

ChatGPT info about agents March 7, 2026

Chat: "ChatGPT info about agents"

You asked what **AI coding agents** are and how they differ from systems like **ChatGPT** or **Claude**. This is one of the biggest changes happening in AI tools right now.

1. The basic idea

Traditional AI assistants (ChatGPT, Claude):

- You **ask a question**
- The AI **answers**
- You repeat the process step by step

This is called a **prompt-response system**.

AI **agents** work differently:

You give them a **goal**, and they **work toward it autonomously**.

Example goal:

Build a small website that plays MP3 files from a folder and allows users to select songs.

An **AI coding agent** might:

1. design the architecture
2. write code files
3. test them
4. run the program
5. debug errors
6. update files
7. repeat until the program works

It behaves more like a **junior programmer working on a task**.

2. How this differs from ChatGPT/Claude

Feature	ChatGPT / Claude	AI Coding Agent
Interaction	prompt → response	goal → multi-step work
Memory	limited conversation	project memory
File control	mostly text output	can read/write files
Execution	cannot run code directly	often runs code
Iteration	user asks for fixes	agent fixes itself
Autonomy	low	moderate to high

So instead of saying:

Write a Python script.

You might say:

Create a Python application that records internet radio and saves files with metadata.

The agent then **builds the whole project**.

3. What agents actually do behind the scenes

Agents typically combine several capabilities:

1. **LLM reasoning** (like ChatGPT)
2. **tool use**
3. **file system access**
4. **code execution**
5. **iteration loops**

A typical internal loop looks like:

Plan

↓

Write code

↓

Run program

↓

See errors

↓

Fix code

↓

Repeat

This is sometimes called an **agent loop**.

4. Examples of AI coding agents (2025–2026)

Some well-known systems include:

- OpenAI Codex
- Devin
- AutoGPT
- Open Interpreter
- Claude Code

These can operate directly on a project directory.

5. What makes agents powerful

Agents can:

- explore a codebase
- modify multiple files
- run tests
- install libraries
- fix errors automatically
- commit changes to git

They behave much more like **an autonomous developer**.

6. Why this matters

This changes programming from:

writing code

to

directing the creation of code

The human role becomes:

- defining goals
 - reviewing results
 - guiding the system
-

7. Where agents struggle today

They are impressive but still imperfect.

Common problems:

- they may wander off task
- they sometimes break working code
- debugging complex systems is still hard
- they need good instructions

So people still **supervise them closely**.

8. Why this might interest you

Based on your projects (like:

- Harry's Boppers
- your media player tools
- web apps you're developing),

an agent could potentially:

- create a **working prototype**
- restructure code modules
- generate documentation
- run debugging cycles

much faster than manual prompt interaction.

9. Simple analogy

Think of the difference like this:

ChatGPT / Claude

You ask questions.

The AI answers.

Agent

You hire a worker.

You give them a task.

They go work on it.

10. The direction this field is going

Many experts think that by **2027–2030** most software development will involve:

- human designers
- AI coding agents
- automated testing systems

working together.

✅ If you'd like, I can also explain something that surprises many people:

why most people initially use AI coding agents the wrong way — and how experienced developers structure projects so agents work dramatically better.

XX

Yes, let's have that

Why people initially use AI coding agents the wrong way

...and how experienced developers make them work much better.

When people first try an AI coding agent, they often treat it **exactly like a chat assistant** such as ChatGPT or Claude. That leads to a few predictable mistakes.

1. The biggest mistake: giving vague goals

A typical beginner instruction might be:

Build a music player website.

For a human programmer that's already ambiguous, but for an agent it's even worse. The agent has to guess:

- language
- framework
- file structure
- database
- UI style
- hosting method

Agents will **invent all of that**, and the results can wander.

What experienced users do instead

They define a **clear specification**.

Example:

Build a simple web app.

Requirements:

- HTML + JavaScript
- Runs from a single folder
- Lists MP3 files in a directory
- Click a file to play it
- No database
- Minimal UI

Agents perform **dramatically better when the target is specific**.

2. Second mistake: asking for huge systems all at once

Beginners often say:

Build an entire application.

Agents do better when tasks are **broken into phases**.

Experienced developers structure projects like this:

Phase 1: basic file listing

Phase 2: audio playback

Phase 3: playlists

Phase 4: UI improvements

Agents work best when they can **iterate**.

3. Third mistake: letting the agent rewrite everything

Agents sometimes decide to **refactor entire codebases**, which can break working features.

Experienced developers instead say things like:

Do not change existing functions.

Only modify the playback module.

Constraining the agent keeps things stable.

4. Fourth mistake: no project structure

Agents work best when the project has clear structure.

Example:

project/
 index.html
 player.js
 styles.css
 audio/

Without structure, the agent may scatter files randomly.

5. Fifth mistake: not using documentation

Agents perform much better when the project includes a **clear README**.

Example README:

This project is a simple music player.

Architecture:

- PHP lists audio files
- JavaScript handles playback
- No database

Agents can read that file and understand the system quickly.

6. Sixth mistake: not giving the agent feedback loops

Agents improve when they can:

1. write code
2. run it
3. see errors
4. fix them

Systems like Open Interpreter or AutoGPT allow the agent to do this automatically.

7. The pattern experienced users follow

Good agent workflows look like this:

Define goal

↓

Define constraints

↓

Define architecture

↓

Build small piece

↓

Test

↓

Expand

This keeps the system stable and understandable.

8. A useful mindset shift

Instead of thinking:

The AI writes the program.

Think:

The AI is a junior engineer on the team.

You still guide:

- architecture

- scope
- priorities

9. Why this matters for projects like yours

For something like **Harry’s Boppers**, an agent could help build pieces such as:

- a **3-D object editor**
- a **path editor**
- a **timeline event system**
- animation export tools

But the best results would come from defining modules like:

module: object generator

module: path editor

module: event timeline

Agents are excellent at **building modules**.

10. A surprising discovery developers have made

The most successful teams now spend **more time writing specifications than writing code**.

The specification becomes the **master blueprint**, and agents build from it.

This is a big change in how programming works.

If you want, I can also show you something very interesting:

the simple architecture most AI coding agents internally use (the “agent loop”) and how that architecture allows them to work on a project for hours at a time without new prompts

XXXXXXXXXXXXXXXXXXXXXXXXXXXX

OK, I wonder if I will understand that. Let me try

Good question — and yes, you will probably understand it. The idea is actually quite simple once you see the structure. Think of an AI agent like a **wombat digging a tunnel system**: it repeatedly digs a bit, looks around, adjusts direction, and continues until the burrow reaches the desired place. The agent does something similar with code.

The basic “agent loop”

Almost every AI coding agent (such as OpenAI Codex, AutoGPT, or Devin) operates with a repeating cycle.

Goal

↓

Plan

↓

Act

↓

Observe

↓

Improve

↓

Repeat

That loop may run **dozens or hundreds of times**.

Step 1 — Goal

You give the system a goal.

Example:

Build a program that lists MP3 files in a folder and lets the user play them.

The agent stores that goal in its memory.

Step 2 — Plan

The AI reasons about what needs to happen.

Example plan:

1. Create HTML interface
2. Write JavaScript audio player
3. Read files from directory
4. Display list of files
5. Connect click events to playback

This planning step uses the same reasoning ability as systems like ChatGPT.

Step 3 — Act

The agent performs actions using tools.

Typical actions include:

- create files
- edit code
- run programs
- install libraries
- search documentation

Example action:

Create player.js

Step 4 — Observe

After acting, the agent checks results.

Example:

Run program

The agent might see:

Error: audio files not loading

Now it has feedback.

Step 5 — Improve

The agent reasons again:

The file path is incorrect.

Fix the directory path.

Then it modifies the code.

Step 6 — Repeat

The loop starts again.

Plan

Act

Observe

Improve

Over time the system gradually improves the program.

What makes agents different from normal AI chats

Chat systems like Claude or ChatGPT usually do this loop **only once per prompt**.

Agents do it **continuously without new prompts**.

That is the key difference.

A simplified internal diagram

Most coding agents internally look like this:

User goal



LLM reasoning engine



Tool system



File system / code execution



Results



Back to reasoning engine

The AI repeatedly cycles through this pipeline.

Why this is powerful

Because the agent can:

- run code
- see errors
- fix them
- try again

It behaves like a **developer debugging a program**.

Why agents sometimes go wrong

The loop can also cause problems.

If the plan is flawed, the agent may:

Fix problem



Break something else



Fix that



Break something new

This is why humans still supervise.

The key insight

The magic of AI agents is **not smarter AI**.

The magic is **letting the AI iterate repeatedly**.

Just like a wombat slowly expanding a burrow, each loop makes a small improvement until the structure is complete.

If you'd like, the next thing I could show you (and it's fascinating) is **what a real AI coding agent session actually looks like internally**—the sequence of thoughts, tool calls, and code edits it performs while building a program. It's very illuminating once you see it.

XXXXXXXXXXXXXXXXXXXXXXXXXXXX

OK

Great — let's look at **what a real AI coding agent session looks like internally**. This will make the idea of the **agent loop** much clearer.

I'll show a simplified example. Imagine you give the agent this goal:

Build a small web page that lists MP3 files and lets the user play them.

An agent such as OpenAI Codex or Devin might go through a sequence like this.

Step 1 — The agent analyzes the goal

Internal reasoning might look something like:

Goal: build MP3 player page.

Plan:

1. Create HTML interface
2. Write JavaScript audio playback
3. Display list of files
4. Connect buttons to play audio

The agent now has a **task plan**.

Step 2 — The agent creates files

First action:

Create file: index.html

Example result:

```
<!DOCTYPE html>
<html>
<head>
<title>Simple MP3 Player</title>
</head>
<body>
<h1>Audio Player</h1>
<ul id="songList"></ul>
<script src="player.js"></script>
</body>
</html>
```

Then:

Create file: player.js

Step 3 — The agent writes initial code

Example:

```
const songs = ["song1.mp3", "song2.mp3"];
```

```
const list = document.getElementById("songList");
```

```
songs.forEach(song => {
  const li = document.createElement("li");
```

```
li.textContent = song;
li.onclick = () => {
  const audio = new Audio(song);
  audio.play();
};
list.appendChild(li);
});
```

Step 4 — The agent runs the program

The agent might run a local server.

Run program

Now it **observes the result**.

Suppose it sees an error:

Error: songs not loading

Step 5 — The agent diagnoses the problem

Internal reasoning:

Problem:

Song paths may be incorrect.

Fix:

Add folder reference.

Step 6 — The agent edits the code

The agent modifies the code:

```
const songs = [
  "audio/song1.mp3",
  "audio/song2.mp3"
];
```

Step 7 — The agent tests again

Run program

Now playback works.

Step 8 — The agent improves the program

Agents often **continue improving things**.

Example reasoning:

Enhancement:

Add play/pause button.

Add nicer UI.

Then more edits happen.

What's important here

The human did **only one thing**:

Give goal

The agent did everything else:

plan

write code

Agent program (CLI tool)

|

AI model (cloud service)

|

Tools on your computer

The agent program acts as a **bridge** between the AI model and your computer.

2. What the agent is allowed to do

Most coding agents can use tools like:

read files

write files

run programs

execute shell commands

install libraries

run tests

Example commands the agent might run:

ls

mkdir

pip install

node server.js

git commit

So the AI isn't just producing text — it is **interacting with the system**.

3. A typical agent session

A developer might open a terminal and run something like:

agent start

Then they give the goal:

Build a simple MP3 player web app.

Now the agent begins working.

You might see activity like:

Planning project structure

Creating index.html

Creating player.js

Running program

Fixing error

Testing playback

It may continue for **minutes or hours**.

4. Popular agent tools (2025–2026)

Several tools are widely used.

Open Interpreter

Runs locally and lets the AI execute commands directly on your computer.

Claude Code

Anthropic's coding agent designed to work inside development environments.

OpenAI Codex

A system designed for autonomous software tasks.

AutoGPT

One of the earliest open-source agent frameworks.

Devin

A well-known experimental autonomous developer system.

5. Safety controls

Because the agent can run commands, most systems include **approval controls**.

Example:

Agent wants to run command:

```
rm -rf build_directory
```

```
Allow? (y/n)
```

The human can approve or reject actions.

Think of it as **supervising a junior programmer**.

6. How developers structure projects for agents

Developers usually place the agent inside a **project directory** like this:

```
project/
```

```
  README.md
```

```
  src/
```

```
  tests/
```

```
  docs/
```

The agent reads the files and understands the system.

The **README** often acts like a **project briefing document**.

7. Why CLI agents are powerful

Running an agent locally gives it abilities chat systems don't have.

It can:

- explore your codebase
- run your programs
- edit multiple files
- run automated tests
- fix errors

This turns the AI into something closer to a **developer assistant**.

8. The emerging workflow

Many developers now work like this:

Human defines project

↓

Agent builds first version

↓

Human reviews

↓

Agent fixes problems

↓

Project improves

It becomes a **collaboration between human and AI**.

9. Why this is becoming important

For decades programming meant **writing every line of code manually**.

They are extremely useful but **not autonomous**.

Level 2 — Coding agents

These are systems such as:

- OpenAI Codex
- Open Interpreter
- AutoGPT

They work toward a **goal rather than a single prompt**.

Example goal:

Create a website that lists MP3 files
and allows the user to play them.

The agent might:

plan the project
create files
run the program
debug errors
improve the code

Characteristics

- multi-step reasoning
- tool use
- file system access
- repeated iteration

They behave somewhat like a **junior developer**.

Level 3 — Autonomous software engineers

This level is still emerging.

Examples include systems like:

- Devin

These systems aim to handle **entire software projects**.

A goal might look like:

Build a bug-tracking web application
with login, database, and reporting.

The system might:

design architecture
write code modules
run tests
fix bugs
deploy the system
monitor results

Characteristics

- long-running tasks
- complex planning
- persistent memory
- project management

They behave more like a **full developer or small team**.

A useful comparison

Think of it like giving a wombat a clear tunnel plan versus dropping it randomly into hard ground. With a plan, the digging progresses smoothly.

1. Start with a written specification

Before the agent writes any code, experienced users create a **simple spec document**.

Example:

Project: Simple Audio Player

Goal:

Play MP3 files from a folder.

Constraints:

- Runs in browser
- No database
- Uses HTML + JavaScript

Features:

1. Display list of MP3 files
2. Click to play
3. Show current track

Agents perform far better when the **goal is explicit and constrained**.

2. Break the system into modules

Agents struggle when everything is one large piece.

Instead define modules:

Module 1: file listing

Module 2: audio playback

Module 3: user interface

Module 4: playlist management

The agent can then work on **one module at a time**.

This approach fits well with tools such as OpenAI Codex or AutoGPT.

3. Give the project a clear directory structure

A predictable layout helps the agent navigate.

Example:

```
project/  
  README.md  
  src/  
    player.js  
    filelist.js  
  ui/  
    index.html  
    styles.css  
  audio/
```

Agents can easily **read files and understand relationships**.

4. Use a README as the “project briefing”

The README becomes a **guide the agent reads repeatedly**.

Example content:

Architecture

- JavaScript handles playback
- HTML displays song list
- MP3 files stored in /audio directory

Many agents will read the README at the start of each planning cycle.

5. Build the system in phases

Instead of requesting the whole application at once, developers use **incremental development**.

Example phases:

Phase 1: list audio files

Phase 2: play selected file

Phase 3: add playlist controls

Phase 4: improve interface

Each phase is easier for the agent to complete reliably.

6. Include tests

Agents become much more reliable if tests exist.

Example test goals:

- audio file list loads correctly
- play button starts playback
- invalid files handled gracefully

The agent can run tests after each change and fix problems automatically.

7. Limit what the agent can modify

Developers often add instructions like:

Do not modify existing playback functions.

Only edit the UI module.

This prevents the agent from breaking working code.

8. Keep modules small

Small modules are easier for agents to understand.

Instead of one 1000-line file:

player.js

playlist.js

ui.js

filelist.js

Agents can reason about smaller components more effectively.

9. Encourage iteration

Good agent workflows follow a loop:

plan

build module

test

fix errors

improve

This is the **agent loop** operating within a structured project.

10. Why this approach works well

Agents are very good at:

- implementing clear instructions
- editing files
- fixing errors

They are weaker at:

- inventing architecture
- deciding requirements
- managing complexity

So humans define **the structure**, and the agent fills in the details.

Why this is especially relevant for your projects

For something like your animation ideas or media tools, a structured design might look like:

Module: object generator

Module: path editor

Module: event timeline

Module: renderer

An agent could build and test each module individually.

The key shift in modern programming

Traditional programming:

human writes code

Agent-assisted programming:

human designs system

AI builds components

human supervises

If you'd like, the next topic that many technically curious people enjoy is **how several agents can work together as a team** (planner agent, coding agent, testing agent, etc.). It's an emerging approach that is becoming quite powerful.

XX

OK, that too

How multiple AI agents can work together as a team

A newer idea in AI development is that instead of using **one agent**, systems can use **several specialized agents** that cooperate—much like a small software team.

Systems such as AutoGPT, Open Interpreter, and experimental frameworks built around OpenAI Codex sometimes follow this pattern.

Think of it like organizing a group of wombats digging different tunnels that eventually connect into one big burrow system: each does a specific job, and together they create something larger.

1. The basic multi-agent architecture

Instead of one AI doing everything, the work is divided:

User goal

↓

Planner agent

↓

Coding agent



Testing agent



Review agent

Each agent focuses on a **specific role**.

2. Planner agent

The planner analyzes the goal and creates a strategy.

Example goal:

Build a web app for playing MP3 files.

The planner might produce a plan like:

1. Create project structure
2. Implement file listing module
3. Implement playback module
4. Build UI
5. Add tests

The planner does **no coding**—it only organizes work.

3. Coding agent

The coding agent writes the actual code.

Tasks might include:

- creating files
- implementing functions
- connecting modules

Example:

Create `player.js`

Implement `playAudio()`

Connect UI buttons

4. Testing agent

This agent checks whether the program works.

It might:

- run automated tests
- simulate user actions
- check for errors

Example result:

Test result:

Playback button not working.

5. Debugging agent

Some systems include a specialized debugging agent.

It analyzes errors and proposes fixes:

Error source: incorrect file path

Suggested fix: update audio directory reference

6. Review agent

A review agent evaluates the overall code quality.

It might suggest improvements like:

- cleaner architecture
- simpler functions
- better comments

This role is similar to a **code reviewer on a development team**.

7. The system loops

After each cycle the agents communicate results.

Example loop:

Planner → Coding → Testing → Debugging → Review

The cycle repeats until the system works.

8. Why multi-agent systems help

One AI trying to do everything can become confused.

Dividing work into roles improves:

- clarity
- reliability
- scalability

It mirrors how real engineering teams operate.

9. What developers are discovering

Many early agent systems tried to build software with **one giant AI loop**.

Newer systems increasingly use **cooperating agents** instead.

This often produces better results.

10. The direction the field may go

Some researchers believe future development environments will look like this:

Human architect

↓

Planner agent

↓

Team of coding agents

↓

Testing agents

↓

Deployment agents

The human focuses mainly on **design and decisions**, while the agents handle implementation.

A thought that may interest you

Your approach of dividing projects into modules (like the components you've described in past projects) actually aligns very well with how AI agents operate.

Designing systems as **independent modules** makes it much easier for agent teams to build them.

If you'd like, the next fascinating thing to explore is **how ordinary people can start experimenting with agents today**, without needing a big lab or expensive systems. That topic often surprises people because the barrier to entry is becoming quite low.

XXXXXXXXXXXXXXXXXXXXXXXXXX

Though I already have an agent that my son set up for me I have very little understanding of what it is or can do. Let me have that information also

That's actually a very good place to be. Many people now **have an AI agent installed before they fully understand it**, especially if a technically inclined friend or family member set it up. Let's walk through what you likely have and how to understand it.

1. What your agent probably is

Since your son mentioned **Codex with a CLI**, you probably have something like this:

AI model (cloud)

|

agent program on your computer

|

terminal / command line

The **agent program** is what runs on your machine.

Common possibilities include:

- OpenAI Codex
- Open Interpreter
- Claude Code
- AutoGPT

The exact one depends on what your son installed.

2. What that agent can usually do

Most coding agents connected to your computer can:

read files

write files

edit code

run programs

install libraries

execute shell commands

Example actions an agent might take:

create file index.html

run python program

install npm package

edit player.js

In other words, it can **work directly on your project directory**.

3. The key difference from ChatGPT

With a normal assistant like ChatGPT:

You ask questions.

The AI answers.

With an agent:

You give a goal.

The AI performs actions.

Example:

Goal: create a small music player web app.

The agent might then:

- create files
- write code

- run tests
- fix errors

without needing constant prompts.

4. Where the agent lives on your computer

Usually it runs in a **terminal window**.

You might start it with a command such as:

codex

or

interpreter

or

agent start

Then you type instructions.

5. What the agent is NOT

Even though it can run commands, it is not a fully autonomous system.

You are still responsible for:

- giving goals
- approving risky commands
- reviewing results

Think of it as a **programmable assistant**, not a replacement developer.

6. The safest way to use it

Most developers treat agents like this:

Create a project folder

Run the agent inside that folder

Let it modify only those files

Example:

```
boppers_project/
```

```
  src/
```

```
  assets/
```

```
  README.md
```

The agent works **only inside that directory**.

7. How you can find out exactly what agent you have

A few quick clues will tell you.

Look at the command your son uses to start it.

For example:

Command Likely system

interpreter Open Interpreter

codex Codex agent

claude Claude Code

autogpt AutoGPT

You can also try typing:

help

inside the agent.

Most agents will print their capabilities.

