ChatGPT

Generative AI Intermediate-to-Advanced Roadmap (8–10 Weeks)

Overview: This 8–10 week roadmap builds on foundational generative AI knowledge and takes you through intermediate and advanced topics. You'll deepen your skills with OpenAI APIs, learn to create structured outputs and tool-using agents, implement Retrieval-Augmented Generation (RAG) with vector databases, and explore cutting-edge multi-agent frameworks (LangGraph, Google ADK, CrewAI, Microsoft AutoGen, etc.). In the later weeks, you'll undertake complex projects (single-agent and multi-agent systems) and delve into the latest trends like test-time reasoning, OpenAI's ChatGPT **Agent** feature, Anthropic's Claude Code, and more. Each week lists study goals, key topics, project ideas, and high-quality resources (free and paid) to guide your learning.

Week 1: Mastering OpenAI & Competitor APIs (Text, Image, Voice)

Study Goals: Get hands-on experience with **OpenAI's APIs** (text completions/chat, image generation, and audio processing) and understand similar offerings from other major providers (e.g. Google's **Gemini** on Vertex AI, Anthropic's Claude). By week's end, you should be comfortable using the OpenAI Playground and programmatic API calls for various modalities, and brainstorm a few multi-modal use cases to implement.

Topics Covered:
- **OpenAI Text APIs:** Using GPT-3.5/GPT-4 via the Chat Completion API (system vs. user messages, parameters like temperature). Experiment in the OpenAI Playground with different modes (chat vs. completion).
- **OpenAI Image API:** Generating images with DALL·E or similar (if available via API) – crafting image prompts and handling results. *(If OpenAI's image API isn't accessible, note alternative image gen models on Hugging Face or stability.ai.)*
- **OpenAI Audio APIs:** Using Whisper for speech-to-text transcription and (if accessible) text-to-speech or voice capabilities.
- **Google's Gemini API:** Understand Google's new multimodal model **Gemini** via Google Cloud Vertex AI (Gemini 1.5/2.0) **– how to access it through AI Studio, its text and image generation capabilities** [1] [2] **. (Gemini 2.0 introduced native image/audio output and tool use** [3] **.)**
- Other Models (Optional): **Awareness of** Anthropic Claude **API and** Meta's Llama 2** (via Hugging Face) for text, and stable diffusion for images. Prioritize OpenAI, but recognize alternatives for a broader perspective.

Practice Ideas: *No major project yet; focus on API "recipes."* For example: **(1)** Use Whisper to transcribe an audio clip, then feed the transcript to GPT-4 to summarize it. **(2)** Generate an image from a text description (e.g. "a sunset cityscape in watercolor") with an image API, then have GPT write a short story about that image. **(3)** Try a conversation with Gemini (or Google Bard) on a topic and compare its style to ChatGPT. Brainstorm 2–3 real-world use cases (e.g. a voice-enabled Q&A bot, or an image + text tutoring assistant).

Key Resources:
- **OpenAI API Documentation & Quickstart:** Official docs on using chat completions, image and audio endpoints [4] [5]. Covers how to format requests and interpret responses.
- **OpenAI Playground Tutorial (Medium):** *"How to Use the OpenAI Playground"* – step-by-step guide for trying prompts and parameters in the UI. Great for non-coders [6].
- **Google Vertex AI Generative Quickstart:** Google Cloud's tutorial on using generative models (PaLM/Gemini) in Vertex AI [7]. Explains accessing the API and trying the web Studio.
- **YouTube:** *"OpenAI API Crash Course"* by freeCodeCamp or similar – a video that demonstrates calling OpenAI's text API in Python and building a simple app (check for updated 2024/2025 versions).
- **Note:** OpenAI offers a $5 credit for new accounts – use it to experiment. Google Cloud may have free trial credits for Vertex AI as well.

Week 2: Advanced Prompting – Tools, Structured Outputs & Complex API Projects

Study Goals: Learn how to make LLMs **produce structured outputs** (like JSON) and call external **tools/APIs** via function-calling. Practice designing prompts and schemas so that model outputs are reliable for downstream use. By the end, you will ideate and outline *5 example projects* that use OpenAI APIs in a complex, resume-worthy manner – focusing on tool use and structured data.

Topics Covered:
- **Function Calling (OpenAI):** Master OpenAI's function calling feature to enable tool use. Understand how to define functions with JSON schemas and let the model decide when to invoke them [8] [5]. Learn best practices for crafting function definitions (clear name, description, and parameters) so the model can reliably pick the right function.
- **Structured Output & JSON Schemas:** Use OpenAI's *"JSON Mode"* or the newer Structured Output feature to force the model's reply to conform to a format [9] [10]. For instance, extracting entities from text into a JSON, or having the model respond with a fixed schema for integration with code. Recognize when to use function calling vs. asking the model directly for JSON (the Vellum AI blog on *JSON Mode vs Function Calling* is a good reference [11]).
- **Prompt Techniques for Reliability:** Few-shot prompting to show the desired JSON format, using system messages to demand *only* JSON output, and handling model errors when it deviates. Also, learn to set `strict=true` in OpenAI's structured output for exact schema conformity [8] [12] (introduced August 2024, it yields 100% valid schema adherence in their evals [12]).
- **Complex Prompting Patterns:** Practice **tool-aware prompts** – e.g. instructing the model to use functions for certain queries (like "If the user asks for current weather, call the `get_weather` function"). Experiment with **chain-of-thought** style prompting where the model outputs reasoning (perhaps as a hidden "# Reasoning" field in JSON) separate from the final answer [13]. This helps in structured workflows (you can capture the reasoning but only show the final answer to users).
- **OpenAI vs Others:** Note that other ecosystems have similar capabilities – e.g. Azure OpenAI Service supports these features (with integration to Azure tools), and Google's PaLM API offers tool use via **MakerSuite** and the Agents API (e.g. in MakerSuite you can create an "AI agent" with tools). But in this week, **focus on OpenAI's approach** for consistency.

Example Projects (5 Ideas): Each of these uses OpenAI's API, demonstrating structured output or tool use, and can be great on a portfolio:
1. **Intelligent Meeting Scheduler:** An assistant that takes a natural language request ("Schedule a 30-min meeting with John next week") and produces a structured JSON calendar event. It could use function calling – e.g. a `find_slots` function to check availabilities – and output confirmed meeting details in JSON. *(Skills: function calling, date parsing, JSON output)*.

2. **Weather & News Chatbot:** A chat assistant that can answer general questions but calls functions for real-time info – e.g. a `get_weather(location)` and a `get_news(topic)` function. The bot might output a nicely formatted answer with a JSON snippet of the raw data fetched. *(Skills: external API integration via function calls, blending static and dynamic info)*.
3. **Personal Finance Analyzer:** The user uploads or describes monthly expenses; the LLM returns a categorized breakdown in JSON (categories as keys, totals as values) [13]. You define a strict schema for the output (e.g. categories: Food, Rent, etc.). This could involve *format enforcement* – the model has to extract numbers from text and fit them into the JSON schema. *(Skills: information extraction, structured output)*.
4. **QA Chatbot with Knowledge Base:** A chatbot that, if it cannot answer from its own knowledge, uses a `search_docs(query)` function to retrieve info from a document database (or Wikipedia API). The model then cites the source in a structured way. For example, user asks a medical question – bot uses a search function to get an answer, then responds: `{"answer": "...", "source": "WebMD"}`. *(Skills: tool calling for retrieval, JSON output with citations)*.
5. **Code Debugger Assistant:** An agent that helps debug code. The user provides an error message; the assistant uses a `run_code` function (which you implement to actually run a code snippet or test) and returns the result. The final output is a JSON with keys like `{"analysis": "...", "error_cause": "...", "suggested_fix": "..."}`. *(Skills: function calling to run code, interpreting and structuring results)*.

Key Resources:
- **OpenAI Blog – *"Introducing Structured Outputs in the API": Explains how to enforce JSON schemas and gives examples [9] [8]. Must-read to understand the motivation and usage (e.g. it highlights that developers often resorted to brittle prompt tricks, which this feature solves [13]).***
- OpenAI Cookbook – Function Calling How-To: A step-by-step guide on implementing function calls [14]. Shows how to define functions and handle the model's `{"function_call": ...}` response. Includes examples of calling a weather API and a calculator.
- **DataCamp Tutorial – *"OpenAI Function Calling: Generate Structured Output"*:** A practical tutorial that builds functions to extract info from student profiles and returns perfectly uniform JSON [15] [16]. It demonstrates multi-function scenarios and how GPT-3.5/GPT-4 auto-selects the correct function for a query [17]. Great for learning by example.
- **Vellum AI Blog – *"When to use Function Calling vs. JSON Mode"*:** Discusses differences between older JSON prompting and the new function call/structured modes [11]. Helps you decide which approach fits your use case and constraints.
- **YouTube – *OpenAI Function Calling Tutorial (2024)*:** Many creators (e.g. James Briggs) have videos walking through function calling. Watch one for a live coding demo.
- **DeepLearning.AI Short Course – *"Building Systems with the ChatGPT API"*:** *(Optional, free audit)* Focuses on chaining and multi-step prompts, but also relevant here. Taught by OpenAI's Isa Fulford and Andrew Ng [18] [19]. It's beginner-friendly but includes valuable lessons on evaluating and structuring prompts, which complements this week's goal of making outputs reliable.

Week 3: Retrieval-Augmented Generation (RAG) & Vector Databases

Study Goals: Understand how to extend LLMs with external knowledge via **Retrieval-Augmented Generation**. Learn to build a pipeline that indexes custom data into a vector database, retrieves relevant chunks, and feeds them to an LLM prompt. Explore advanced RAG techniques (chunking, hybrid search, re-ranking) and get familiar with frameworks like **LangChain** and **LlamaIndex** for simplifying RAG development. By the end, you should implement a basic RAG workflow on a sample dataset and grasp what "Advanced RAG" entails.

Topics Covered:
- **RAG Concept Refresher:** LLMs have limited knowledge cutoff and context windows. RAG mitigates this by retrieving relevant **external data** (documents, knowledge base) and **augmenting** the prompt with that information [20]. The LLM's answer is thereby "grounded" in real data, reducing hallucinations. *Key point: "RAG helps LLMs generate more accurate and useful responses by retrieving relevant info they weren't trained on, often private, recent, or domain-specific, giving the LLM context to provide grounded answers."* [20].
- **End-to-End RAG Pipeline:** The typical flow: **Ingestion** (load your data), **Chunking** (split into passages, embed them into vectors), **Indexing** (store embeddings in a vector database), **Retrieval** (given a user query, compute its embedding and find similar document vectors), and **Generation** (prompt the LLM with the query + retrieved text, to get an answer) [21] [22]. Learn these stages and why each is important (e.g. good chunking ensures relevant info is retrieved).
- **Vector Databases:** Overview of popular vector DBs – **Chroma, Pinecone, Weaviate, Milvus, Qdrant**, etc. Understand basic operations: insert documents (with embeddings), similarity search, filtering by metadata. No need to master all, but know their purpose: efficient nearest neighbor search in high-dimensional space. Many are accessible with a Python SDK or REST API.
- **Using LangChain for RAG:** LangChain provides chains and ready integrations (for instance, `VectorstoreIndexCreator` to chunk, embed (using OpenAI or other models), and store in a vector DB in a few lines). It also has utilities to combine retrieval with LLM QA (the *RetrievalQA* chain). Try a LangChain tutorial that builds a Q&A chatbot over a custom PDF or website [23].
- **Using LlamaIndex (GPT Index):** LlamaIndex offers an abstraction to index documents and query them with various strategies. It can connect to vector stores under the hood. Explore LlamaIndex's documentation on building a simple RAG pipeline [24] and note features like composite indices, query transforms, etc., which represent more advanced RAG.
- **Advanced RAG Techniques:** Go beyond the basic "retrieve top-3 and stuff into prompt." Learn about **hybrid search** (combining keyword + vector search), **re-ranking** retrieved docs by relevance (using a cross-encoder or feedback from the LLM), **iterative retrieval** (the LLM can reformulate the query if initial results aren't good), and **chunking strategies** (overlap sliding windows to capture context). For example, chunk size vs. overlap is a trade-off: too large and you risk irrelevant text, too small and you lose context. Advanced RAG involves tuning these and possibly multi-step retrieval (retrieve, then use LLM to decide if more info needed). *Resource:* Tim Spann's All Things Open 2024 talk on Advanced RAG techniques [25] or the Towards AI article *"Advanced RAG: Illustrated Overview"* [26] – these cover chunking, re-rankers, query expansion, etc.
- **One Use-Case Deep Dive:** Pick a domain to solidify ideas – e.g. **RAG for Tech Support**: ingest product manuals into a vector DB, then user asks troubleshooting questions and the system retrieves relevant sections to answer. Consider what "advanced" features you might need (perhaps search by error codes = keyword, plus semantic search for general symptoms; ensure the answer cites the manual page). This will prepare you for designing your own RAG project next week.

Hands-On Practice: Implement a **simple RAG Q&A bot** on a small dataset (for example, a set of Wikipedia articles, or your personal notes). Use either LangChain or LlamaIndex to simplify the process. Steps: ingest docs, build index, and write a prompt template that includes retrieved text. Test it with a few queries. This practice will surface issues like irrelevant retrievals or token limit concerns, which are part of learning. If you can, also try an **Advanced RAG tweak** – e.g., use **multiple retrievers** (LangChain's `MultiVectorRetriever` or LlamaIndex Router) to combine keyword and semantic search [27], or use a *re-rank step* (scoring passages with a question-answering model to choose the best). Even if you just experiment with these, it will deepen your understanding.

Key Resources:
- **DeepLearning.AI – *"Retrieval-Augmented Generation (RAG) Course"*:** *Highly recommended (free audit)*. A full course on RAG by instructor Zain Hasan. Covers search techniques (BM25, semantic vectors,

hybrid) [27], prompt design for RAG, and even deployment considerations. It includes hands-on labs building a RAG system on real datasets and discusses *evaluating RAG* performance [28] [29]. This will solidify both fundamentals and advanced skills.
- **LlamaIndex Docs – *Introduction to RAG*:** Succinct explanation of RAG and key stages, directly from the LlamaIndex perspective [21] [30]. Also read *"Stages within RAG"* and *"Important concepts"* in that doc to learn terminology (Nodes, Indexes, Retrievers, etc.) which apply generally.
- **LangChain Documentation/Tutorials:** The LangChain docs have a tutorial *"Build a QA system with RAG"* [23] and an example of using **Chroma** or **FAISS** as a vector store. Follow one of these to practice coding. LangChain's `RetrievalQA` chain or `ConversationalRetrievalChain` can handle a lot for you (including keeping conversation history in context).
- **Medium Article – *"Advanced RAG in LangChain and LlamaIndex"*:** (TowardsAI, 2024) Explores how to create basic vs. advanced RAG, including using both **LangChain and LlamaIndex together** [31]. A good read to see how the two frameworks can complement each other.
- **Vector DB Free Tiers:** Many vector DB services have a free tier or local option. E.g. **Pinecone** (free for small projects), **Weaviate** (can run locally or use their demo instance), or **ChromaDB** (open source, you can just `pip install chromadb` and use it in-memory). Use these in your experiments – e.g. Chroma for quick local tests, Pinecone if you want persistence in the cloud.
- **Video – *"All You Need to Know about Vector Databases for LLMs"*:** (e.g. a YouTube talk by **Sam Charrington/DEEpLearning AI**) – helps reinforce why vector DBs are crucial and how they work conceptually.

Week 4: Building a Complex Generative AI Project (Integrated Models + RAG)
Study Goals: Synthesize everything learned so far by building a **complex project** that integrates multiple components: LLMs, function/tool calls, and RAG for custom data. The goal is to simulate a real-world application development cycle – designing the system, implementing a prototype, and evaluating its performance. By the end of the week, you should have a portfolio-worthy project (or at least a well-scoped plan and partial implementation) that demonstrates intermediate/advanced generative AI skills.

Project Build Focus: Now that you've practiced various pieces (APIs, prompting, RAG), choose a substantial application to develop. Some suggestions:
- **"Ask My PDF" Research Assistant:** Build a chatbot that can answer questions about a collection of PDFs or knowledge base documents. For example, an HR policies chatbot that employees query. This requires: ingesting PDFs (RAG), using LLM to answer based on retrieval, and possibly function-calling for simple math or date lookup if needed. You'd implement retrieval with a vector DB from Week 3, and the generation via GPT-4. Ensure the bot's answers cite which document/page they came from (this can be structured output: e.g. answer + source).
- **Personal AI Planner:** An assistant that can take high-level goals from a user and break them into tasks/dates. For instance, "Plan a 5-day trip to Japan under $2000" – the agent should possibly use tools: call a flight search API, call a hotel API (you can mock these functions with sample data), and then produce an itinerary. This project would involve writing prompts for the agent to reason (maybe chain-of-thought) and call functions like `search_flights`, `search_hotels`, etc. It combines Week 2's function-calling mastery with creative prompt engineering.
- **Multi-modal Hobby Project:** If you're inclined, incorporate an image or audio component for extra flair. E.g. a "Podcast Prep Assistant" – you give it a topic, it retrieves info (RAG) and then even generates an image cover (via DALL·E) and outlines a script. This would use text + image generation together. Or a voice-enabled Q&A: use speech-to-text (Whisper) for input, answer via GPT, and text-to-speech for output (could use a service like ElevenLabs or Azure TTS for voice). This shows you can combine modalities.

System Design Considerations: As you build, consider **workflow** and **integration**: how will components talk to each other? For example, if using LangChain, you might orchestrate steps in a chain/agent. Or if coding manually, you'll have a Python script that calls OpenAI for completion, Pinecone for retrieval, etc., in a loop. Draw a diagram of your system's flow (user -> components -> response). This is good practice for explaining your project later (e.g. in interviews). Also plan for **error handling**: What if the model doesn't use the function correctly or output is malformed? Implement basic checks or retries (for structured output, you might re-prompt on JSON parse failure). These are the polish points that show advanced competency.

Evaluation: Test your project with several scenarios. Evaluate accuracy and coherence of outputs. For RAG, ensure that the model is indeed using the provided data (not hallucinating). One trick: include a slightly wrong fact in the retrieved text and see if the LLM uses it blindly – if it does, you might need to prompt it to double-check or increase retrieval quality. If you have time, incorporate simple metrics or user feedback: e.g. count how often the answer was correct or needed tweaks.

Document & Showcase: Write a README for your project, describing the approach, tools, and how to run it. This will help solidify your understanding and is useful if you share it on GitHub or in a portfolio. Highlight that it uses advanced features (like "OpenAI's function-calling to integrate live data" or "retrieval augmented generation for grounding answers"). These keywords stand out to recruiters.

Resources & Examples:
- **LangChain Example Apps:** Check LangChain's Hub or examples for similar projects. For instance, the *"Chat with PDF"* example uses LLM + vector DB; LangChain provides sample code for that [23]. Use it as a starting point.
- **Gradio Tutorials:** If you want a UI for your project, Gradio is an easy way to make a web demo (DeepLearning.AI's short course *"Building Generative Apps with Gradio"* is useful). A simple Gradio interface can make your project interactive (e.g. file upload for PDFs, chat box for Q&A).
- **OpenAI Cookbook – *"ChatGPT with Retrieval and Tools"*:** Look for cookbook examples that combine techniques. One example, *"Conversational Retrieval QA"*, shows how to maintain a chat history along with RAG. Another, *"Function Calling with OpenAI and external APIs"*, demonstrates step-by-step tool usage. These can guide your implementation details.
- **Community Projects for Inspiration:** The GitHub repo **"awesome-llm-apps"** [32] lists many open-source LLM applications that use RAG and agents. Browsing through a few (like "AI Medical Q&A" or "Regulations GPT") can spark ideas on system design and give insight into how others structure their code. It's also motivating to see real examples of what you're trying to build.
- **Hugging Face Spaces:** Similar to above, many demo apps on HF Spaces showcase LLMs with custom data. Two to check: *"PrivateGPT"* (question-answering on your documents locally) and *"LangChain Book QA"*. The repositories behind these demos are often open – reading their code can be educational (e.g. how they do chunking, or how they handle long answers).

By the end of Week 4, you should have a working prototype of your project. This not only reinforces the concepts from weeks 1–3, but also gives you a concrete piece to discuss or expand on in the next stage of advanced topics.

Week 5: Agents and Agentic Frameworks (Advanced Decision-Making with LLMs)

Study Goals: Transition from single-turn or single-task systems to **agentic systems** – where an AI agent can make decisions, invoke tools, and even collaborate with other agents autonomously. This week, you will explore the landscape of agent frameworks, understanding how they differ and what problems they solve. Key frameworks include **CrewAI, Microsoft's AutoGen, LangChain's LangGraph extension, and**

Google's Agent Development Kit (ADK). By week's end, you should grasp the core ideas of **LLM-based agents**, know the strengths of major frameworks, and be able to choose one for a given project.

Topics Covered:
- **What is an "AI Agent"?** In this context, an agent is an AI system (often powered by an LLM) that can **observe**, **reason**, and **act** in an environment towards a goal [33]. Unlike a single prompt-response, an agent might loop through thought processes, call functions/tools, and even spawn other sub-agents. It's the difference between a one-shot QA versus an autonomous problem-solver that figures out what steps to take. You likely saw simple agents in Week 2 (function calling is a primitive form of agent actuation). Now we formalize it.
- **Frameworks Overview:** There has been an explosion of **agentic frameworks** in 2024–2025. Each provides infrastructure for building complex AI systems:
- **LangChain's Agents & LangGraph:** LangChain introduced agents that use an LLM to decide which tool to use and when. **LangGraph** is a new extension focusing on *explicit control flow* – it lets developers define agents and their interactions as a **graph of nodes and edges (stateful workflow)** [34]. LangGraph gives fine-grained determinism: you can enforce an exact sequence or loops in an agent's reasoning, which is useful for reliability in long workflows. *Core idea:* treat multi-agent processes like a flowchart, allowing debugging and state tracking [35] [36]. LangGraph is great when you need "agents that don't veer off course" by constraining their possible transitions [37].
- **Google's Agent Development Kit (ADK):** Open-sourced by Google in 2025, ADK is designed for **hierarchical multi-agent systems** and tight integration with Google's ecosystem [38]. It follows a **"code-first"** approach – meaning you structure agents in code similarly to how you'd write a software program (with classes, functions, etc.). ADK emphasizes building *teams of agents* with clear roles (one coordinator delegating tasks to specialist sub-agents) [39] [40]. It comes with many built-in tools and supports models like Gemini and others via Vertex AI [41] [42]. If you're in the Google Cloud world or want a more **enterprise-scale framework** (with evaluation, debugging UIs, deployment support), ADK is a strong choice [43] [44].
- **CrewAI:** A lean Python framework created independently (open-source, community-driven). CrewAI is built from scratch, *not* on LangChain [45]. It's known for high performance and flexibility in defining multi-agent "crews." CrewAI introduces concepts like **Crews, Flows, Tasks** – where a Crew is a group of agents working together, and Flows allow structured orchestration of those agents [46] [47]. CrewAI emphasizes **role-based agents** (each agent has a specific role/skill), **collaboration** (agents share information), and **observability** (tools to monitor agent interactions) [48] [49]. Many see CrewAI as a more low-level but powerful alternative to LangChain's ecosystem. (IBM's write-up calls it "production-grade orchestration for role-playing autonomous agents" [50] [51].) If you want to custom-build agent behaviors and need speed, CrewAI is compelling.
- **Microsoft AutoGen:** An open-source framework from Microsoft Research for multi-agent conversations [52]. AutoGen gained attention with patterns like "**Commander and Solver**" agents – e.g. one agent breaks a problem into sub-tasks, another executes or solves them, and they chat back-and-forth. AutoGen simplifies creating such multi-agent dialogues with LLMs cooperating [52]. It supports asynchronous messaging and dynamic agent creation, making it flexible for both research and practical apps. AutoGen v0.4 (late 2024) introduced an event-driven architecture for better scalability and debugging [53] [54]. If you're already using Azure/OpenAI, AutoGen integrates well (and can leverage Microsoft's Azure OpenAI tools).
- **Agents vs. Hardcoded Logic:** Reflect on why we use LLM-based agents. They shine in **open-ended, complex tasks** where we can't pre-program every step. Instead of writing a fixed algorithm, we let the LLM decide – for example, in a game, an agent can decide whether to attack or defend based on the situation (no fixed script). However, this flexibility comes with unpredictability. Frameworks like LangGraph and ADK aim to give developers control back (through constraints, testing, and approval steps) [35] [55]. A key learning is when to use an agent vs. a deterministic pipeline. If your use case needs adaptability and chaining multiple unknown steps (like "browse the web -> extract info -> draft report"),

an agent is powerful. If it's straightforward ("call API A then B"), simpler scripting might do.
- **Videos & Conceptual Learning:** Watch concept videos like **DeepLearning.AI's** *"What are AI Agents?"* (from the *Generative AI with LLMs* specialization) or **Two Minute Papers** episodes on autonomous agents. These give an intuitive sense of what it looks like for an AI to "decide and act." Also, check any recorded talks from OpenAI or Anthropic on agents – e.g. OpenAI's DevDay had hints of agentic features (which became ChatGPT's agent mode later).

Comparing Frameworks (Differences & Similarities): It's useful to create a small comparison chart for yourself. For example:

- *LangChain Agents:* Easy to start (especially if you used LangChain for RAG), good tool integrations, but could become unpredictable in complex scenarios – hence LangGraph was added for structure.
- *ADK:* Great for complex, multi-agent workflows especially if deploying on cloud; has **built-in evaluation, hierarchy support** [56] [57]. Tied to Google's ecosystem (works best with Vertex AI, etc.).
- *CrewAI:* Highly customizable and independent; might require more coding, but offers clear separation of agents, tasks, and flows – which can be a cleaner mental model for large projects [58] [59].
- *AutoGen:* Focused on multi-agent communication patterns and research, with strong support for conversation-based collaboration. If your problem is naturally "let these two LLMs talk to solve X", AutoGen is a fit.

Also note commonalities: All these frameworks allow an agent to use **tools** (APIs, functions) and have some notion of **memory/state** to carry info over steps. They differ in how much they expect you to hard-code the sequence vs. let the LLM figure it out.

Key Resources:
- **Medium – *"LangGraph vs ADK: Choosing the Right Agent Framework"* (2025)**: An excellent comparison of LangGraph and Google ADK [60] [61]. It describes each framework's philosophy (LangGraph for fine control in complex flows [35], ADK for scalable agent teams with code-first design [62] [39]) and gives guidance on when to pick one over the other [63] [64].
- **CrewAI Documentation – Introduction:** Read the first sections of CrewAI's official docs [65] [46]. It defines the building blocks (Agents, Crews, Flows, Tasks) in a clear way and highlights key features like role specialization and collaborative intelligence [48] [49]. This will help you conceptualize multi-agent systems in general, not just CrewAI.
- **Microsoft AutoGen GitHub/Docs:** Look at the README of the AutoGen GitHub repo or the MSR project page [52]. It likely has examples such as the classic "Commander & Solver" or using AutoGen to have a Python helper agent. This will show how to implement multi-agent dialogues in code.
- **DeepLearning.AI Short Courses on Agents:** They have several new short courses (free to audit) that are timely: *"Multi AI Agent Systems with CrewAI"*, *"AI Agentic Design Patterns with AutoGen"*, and *"AI Agents in LangGraph"*. For example, the **LangGraph course** (by Harrison Chase) teaches building an agent from scratch and then with LangGraph [66] [67]. These courses being short (~1.5 hours each) are perfect to quickly get practical exposure to each framework with expert guidance. Consider taking one for the framework you find most interesting.
- **Google Cloud Blog – *"Making it easy to build multi-agent applications"* (re: ADK):** This blog post introduces ADK with an example of multiple agents (WeatherAgent delegating to GreetingAgent) [68] [69]. It's a friendly explanation of why hierarchical agents matter and how ADK handles delegation via descriptions and sub-agents. After reading, you'll understand how Google envisions agent systems (which is transferable knowledge even outside ADK).
- **Mahimai Raja's Medium – *"Understanding Agentic Frameworks (CrewAI)"*:** An easy-to-read article

explaining agentic thinking and a demo using CrewAI [70] [71]. It's useful to see an example scenario (writing an article with Planner, Writer, Editor agents) described step-by-step, which makes the abstract concepts concrete.

By absorbing these materials, you'll be well-prepared to start **building agents** next. You'll know the "language" of agent frameworks and can decide which one to experiment with for your use case.

Week 6: Multi-Agent Systems in Practice & Real-World Use Cases

Study Goals: Dive into **multi-agent system behaviors** and learn how multiple AI agents can collaborate or compete to solve problems. This week is about seeing concrete examples of agents interacting (through dialogues or shared tasks), and identifying real-world applications where multi-agent setups shine. You will also come up with *5 potential use cases* for multi-agent systems that could be implemented with the frameworks from Week 5, solidifying your understanding of their value.

Topics Covered:
- **Multi-Agent Collaboration Patterns:** Learn the common patterns in which agents work together: **master-worker** (a chief agent delegates tasks to sub-agents, as in ADK's hierarchy [56] [72]), **peer collaboration** (agents with different skills communicate to jointly solve something), and even **adversarial or debate** setups (agents critiquing each other's answers to refine quality). A famous example is the *"two agents debating to find the truth"* – some research suggests multi-agent debate can improve reasoning. Understand that designing multi-agent prompts often involves giving each agent a clear identity/role and sometimes a shared memory or interaction protocol (like a turn-taking conversation).
- **Example Multi-Agent Scenario:** Revisit the *CrewAI blog demo* of **Planner, Writer, Editor** agents co-writing an article [73] [74]. Step through how it works: The Planner agent creates an outline (Task 1), the Writer expands it (Task 2), the Editor polishes it (Task 3) [75] [76]. This illustrates sequential collaboration: output of one agent is input for the next. Another example: **Coding with an AI pair** – one agent writes code, another reviews and tests it. Microsoft's AutoGen showcased such patterns (having a "checker" agent verify the "coder" agent's work). By studying these examples, you see how agents can compensate for each other's weaknesses (e.g., one agent keeps the other on track).
- **Agent Communication & Memory:** Multi-agent systems often involve agents communicating in natural language (which is basically prompt messaging). You'll learn how frameworks implement this: for instance, in AutoGen or LangChain, you might have a loop where each agent's response (a message) is fed as input to the next agent. Some frameworks provide a "message broker" or a shared memory list of chat history. It's important to design the protocol – e.g., Agent A's message might include a question to Agent B, etc. Think of it like roleplaying: you must prompt each agent with not only the user query but also what the other agents have said. Many frameworks handle this bookkeeping for you.
- **Emergent Behaviors:** Multi-agent setups can exhibit surprising behaviors – positive ones like creative problem-solving, and negative ones like getting stuck in loops or groupthink. Be aware of issues like **coordination** (ensuring agents don't talk over each other or fall into infinite back-and-forth on trivial points) and **consistency** (they might contradict each other or the final answer might need reconciliation). For example, if one agent is a "critic" and one is a "proposer," you have to eventually decide which output to present to the user. This often requires an extra step or an outside arbiter (which could be another LLM or simple logic).
- **5 Real-World Use Cases for Multi-Agent Systems:** Identify and describe five compelling applications where using multiple agents yields an advantage. For each, think about what roles the agents would play and how they'd interact. Here are five to consider:
1. **Software Development Assistant:** One agent is a **Code Generator** (writes code given a spec), another is a **Code Reviewer/Tester** (checks for bugs, runs tests). They iterate until the code passes the tests. This mimics a two-engineer pair programming or GitHub Copilot + a reviewer. Real-world value:

could auto-resolve simple pull request comments or write unit tests for generated code. *(Similar patterns have been demonstrated, where an AI "architect" designs a solution and a "coder" implements it, then an "inspector" reviews it)*.
2. **Research Analyst Team:** Imagine a complex research question (legal, market research, etc.). You can have a **Researcher Agent** that searches for information (uses tools to scrape web or database) and a **Synthesizer Agent** that takes those findings and writes a coherent report. Perhaps also a **Fact-Checker Agent** that verifies claims. Together, they can handle more breadth and verification than a single agent. (Notably, OpenAI's new ChatGPT agent essentially combines web browsing and analysis in one, but we can conceptualize them as cooperating modules [77] [78].)
3. **Customer Support Automation:** In a company setting, you might deploy multiple specialized agents: e.g., **Policy Enforcer Agent** (ensures responses don't violate company policy or legal terms) and **Problem Solver Agent** (actually addresses the user's issue). When a customer question comes in, Problem Solver drafts a reply, Policy Agent reviews it and either approves or corrects it. This is analogous to how human support has tier 1 and compliance review. Multi-agent AI could increase accuracy and safety.
4. **Creative Content Generation:** For writing stories, scripts, marketing content – multiple agents can simulate a **writer's room**. For instance, a **Plot Generator Agent** brainstorms ideas, a **Character Developer Agent** focuses on character backstories, and a **Editor Agent** ensures consistency and style. Their interplay can lead to richer content. (There's a known project "Guided Narrativ" where one AI generates and another critiques, resulting in better stories – similar principle).
5. **Planning & Task Execution (Personal Assistant):** A personal AI but with an internal team: A **Manager Agent** that breaks a user's high-level goal into tasks, then dispatches a **Tool Agent** to use APIs for each task. For example, user says "Organize my next week's agenda." Manager agent creates tasks: "check emails for invites", "schedule workouts", "set up meetings", etc., and for each task a different agent (or a same agent with different tool) executes it. This is essentially an *Autonomous AI executive assistant*. It aligns with what some frameworks like ADK facilitate – primary agent delegating to sub-agents [79] [68].

For each use case above (or ones you come up with), outline how agents would interact. This thought exercise bridges abstract frameworks to concrete designs. It also prepares you for the next weeks where you might implement one of these.

Case Study – Google's Weather Agents: To reinforce, recall Google's blog example: **WeatherAgent** delegating greeting tasks to **GreetingAgent** [68] [80]. The WeatherAgent didn't try to do everything – it knew if user says "Hi", pass it to GreetingAgent. This is a simple multi-agent case but very illustrative: specialization leads to simpler sub-tasks and the overall system is more robust (the weather logic is separate from greeting logic). This pattern can be seen in many use cases (detect the user's intent and route to the appropriate specialized agent). It's essentially an agent orchestrator doing *intent classification and delegation*, which is a common requirement in complex applications.

Key Resources:
- **Google Developers Blog – *"ADK WeatherAgent example": Walks through the Weather + Greeting agent scenario in detail [68] [81], including how delegation rules are defined (in the root agent's description) [82] [80]. It's a perfect, concrete tutorial of multi-agent interaction with tools.***
- Shubhamsaboo's GitHub "Awesome LLM Apps": In the **"Multi-agent Teams"** section, browse entries like *"AI Travel Planner Agent Team"* [83] or *"AI Company Analyst Team"*. Each entry (if you find the repo or description) will describe multiple agents and their roles. It's inspiring to see how multi-agent systems are being applied (some entries even mention using CrewAI or ADK under the hood).
- **YouTube – *"Building multi-agent AI"*** (e.g., a WandB webinar): Often shows a demo of agents chatting. Seeing a visual demo of agents in a console (like one agent says: "I will do X", another says "I will do Y") makes the concept less abstract. Look for recent videos (2024/2025) where developers show off

AutoGPT-like systems; many have a multi-agent aspect.
- **Academic Paper (optional) – *"Chain-of-Agents" (Arxiv 2023)*:** This is researchy, but if interested: it explores letting multiple LLM agents handle long context by passing info along a chain [84]. It's beyond our scope to implement, but reading the abstract can broaden your view of what's possible (e.g., an agent handing off to another for different context windows).
- **DeepLearning.AI Event – *"Agents that Plan and Critique"*:** Sometimes there are recorded events or interviews (perhaps on The Batch or YouTube) discussing emergent behaviors when agents interact. These can be insightful to understand pitfalls (like two agents might get into an endless polite loop: "You go ahead." "No, you go ahead." – which is something a good framework will guard against!).

By the end of this week, you should feel comfortable imagining an AI solution that involves more than one agent. You'll have concrete ideas of *when and why* to use multi-agent setups, and how to articulate those in terms of roles and workflows. This sets the stage for actually **building** agent systems in the next weeks.

Week 7: Project – Building a Complex Single-Agent System (Tools & Function Calls)

Study Goals: Now it's time to build with agents. This week, focus on a **single-agent system that uses tools/functions autonomously** to achieve a complex objective. The idea is to create an agent (one LLM "brain") that can handle multi-step tasks by calling external functions – essentially implementing a custom version of something like the early ReAct agents (Reason+Act) but tailored to your needs. By the end, you will have built and tested an agent that demonstrates advanced tool use and reasoning, ready to be integrated or scaled up.

Project Outline: Design a **single agent with a toolkit** that can solve a non-trivial problem for a user. Some project ideas:
- **"Smart Data Analyst" Agent:** This agent accepts a user's request in natural language (e.g. "Compare our sales and expenses over the last quarter and summarize the key insights"). It then uses tools to fulfill it: maybe a `query_database` function to get sales and expense data, a `plot_graph` function to visualize it, etc. The agent should decide which tools to call and in what order, then produce a final report (possibly with an image or stats) for the user. This mimics what OpenAI's Code Interpreter plugin did – an LLM that can run Python to analyze data. You'll need to implement dummy versions of these tools (or real ones if you have data) and prompt the agent to utilize them step by step.
- **Autonomous Web Researcher:** An agent that can fetch information from the web (using, say, a `web_search` function and a `web_read` function) to answer a query like "What were the main outcomes of the G20 summit this year? Provide a brief summary with quotes." The agent would need to search for the topic, read relevant snippets, maybe search again for clarification, and then compile an answer. Essentially, you're making your own miniature version of the ChatGPT web browsing agent (before ChatGPT had an official one, people built similar with LangChain + browser API). This single agent would loop: decide a search query, call the search function, get results, decide to call `web_read(url)` on one result, and so forth until it gathers enough info to answer.
- **Task Automation Agent:** Consider a scenario like booking a meeting or ordering a product. A single agent could orchestrate multiple steps: e.g., user says "Book me a flight to London next Monday and schedule a cab to the airport." The agent might parse this and call `search_flights(destination, date)` then `book_flight(flight_id)` then `schedule_ride(pickup_time)`. Each function could return confirmation or ask for more info, which the agent handles. This is along the lines of "agent as a workflow engine." It's a single agent because one AI is making all decisions and calling all necessary tools in sequence to accomplish the user's goal.

Choose a scenario that excites you and is feasible to simulate. **Keep it scope-bound** (maybe 2–3 tools max, and a clear end goal).

Implementation Approaches: You have two main ways to build the agent logic:
- *Using a Framework:* LangChain's **ReAct Agent** or **Toolkit** can do a lot: you define a set of tools (functions) and LangChain provides an LLM-driven agent that will choose actions (tools) and use them until done. This is straightforward – you mostly configure it and perhaps customize the prompt (the default ReAct prompt pattern: *Thought* -> *Action* -> *Observation* -> *...* -> *Answer*). If you use this, study how to add stop criteria and how to give the agent memory of intermediate steps (LangChain handles memory of tool outputs automatically in the agent's state).
- *Manual Loop with OpenAI API:* For more control, you can write your own reasoning loop. For example, maintain a list of messages (like a conversation where the assistant "thinks out loud"), and after each tool execution, append the result as a system message, then prompt the model for the next step. OpenAI's function calling is actually very well-suited here: you can have the model "think" and then return a `function_call` when it decides to use a tool. The **OpenAI Cookbook** has an example of combining *GPT-4's reasoning and function calling* for multi-step tasks [85] [86]. The key is to craft the prompt to encourage chain-of-thought and to include all prior steps in each new API call so the model has context. You may also use the `assistant` role messages with a special content like: "Thought: I should search for flights. Action: search_flights('London, 2025-08-01')". This mimics the ReAct format. OpenAI's newer "reasoning model" (`gpt-4o` or similar) might even handle this more natively, as they have output types for reasoning and action [87] [88].

Focus on Tools and Functions: Ensure your agent actually *uses* the tools, not just answer from its own knowledge. To test this, you can, for instance, give the agent a question that *requires* tool use (like current date, or data it couldn't possibly know). This is similar to evaluation – see if it calls the correct function. If it fails, adjust prompts or tool descriptions. It's an iterative process to get the prompting right so that the agent understands it *must* use functions for certain info. (OpenAI's function calling allows forcing a function call by setting `function_call={"name": ...}` in the API, but ideally your agent decides itself.)

Add Complexity (Optional): If time permits, incorporate a form of **memory** or state beyond a single session. For example, your agent could store results from previous runs in a file or database and recall them. This edges into multi-turn agents, but even a single agent can have a long-running state. Alternatively, implement a simple **error recovery**: e.g., if a tool execution returns an error (flight not found), the agent can handle it (maybe try a different date). Exhibiting this kind of robustness is a plus at advanced level.

Testing: Try edge cases. For the Data Analyst, what if the data API returns nothing? Does the agent explain "I couldn't find data for X"? For the Web researcher, what if the first search yields irrelevant results – does the agent reformulate the query? Keep an eye on token usage as well; with a long chain of thought, you might hit context limits. You might need to truncate or summarize intermediate steps in a real system; just be aware of this issue.

Key Resources:
- **OpenAI Cookbook – *"How to call functions with chat models"***: Shows how to structure prompts and loop when using function calling [14] [85]. There's an example of chaining calls where the model might call one function, get info, then decide to call another. This is exactly the behavior you want.
- **OpenAI Cookbook – *"Handling function calls with reasoning models"*:** An advanced notebook demonstrating use of GPT-4 with the `functions` API to perform multi-step reasoning (with the model's own chain-of-thought preserved) [89] [90]. If you're going the manual route, this is gold: it

illustrates preserving the conversation state and ensuring the model's reasoning steps are fed back in so it "remembers" what it already did [91] [88].
- **LangChain Docs – Agents:** Specifically, read about the *"AgentType.ZERO_SHOT_REACT_DESCRIPTION"* and tool specification in LangChain. The LangChain documentation has a section where they implement an agent to use Python and Wikipedia tools to answer a question. Even if you don't use LangChain, seeing the prompt it auto-generates and how it formats tool outputs is insightful (LangChain's standard prompt format for ReAct has phrases like `Action: <tool>` and `Observation:` which structure the agent's decision loop).
- **DeepLearning.AI – *"Building Agentic RAG with LlamaIndex"* short course:** This course (with instructor Jerry Liu, co-founder of LlamaIndex) teaches building an agent that can **do tool use + RAG** [92]. It might be directly relevant if your project involves both retrieving data and then acting on it. You'll learn how to combine an index query as just another tool in the agent's toolkit.
- **Community Implementations:** Search the OpenAI Community or LangChain community for "function calling agent example" – many users have shared how they built, say, a mini ChatGPT plugin using function calls. One example: a community post *"OpenAI function calling to create a multi-step assistant"*. Reading others' experiences can highlight pitfalls (like the model sometimes ignoring the tool and answering directly – solution: adjust system prompt to remind it to use tools).
- **YouTube – *"OpenAI Function Calling Full Tutorial"* by DataLumina (2024):** A video that walks through an example of creating a JSON output and using a calculator function. Watching the debugging process (when the model returns slight format errors and how to fix them) will prepare you for similar troubleshooting in your project.

By completing this single-agent project, you set the groundwork for the multi-agent project next. You'll have dealt with the complexities of guiding an LLM through multiple actions. Many of those lessons (prompt engineering, result handling) will carry over when you add more agents to the mix.

Week 8: Project – Building a Multi-Agent System (Agents Collaborating)

Study Goals: This week, level up from a single agent to a **multi-agent system**. You will design and implement a system with **multiple agents working together** (or in sequence) to solve a problem. This will likely involve using one of the frameworks (CrewAI, ADK, LangChain, AutoGen, etc.) to manage agent interactions. By the end, you should have a prototype where at least **two or three agents** interact, and you'll have learned how to handle the communication and orchestration between agents.

Project Ideas: Consider building one of the multi-agent use cases you outlined in Week 6. Aim for 2–3 agents (to keep it manageable) with clear roles. Some feasible scenarios:
- **AI Pair Programmer:** Implement the **Code Generator + Code Reviewer** duo. Agent A writes a function based on a spec, Agent B reviews the code for errors or improvements. They can have a few back-and-forth iterations. For instance, user asks for a certain algorithm implemented – the coder agent produces code, the reviewer agent finds a bug and explains it, then the coder agent fixes it, and so on until the reviewer is satisfied. This can be done with AutoGen (they have an example conversation in their README with a user, a "helper" agent, and a "resolver" agent). Or you can orchestrate it manually by alternating calls to GPT-4 with different system prompts ("You are the Coder...", "You are the Reviewer..."). This showcases multi-agent collaboration and is very relevant to real-world AI coding assistants (Anthropic Claude 2, for example, introduced something like this internally, and projects like Camel have explored dev roles).
- **Complex Question Answering Team:** Pick a difficult, open-ended query (e.g. a business strategy question, or a detailed medical query). Set up Agent 1 as a **Researcher** (who can use a tool to gather info, e.g. via web search or a vector DB of texts), and Agent 2 as an **Analyst** or **Summarizer** (who waits

for info from agent 1 and then synthesizes an answer). The user's question is given to the Analyst who delegates to the Researcher for facts, then composes the final answer. This can be implemented with a framework like ADK (primary agent auto-delegates as in the WeatherAgent example [68] [80]) or by manually scripting a turn-based chat where the Analyst asks the Researcher for certain info, the Researcher returns it, and then the Analyst responds to the user. This tests multi-agent info sharing and is useful for any domain where fact-gathering and analysis are distinct (think: consultant (analyst) + internet researcher).
- **Multi-Agent Task Force (Planner & Executors):** Design a system where a **Manager agent** breaks a high-level task into parts and assigns each part to a specialized **Worker agent**. For simplicity, you (the developer) can simulate the environment responses. For example, user goal: "Plan a charity event." Manager agent creates tasks: (1) Find venue – assign to VenueAgent (which could be a dummy agent that picks a venue from a list), (2) Get catering – assign to CateringAgent, etc. Each specialized agent returns their result, and Manager compiles a final plan. You could implement this with Google ADK or CrewAI, as they natively support hierarchical agents and parallel task execution [39] [93]. Or do it manually by sequentially calling different agent prompts and storing the interim results. The key is that the Manager is not hardcoded – you prompt it to generate tasks, so it's flexible to any event. This pattern reflects real applications like complex workflow automation or project management via AI.

Choosing a Framework: For multi-agent, a framework can save you a lot of headache. If you use **Google ADK**, you'll get structure like `root_agent` with `sub_agents` and built-in delegation rules (like the weather/greeting scenario). If you choose **CrewAI**, you'll define Crew and Tasks as in the blog example – which can be very straightforward for sequential flows. **LangChain** also allows multiple agents but is less opinionated; it might require you to manage the dialogue between agents (unless using LangChain's experimental multi-agent manager). **AutoGen** is specifically built for letting agents talk, so that's great for e.g. the coding pair. Evaluate which framework aligns with your project idea:
- *If your agents need to talk in a free-form conversation:* AutoGen or a custom loop is ideal.
- *If you have a clear workflow with steps:* CrewAI or ADK, where you can explicitly set the sequence or parallel tasks, might be easier.
- *If you love Python and want quick setup:* LangChain with a bit of custom prompting can do 2-agent loops (you'll basically alternate `.run` calls with different personas).

Regardless of framework, you must carefully design the **prompts for each agent's role**. For example, for the coder/reviewer, create two system prompts: one that instructs an agent "You are a helpful AI coder. Your goal is to write correct code given requirements and fix errors pointed out by the reviewer." and another: "You are a strict code reviewer AI. You will point out any bugs or improvements in the given code." Provide them with the right context each turn (the code, the error messages, etc.). The quality of these role instructions will directly impact performance.

Communication Control: Decide on how agents address each other and how to stop the loop. You might give them names (Agent A, Agent B) and encourage them to refer to each other in conversation (some frameworks do this automatically). Also implement a **termination condition**: e.g., in code review, stop after X iterations or when reviewer says "Looks good." In research Q&A, stop when the analyst agent has an answer and researcher has no more info to add. Many frameworks provide an "end-of-dialogue" detection (like a message type or a token). If coding manually, you can include an "[DONE]" signal in the protocol.

Multi-Agent Debugging: This can be tricky. If something goes wrong, try to figure out which agent's output was off. One agent might misunderstand the other. Techniques: enforce format (e.g., researcher agent replies with a numbered list of facts, so the analyst can easily read them), or have a simple summary exchange ("Researcher: Here are 3 key points..."). Sometimes you might need to insert a system message mid-dialogue to correct course if they get confused or hostile. This is a normal part of aligning multi-agent dialogues.

Test Use Cases: Evaluate your multi-agent system on a few different inputs. Does it generalize? For coding, try a couple different functions; for Q&A, different questions. Observe if agents properly follow their roles. A common hiccup: an agent might stray from its role (e.g., the researcher tries to do analysis, or the analyst tries to search on its own). If that happens, reinforce the role definitions in the system prompts ("Remember: If you are the Analyst, do not do web searches, ask the Researcher."). This is analogous to how humans in teams need clarity of responsibilities – AI agents do too, via prompting.

Showcase & Document: As always, document how your agents are set up and maybe include a sample dialog in your README to illustrate the outcome. Multi-agent dialogues can be fun to read (almost like a chat transcript between AIs). This will be a standout piece in your portfolio because multi-agent systems are cutting-edge and not many have hands-on experience with them yet.

Key Resources:
- **AutoGen Paper/GitHub Examples:** The AutoGen project page or GitHub likely has example conversations. For instance, *"Example: Chat between a math problem Commander and Solver."* Look at those to understand how they structure turns and what the system prompts contain (the MSR paper on AutoGen had pseudocode for agent loops which could be insightful).
- **CrewAI Example (Content Writing):** The Medium article we referenced in Week 6 [73] [94] and CrewAI's own tutorial code show how to set up multiple agents and tasks. Use that pattern for your project if applicable – it provides a clear template of defining agents, tasks, then `crew.kickoff()`. Adapting it to your scenario (e.g., Code Planner, Coder, Tester as tasks in sequence) is straightforward.
- **Google ADK Documentation:** Google's ADK docs have templates for multi-agent structures. Specifically, look at how to define multiple agents in code and how the **LlmAgent** class can have `sub_agents` [95] [80]. The WeatherAgent example code in the blog is valuable because it shows how to encode delegation logic in the agent's description ("if greeting, delegate to greeting_agent") [81] [96]. You can mimic that style of instruction in any framework.
- **LangChain Multi-agent Example:** LangChain's documentation has a page about agents talking to agents (perhaps an example where two chatbots talk to each other). There is also a known project called **CAMEL** where two GPT agents (user and assistant) communicate to solve tasks (like coding) – essentially role-playing. You can search for "Camel AI multi-agent" or see if LangChain integrated something similar. Those examples might provide prompts that keep the conversation productive.
- **Anthropic Claude's Self-Dialogue:** Anthropic published some snippets of Claude engaging in self-dialogue (where one instance plays "assistant" and another "user" to stimulate a solution). While not a separate framework, it's educational to see how simply having two instances of an LLM with different instructions can lead to problem-solving. It reinforces that multi-agent doesn't always need new infrastructure; it can be two API calls with carefully crafted roles.
- **Forum/Reddit Discussions:** Check out r/LanguageModel or r/MLEngineering for threads on multi-agent systems. People often discuss experiences with AutoGPT, BabyAGI, etc., which, while not exactly multi-agent in our framework sense, share similar ideas (multiple "subtasks" and sometimes multiple models collaborating). You might pick up tips like how to avoid agents looping or how to maintain focus on the goal.

At this point, congratulations – you've built both a single-agent and a multi-agent advanced AI system! You've essentially replicated in miniature many components of state-of-the-art systems (ChatGPT's tool use, and multi-LLM collaboration like AutoGPT). This experience is invaluable and relatively rare, putting you well into "advanced" territory.

Week 9: Advanced Topics – Test-Time Reasoning, Chain-of-Thought & “Thinking” Models

Study Goals: In this week, step back from hands-on building and update yourself on the **latest research and techniques** that push the frontier of generative AI reasoning. Focus on concepts like **Test-Time Compute scaling**, advanced **Chain-of-Thought (CoT)** strategies, and how new models are designed for better “thinking” at inference. By the end, you should understand how smaller models can “think” more to perform better, what methods like self-consistency, tree-of-thought, etc. are, and how industry is incorporating these ideas (e.g. OpenAI’s *GPT-4 Turbo with 128k context* or Google’s *Gemini 2.5 reasoning mode*).

Topics Covered:
- **Test-Time Compute (TTC):** This refers to allowing an LLM to do more computation *during inference* (as opposed to having a static single forward pass). Read about how giving a model the ability to take multiple reasoning steps or deliberations can dramatically improve performance without retraining a bigger model [97] [98]. A key insight: *“By optimally scaling test-time compute, a smaller model (e.g. 1B parameters) with extensive thinking can outperform a much larger model (405B) on complex tasks”* [99]. Essentially, instead of relying purely on model size, we let the model “use its brain longer” on hard problems. This is analogous to a person taking more time to solve a tough puzzle. Techniques enabling TTC include **chain-of-thought prompting** (the model outputs step-by-step reasoning) [100], **self-consistency** (generate many CoT paths and pick the most common answer), and **external tools or scratchpads** (like letting the model write notes or run code).
- **Chain-of-Thought & Variants:** CoT prompting (e.g. “Let’s think step by step”) was a breakthrough for reasoning tasks. Ensure you understand it and see examples (like the classic adding two numbers with reasoning vs. directly, or logical puzzles). Then see advanced variants: **“Tree-of-Thought”** (where the model can branch into multiple thoughts instead of a single linear chain, exploring different possibilities) and **“Graph-of-Thought”** (a generalization of Tree-of-Thought allowing recombination of branches). Microsoft researchers have a paper on Graph-of-Thought in 2023, and others on tree search approaches. The idea is to overcome the limitation of one biased chain by **searching** the thought space. Another concept is **Reflexion** – where the model reflects on its answer and tries again (this is like having the model be its own critic and then improving).
- **Reasoning-Optimized Models:** Companies have started creating specialized model variants or modes for reasoning. For example, OpenAI’s **“o-Series” models** (like GPT-4o) are mentioned as reasoning-optimized, showing the chain-of-thought in the API [101]. Anthropic’s Claude has modes like “Claude Instant vs Claude 2 – and they specifically increased the context and reasoning abilities (Claude 3.7 Sonnet was noted for multi-step reasoning visible to user [102]). **Google’s Gemini 2.5 Pro** introduced a “Flash Thinking” feature [101] and topped some reasoning benchmarks [103]. Learn *why* these are separate – often, a model that’s good at reasoning might be a bit slower or trained slightly differently (maybe with supervised CoT data or fine-tuned to not rush to answers). The term **“thinking models”** sometimes refers to these – models explicitly fine-tuned to output their reasoning process or to allocate more computation per query.
- **Dynamic Inference Strategies:** Look into methods like **early stopping and self-evaluation** – e.g., the model generates an answer, then a separate pass where the model (or another model) evaluates if that answer seems correct or needs another attempt. This ties into *‘tool use vs self-contained reasoning’* – sometimes a model might know it should use a calculator for precise math instead of doing it mentally. These strategies are being actively researched. For instance, there’s the idea of **“meta-cognition”** in LLMs: the model gauges its own confidence and decides to double-check via another chain-of-thought if not confident.
- **Latest Research Bits (last 1–2 months):** This field moves fast, so check current discussion forums or newsletters (like **Sebastian Raschka’s LLM newsletter** [104] or **Lil’Log’s blog**) for any fresh

breakthroughs. As of mid-2025, some trending topics: **Modular reasoning (using multiple specialized models collaboratively)**, **Longer context via smart retrieval (beyond 100k tokens)**, and **mental models like "Reason+Act+Reflect" loops**. Also, there's buzz about **Meta's planned Llama-3** and **OpenAI GPT-5 rumors** focusing on reasoning and efficiency, but stick to concrete known info. The concept of *"Test-time optimization"* (letting the model adjust some internal parameters during inference to better suit a problem) is experimental but fascinating to note.

Industry Impact: Understanding these advanced reasoning techniques isn't just academic – they directly affect product features. For example, OpenAI's system card for GPT-4 mentioned they used a "chain-of-thought improver" in the final stage to boost performance on difficult tasks. Companies like Anthropic lean on their **Constitutional AI** (where the model self-critiques according to principles – a kind of guided reasoning to avoid bad outputs). **NVIDIA's CEO** even talked about "predicting the future is not just big models but models that think more per query" [105] – emphasizing test-time compute. Realize that as an advanced practitioner, you might be expected to know how to get the best out of models not just by fine-tuning (which was the 2010s approach) but by *inference-time techniques* (which is now).

Practical Exploration: If you want to experiment: try using the OpenAI API with **temperature=0 vs higher, and multiple attempts** to simulate *self-consistency*. For example, ask a tricky riddle 5 times with temp 1, and see if answers vary; if one answer appears 3/5 times, that's self-consistency voting. Or use the `n` and `best_of` parameters to have the API return several answers in one call, then majority vote – a hacky version of self-consistency (though not as good as doing it on reasoning traces). Another small experiment: try the prompt *"Let's think step by step"* vs not using it on a multi-step math word problem – observe the difference in the correctness of the answer. These quick tests reinforce the theory.

Key Resources:
- **Geodesic Capital Blog – *"Test-Time Compute: Thinking, Fast and Slow"* (2025):** An excellent overview of test-time compute concepts in lay terms [106] [107]. It draws analogies to human slow vs fast thinking and cites industry leaders emphasizing this shift (Ilya Sutskever and Sam Altman on the end of purely scaling parameters) [108] [109]. It also gives examples and even data points: e.g. the **1B vs 405B model result on MATH dataset** [99] and that adding a simple "let the model check work by saying 'wait'" improved a 32B model's math score significantly [110]. This is a must-read to ground your understanding with concrete numbers and trends.
- **Medium – *"Train Less, Think More: Advancing LLMs through Test-Time Compute"* (2024):** A Medium article (perhaps by an independent researcher) summarizing various approaches to enhance model thinking at inference [111]. It might cover things like self-consistency, scratchpad methods, and mention specific research papers (like "Optimizing Test-time Compute" by Google or "RRHF" etc.). Use this to get a broad survey.
- **ArXiv Paper – *"Scaling Transformerlms vs. Scaling Compute at Inference"* (2024):** If you can handle an academic paper, look at *"Scaling LLM Test-Time Compute Optimally Can be More Effective than Scaling Model Size"* [112]. The abstract and intro will likely give you the main idea (the title itself is the insight!). It might also describe a method to decide how many reasoning steps to take for a given query (perhaps based on estimated difficulty).
- **OpenAI & Anthropic Model Card Excerpts:** Skim the system card or model card for GPT-4 or Claude 2. These sometimes mention the use of chain-of-thought in training or the model's ability to internally reason. For example, Anthropic released a doc about Claude's "Constitutional AI" which involves the model reflecting on outputs and adjusting them – an inference-time alignment technique. Reading these will show how advanced reasoning ties into safety and alignment (e.g. a model thinking more might catch itself before saying something incorrect or disallowed).
- **Two Minute Papers – *"AI learns to think before answering"*:** TMP likely has a video on chain-of-

thought or tree-of-thought papers. These short videos help solidify *why* a technique works with visuals. Search their channel for terms like “Chain-of-Thought” [113] or “Tree-of-thought.”
- **Emerge AI Atlas – *“Test-Time Compute in Generative AI”*:** A report or article that might illustrate how an LLM can generate multiple thoughts and evaluate them [114]. It could include examples like “the model generates intermediate thoughts or multiple candidates and then evaluates,” which is basically what we discussed [115]. Such articles often have simple diagrams or pseudo-code that make these concepts clear.
- **Current News/Blogs (July 2025):** Look at the latest posts on O’Reilly, Hugging Face blog, or Multimodal.pub. For instance, Hugging Face may have a blog on “Long context and planning with transformers” or something about their newest 100k context model *StarCoder*. Keeping an eye on these sources ensures you catch any new buzzwords or techniques (like recently, *“Q* heuristic for chain-of-thought” was on arXiv – niche, but interesting).

By the end of Week 9, you’ll have a cutting-edge perspective. This knowledge will not only make you conversant in the latest AI discussions (great for interviews or networking), but also inform how you might improve your own projects (maybe you’ll try adding a self-check agent or use a smaller model with multiple passes instead of paying for a bigger model).

Week 10: Latest Platforms, Tools & Future Trends (ChatGPT Agent, Claude Code, and Beyond)

Study Goals: In this final week, round out your intermediate-to-advanced journey by exploring the **very latest platforms and features** announced in the last couple of months (circa mid-2025). You’ll investigate OpenAI’s new **ChatGPT Agent** capabilities, Anthropic’s **Claude Code SDK**, and other state-of-the-art developments (like expanded context windows, multimodal integrations, etc.). Also, take time to reflect on **ethical and societal implications** of these powerful AI systems and peek into what’s next (regulatory trends, open-source movements, etc.). By the end, you’ll be up-to-date and have a forward-looking understanding of generative AI.

Topics Covered:
- **OpenAI ChatGPT Agent (July 2025 release):** OpenAI has introduced an **agentic version of ChatGPT** that can “think and act” on your behalf [116] [77]. Learn what it can do: it can use a “toolbox” of skills on its own, like browsing the web, running code, accessing APIs, and even managing files – all within a controlled sandbox [117] [118]. Essentially, ChatGPT can now autonomously complete complex tasks start-to-finish (the example given: “plan and buy ingredients to make Japanese breakfast for four” – ChatGPT will browse recipes, make a shopping list, go to Instacart and fill the cart, etc.) [77] [119]. This is exactly the kind of multi-step agent we’ve been learning to build! Understand how OpenAI implemented this safely: ChatGPT Agent uses a virtual browser with user permissions, and it **asks for permission** before taking actions like purchases [120]. It also blends Operator (the old browser plugin) and code execution into one system [119]. For you, the key takeaway is that the leading products are now embracing *integrated agentic behavior*. This validates the importance of what you’ve learned. From a practical view, learn *how to use* ChatGPT Agent (if you have ChatGPT Plus or Enterprise): how to turn on Agent mode from the UI [121], and what types of tasks it excels at. Also note limitations: it’s early and might be slow or constrained to certain websites and APIs for now.
- **Anthropic Claude Code:** Claude (Anthropic’s LLM) has launched a specialized offering called **Claude Code** and an SDK [122]. This is Anthropic’s answer to ChatGPT’s Code Interpreter and GitHub Copilot X. The Claude Code SDK allows Claude to integrate deeply with a developer’s environment – it can access the file system, run as a subprocess, integrate with CI/CD pipelines, etc. [122] [123]. It’s built around Anthropic’s **Model Context Protocol (MCP)** [123], meaning developers can inject *live context* (like your whole codebase, or real-time data) into Claude’s sessions. The SDK supports Python, TypeScript, and

CLI, enabling automation of code review, refactoring, etc. [122]. Essentially, you could have Claude automatically open a pull request and suggest edits across an entire repository – a leap beyond just single-file code completion. As an advanced learner, consider what this means: we're moving towards **AI DevOps** – AI not just suggesting code, but operating on code repositories and workflows. This raises new considerations like how to trust and verify AI changes (the InfoQ article notes some devs worry about juniors using it to auto-fix without learning [124]). But also it promises huge productivity leaps (one quote: *"Claude Code's ability to understand context and generate production-ready code has transformed my workflow"* [125]). Action item: if possible, try out Claude on a coding task via their API or the free Claude 2 interface, to see how it differs from ChatGPT in explaining and writing code. If you can access the Claude Code beta, experiment with the SDK in a sandbox repo.
- **Other Notable Developments:**
- **Context Window Explosion:** GPT-4 now offers up to 128k token context in limited availability (and other models like Claude 2 handle 100k). This means entire books or massive logs can be input. Tools like **NotebookLM** (by Google) leverage this to have an AI read and summarize whole research papers. As an advanced user, know how to use large context effectively (embedding chunk + retrieve is still more reliable for truly huge data, but sometimes just throwing a whole document in works).
- **Multimodal Progress:** After images and text, now **voice** is a big focus. OpenAI's ChatGPT mobile app introduced voice conversation (using new text-to-speech models). Google's Gemini is multimodal by design (possibly able to output images or handle video). Meta's CM3leon model generates images from text. Keep an eye on how these modalities converge – e.g., an agent that can see (images) and talk (voice) and act (tools) all-in-one. The future assistants likely have all these; already, Microsoft's **Jarvis** (HuggingFace demo) chains vision, audio, and LLMs.
- **Regulation & Governance:** Advanced practitioners should be aware of the AI policy landscape. The EU's **AI Act** is coming, likely requiring transparency and risk checks for AI systems. The US has had hearings with OpenAI's CEO calling for licensing of advanced models. On the flip side, there's a push for **open-source** – Meta released LLaMA 2 (70B) openly in 2023, and by 2025 there are even some open models approaching GPT-3.5 level that you can fine-tune yourself. This could influence your choices in building products (when to use closed API vs open model for privacy/cost). Since you're focusing on OpenAI primarily, note that OpenAI is also launching tools to accommodate enterprise needs (data privacy assurances, and possibly allowing fine-tuning or plug-ins for GPT-4).
- **New research to watch:** *"Generative agents"* (Stanford paper where AI characters lived in a Sims-like world) is a fun one – it combined memory, planning, and multi-agent interaction, akin to what you learned. Also, *"Self-healing"* AI workflows where the AI can diagnose its failures (e.g. if a tool fails, try something else) – some frameworks like Dust or Cognosys are exploring that.
- **Tool Networking:** Interesting concept – letting one agent not just use tools, but spin up new tools or even new agents on the fly (there's the idea of an agent recursively spawning helpers; some frameworks support that). It's like dynamic AI organizations. While experimental, you essentially did a fixed version of that in Week 8 by pre-defining multiple agents.

Career and Learning Trajectory: Think about how to keep learning after this roadmap. The AI field will continue to evolve, so cultivate habits like: following top AI researchers on Twitter/LinkedIn, subscribing to newsletters (e.g. The Batch, ImportAI), joining communities (Discords for LangChain, etc.), and possibly contributing to open-source projects. Also consider **certifications** or credentials for the skills you gained: for example, **DeepLearning.AI** offers certificates for their short courses (you did several) – having those on your resume signals your up-to-date knowledge. Hugging Face also has a new LLM course certificate. And if you built cool projects, showcase them on GitHub with a nice README and maybe a short demo video or screenshots. Real-world proof of skill often trumps certificates.

Key Resources:
- **OpenAI Announcement – *"Introducing ChatGPT Agent"*** [116] [117] and related TechCrunch/Reuters articles [126] [127]. These outline the feature set (e.g., *"ChatGPT will intelligently navigate websites, prompt*

you to log in when needed, run code, and even deliver slideshows/spreadsheets summarizing its findings." [128]). The Reuters piece gave an example of ordering an outfit with given requirements – showcasing transactional capabilities [127] [129] . Reading these will help you understand how far OpenAI has taken the agent concept in a production setting (and also the *safety mitigations*, like always asking permission [120] , which is crucial for responsible use).
- **OpenAI's ChatGPT Agent System Card:** OpenAI usually releases a technical/system card for major updates (the link in search results [130] might be it). Skim it for details on how they ensure the agent doesn't go rogue – likely they have an internal supervisor model that stops unsafe actions. This is interesting as an advanced topic: the idea of a *'governor agent'* overseeing the main agent (some frameworks like Hugging Face Transformers Agent have something akin to this too).
- **Anthropic Claude Code Blog/InfoQ** [122] [123] : The InfoQ article we cited is very informative, giving context on why the Claude Code SDK matters (lack of context in coding assistants was a problem, and this addresses it by hooking Claude into the dev environment tightly [123]). Also check Anthropic's own blog for "Claude Code" – they had a post *"Claude Code: Deep coding at terminal velocity"* (just as OpenAI had one for Code Interpreter). That likely outlines use cases like reading an entire codebase to answer questions in seconds [131] (Anthropic said Claude can map a whole codebase).
- **Developer Blogs:** After these launches, devs often blog "I tried ChatGPT's agent on task X, here's what happened" or "First impressions of Claude Code". Reading one or two such blog posts gives practical insight (e.g., someone might note that ChatGPT Agent was great at web browsing but struggled with a specific site's login). These real-world tests teach you about limitations and potential creative uses.
- **High-Quality Courses on New Topics:** Consider taking an advanced course or certificate now that you've self-studied. Two recommendations: **AI & Law/Policy** (if interested in the legal/societal side, check out courses from maybe Stanford or DeepLearning.AI's *"AI for Good"* offerings) and **Advanced Prompt Engineering/AI Programming** (some platforms offer courses on building AI plugins or custom agents – e.g., there's a Coursera course on building ChatGPT plugins). Since you're already hands-on, these can fill any gaps and give credentials.
- **Community and Open-Source Engagement:** Join the **LangChain Slack or Discord**, the **OpenAI API forum**, or the **Hugging Face Hub discussions**. Being active in these will keep you updated daily. For instance, when someone finds a way to jailbreak an agent or a new feature in OpenAI API (like function calling got an update), you hear it there first. Also, consider contributing to a project like an "Awesome Generative AI" list or a small fix to LangChain – it's a great way to solidify knowledge and get recognition.

Ethics & Final Thoughts: As a concluding note, reflect on the ethical side (which was also Week 12 of the beginner roadmap). Now that you wield advanced AI techniques, always consider **bias, hallucinations, and misuse**. For example, an agent that can browse and execute code is powerful – it could be misused to crawl for vulnerabilities or generate disinformation at scale. As an AI builder, commit to using these skills responsibly: implement those permission checks, content filters, and know the limits of your systems (be transparent about them). This not only is ethically right but also aligns with the direction the industry is taking (AI Ethics and safety is a growing field, and expertise there is a plus).

Congratulations on completing the intermediate-to-advanced generative AI roadmap! You've gone from using off-the-shelf models to customizing and orchestrating complex AI systems. With these skills and knowledge, you're well-equipped to build state-of-the-art generative AI applications and to continue learning as the field evolves. Good luck, and happy building!

Sources: [32] [132] [100] [68] [117] [128] [122] [101]

---

[1] Introducing Gemini: our largest and most capable AI model
https://blog.google/technology/ai/google-gemini-ai/

[2] [3] Google introduces Gemini 2.0: A new AI model for the agentic era
https://blog.google/technology/google-deepmind/google-gemini-ai-update-december-2024/

[4] [5] [8] [9] [10] [12] [13] Introducing Structured Outputs in the API | OpenAI
https://openai.com/index/introducing-structured-outputs-in-the-api/

[6] How to Use the OpenAI Playground: A Step-by-Step Guide for 2024
https://smartblogger.com/openai-playground/

[7] Quickstart with Agent Development Kit | Generative AI on Vertex AI
https://cloud.google.com/vertex-ai/generative-ai/docs/agent-engine/memory-bank/quickstart-adk

[11] When should I use function calling, structured outputs or JSON mode?
https://www.vellum.ai/blog/when-should-i-use-function-calling-structured-outputs-or-json-mode

[14] How to call functions with chat models - OpenAI Cookbook
https://cookbook.openai.com/examples/how_to_call_functions_with_chat_models

[15] [16] [17] OpenAI Function Calling Tutorial: Generate Structured Output | DataCamp
https://www.datacamp.com/tutorial/open-ai-function-calling-tutorial

[18] [19] Building Systems with the ChatGPT API - DeepLearning.AI
https://www.deeplearning.ai/short-courses/building-systems-with-chatgpt/

[20] [27] [28] [29] Retrieval Augmented Generation (RAG) Course - DeepLearning.AI
https://www.deeplearning.ai/courses/retrieval-augmented-generation-rag/?utm_campaign=rag-launch&utm_medium=headband&utm_source=dlai-homepage

[21] [22] [24] [30] Introduction to RAG - LlamaIndex
https://docs.llamaindex.ai/en/stable/understanding/rag/

[23] Build a Retrieval Augmented Generation (RAG) App: Part 1
https://python.langchain.com/docs/tutorials/rag/

[25] Advanced Retrieval-Augmented Generation (RAG) Techniques
https://thenewstack.io/advanced-retrieval-augmented-generation-rag-techniques/

[26] Advanced RAG Techniques: an Illustrated Overview - Towards AI
https://pub.towardsai.net/advanced-rag-techniques-an-illustrated-overview-04d193d8fec6

[31] Retrieval-Augmented Generation (RAG) using LangChain ...
https://pub.towardsai.net/introduction-to-retrieval-augmented-generation-rag-using-langchain-and-lamaindex-bd0047628e2a

[32] [83] [132] GitHub - Shubhamsaboo/awesome-llm-apps: Collection of awesome LLM apps with AI Agents and RAG using OpenAI, Anthropic, Gemini and opensource models.
https://github.com/Shubhamsaboo/awesome-llm-apps

[33] [70] [71] [73] [74] [75] [76] [94] Understanding Agentic Frameworks | CrewAI | Medium
https://mahimairaja.medium.com/understanding-agentic-frameworks-crew-ai-224f62ec711f

[34] [35] [36] [37] [38] [39] [40] [43] [44] [55] [60] [61] [62] [63] [64] LangGraph vs ADK: A Developer's Guide to Choosing the Right AI Agent Framework | by Jitendra Jaladi | Jun, 2025 | Medium
https://medium.com/@jjaladi/langgraph-vs-adk-a-developers-guide-to-choosing-the-right-ai-agent-framework-b59f756bcd98

41 42 56 57 68 69 72 79 80 81 82 93 95 96 Agent Development Kit: Making it easy to build multi-agent applications - Google Developers Blog
https://developers.googleblog.com/en/agent-development-kit-easy-to-build-multi-agent-applications/

45 46 47 48 49 58 59 65 Introduction - CrewAI
https://docs.crewai.com/en/introduction

50 What is crewAI? - IBM
https://www.ibm.com/think/topics/crew-ai

51 Comparing AI agent frameworks: CrewAI, LangGraph, and BeeAI
https://developer.ibm.com/articles/awb-comparing-ai-agent-frameworks-crewai-langgraph-and-beeai/

52 53 54 AutoGen - Microsoft Research
https://www.microsoft.com/en-us/research/project/autogen/

66 67 AI Agents in LangGraph - DeepLearning.AI
https://www.deeplearning.ai/short-courses/ai-agents-in-langgraph/

77 78 116 117 118 119 120 121 128 Introducing ChatGPT agent: bridging research and action | OpenAI
https://openai.com/index/introducing-chatgpt-agent/

84 Chain of Agents: Large Language Models Collaborating on Long ...
https://arxiv.org/html/2406.02818v1

85 87 88 89 90 91 Handling Function Calls with Reasoning Models | OpenAI Cookbook
https://cookbook.openai.com/examples/reasoning_function_calls

86 o3/o4-mini Function Calling Guide | OpenAI Cookbook
https://cookbook.openai.com/examples/o-series/o3o4-mini_prompting_guide

92 Building Agentic RAG with LlamaIndex - DeepLearning.AI
https://www.deeplearning.ai/short-courses/building-agentic-rag-with-llamaindex/

97 98 99 100 101 102 103 106 107 108 109 110 Test-Time Compute: Thinking, (Fast and) Slow | Geodesic
https://geodesiccap.com/insight/test-time-compute-thinking-fast-and-slow/

104 The State of LLM Reasoning Model Inference - Ahead of AI
https://magazine.sebastianraschka.com/p/state-of-llm-reasoning-and-inference-scaling

105 Scaling test time compute is back ? : r/singularity - Reddit
https://www.reddit.com/r/singularity/comments/1hpwr3y/scaling_test_time_compute_is_back/

111 Train Less, Think More: Advancing LLMs Through Test-Time Compute
https://medium.com/electronic-life/train-less-think-more-advancing-llms-through-test-time-compute-a46832e973e9

112 Scaling LLM Test-Time Compute Optimally Can be More Effective ...
https://openreview.net/forum?id=4FWAwZtd2n

113 Chain-of-thought prompting - Explained! - YouTube
https://www.youtube.com/watch?v=AFE6x81AP4k

114 115 Test-Time Compute in Generative AI: An AI Atlas Report
https://www.emerge.haus/blog/test-time-compute-generative-ai

122 123 124 125 Anthropic Releases Claude Code SDK to Power AI-Paired Programming - InfoQ
https://www.infoq.com/news/2025/06/claude-code-sdk/

126 OpenAI launches a general purpose agent in ChatGPT - TechCrunch
https://techcrunch.com/2025/07/17/openai-launches-a-general-purpose-agent-in-chatgpt/

[127] [129] OpenAI unveils ChatGPT agent to handle tasks as AI apps evolve
https://www.reuters.com/business/openai-unveils-chatgpt-agent-handle-tasks-ai-apps-evolve-2025-07-17/

[130] ChatGPT agent System Card - OpenAI
https://openai.com/index/chatgpt-agent-system-card/

[131] Is Claude Code About to Disrupt the Billion-Dollar AI Coding Tools ...
https://medium.com/data-science-collective/is-claude-code-about-to-disrupt-the-billion-dollar-ai-coding-tools-market-813a67b8a5f0