI Agents | Built In
https://builtin.com/artificial-intelligence/autogpt

51 52 64 76 77 78 GitHub - xlang-ai/OpenAgents: [COLM 2024] OpenAgents: An Open Platform for Language Agents in the Wild
https://github.com/xlang-ai/OpenAgents

54 59 67 79 80 81 82 GitHub - TransformerOptimus/SuperAGI: <⚡> SuperAGI - A dev-first open source autonomous AI agent framework. Enabling developers to build, manage & run useful autonomous agents quickly and reliably.
https://github.com/TransformerOptimus/SuperAGI

60 AgentGPT
https://agentgpt.reworkd.ai/

72 74 75 83 GitHub - reworkd/AgentGPT: Assemble, configure, and deploy autonomous AI Agents in your browser.
https://github.com/reworkd/AgentGPT

85 microsoft/autogen: A programming framework for agentic AI - GitHub
https://github.com/microsoft/autogen

88 ChatGPT agent mode guide - KI Company
https://www.ki-company.ai/en/blog-beitraege/agent-mode-how-to-use-chatgpt-agent-mode-step-by-step

90 91 95 [NeurIPS 2023] Large Language Model-Based Autonomous Agents - LG AI Research BLOG
https://www.lgresearch.ai/blog/view?seq=409