INTRO TO 'AI'

A Bit Of Historical Perspective:

The term 'artificial intelligence' was coined at Dartmouth in 1956. During the 1990s–2000s, the 1st practical 'narrow AI' systems were built to perform speech & character recognition. In 2012 the 'deep learning' AlexNet project dramatically improved upon previous image recognition benchmarks. In 2016 Google DeepMind's AlphaGo system outperformed the top human professional Go player Lee Sedol. In 2017 'Transformer' architecture was introduced in a paper titled 'Attention Is All You Need', by a group of researchers at Google. Transformers now enable the majority of well known AI tools.

LLMs, or 'Large Language Models' are the most prevalent AI systems in mainstream use today. ChatGPT, Claude, Gemini, Grok, Deepseek, Kimi, GLM, Minimax, and Qwen are all LLMs.

'Diffusion' models are a different architecture which is widely used to generate images, video and audio, are often combined with LLMs to create systems with 'multimodal' (text and other media-related) capabilities.

To learn some more about the historical foundations and culture of AI technology, search Google for info on Geoffrey Hinton, Yann LeCun, Yoshua Bengio, Demis Hassabis, Jürgen Schmidhuber, Ashish Vaswani, Andrew NG, Fei-Fei Li, and Andrej Karpathy.

LLMs, Tokens:

LLMs were initially engineered to do only one thing: predict the next 'token' (part of a word) which should most likely follow a series of previous input tokens. LLMs are trained to do this by being fed enormous corpuses of text, including scrubbed versions of all the writing on the Internet, enormous collections of books, and other publications.

LLM systems use a 'tokenizer' which transforms a user's input into numerical values. They break words into smaller components and characters to reduce vocabulary size and to handle rare words, for better generalization. In English, 1 token is equivalent to roughly 0.75 words. Modern tokenizers convert a vocabulary of typically 30,000-128,000 pieces of words, punctation, etc. into numerical representations.

LLM systems process the converted tokenized numbers through a series of mathematical computation stages, to output a single predicted next token response which is most likely to be correct in context. That new token/word is then appended to the input text, and the entire process is repeated from the beginning, to produce a sequential list of useful output tokens.

Using a trained LLM system to produce such predicted output is called 'inference'.

Training, Parameters, GPUs, Nvidia, Cuda, and Python:

LLMs are a type of 'neural network', trained using 'deep learning' methodologies. The training process involves using 'GPU' hardware (Graphic Processing Units), which are chips originally designed to speed up graphic display performance in video games. The sorts of intensive quadratic mathematical computations needed to build/run neural networks are similar to those needed to quickly calculate/render real time visual 3D scenes.

Nvidia is the company which has been at the center of the GPU hardware revolution, and their 'CUDA' software system is typically used as the foundation for building most common types of modern AI tools. The Python programming language has become the default interface used to manipulate and interface with CUDA systems (this has helped Python become the most popular general purpose computer programming language in modern use).

Inside LLMs, there are billions-trillions of 'parameters', which can be thought of as a massive set of knobs that collectively shape how an AI model transforms input tokens into output tokens. Parameters are like the digital analogues to neurons in biological brains. They connect together to form a computational mesh which processes data based on the combined activity of their collective settings.

By going through an enormous number of repetitive 'back propagation' training cycles, LLM parameters get set to particular numerical values, or 'weights'. Weights are adjusted to progressively improve the transformer's ability to predict the next expected token in any input sequence.

Python libraries such as PyTorch, Tensorflow, Keras, Scikit-learn, and JAX are used to perform this training, in parallel on multiple extremely powerful GPUs. This tremendously heavy computational process would take millions of years if performed on a typical home computer CPU, and it requires a huge amount of electricity to complete.

During training, a 'loss function' measures the level of output error, and the process continues to update parameters until the LLM can generate reliable predictions about virtually any combination of input characters.

How LLMs Are Able To 'Reason' Intelligently:

LLMs simply process human language. The idea of how related meanings exist between words, is very important in the way LLMs actually end up generating useful inference output.

Transformer systems don't just return the statistically most likely next word in a sequence. They instead look at all the words in an input series, and weigh their relationships against the complex multi-dimensional set of relationships which exist between every other word in the input text. The parameter weights which were set

during training affect how the LLM processes these relationships between input text tokens. Those weights reflect how likely a predicted next output word is, according to how every word in the LLM's training texts were related to one another.

It's important to understand that LLMs don't use databases or files to store information 'learned' during the training process. Instead, the knowledge they're able to output exists within them, and is retrieved solely according to the likelihood of words being output in a predicted order, in response to a given input text.

So, the inference process produces output words which essentially 'remember' information from all the texts in the training data, by re-assembling that information as predicted next words. This is a hard concept to grasp. It's mind-bending to realize that such a stochastic method of data compression actually works, but the paradigm has been engineered and improved to perform incredibly well.

Here's one example which may help you better grok how LLMs work:

If you feed an LLM all the text of a newly authored murder mystery novel, when it reaches the end point of the story that reads 'and the murderer is...', the LLM won't just predict context-insensitive next words from a list that are simply the most statistically common throughout its training texts. Instead, the LLM is designed to return the words which are most likely to make sense within the context of meanings formed by the complex relationship mesh of tokens/words that have been combined throughout the unique text of the new mystery novel.

For example, 'the butler' may be the most common next words in the largest percentage of mystery novels in the LLM's training material - but those words may not be correctly meaningful for content of the current mystery novel text. The LLM will instead choose next words which make sense within the combined *meaning of the new story text - 'Professor Thomas', for example, if the words in the new story were arranged in a way to indicate he is the killer.

The choice of a predicted next word in inference output is therefore effectively influenced by the *meaning formed from all the relationships of all the particular words, in the particular order they appear throughout the new novel text, and inferred output is based on how the LLM parameters have been set, according to all the text relationships in its pre-training corpus.

So, the predicted next word actually represents what is effectively a thoughtful, meaningful choice, or what humans might consider to be an *understanding of the textual material, all supported by the massive universe of words and their intertwined relationships, in both the training and the input texts.

It's hard to believe this actually works reliably in any way, for any input text, but the complicated mathematical mechanisms which make up an LLM's internal machinery, combined with the truly stupendous volume of data and statistical processing involved, actually do lead to a well established paradigm which is astoundingly effective at inferring 'intelligent' output.

Base Models And Fine Tuning:

One of the important things to understand about LLMs is that when they finish their first stage of training, called 'pre-training', on the massive corpus of input text they loop over - reaching the point where parameters have been set so that the loss function consistently returns an acceptably low error value, the LLM's neural net 'brain' has basically finished its first stage of formation.

All any LLM can do at that point is produce a stream of sequential next tokens (words). A newly pre-trained LLM will simply ramble on through continuations of any text which is fed into it. At this point in its development, the LLM doesn't know how to answer questions, how to chat, or how to interact in any other meaningful way with humans.

This raw 'base' or 'foundation' model then goes through another stage of training, in which it learns to be useful. Typically this involves feeding in many millions of human-curated question and answer pairs, to influence the sort of output it produces, given certain types of expected input.

The transformation of Base LLM models into useful, conversational chatbots is called 'fine-tuning'. This stage of training involves processes such as 'SFT' (Supervised Fine-Tuning, also called Instruction Tuning) to teach it how to respond to tasks. This is typically followed by 'RLHF' (Reinforcement Learning from Human Feedback), and/or 'DPO' (Direct Preference Optimization) to 'align' responses with human values, to improve safety and helpfulness.

The fine-tuning process turns pre-trained pattern-recognition abilities into interactive, useful agents. Models learn to follow user queries and act as an assistant, to summarize text, to work in the 'chatbot' format, to produce more reliable, guided, safe, appropriate, helpful content, aligned with human preferences. RLHF uses human annotators to rank multiple model-generated answers from best to worst. A separate 'Reward Model' is trained to predict those human preferences. With PPO (Proximal Policy Optimization), chatbots are trained to maximize the reward from the Reward Model, effectively learning what humans prefer. DPO is a simpler alternative to RLHF which skips the reward model stage and aligns the LLM directly on human preference data. PEFT (Parameter-Efficient Fine-Tuning) uses techniques such as 'LoRA' (Low-Rank Adaptation) to update only a small subset of the model's parameters rather than the whole model - it's often used to train an existing model on a particular data set, for example, operation manuals and/or routines that are important to a particular company, or for use in some specialized process that requires knowledge of more focused proprietary data.

Thinking Models

By the end of 2024, OpenAI (the company which created ChatGPT), and other companies such as Deepseek (which created one of the first significant free, open source LLMs) began to release 'thinking' models. Base LLM models are trained to become thinking models (also called 'reasoning' or 'inference-time' compute models) by moving from

simple next-token predictions, to generating structured, intermediate 'Chain-of-Thought' steps, before returning final text answers to the user. This involves specialized post-training on step-by-step logic data, as well as additional reinforcement learning steps which force the LLM to examine and compare its own output, read its own answers as input, and evaluate that cycle using specialized <think> tokens. Chain of thought reasoning helps improve the quality of results and cuts down on 'hallucinations', or result text which LLMs generate simply because the output makes sense as related words, even though it may be factually incorrect or entirely made up.

Emergent Capabilities:

Because LLMs are trained on trillions of words which have been pulled from curated, knowledgeable source material (cleaned of garbage text), what we've seen develop is an almost magical 'emergent' ability for them to *reason in useful ways.

The words they generate make sense, because they construct output texts based on the relationships between words in their training materials, and that text contains inherent knowledge developed by humans (text is a way humans store/represent that knowledge).

This leads to an effect in which LLMs actually appear to demonstrate educated intelligence. They can actually be used to *accomplish novel, complicated, flexible work goals which were only previously achievable by human intellect. But this doesn't necessarily mean they're conscious - that's a separate characteristic.

The most important thing to understand about LLMs' emergent abilities to 'reason' is that engineers don't intentionally 'program' any such logical ability into a base LLM, at least not in the same way that traditional computer software is programmed with algorithmic logic. No one writes deterministic 'if this do that' rules into the system (at least not into a base LLM). In fact, during the fine tuning stages, engineers basically input more curated data in the training corpus, to *influence how the LLM system learns to interpret input text. The LLM system continues to 'learn' more from curated fine-tuning examples, in general ways that affect how output text should be shaped, but there are no hard and fast deterministic rules programmed directly into the system parameters, as they are defined in traditional software systems.

The truth is, emergent capabilities are not something anyone in the AI industry fully understands. They simply appear naturally within an extraordinarily complex system that was initially engineered solely to predict the next token after any given input stream of tokens.

It's a bit mind blowing to understand that this is the case, but it's the truth. Reasoning abilities in any LLM are not built, but 'grown'. Engineers simply build these 'brains' to predict next tokens, and that, somehow, leads to an ability to reason intelligently, almost in the same way biological brains seem to have just evolved naturally, without guidance, to become intellectually capable.

It's critical to understand that by building LLMs, humanity has stumbled upon a recipe to create thinking machines which turn out to be much more intelligently capable than they were specifically engineered to be. This is very different from how traditional computing systems are engineered to solve problems.

Explained in another way, the 'multi-dimensional' relationship of trained parameters connecting all potential tokens processed by an LLM, via multiple pathways through every other token, leads to a nearly impossible to conceive level of complexity inherent in the web of relationships represented by the parameter settings which make up an LLM system, all set by an almost incomprehensible volume of repetitive training. This deeply complex architecture of interconnected neurons just ends ups being capable of displaying characteristics of intelligence, in ways which were originally unexpected.

What has become absolutely apparent, as LLM technology has matured, is that the more LLM systems get scaled up in parameter count, the more complex their emergent abilities become. The predictable notion that larger LLMs display more complex emergent capabilities, is referred to as a 'scaling law'. The rate of progress at which scaling laws have materialized progress during the past few years, has been staggering, especially considering that this progress has required billions of dollars in funding to ramp up development of infrastructure and energy resources.

LLM systems went from barely being able to produce sentences that made meaningful sense, to solving intellectual tasks which exceed the ability of human PHDs in many cases, all since ~2019.

Tool Use:

One issue which was common with the first popular LLMs was that they were limited to accessing only the data encountered in their pre-training corpus text. When users submitted prompts about recent news events, for example, LLMs would respond with output such as 'this question involves data and/or events which occurred after my last training cut-off date' (the models were fine tuned to provide that response).

In order to better handle this situation, LLMs began to be trained to use search 'tools' and RAG (Retrieval-Augmented Generation) routines which enabled looking through fresh new text, to reason about information that wasn't included in the model's original pre-training data. Additionally, models were trained to use deterministic tools such as Python code interpreters installed on the LLM server machine, and to more effectively perform 'in-context learning' to understand input data. This sort of engineered tool use has dramatically improved upon the capabilities of all those early LLM systems.

One of the most important emergent capabilities of LLMs, from the very beginning, has been their uncanny ability to write actually useful software code. It was quickly learned that early LLM models could reason through solutions to simple games such as Towers of Hanoi, but only at the most rudimentary level, involving just a few steps. Reasoning which required too large a scope of connected thoughts (i.e., too many plate moves in the Towers of Hanoi game) would be more than the model could

chew on all at once, so it couldn't complete the problem.

If, however, users asked the same model to write code to solve the Towers of Hanoi game algorithmically, the LLM could handle games of virtually endless complexity. In this way, providing code execution tools the system could run and interact with, is one of the most important ways LLMs have been enabled to solve more complex problems.

Agentic Systems:

By combining the emergent ability to make reasoned decisions, to use tools (typically via a system called MCP (Model Context Protocol)), and to write code which operate those tools, LLMs began to evolve into 'agentic' systems.

Agentic systems, or 'Agents', which make use of all this reasoning, code writing, and tool-use capability, have begun to gain real ability to interact with the world, entirely on their own. Agentic systems which can complete long, complicated tasks in the real world, represent the state of the art in 2026 LLM technology.

Earlier LLM chatbots simply output text which could be useful to a user. To get work accomplished, the human user needed to interact with some sort of real life work environment, and explain step by step to the LLM, in a chat session, what was happening in that work environment.

Agentic systems improved on that situation because they can be given access to tools which actually get work accomplished in those real life work environments, all on their own. The use of connected tools enables LLMs to 'put hands on' real world systems, so they're able to interact with all sorts of private accounts, web sites, databases, etc., which have actual consequences in the brick and mortar world.

Agentic systems such as Claude Code, OpenClaw, Goose and others can be set up with a collection of MCP tools, and a set of 'recipes' which explain how to solve specific problems, to go out in the world and make things happen beyond the digital domain.

An Example Of An Agentic Software Development Workflow, Compared To A Chat Bot Workflow:

To understand more about how agentic systems are useful, it can be helpful to understand, for example, how software development methodologies have changed over the past few years. When ChatGPT and the first generation of LLMs initially appeared, software developers began to realize they were capable of generating useful code.

To make use of that capability, developers needed to work in their traditional IDE (Integrated Development Environment) to build applications, and whenever they wanted to write some code for a particular functionality, they would provide all the context of that functionality to the LLM - everything about how connected pieces of the application code worked - in a chat conversation, where the LLM would then

output some useful code that the human developer could paste back into their IDE.

Typically, this process involved providing lots of written context to the chat bot about the purpose of the code, and often lots of specific details about arguments/parameters (data values) sent from other functions in the code base, along with what sorts of values needed to be returned from the newly generated function, and other details. Even though this process required a lot of verbose explanation, and was often error prone, it still improved developer productivity.

During the initial phase of ChatGPT's first popularity, one common practice when it came to using LLMs to write code, was to make use of its ability to learn in-context, to write code, for example, for 'REST API's (web based Application Programming Interfaces), which are a core way developers connect applications across network and Internet connections. REST APIs are like URLs (web addresses) which take in some data (parameter values), and return some useful processed data to the system that connects with the API. Developers use REST APIs to connect with virtually any sort of available published service, such as banking systems, communications systems, database systems, etc.

Since there are an unlimited number of potential REST APIs made by vendors of all sorts around the world, LLMs would only be pre-trained on how to use the most popular of them. Documentation for many secure APIs, for example, would often be shared only behind secure logins, so that text documentation never made it into the pre-training corpus of any LLM.

Luckily, LLMs are fantastically good at understanding technical docs such as API specifications, and if a developer pastes the required documentation into a chat bot, that bot's LLM would typically be able to write working code based on the pasted documentation (!!!), which it learned about only in the context of the chat, and by understanding a carefully formed question submitted by the developer.

The issue with all this was that in-context learning is not remembered by any LLM, beyond the context of a transient conversation. The parameters (knobs) inside an LLM model are not affected by learning acquired during the context of any conversation. In fact, the way chat bots work, in the most basic sense, is that you start a conversation, and the LLM responds. Then when you ask your next question, the text of the entire previous conversation (your question(s) and the LLM's response(s)), are all simply fed back into the LLM, so it can read the entire existing conversation text. It then continues responding to the meaning of that entire context. Otherwise, without the help of such software techniques, LLMs have absolutely no inherent 'memory' whatsoever.

The problem with all this is that LLMs all have a limited amount of context that they can deal with. The first version of ChatGPT (GPT 3.5) could only deal with a 'context window' of about 4000 tokens - which means it could pay attention to less than a single page of text in any chat interaction. What that meant for a software developer working with some API documentation, for example, was that only the documentation for perhaps a single function, the parameter(s) it accepted, and some basic description about the intended purpose of the output code, could be typed into a conversation, before the LLM got overwhelmed and couldn't keep everything it

needed to 'remember' about the task, within its working attention limit - anything beyond that limit was simply disregarded.

Over the past few years, the context window size of all the leading frontier model LLMs has increased dramatically. Most can handle at least 100,000 tokens at a time, and several can now handle more than 1 million tokens in a single 'thought' - that's far better attention than most people are capable of, even if they're experts in a field.

What that means is that a developer can, for example, potentially paste the entire contents of an API's full documentation text, as well as lots of their own existing code which needs to be connected to a new function, as well as lots of description and explanation about the purpose/goal of the required code, and the LLM can make sense of all that related information in a single chat interaction, and output a result based upon all that combined information, over the course of a single inference operation.

Together with improved reasoning capabilities, more deeply focused pre-training on programming languages and programming ecosystems, the ability to look up potentially needed information using a search tool, and even the ability to run generated code using a built-in interpreter/compiler tool, directly on the system that hosts the LLM, the result is far better generated code output.

So this is all a great improvement, but large context capabilities don't solve every problem. Using LLMs solely through a chat interface still requires a huge amount of interaction with the chat bot, because it can't see your entire code base - users need to spend time talking with chat bots, to provide every bit of required information about any given problem.

Well, with the advent of more and more established, pre-wired tools, existing MCP servers, pre-defined recipes and 'skills' which an agentic system can make use of, agentic frameworks such as Claude Code, OpenClaw, Goose, etc., which come with all these features built-in, can now be installed directly on your computer - and more features can be added from connected repositories.

This enables an intelligent LLM 'brain' to get long form work done on your computer. The framework forms a complete environment where the LLM can operate. It can read and write to any files that exist on your local file system, or read/write to/from every file in a Git repository which stores all the code related to a complete software development project, etc. The LLM's reasoning ability can then make sense of how all those files relate to one another, and the model can make changes to all the functions in any file, as needed, and even run any code it generates, to debug errors in an iterative loop, just like a human developer would - all from a single initial prompt by a human user.

Human users, therefore, no longer need to perform so much monotonous work. Agentic systems now provide LLM brains all the tools they need to take action and actually work towards achieving any longer context goals dreamt up by its user.

Instead of a human having to type text into a chat bot, to explain every step

they've taken in a development process, the agent simply accesses all the files it needs directly, it runs generated code files at the command line of your local computer (or on your remote development or production servers, for example), and sees any errors that are generated by the running code, it can debug those errors, look up relevant documentation and code examples which exist online, to help deal with any issue encountered, push completed code to repositories, and just continually loop through any of those required steps, testing output on a real computer, until a working application is completely formed and able to be used by users.

Since LLMs can work 1000s-millions of times faster than humans, the agentic ability of LLMs to loop through work with useful tools, means that real, complicated goals can be achieved by these systems, which would have previously taken large teams of highly trained professionals to complete, even just a few months ago.

Dealing With Context Window Size Limitations:

Agentic systems are also built from the ground up to help eliminate issues related to context size limits. LLMs have always been good at summarizing text. One of the techniques which has been implemented since the first versions of ChatGPT, is that previous messages in a conversation are 'compacted' into summarizations which are useful in further chat interactions. Obviously, details are lost during this compaction process, but it provides some ability for the LLM to 'remember' the basic history of a conversation, beyond the scope that can be handled by forcing an entire conversation into the context window size limit of the LLM that powers the chat bot.

Agentic systems are typically now engineered with the general ability to spawn *new contexts, to get any piece of a job completed. So, for example, if the LLM used in an agentic workflow reasons that documentation about a particular API function must be looked up online, in forums or repositories, it can start a new process to do all that particular search work, then summarize everything it's learned in-context from that web search, send that newly generated information to yet another spawned process which generates the output code needed to complete part of a project workflow, and so on - all without ever polluting the context of the original human interaction, which must keep track of the overall goal and progress of a user request.

This sort of native context spawning eliminates much of the work that a human needs to do to complete larger projects. We're starting to see LLMs which can work for many hours, and even multiple days, to complete extraordinarily complex goals, which would take talented humans weeks to accomplish. They just never run out of useful context, and they keep track of progress without losing site of the originally requested goal.

Agents Are Not Just About Software Development:

Very importantly, the idea of agentic workflows and agent systems does not apply only to software development. Agents which were originally designed to write

software code, are now more and more often used to accomplish general goals of almost any sort.

Anything which previously required a human working at a computer screen, can likely be handled by an agentic system with a capable LLM and a pile of attached MCP tools that have been given access to real world accounts and hardware.

Most companies building LLMs have been focused since the beginning on training their LLMs to get better at coding, because coding unlocks many other capabilities. Now that LLMs are better at writing software code than many human professionals, that capability can be leveraged to unlock technical ability of almost any sort.

The frontier model LLMs are now actually helping their human creators (companies like OpenAI and Anthropic) to build and improve new, more powerful LLMs, not just by performing research, but by writing code, running tests, and building new systems, in steps which are completed much more quickly than humans could previously perform those iterations on their own. What this means is that we're now beginning to ride a fast exponential curve of 'recursive self-improvement' which will quickly explode LLM capabilities at a progressively faster and faster pace.

So, How Does Someone Without A Tech Background Take Part In This Revolution?

At the moment, learning to use the most popular AI chat interfaces is the place to start. Getting to know the personalities and strengths of ChatGPT, compared to Gemini, Claude, Grok and other LLMs, is essential learning, as is learning how to form text prompts which get the best performance out of each model ('prompt engineering'). Learning to communicate very detailed and exact requirements via a prompt is perhaps the most important skill to build. Gaining experience in how each unique LLM succeeds or fails at achieving intended goals, based on the content and quality of prompt inputs, is an art which only improves with practice and experience.

Learning how to use existing agentic systems is a great way to focus your efforts next, if you're just getting started using modern AI. All the companies making LLMs (both commercial and open source) are engaged in a multi-trillion dollar race to build easier to use and more capable agentic systems. Those agentic systems are increasingly packaged directly into their chat bot interface systems (so chat bots now have the built-in ability go out and get things done in the real world).

For example, you can now upload a zip file of an entire software project to ChatGPT, give it some instructions to add or change a feature of the software, and the main chat context will use a tool that has been pre-installed on the ChatGPT server, to unzip the package, and then it will continue to spawn a chain of contexts to inspect all the files in the package, make required adjustments to the existing code in those files, run and debug the changes it made, search online for required documentation and code examples to fix bugs, and repackage the new completed functionality into an updated zip file for you to download - all entirely within the hosted chat environment (all for $20 per month).

Even the least expensive LLM alternatives such as Kimi 2.5, Minimax, and other open source options have deep agentic capabilities built right into their online chat environments - they can go out and get work done on the Internet, work with accounts you provide access to, etc. Kimi provides pre-packaged instances of the popular 'OpenClaw' agent system, as well as a built-in interactive software development environment, with hard drive storage space to save everything, etc., all for little to no cost for various levels of use. Minimax recently released an agentic system meant to help non-technical users build agentic systems. Users type in an explanation of what they want their agentic system to be able to accomplish, and it collects all the required MCP servers, writes the code and natural recipes needed to complete the class of tasks assigned to the agent, hosts all those pieces, provides interfaces for the human user to manually edit everything about the newly created agent system, and provides a natural language interface to interact with the system. With this infrastructure by Minimax, you can begin creating agentic systems immediately, even if you have little to no technical background whatsoever.

Things To Learn About, To Go Farther:

You no longer need to be a career software developer or a deeply trained data scientist to use LLM tools effectively, but if you want to dive more deeply into technology and AI systems, you should learn at least the basics of the Python programming language ecosystem. It is the lingua franca of the AI industry, and more commonly in recent years, the entire computing industry.

It's also really worth while to learn how to use the 'Linux' operating system, and to develop a very strong working knowledge of remote 'servers' - especially inexpensive hosted 'VPS' (Virtual Private Server) accounts. For $50 per year, you can put together a hosted server machine which can run applications of any sort on the Internet, connect to API's from all the biggest LLM providers, and build systems that can accomplish virtually any goal - all without having to use any local hardware. The machinery, Internet connection, OS installation/configuration, etc., is all managed by a 'hosting' company, and you can purchase a URL, or 'domain name' at which your server can be reached by web browsers and network tools.

Companies such as Atlantic.net provide managed servers to satisfy even the most stringent security requirements needed to maintain HIPAA, PCI, and other compliance obligations which are required by law when dealing with sensitive data in finance, healthcare, and other sensitive workflows. These hosting companies perform the necessary IT work (network monitoring, routine backups, maintenance, etc.) which would otherwise be very expensive for a small company to maintain, all for very inexpensive relative fees.

Learning the basics of how to connect with LLM REST APIs from Anthropic (Claude), OpenAI (ChatGPT), and providers such as OpenRouter and Syntheic, which provide access to hundreds of the most commonly used LLM models, is another essential requirement if you want to use AI technology. Agentic systems can now make use of virtually any LLM service hosted by these companies. You'll need to understand, at very least, how to wire them up to your account, with an API key, and how to

interact with the LLM over an API programming interface (instead of simply typing natural language into a public chat interface developed for non-technical users).

You'll likely need to learn how to use Git and other software development tooling such as databases, UI tools, environment installation/management tools such as pip, tmux, various IDEs, and more, along the way towards deeper AI mastery.

Buying a physical server machine which includes a powerful GPU, and learning to use the whole Nvidia CUDA ecosystem, as well as hosted services such as Vast.ai (which provides rented raw server access, to perform work on machines with GPU setups that would otherwise cost 10s-100s of thousands of dollars to own locally), is all something you'll likely need to do if you want to get much more deeply involved in serious AI technology work.

You'll need to learn at least a little about software development frameworks (especially in the Python and JavaScript/web development ecosystems), and also about all the common tools used to run AI inference, such as Ollama, LM Studio, LlamaCpp/KoboldCpp, vLLM and others.

You will get to know the 'personalities', strengths and weaknesses of every well known LLM (some excel at particular tasks, or have particular agentic abilities trained in). You'll begin to intuitively understand what sorts of knowledge depth and reasoning capabilities should be expected from a trillion parameter model, compared to a 100 billion parameter model, and/or the expectations you should have about the ability to produce factual information and quality code versus hallucinated content, in different sized models of the same series.

For example, anyone experienced with LLM use knows that the 600 million parameter Qwen3 model can barely speak sensibly, whereas the 32 billion parameter model is quiet intelligent - and the same variation in intelligence and capability is true throughout the ranges of model sizes in the Google Gemma series (270 million to 27 billion), as well as those made by all other providers.

From experience, you'll develop a sense of how 'quantization' (compression) level affects the level of accuracy expected from the exact same model with the same number of parameters. For example, a model quantized to 2 bits will not be able to form logical ideas to nearly as good a degree as the exact same model at full f16 precision. Smaller parameter count models with lower quantizations will perform much faster on lower powered consumer GPUs (which cost hundreds to thousands of dollars), but their performance with suffer compared to their larger counterparts which require serious commercial hardware (costing tens to hundreds of thousands of dollars).

You'll get to know, for example, exactly the sort of work which can be performed successfully on an inexpensive RTX 3090 GPU with 24 GB VRAM, using the GPT-OSS:20b model, compared to what can be accomplished with the GLM 5 model on a server with multiple RTX Pro 6000 GPUs that each have 96 GB VRAM installed. This understanding, and the ability to choose the right tools for a job, only comes from lots of experience, which takes time, legitimate involvement, and work solving real problems, to acquire.

You can continue by diving into learning about fine tuning with systems such as Lora, and then jump into the deep end to learn all the math, statistics, and data science disciplines needed to more deeply understand how foundational AI research is performed, and to work with teams/organizations who create LLMs and other models.

Take Some Concrete First Steps:

To get started, just dive into using LLM systems such as ChatGPT, Claude, Gemini, Grok, Deepseek, Kimi, GLM, Minimax, and Qwen. Most of the companies which make those LLMs also make other tools such as image and video generation tools. Try using other specialized tools such as Suno to create music.

You can do a LOT in the real world, just learning how to use those packaged, hosted systems, without any special technical knowledge at all. During the coming years, less and less background knowledge will be required to use AI systems, because eventually, as AI systems get smarter and smarter, they'll be able to guide anyone through accomplishing even the most complex tasks. So now is a great time to dive in, and it will continue to get easier to learn in the future.

Just get *started using and exploring LLMs and all the other tools in the mainstream AI ecosystem. Gain experience writing more effective prompts for LLMs. You'll learn quickly.

Resources:

One simple place to start on a deeper journey is with the videos and projects created by Andrej Karpathy, especially his nanoGPT and nanochat projects, which take students through all the phases of building an LLM from scratch:

https://www.youtube.com/andrejkarpathy

Another great YouTube channel which provides concise and approachable explanations about how AI systems work internally, including some simplified math topics, is 3blue1brown:

https://www.youtube.com/@3blue1brown

If you've never done any programming, the following tutorial will get you started using Python with the Anvil framework. This text is ~200 pages, with hundreds of screen shots, and will teach you from the ground up how to build more than 20 interesting apps, as well as how to begin approaching common software development methodologies which involve learning to think algorithmically, all in a very simple environment that can produce powerful commercial applications:

https://pythonanvil.com

The following tutorial about Linux, is particularly useful for getting started with

the essentials of using inexpensive VPS servers:

https://com-pute.com/nick/linux_server_basics.txt

Another relevant tutorial is this full case study which demonstrates how a complete application was built in real-time with an earlier version of ChatGPT (from 2 years ago). It includes the full chat conversation and all the prompts used to build every detail of a typical full stack app, with all the normal database, auth, UI, and other common software componentry:

https://com-pute.com/nick/brython-tutorial-and-GPT-case-study.txt

The thread below follows a casual history of writings by this author over the past 3 years, as LLM technologies progressed, in real time:

https://rebolforum.1y1z.com/topic/819

If you're wondering how AI relates to traditional concepts, such as 'how does all this come from 1s and 0s?', the text below explains everything from the very beginning, starting with basic concepts such as how binary representation works to make those 1s and 0s do useful things, and defines many of the most important terms and concepts needed to understand what computing generally is. It traces a history through how all the essential hardware, operating systems, programming languages, frameworks, and other pieces of computer science evolved, so you understand enough context and verbiage to fit everything about AI into a traditional understanding of computing. You won't necessarily gain any skills by reading this ~25 page text, but you'll likely have a lot fewer questions about how everything works, once you've read it:

https://com-pute.com/computing-primer.txt

A number of the video tutorial links on the following page (especially the ones about Baserow), are useful for users who need an introduction to database systems, all using no-code development tools:

https://compute.anvil.app/#?page=nocode

Additional varied programming tutorials spanning nearly 4 decades of software development, IT, and other applied computer technology work are available at:

https://compute.anvil.app/#?page=tutorials

Final Thoughts:

Traditional computing is typically thought of as 'deterministic' - it relies on algorithms humans have devised to solve problems. It makes use of systems such as databases and programming languages which always return the exact same output, when the same input is entered.

AI systems flip the script and teach themselves to work out solutions to problems, simply by being fed large amounts of data and running through training regimens, until they tend to return better results. They might not always produce the exact same output, but they can reason about generalities, and work towards achieving new and novel goals, just like humans can. As AI technology progresses, they will begin to lead humans towards achieving goals, and they will eventually surpass all human capability.

By learning the basics about traditional computing, you'll have a foundation needed to deal with AI tools. At very least, build some understanding about how to work with the Python programming language, learn some Linux OS, and learn to use VPS servers. That will get you past simply using packaged tools such as ChatGPT.