# Introduction to Large Language Models, and RAG Agentic Workflow Engineering

**Instructor:** Alejandro Morera Alvarez, alejandro.morera@profuturo.com.mx

In this document, you will learn about the necessary theory behind selected topics in artificial intelligence, as well as how to implement this in practice. We shall begin with a general introduction to large language models (LLMs), continue to a quick and dirty introduction on how we store and retrieve data effectively, and end with the difference between two agentic architectures, whose names are often used interchangeably in practice, yet whose distinction is of key importance. We then introduce the general practical components of a generic automated agentic workflow platform: knowledge, tools, topics, activity, analytics, and channels in the context of Copilot Studio. Afterwards, we shall discuss a similar yet more flexible platform for workflow design. Finally, the last chapter will introduce the reader to a programming-language-based solution to the same problem.

I have left questions after most of the parts discussed in this manual. The reader is strongly encouraged to answer the questions after reading each of the parts to promote active reading. Without further ado, let's get to it!

# Chapter 1: The Black Box

## Part 1: Large Language Models (LLMs)

**Goal:** By the end of this part, you should be able to explain in plain words (i) what an LLM is, (ii) why it works so well, (iii) where it fails, and (iv) how we make it reliable for real-world applications.

**What is an LLM?** One can think of a Large Language Model (LLM) as a program that continues text one piece at a time, based on everything it has seen so far. If you've used phone autocomplete, imagine that—but trained on much larger text collections and with far more nuance. The model doesn't ”know” facts like a database; instead, it has learned patterns of language and reasoning from examples.

**Tokens: the pieces the model sees.** Computers don't read whole words the way we do. They break text into small pieces called **tokens** (often parts of words). For instance, “retirement planning” might become two or three tokens. Working with tokens makes the system efficient and consistent across languages and rare words.

**Embeddings: turning tokens into numbers with meaning.** Each token is turned into a list of numbers called **vector** that captures some of its meaning and context. You can think of this as placing tokens on a huge “map of meaning”, where related tokens live closer together. The exact numbers don't matter to us; what matters is that similar things land nearby. In more mathematical words, embeddings leave semantic closeness invariant in the codomain's geometry.

**Transformers: how the model pays attention.** LLMs use an architecture called a **transformer**. Its key idea is **self-attention**: for every new token it wants to write, the model looks back at the previous tokens and decides which ones are most relevant, then mixes that information to produce the next step. Self-attention is like having a group discussion where each word can "look around" and borrow the most useful hints from the rest of the sentence or paragraph. Stacking many of these attention layers lets the model handle long sentences and complex instructions.

**Training: learning by predicting the next piece.** During training, we show the model lots of text and ask it to guess the next token many billions of times. When it guesses poorly, we nudge its internal settings so it'll guess better next time. Repeating this at huge scale teaches the model grammar, facts that appear in the data, writing styles, and common steps of reasoning. Later, a short phase called **instruction tuning** teaches it to follow directions, and **preference tuning** uses curated examples to steer it toward helpful, safe answers.

**What's "probabilistic" about LLMs?** For every next token, the model produces a list of possibilities with probabilities. At generation time we can pick the most likely one for stable answers or sample with some randomness for more creative text. Two simple dials control this: **temperature** (lower = more predictable, higher = more varied) and **top-p/top-k** (limit choices to the most promising few). Top-P is more dynamic and better for diverse outputs, whereas top-K is better for deterministic, consistent text.

**Why do LLMs work so well?** Three ingredients: (1) **attention** lets the model focus on what matters in the prompt; (2) **scale** (lots of data and compute) uncovers rich patterns of language and reasoning; and (3) **alignment** (those tuning steps) makes the model follow instructions and be helpful. As a result, with the right prompt, LLMs can summarize, explain, draft, translate, and reason through everyday tasks.

**The catch: limits you should know.** LLMs do not automatically know the latest policies or private company details; their memory is what they were trained on. They also have a **context window**—only a certain amount of text can be considered at once. And if asked about things they haven't seen or can't look up, they may produce **hallucinations**—confident-sounding but incorrect statements. Finally, raw internal data should not be "absorbed" into the model itself for privacy and compliance reasons.

1. In one or two sentences, explain to a non-technical colleague what an LLM is and how it is similar to, but much more capable than, phone autocomplete.

2. What are *tokens*, and why do LLMs use tokens instead of full words? Give a short example.

3. Describe *embeddings* using the "map of meaning" analogy. What does "closer" mean on this map?

4. In plain words, what is *self-attention* and why does it help the model handle long or complex inputs?

5. Summarize the training process in one paragraph: next-token prediction, instruction tuning, and preference tuning. What does each stage add?

6. What does it mean that LLMs are *probabilistic*? Name the two decoding dials and explain how tweaking their parameters changes outputs.

7. List three reasons LLMs work so well in practice. Give a one-sentence justification for each.

8. Name three important limitations of LLMs. For each, give a concrete consequence in real use.

9. Scenario: A user asks about a policy that changed last week. Without any external documents, what failure mode might occur, and why?

10. You must generate (a) a legal-style summary and (b) creative slogan ideas. For each case, specify suitable decoding settings and briefly justify.

## Part 2: (Optional, Source: 3B1B) A more thorough Introduction to LLMs and Transformers

**What Are Large Language Models and How Do They Work? Large Language Models (LLMs)** like ChatGPT are fundamentally sophisticated mathematical functions that predict what word comes next for any piece of text. To understand this seemingly magical capability, imagine a powerful machine that can complete a torn movie script by repeatedly predicting the next word based on what it has seen so far.

The core process is surprisingly straightforward: when you interact with a chatbot, the system takes your input, adds it to a template describing an interaction between a user and an AI assistant, then repeatedly predicts the next word that such an assistant would say. The model doesn't predict just one word with certainty – instead, it assigns **probabilities** to all possible next words. This probabilistic approach allows for more natural-sounding responses, as the system can select less likely words at random, making the output less deterministic even though the underlying model is fixed.

The scale of training is mind-boggling. To train GPT-3, researchers used an enormous amount of text – if a human were to read it non-stop 24/7, it would take over 2,600 years. Larger models since then train on vastly more data. The training process involves repeatedly showing the model text examples and adjusting hundreds of billions of **parameters** (the "knobs and dials" that determine behavior) to make the model slightly more likely to predict the correct next word.

To put the computational scale in perspective: imagine you could perform one billion mathematical operations every second. Training the largest language models would still take you well over 100 million years. This staggering computation is only possible using specialized computer chips called **GPUs** that can run many operations in parallel.

However, the initial training (called **pre-training**) is just the beginning. The goal of auto-completing random internet text is quite different from being a helpful AI assistant. This is where **reinforcement learning with human feedback** comes in – workers flag unhelpful or problematic responses, and these corrections further adjust the model's parameters to align with human preferences.

The breakthrough that made modern LLMs possible was the **transformer architecture**, introduced by Google in 2017. Unlike previous models that processed text sequentially word by word, transformers can "soak in" all the text at once, processing it in parallel. This parallelization is crucial because it allows the massive computational power of modern hardware to be fully utilized.

**1) The Transformer Architecture: Converting Words to Meaning**

The transformer's first crucial step is converting human language into numbers that computers can process. Text is broken into small pieces called **tokens** (usually words or parts of words), and each token gets associated with a high-dimensional vector – essentially a long list of numbers that somehow encodes the meaning of that piece.

This process, called **word embedding**, is far from arbitrary. Through training, the model

learns to position words in a high-dimensional space where similar meanings cluster together. Think of this as a vast multidimensional map where directions have semantic meaning. For example, the difference between the vectors for “woman” and “man” captures gender information – if you take this difference and add it to “uncle,” you land very close to “aunt.”
These embedding spaces can capture remarkably subtle relationships. In some models, if you take the vector for Germany, subtract Japan, and add it to sushi, you end up near bratwurst. The direction from Italy to Germany, when added to Hitler, points toward Mussolini. These aren’t programmed relationships – they emerge naturally from the training process as the model learns that certain directions in the space encode useful semantic information.
The **embedding dimension** in GPT-3 is 12,288 – meaning each word is represented by a list of over 12,000 numbers. This high dimensionality is crucial because it provides enough distinct directions to encode the rich variety of human language and knowledge.
But initially, each vector only encodes the meaning of a single word without any context. The word “bank” would have the same representation whether it appears in “river bank” or “bank account.” The transformer’s job is to progressively refine these embeddings so they absorb contextual meaning from surrounding words.
The model processes a fixed number of tokens at once (2,048 for GPT-3), creating a **context window** that limits how much text the model can consider when making predictions. This limitation explains why early chatbots would sometimes lose track of long conversations.
At the end of processing, only the final vector in the sequence is used to predict the next word. This means that this last vector, which might have started as a simple embedding for a word like “was,” must somehow encode all the relevant information from the entire context. In a mystery novel ending with “therefore the murderer was,” this final vector needs to have absorbed information about characters, motives, and clues from potentially thousands of preceding words.
The transformation from simple word embeddings to these information-rich contextual vectors happens through two main types of operations repeated many times: **attention** blocks (which allow information to flow between positions) and **multi-layer perceptrons (MLPs)** (which provide additional processing capacity). Understanding these components is key to understanding how transformers work.

## 2) Attention: The Heart of Understanding Context

The **attention mechanism** is what allows transformers to understand that “tower” means something different when preceded by “Eiffel” versus “control,” or that “mole” has different meanings in “American shrew mole,” “one mole of carbon dioxide,” and “biopsy of the mole.” This is the transformer’s solution to the fundamental challenge of language: the same word can mean vastly different things depending on context.
Think of attention as a sophisticated communication system between words. Each word asks a question (“What information do I need from the other words to refine my meaning?”) and other words respond with relevant information. For example, consider the sentence “a fluffy blue creature roamed the verdant forest”; the word “creature” might ask “Are there any adjectives describing me?” and “fluffy” and “blue” respond “Yes, we’re adjectives and we’re here to describe you.”
This process involves three key mathematical objects for each word: **queries**, **keys**, and **values**. The query represents the question a word is asking (“What should I pay attention to?”). Keys represent the potential answers other words can provide (“I’m an adjective in this position”). Values represent the actual information to be passed (“Here’s what it means

to be fluffy and blue").
Computing attention starts with **matrix operations**. Each embedding is multiplied by three different matrices – the query matrix, key matrix, and value matrix – producing query, key, and value vectors for every word. These vectors are much smaller than the original embeddings (128 dimensions versus 12,288 in GPT-3), making the computation more efficient.
The magic happens when queries and keys interact. For every possible pair of words, we compute how well the key matches the query by taking their **dot product**. This creates a grid of attention scores – larger numbers mean stronger relationships. In our example, if the system learned correctly, the scores would be high between "creature" and its adjectives "fluffy" and "blue," but low between "creature" and irrelevant words like "the."
These raw scores are then normalized using a **softmax function**, which converts them into probabilities that sum to 1 for each word. This creates an attention pattern – a precise specification of how much each word should influence every other word.
But attention doesn't just identify relationships – it transfers information. The value vectors encode what information should actually be passed. When "fluffy" attends to "creature," it doesn't just signal relevance – it provides specific updates that move "creature's" representation toward encoding "fluffy creature" rather than just "creature."
The final step combines everything: for each word, we take a weighted sum of all value vectors, where the weights come from the attention pattern. The word "creature" receives large contributions from the value vectors of "fluffy" and "blue," and tiny contributions from irrelevant words. This produces a change vector that gets added to the original embedding, creating a new representation that incorporates contextual information.
One crucial technical detail is **masking**: during training, later words are prevented from influencing earlier words. This ensures the model can't "cheat" by looking ahead to predict what comes next. The attention pattern has the upper triangle forced to zero, making it look like a lower-triangular matrix.
**Multi-headed attention** runs this entire process in parallel with different sets of query, key, and value matrices. GPT-3 uses 96 attention heads in each block, allowing the model to capture many different types of relationships simultaneously. One head might focus on adjective-noun relationships, another on subject-verb connections, and another on long-range dependencies between sentences.
**3) Multi-Layer Perceptrons: Where Facts Are Stored**
While attention gets most of the spotlight, the majority of parameters in large language models actually reside in **multi-layer perceptrons (MLPs)**. Recent research suggests these components serve as the primary storage mechanism for factual knowledge – when a model "knows" that Michael Jordan plays basketball, that information likely lives in the MLP blocks.
The computation in an MLP is surprisingly simple compared to attention: just two matrix multiplications with a simple nonlinear function in between. Yet interpreting what these computations accomplish is extraordinarily challenging, as they operate in the same high-dimensional embedding space where directions encode semantic meaning.
Let's trace through how an MLP might store the fact "Michael Jordan plays basketball." Imagine the embedding space has specific directions for "first name: Michael," "last name: Jordan," and "sport: basketball." When a vector encoding both "Michael" and "Jordan" flows through the MLP, the goal is to add the "basketball" direction to that vector.
The first step multiplies the input vector by a large matrix (about 50,000 rows for GPT-3).

Each row can be thought of as asking a specific question about the input. One row might correspond to "first name Michael + last name Jordan," which would produce a high value (say, 2) only when the input vector contains both name components. A bias term then adjusts this result – subtracting 1 gives us a value that's positive only for the complete name "Michael Jordan."

The next step applies a simple nonlinear function called **ReLU (Rectified Linear Unit)**, which clips all negative values to zero while leaving positive values unchanged. This creates a clean "yes/no" signal – the output is positive only when the input matches the exact pattern we're looking for (both first and last name), effectively implementing an AND gate in the high-dimensional space.

The final step multiplies by another large matrix to project back to the embedding space. The columns of this matrix represent what gets added to the output when specific "neurons" are active. The first column might be the "basketball" direction, so when our "Michael Jordan" detector fires, basketball information gets added to the vector.

This two-step process – expanding to a higher-dimensional space, applying nonlinearity, then projecting back – gives the model tremendous capacity to store and retrieve factual associations. Each MLP can implement thousands of such fact-retrieving circuits in parallel.

The scale is enormous: GPT-3 has 96 layers, each containing an MLP with about 1.2 billion parameters. Together, the MLPs account for roughly two-thirds of the model's 175 billion total parameters, explaining why they're believed to be the primary knowledge storage mechanism.

However, the reality is more complex than our toy example suggests. Individual neurons rarely represent single clean concepts like "Michael Jordan." Instead, the model appears to use a phenomenon called **superposition**, where many features are represented by nearly-perpendicular directions in the high-dimensional space. This allows storing exponentially more concepts than the space's dimensionality would normally permit.

This superposition principle helps explain both why models are difficult to interpret (features aren't visible as individual neurons) and why they scale so remarkably well (a space with 10 times more dimensions can store far more than 10 times as many independent concepts). In a 100-dimensional space, you can pack in thousands of nearly-perpendicular vectors, each representing distinct features or facts.

The combination of attention and MLPs creates a powerful architecture: attention moves information between positions and integrates context, while MLPs store and retrieve the vast knowledge needed for language understanding. Together, they enable the remarkable capabilities we see in modern large language models – the ability to engage in coherent conversations, answer questions, write creative content, and demonstrate broad knowledge across countless domains.

Understanding these mechanisms provides insight into both the power and limitations of current AI systems. They're not truly "understanding" language in a human sense, but rather performing sophisticated pattern matching and information retrieval in high-dimensional mathematical spaces. Yet this process can be so effective that it often appears indistinguishable from genuine comprehension, highlighting both the remarkable capabilities and the fundamental questions that remain about artificial intelligence.

1. What does a LLM do when generating text?

2. Why does sampling from a probability distribution over possible next words (instead of always

taking the most likely one) make outputs feel more natural and less deterministic?

3. The text mentions that training involves “hundreds of billions” of parameters and massive amounts of data. What is actually being adjusted during training, and why?
4. Explain why specialized GPUs and parallelization are critical for training LLMs, given the scale of required computations.
5. What gap does reinforcement learning with human feedback (RLHF) close between pre-training on internet text and functioning as a helpful AI assistant?
6. What problem did the 2017 transformer architecture solve compared to earlier sequential models, and why does parallelization enable today’s scale?
7. Define tokens and embeddings. How does a high-dimensional embedding space (such as 12,288 dimensions in GPT-3) capture semantic relationships? Give one example.
8. What is a context window (for example, 2,048 tokens in GPT-3), and how does this limitation affect long conversations or documents?
9. Describe the attention mechanism using queries, keys, and values. How are attention scores computed and normalized, what information actually flows, and why is masking required during training?
10. Compare the roles of attention and multi-layer perceptrons (MLPs). Which component primarily routes context between positions, which one is believed to store factual associations, and what does the concept of “superposition” mean in this setting?

## Part 3: Storage, Retrieval, and Allignment

**Grounding answers with Retrieval-Augmented Generation (RAG).** To make answers accurate and up-to-date, we **separate storage from writing**. First, we load company documents (PDFs, Word, transcripts, spreadsheets) into a special index called a **vector database**. We split documents into small chunks and convert each chunk into a vector on that “map of meaning.” When a user asks a question, we convert it into a vector too and quickly find the most similar chunks. We then **feed** those chunks to the model along with the question, so it writes the answer using the right sources.
In short: instead of “LLM answers from memory,” we get “LLM answers **with the right passages in view**.”
**Guardrails and evaluation: making it safe for real use.** We apply checks before and after model calls: removing sensitive data, restricting allowed tools, and enforcing structured outputs (for example, “always return a JSON with these fields”). We also measure performance continuously: accuracy against trusted documents, coverage of common questions, response time, cost per answer, and safety violations. If something drifts, we adjust the index, the prompts, or the workflow.
**Choosing a model for service operations.** Start small when possible: **small language models** (SLMs) can be fast and affordable, and often perform well once grounded with RAG. Scale up only when accuracy gains justify the extra cost and latency. Prefer platforms that connect smoothly to your tools (SharePoint, Teams) and support good engineering hygiene: separate environments (dev/test/prod), reusable environment variables, and connection ref-

erences—so solutions move cleanly from testing to production.

1. In your own words, explain “separating storage from writing.” Which part stores facts, and which part writes the answer?
2. Outline and roughly explain each of the RAG steps.
3. Using the “map of meaning” idea, explain how the system finds relevant passages for a new question without reading every document.
4. Why does answering as $LLM$(q, C(q)) (with retrieved context) tend to be more accurate and auditable than $LLM$(q) alone?
5. What are guardrails? Give two pre-answer and two post-answer examples that improve safety or compliance.
6. Pick three metrics from the text (e.g., accuracy vs. trusted docs, coverage, latency, cost per answer, safety violations). For each, explain how a bad value would be detected and fixed.
7. When would you start with a SLM instead of a LLM? Name two cost/latency reasons and one accuracy caveat.
8. List two Application Lifecycle Management (ALM) practices (e.g., environments, environment variables, connection references) and explain how each helps promote a solution from dev to prod without rework.

## Part 4: Agentic Workflows

**Workflows and Agents** A **workflow** is a clear recipe which is to be automatized. An **agent** is more free-form and can come up with solutions by itself. Pure agents are flexible but can be unpredictable; pure workflows are safe but rigid. In practice, we combine them into what we call an **agentic workflow**: agents operate inside a workflow so we keep creativity where it helps and guardrails where it matters.

1. Define a workflow and an agent in one sentence each. What is the main trade-off between them? Define an agentic workflow.
2. Give one concrete task that is safer as a fixed workflow and one that benefits from agent flexibility. Explain.
3. Explain why placing agents *inside* a workflow can give the best of both worlds. Mention at least one guardrail you would keep in place.
4. Sketch a simple orchestrated flow for answering “What are my withdrawal options?” Include: retrieval, policy check, template-based drafting, and a point where the agent can adapt (e.g., follow-up questions).

# Chapter 2: Practical Aspects of LLMs and No/Lo-Code Agentic Workflows

## Section 1: Copilot Studio

### Part 5: Prompting

A good prompt makes the model's job unambiguous: you state the role, the task, the sources it may use, the constraints it must follow, and the shape of the output. Start by setting **context** ("You are a retirement service assistant that must answer only from attached Knowledge") and the **goal** in one sentence ("Give the user's withdrawal options"). Then, list required **inputs** (for example, "Ask for plan type and state"), and specify **source boundaries** ("Cite only SharePoint Knowledge; if missing, say so explicitly").
Next, tell the copilot how to work: request a brief **plan** ("If needed, ask one clarifying question before answering"), allow tool use ("Call `RetirementCalculator.estimate` when the user provides age, salary, and contribution"), and define **safety constraints** ("No personal advice; no PII in the reply"). Finally, fix the **output format** so downstream steps are reliable ("Return JSON with fields `policy_name`, `eligibility`, `steps`, `citations`"). Keep **language requirements, tone, and length** explicit ("Professional, neutral Spanish; under 120 words unless policy requires detail").
As a template you can adapt:

> **Role:** You are *[agent role]* and must answer only from *Knowledge* and approved tools.
> **Goal:** *[one-sentence task]*. If information is missing, ask a single clarifying question, then proceed.
> **Inputs required:** *[list fields]*.
> **Allowed tools:** *[tool names]* — use only when inputs are present; otherwise explain what is missing.
> **Constraints:** No hallucinations; cite documents by title.
> **Output:** Return `JSON` with {`field1, field2, citations`} and a short user-friendly paragraph.

consistently yield faster, cheaper, and more accurate answers. I have created this agent in order to help you create descriptions and instructions for agents in Copilot Studio.

1. Test different prompts with real questions and tasks.

The following is just a brief introduction to some of Copilot studio's functionalities. For a more hands-on experience, I strongly recommend the reader to follow Microsoft's Copilot Studio course, which includes several use-cases, which introduces one to some of the features of this agent-creating platform. Here, I will only summarize two of the most theoretical parts of the course.

### Part 6: Copilot Studio – A Brief Introduction

**Overview.** This is the control room: it shows the copilot's name, version, environment (typically `DEV`, `TEST`, or `PROD`), and whether it is published. An **environment** is a safe space with its own settings and data so you can experiment without affecting users; the **publish**

**state** tells you which version end-users actually see. Use Overview to confirm you are editing the right copy before changing prompts, knowledge, or connections.
**Knowledge.** Knowledge is the set of document sources the copilot is allowed to cite at answer time (for example, a SharePoint library). The platform crawls, splits, and embeds these files into a searchable index so a user's question retrieves the most relevant passages first; this is the "storage" side of retrieval-augmented answers. Because permissions are respected, users only see passages they are allowed to read; after document updates, trigger a refresh so the index reflects the latest state.
**Tools.** A tool is an external **action** the copilot can call to fetch data or perform work (calculate, look up, send). Many tools are exposed through an **API**—a documented interface on a server—accessed at a specific **endpoint** (a URL path that represents one function, such as `/calculator/estimate`). A **connector** is a prebuilt integration that knows how to talk to a service (SharePoint, Outlook) so you do not hand-craft requests; an **internal function** is a small action you define yourself (for example, a Power Automate flow or a custom web service). **Credentials** (keys, secrets, or OAuth tokens) authenticate these calls so only authorized agents use them; set them here once, then topics can invoke the tool safely.
**Agents.** This tab lists the copilot itself and any collaborating sub-agents. In an agentic workflow, an **orchestrator** is the agent that coordinates others: it decides which specialist to ask, passes context, and aggregates results. Use this area to compose an orchestrated solution—sharing tools or knowledge sources across agents when appropriate.
**Topics.** Topics are the conversational blueprints that guide turn-by-turn behavior. A topic starts on a **trigger**—a detected intent such as "withdrawal options"—then collects needed inputs, optionally **calls tools** (that is, executes actions you defined in Tools), consults **Knowledge**, and finally formats the answer. Because topics define the order of steps and the required fields, they act as guardrails around otherwise flexible agents, ensuring consistent outputs and safer execution.
**Activity.** This is your run log: conversations, tool calls, retrieved passages, and any errors. Open this tab when debugging mismatched intents, permission issues, or failing endpoints; it shows exactly what was passed into tools and what came back.
**Analytics.** Dashboards here summarize usage and quality over time. **Latency** is the time from a user's message to the copilot's reply (including tool calls and retrieval); keep it low by trimming unnecessary steps and large contexts. **Cost** is mainly driven by model tokens (input + output) and any metered tool/API usage; reduce it by shortening prompts, constraining outputs, and preferring smaller models when accuracy allows.
**Channels.** Channels determine where the copilot is available (Teams, web, custom sites) and how users authenticate there. A **channel endpoint** is the published entry point the host uses to reach your copilot; when you enable a channel, you bind identity (for example, Microsoft 365 sign-in) so permissions and data access remain consistent from user to knowledge to tools.

## Section 2: Structure and Pipelines

Structure gives you *governance without friction.* Pipelines give you *speed without drama.* Put them together and you get a Copilot Studio practice that scales from a developing setting, to a large-scale production one: clear ownership, safe promotion across environments, and the ability to move fast with a net.

## Part 7: Structuring Your Copilot Studio Projects

**Goals:** We'll first make the case for why structure in Copilot Studio is non-negotiable; second, we'll show the exact setup that gives you speed without chaos. Think "mise en place" for software: a few minutes of order up front, and future-you stops firefighting and starts shipping.

**Why structure matters:** If you skip structure, you will possibly get technical debt: beautiful agents that can't leave DEV, teammates overwrite each other's work, among other problems. Governance isn't red tape; it's guardrails that let you move fast *on purpose*. The goal is simple: portability across environments, clean traceability, and changes that are reversible.

**The golden path (what to set up first)** There are three foundational moves. Do these once; benefit every day.

1. **Create a *solution* and a custom *publisher*.** A **solution** is your container—the single package that holds topics, tools, data sources, and everything your agent needs. It's the unit you'll move from DEV to TEST to PROD. A custom **publisher** (with your org's prefix) gives you clean naming, ownership, and zero collisions. This keeps your estate readable six months from now when you've slept since then. In order to create a solution, log into Copilot Studio, click on the ellipsis on the left-hand side, and then on Solutions. Within the solution, you will be able to create your own publisher.

2. **Add *environment variables* and *connection references*. Environment variables** store settings that change by environment (URLs, IDs, toggles). No find-and-replace hacks, no fragile hard-coding. **Connection references** point to external systems (SharePoint, CRM, etc.) in a way that can be re-bound per environment. Result: the same solution just works in TEST and PROD with the right connections and, most importantly, without rebuilds.

3. **Set up *Git source control*. (Optional)** Connect your solution to a Git repository. Every change gets a history. You can branch, review, revert, and audit. In other words, this is like collaboration without stepping on toes, and rollbacks that take minutes, not meetings.

**Compactly stated: Solution + Publisher** (organization), **Env Vars + Connection Refs** (portability), **Git** (traceability). That's your operating system for Copilot Studio.

1. In your own words, make the business case for why structure in Copilot Studio is non-negotiable. Name at least three failure modes it prevents and map each one to the benefits of portability, traceability, or reversibility.

2. Distinguish the roles of a **solution** versus a custom **publisher**. Explain why both are required for sane operations, and propose a concrete publisher prefix your org should use—justify.

3. Your agent works in DEV but faceplants in TEST due to hard-coded URLs/IDs. Write the exact remediation plan using *environment variables* and *connection references* so the same solution deploys to TEST/PROD with zero code changes.

4. (Optional) Propose a lightweight Git workflow (branching model, commit discipline, rollback protocol) that fits this setup. Explain precisely how it eliminates "who touched this?" autopsies and reduces rollbacks from days to minutes.

## Part 8: Deployment with Confidence: Automating the Path to Production

Scaling: let us move from "it works on my machine" to "it works for customers." Automation is how we trade heroics for reliability.
**Pipelines** are your paved road from DEV to PROD. Use **Power Platform pipelines** to define a clean, mandatory path—typically **DEV $\rightarrow$ TEST $\rightarrow$ PROD**. An pipeline is an ordered structure which is used to process data, as well as to train, evaluate, and deploy machine learning models. Pipelines enforce sequence, apply checks, and export your solution as a **managed solution** on the way into TEST/PROD. This is how you reduce blast radius and bake in change management.
Here are a few examples of what the pipeline actually does for you:

- **Validation.** It verifies that **environment variables** and **connection references** are properly mapped for the target environment and surfaces gaps early—where they're cheap to fix.
- **Repeatability.** Same steps, same order, every time.
- **Managed delivery.** Your TEST/PROD installs are locked down and auditable by design.

**Not everything exists inside a solution:** Some settings are *not* solution-aware: authentication configurations, channel security, certain monitoring/telemetry hooks. These require a short, explicit **post-deployment checklist**. Don't fight this—embrace it. Keep the checklist next to your pipeline runbook and tick it off every time.
**The minimal, repeatable workflow:**

1. **Create your pipeline and wire the stages.** In DEV, define the pipeline and point it at TEST and PROD in that order. This becomes your only road to PROD.
2. **Run the deployment and handle post-deploy steps.** Kick the pipeline, resolve any variable/connection prompts, let it ship the **managed solution**, then execute your **manual checklist** for the non-solution-aware pieces (auth, channels, monitoring).
3. **Commit the state to Git.** Record the changes and deployment metadata in your repo so history stays complete: what changed, when it moved, and where it landed. Tomorrow-you—and audit—will thank you.

1. Make the business case for a mandatory **DEV $\rightarrow$ TEST $\rightarrow$ PROD** pipeline. Explain how *managed solutions*, enforced sequence, and built-in validation reduce blast radius and turn "heroics" into boring reliability.
2. Your deployment to TEST fails because a *connection reference* isn't bound and an API URL was hard-coded in DEV. Write the remediation plan that fixes this *without code changes*: include how you'll use *environment variables*, re-bind *connection references*, and re-run the pipeline.
3. Draft a **post-deployment checklist** for the *non-solution-aware* pieces (authentication, channel security, telemetry/monitoring). Specify what you verify, how you verify it, and what constitutes a pass/fail gate before declaring the release "done."

4. Propose the **promotion policy** from TEST to PROD: list the required artifacts (versioned managed solution, commit/tag, release notes), approvals, and automated checks the pipeline must pass. Keep it lean but audit-proof.

5. It's Friday evening and someone quickly tweaked PROD. Describe the **containment and rollback** play using managed solutions and Git history, then state the preventative controls you'll implement (permissions, pipeline-only promotions, branch protection) to keep PROD gloriously boring.

## Section 3: n8n

The platform n8n is an example of an automation platform for agentic AI solutions. In this section, we will learn how to choose the right **AI Agent** (Tools, Conversational, Plan-and-Execute—or the niche ones when needed), wire a sane trigger, chain steps cleanly, set a simple memory window, and enforce **Structured Output** so downstream nodes don't guess. Add observability (run logs, errors), credentials hygiene (least privilege, masked secrets), and a boring DEV → TEST → PROD rollout. My main go-to source to learn how to use n8n is Nate Herk's YouTube channel. There, he presents several use cases, which eventually lead to him creating a sort of "brain", which automates many tasks for him. Following his videos and the following theory, you will learn how to create efficient automatized workflows with the help of AI, as well as to connect to external sources, which is the point where these tools start to become useful.

### Part 9: The Six Types of n8n Agents

When configuring an **AI Agent** node in n8n, the first choice is which specialized agent type to use. While all six are available, the most common are the **Tools Agent**, the **Conversational Agent**, and the **Plan-and-Execute Agent**; the others (**React Agent**, **SQL Agent**, **OpenAI Functions Agent**) address more specific needs.

**Tools Agent.** A flexible, general-purpose agent that can call external **tools** such as calculators, HTTP requests, or custom code. It is meant for actions that go beyond plain dialogue, orchestrating concrete operations and then returning results. **Use case:** receive user input, call a web API to look up data, and send an email with the results.

**Conversational Agent.** An agent optimized for multi-turn dialogue that maintains **context** across messages. It remembers prior user messages and uses them to craft coherent, context-aware replies, making it suitable for chatbots and virtual assistants.

**Plan-and-Execute Agent.** An agent for complex, multi-step objectives. It first **plans** by decomposing a goal into ordered steps and then **executes** those steps. It shines when a task requires structured progress more than long-term conversational memory. **Use case:** project planning or event organization, breaking the goal into tasks and executing them in order.

**React Agent.** A reasoning-first agent that selects the next action based on current observations rather than past chit-chat. It is useful when the agent must analyze a state, choose a tool, and justify the choice. **Use case:** analyze survey responses and decide the appropriate follow-up workflow.

**SQL Agent.** A specialized agent that interfaces directly with **SQL** databases. It translates natural language into SQL queries to retrieve or modify data, enabling non-SQL users to work with relational stores. **Use case:** "Show me all customers who upgraded in the last 30 days."

**OpenAI Functions Agent.** An agent that leverages OpenAI's native **function calling** to pick and execute the correct tool from a provided list, producing **structured outputs**. It is precise about arguments and return formats, which is valuable for downstream automation. **Use case:** map a user request to a specific function with validated parameters and return JSON.

1. Pick the right **agent type**: For each scenario, choose **Tools**, **Conversational**, **Plan-and-Execute**, **React**, **SQL**, or **OpenAI Functions** and justify in one sentence. (a) "Pull website metrics and email a summary." (b) "Chat with users, remember the last 5 turns, and escalate

on keywords.” (c) “Break down onboarding into steps and tick them off.” (d) “Given a JSON payload, decide the next action with an explanation.” (e) Given a database, answer: ”Which customers upgraded last 30 days?”

2. **Conversational Agent exercise:** Configure a bot that schedules meetings. Specify the **Memory** window, what user slots (date, time, attendees) you will collect, and how the agent confirms before finalizing.

3. **Plan-and-Execute exercise:** Given “launch a webinar,” define the **plan** (at least 5 ordered steps) and which steps are delegated to **Tools Agents** (e.g., create calendar event, email invites, post reminder).

4. **React Agent exercise:** You receive a JSON survey result. Describe the prompt that makes the agent (i) classify sentiment, (ii) decide an action, and (iii) output a short rationale.

5. **SQL Agent exercise:** Provide a natural-language query and the expected SQL (including table/column names). Add two **guardrails** to avoid destructive writes and leaking.

## Part 10: Constructing an Agent Workflow in n8n

**Triggers and chaining.** An **agent workflow** begins with a **trigger** node that starts the run. Common triggers include a **Webhook** (fires on incoming HTTP requests), a **Google Sheets Trigger** (fires when a new row appears), or a manual **Chat Message** input for testing. n8n supports **chaining** via the **Execute Workflow Trigger**, allowing one agent-driven workflow to call another; this enables **multi-agent systems** where each agent specializes. For example, a **Conversational Agent** gathers requirements, hands them to a **Plan-and-Execute Agent** to produce a task list, and individual **Tools Agents** perform each task.
**Core configuration.** Every **AI Agent** node needs three pillars. First, a **Chat Model**—the large language model that provides reasoning (e.g., an OpenAI chat model). Second, **Memory**—so the agent can recall prior turns when appropriate; the built-in **Window Buffer Memory** retains the most recent messages within a configurable window. Third, **Output Parsers**—to make responses machine-reliable for downstream nodes. Enabling “**Require Specific Output Format**” lets you attach an **Output Parser** such as **Autofixing**, **Item List**, or **Structured Output** to enforce shapes like JSON and recover from minor formatting errors. Together, these ensure the agent thinks with the right model, remembers what matters, and returns data in a predictable schema.
**Putting it together.** Choose the agent type that matches the problem, wire a suitable **trigger**, connect the **Chat Model**, add **Memory** if multi-turn context is needed, and enforce a **Structured Output** when passing results onward. When tasks get complex, **chain** agents: let a planner outline steps, let tools execute them, and let a conversational layer report progress back to users.

1. **Trigger selection:** For each source—**Webhook**, **Google Sheets Trigger**, **Chat Message**—give one concrete business event it fits best and explain why another trigger would be worse.

2. **Chaining design:** Sketch a **multi-agent** chain where a **Conversational** agent collects requirements, a **Plan-and-Execute** agent creates tasks, and two **Tools** agents perform them. List the data passed at each handoff and the stop condition.

3. **Chat Model choice:** Choose a small vs. large **Chat Model** for: (a) FAQ triage, (b) legal summary drafting. State the cost/latency trade-off and a fallback rule when tokens exceed budget. **Bonus:** create a cheap agentic workflow called "Brain WF" which receives tasks as inputs and outputs the appropriate model for the task.

4. **Memory configuration:** Set a **Window Buffer Memory** policy for a support bot: window size, what fields are excluded, and when to flush memory between threads. Give an example of when a simple memory is not sufficient for the task.

5. **Output Parser:** Define a **Structured Output** schema for "create ticket" with fields `summary`, `priority`, `category`, `citations` for a customer service agent. Write two validation rules and how the parser should **autofix** minor formatting issues.

6. **Error handling:** Describe how your workflow reacts when an **HTTP Request** tool times out: retry policy, backoff, a user-facing message, and a log entry. Where in the chain do you catch and route the error?

7. **Security & credentials:** List the **credentials** you need (API keys/OAuth) and where they are stored for a simple FAQ use-case where the knowledge comes from a SharePoint.

8. **Testing & rollout:** Propose a **DEV** $\rightarrow$ **TEST** $\rightarrow$ **PROD** plan in the context of n8n. **Hint:** Take a look back to the section on Copilot Studio.

## Part 11: Vectorized RAG in n8n: Pinecone, Supabase, and Company

TODO: introduction of how to, importance embedding dimension and model,... include the link to the WF in my n8n.

# Chapter 3: Programming (Optional)

## Part 12: From No-Code to Code: Your Local AI Stack on Windows

This section shows how to run an AI assistant *locally*—no cloud, no external data leaving your PC—using three pieces: **Ollama** (runs language models on your machine), **Miniconda** (manages an isolated **Python environment**), and **Open WebUI** (a simple chat interface in your browser). You will (i) install two trusted programs, (ii) create a clean Python space, (iii) download a model (**qwen3:latest**), and (iv) start a tiny **web server** on **localhost** (your own computer) at **port 8080**. Throughout, we explain what each command does to your system.

**1) Install Ollama — the local model runner.** **Ollama** is a small runtime that downloads and serves language models, using your CPU or GPU if available. Installing it adds a command-line tool called `ollama`. After the installer finishes, open **Windows Terminal** (PowerShell) and check: `ollama --version`. This verifies the command is on your PATH. *What this changes on your PC:* a program is added under `C:\Program Files` (or similar) and a background service may run to host models. No system settings are altered beyond PATH entries.

**2) Install Miniconda — clean Python, no conflicts.** **Miniconda** is a minimal Python distribution plus a package manager. We use it to keep AI tools isolated from corporate Python or other projects. After installation, open a new PowerShell and create an **environment** (a named, sandboxed Python): `conda create -n myenv python=3.11` then `conda activate myenv`. *What this changes:* it creates a folder `myenv` containing its own Python and libraries; it does not modify the system Python.

**3) Add the chat interface.** Inside the activated environment, install **Open WebUI**: `pip install open-webui`. This is a small web app that talks to Ollama and renders a chat page. *What this changes:* it downloads Python packages into `myenv`; nothing is installed system-wide.

**4) Download a model** Run `ollama pull qwen3:latest`. The verb **pull** fetches the model weights to your disk. **qwen3:latest** identifies the model family and tag. *What this changes:* several gigabytes may be saved under Ollama's models directory; ensure you have free disk space. You can see installed models with `ollama list` and remove one with `ollama rm qwen3:latest`.

**5) Start the local server and chat.** Launch the UI with `open-webui serve`. The word **serve** starts a tiny **HTTP server** bound to **localhost** (your own machine) on **port 8080**. If Windows asks about firewall access, allow *private* networks. Now open a browser and visit `http://localhost:8080/`. You should see a chat interface; select **qwen3:latest** and start testing. *What this changes:* a process is listening on port 8080 until you stop it (`Ctrl+C` in the terminal). No public exposure—**localhost** means only your computer can access it.

**6) Resource notes and safe exits.** Models use RAM/VRAM and CPU/GPU. If the UI feels slow, try a smaller model (e.g., a 7B variant) via `ollama pull <model:tag>`. To stop the UI, press `Ctrl+C`. To leave the Python environment: `conda deactivate`. To remove it entirely: `conda remove -n myenv --all`. To stop/remove a pulled model: `ollama list` then `ollama rm <name>`.

**Why this path (and when to use it).** Running *locally* gives you privacy (data never leaves your PC), predictable **cost** (no cloud tokens), and offline capability—great for prototypes, internal demos, and sensitive text. It complements the no-code Copilot approach: Copilot is ideal for production integrations and M365 workflows; the local stack is ideal for quick experiments, custom prompts, and testing retrieval ideas before wiring them into enterprise tools.

**Copy–paste cheat sheet**

```
conda create -n myenv python=3.11
conda activate myenv
pip install open-webui
ollama pull qwen3:latest
open-webui serve
```

**Troubleshooting quick wins.** If `conda` is not found, re-open Terminal or run `conda init powershell` then restart. If `ollama` is not found, log out/in after install. If the page does not load, ensure the command shows "Serving on 127.0.0.1:8080" and no VPN or proxy blocks localhost.

## Part 13: Add Your Own Documents (Local RAG on `localhost`)

Once the chat page is running at `http://localhost:8080`, the next step is to let the assistant **ground** its answers in your files. Open WebUI includes a lightweight **knowledge** feature that builds a local index and never sends content off your machine.

**Where to click.** In the left sidebar, open **Knowledge** (sometimes labeled "Documents" or "Collections"). Create a new **collection** and give it a clear name (e.g., `HR_Policies_2025`). This creates a local folder and a tiny metadata database under your Open WebUI data directory.

**What to upload.** Drag in PDFs, DOCX, TXT, or Markdown. CSV works for tabular snippets. If a PDF is *scanned* (images), run OCR first; the indexer reads text, not pixels.

**What indexing does.** When you click **Index** (or it starts automatically), the UI: 1) **splits** each file into overlapping **chunks** so passages can be retrieved efficiently; 2) computes **embeddings** (vectors) for each chunk; 3) stores those vectors in a local **ANN** index (recall we saw this in the first chapter). Typical defaults work well: chunk size around 800–1,000 tokens with 100–150 overlap.

As a rule of thumb, keep in mind that larger chunks keep context; smaller chunks make retrieval more precise.

**Pick an embedding model.** The UI lets you choose a local embedder via Ollama. Good starters are `nomic-embed-text` (fast, small) or `mxbai-embed-large` (higher recall, heavier). Pull one first:

```
ollama pull nomic-embed-text
ollama pull mxbai-embed-large
```

Embeddings are just numbers; they stay on disk and are reused across sessions unless you re-index.

**Use the knowledge in chat.** Back in **Chat**, select your **model** (e.g., `qwen3:latest`) and flip on "**Use Knowledge**." Ask questions like "What is the leave carryover limit?" The model receives your question plus the top retrieved chunks; you'll see **citations** to the exact files and pages. If you update a document, click **Re-index** so the vectors reflect the new content.

**What this changes on your PC.** The uploaded files and the vector index are stored under Open WebUI's data directory; no registry edits or system-wide changes. Deleting a collection removes its vectors but not your originals.
**Tip.** Create one collection per department (HR, IT, Finance) and keep filenames tidy (policy_name_vYYYYMMDD.pdf). This makes citations and audits easy.

# Chapter 4: Microsoft Azure

Copilot Studio often requires heavier additional tools from Microsoft Azure. This section is heavily based on the following courses (Coursera):

- Artificial Intelligence on Microsoft Azure;
- Microsoft Azure Machine Learning;
- Computer Vision in Microsoft Azure;
- Natural Language Processing in Microsoft Azure;
- Preparing for AI-900: Microsoft Azure AI Fundamentals exam.

We shall go through the practical exercises and diving deeper in the theory behind each of the tools. In this script, I have chosen to explain the second one in great detail.

## Part 14: Azure Machine Learning

### Mise en Place

Just like when we cook, we need to have everything ordered and ready before we begin the actual training. To create an Azure ML workspace, we go to Microsoft Azure and create a new Machine Learning resource. This step requires having an Azure subscription. Note that you can make one for free and get 200 dollars of credit(!). Choose one of your subscriptions, create your custom resource group, as well as a name for the workspace, the region closest to where you are geographically located, and fill in the rest of the details required. When it comes to public network access, in our case, we shall leave the "All networks" option selected — change it if your workspace will go into production. Once your workspace has been created, open your workspace in Azure AI Machine Learning Studio by either following the previous link, or, in Microsoft Azure, accessing your created workspace. In *Overview*, you should be able to see the Studio web URL.
Next, we shall create compute resources. Indeed, in order to train and deploy models, we need compute power on which to run the training process, and to test the trained model after deploying it. The tab *Manage >> Compute* is where you manage the compute targets for your data science activities. There are four kinds of computing resources:

- *Compute Instances:* Development workstations that data scientists can use to work with data and models.
- *Compute Clusters:* Scalable clusters of virtual machines for on-demand processing of experiment code.
- *Inference Clusters:* Deployment targets for predictive services that use your trained models.
- *Attached Compute:* Links to existing Microsoft Azure compute resources, such as Virtual Machines or Azure Databricks clusters.

Since we are in the development phase, let us create a new compute instance. Create a com-

pute cluster in case your application requires inferencing afterward. Choose an appropriate compute name, select CPU as our virtual machine type, and Standard_DS11_v2 as the virtual machine size, and, if you are only playing around, leave everything else as is.
After this, we are ready to explore and prepare data: creating a dataset, adding it to your pipeline canvas, and applying necessary transformations to prepare your data for further analysis or modeling. The *Assets >> Data* page contains specific data files or tables that you plan to work with in Azure ML. You can create datasets from this page as well. After the dataset has been created, open it and view the *Explore* page to see a sample of the data.
After creating your dataset, we are ready to move on to the next step: designing a pipeline. In order to create a pipeline, go to *Authoring >> Designer* and click on *Create a new pipeline*.
Now, to the training part. In our example, note that there are some rows with missing data, and that most of these columns are numeric, but each feature is on its own scale. For the former observation, we can use the *Clean Missing Data* node to prepare the data. Sometimes it is also helpful to use the *Select Columns in Dataset* in case one is only interested in certain columns. Concerning the latter observation, note that, for example, in the dataset from the course, Age values range from 21 to 77, while DiabetesPedigree values range from 0.078 to 2.3016. When training a machine learning model, it is sometimes possible for larger values to dominate the resulting predictive function, reducing the influence of features that are on a smaller scale. Typically, data scientists mitigate this possible bias by using the *Normalize Data* node, so they're on similar scales.
Now that the data is filtered and normalized, it is ready to be processed for training — the next step is to create and run a training pipeline. It is common practice to train the model using a subset of the data (say 70% of it), while holding back some data with which to test the trained model. This enables you to compare the labels that the model predicts with the actual known labels in the original dataset. Connect the pre-processed data, insert an *Untrained model* node as well as the processed data. Then, drop a *Train model* node below these two. Finally, score the trained model against the processed data by using a *Score model* node. After running this pipeline, if you want, you can also insert an *Evaluate model* node below the Score model node.

**1) Regression Models in Azure ML**

Regression predicts a numeric value from features: for instance, given engine size, seats, mileage, we would like to predict car price. It is an example of supervised learning because you train on data that includes both inputs and known numeric labels to learn that mapping.

**2) Classification Models in Azure ML**

Classification is a machine-learning method that predicts which category/class an item belongs to. It is a supervised approach: you train a model on data that includes both the features (inputs) and the known label (output), so the model learns how feature combinations map to the label and can later output class probabilities for new, unlabeled items.
For example: a clinic uses patient measurements—age, weight, blood pressure, glucose, etc.—as features to predict a binary label: diabetic (1) vs. non-diabetic (0). After training, the model takes a new patient's measurements and estimates the probability of each class.
The result of evaluating the trained model yields a so-called confusion matrix — one in which true positives and negatives, as well as false positives and negatives, are plotted. The confusion matrix shows cases where both the predicted and actual values were 1 (known as true positives) at the top left, and cases where both the predicted and the actual values were 0 (true negatives) at the bottom right. The other cells show cases where the predicted and

actual values differ (false positives and false negatives).
These are some metrics for a confusion matrix:

- *Accuracy:* The ratio of correct predictions (true positives + true negatives) to the total number of predictions. In other words, what proportion of diabetes predictions did the model get right?
- *Precision:* The fraction of positive cases correctly identified (the number of true positives divided by the number of true positives plus false positives). In other words, out of all the patients that the model predicted as having diabetes, how many are actually diabetic?
- *Recall:* The fraction of the cases classified as positive that are actually positive (the number of true positives divided by the number of true positives plus false negatives). In other words, out of all the patients who actually have diabetes, how many did the model identify?
- *F1 Score:* An overall metric that essentially combines precision and recall.

After creating and testing an inference pipeline, the final step is to deploy a predictive service for client application use. We shall deploy the web service to a Microsoft Azure Container Instance (ACI). This type of compute is created dynamically, and is useful for development and testing. For production, you should create an inference cluster instead, which provides a Microsoft Azure Kubernetes Service (AKS) cluster that provides better scalability and security.
**3) Clustering Models in Azure ML**
Clustering groups similar items into clusters based solely on their features — no labels, no hand-holding. It's unsupervised learning: the model discovers structure in the data on its own. For example, a researcher measures penguins and groups them by similar proportions. In Azure Machine Learning Designer, you work with a dataset (the container for your training data) and can train, infer, and deploy a clustering model with drag-and-drop — zero code.
Contrast this with classification: classification is supervised (you train with features and known labels, like "diabetic vs. non-diabetic"); clustering has no labels and simply separates items based on similarity. In other words, classification predicts a known bucket, whereas clustering finds the buckets.

**Choosing the Right Machine Learning Algorithm in Azure ML Designer**

The following text is a detailed scripted version of this. When you choose a model, your first job is to name the problem in plain terms: are you predicting a number, assigning a label, finding weird outliers, grouping similar things, reading text, classifying images, or recommending items? The answer narrows the field dramatically. Next, respect your operational constraints: how much data you have, how quickly you need results, whether you must explain the decision to a non-technical stakeholder, and how much you can spend on training and serving. With that framing in place, the following narrative walks through the major families in Azure Machine Learning Designer and shows when each algorithm shines, using examples you could defend in a boardroom and publish in a lab report.
**a) Two-Class Classification (exactly two outcomes)**

When the question is a simple yes/no—for example, "will this transaction be fraudulent?"—start simple and scale up only if needed. The most direct option is **Two-Class Logistic Regression**, which draws a straight-line rule in the space of your input variables. "Straight line" here means it combines the inputs with weights and then converts that score into a probability. If you must explain "why" to an auditor, this is your friend, because each weight tells you how a specific input pushes the decision one way or the other. Imagine a credit approval engine with forty or fifty tabular fields (age of account, payment ratio, missed payments, and so on). With historical approvals and defaults, **Two-Class Logistic Regression** provides a credible baseline you can ship fast, and the resulting coefficients double as a clear story for risk and compliance.

If your data calls for a sharper boundary between yes and no, **Two-Class Support Vector Machine** focuses on the edge cases where the model is most uncertain and tries to maximize the distance between the two classes. Think of it as placing a wide "no man's land" between legitimate and fraudulent events. This tends to work best when the number of inputs is moderate and the signal is reasonably clean. A practical case: blocking suspicious logins using device fingerprint features (browser traits, time-of-day patterns, and location consistency). The model sets a sturdy separating rule that discourages borderline impostors.

When you need to learn from a continuous stream and update the model frequently with minimal cost, **Two-Class Averaged Perceptron** is a fast, incremental learner. Picture a moderation system that flags abusive chat messages in near real time. New labeled examples arrive all day; you retrain quickly and keep latency low. The Perceptron learns a direct rule and averages many such rules to stabilize the final decision.

If your inputs interact in complicated ways (for example, the effect of one variable depends heavily on another), tree-based ensembles are a strong default. **Two-Class Decision Forest** builds many diverse decision trees and averages them, which makes it robust to messy, mixed data (numbers, categories) with minimal manual data cleaning. A marketing use case fits well: predicting whether a user will click a personalized offer given dozens of behavioral and demographic fields. When you want to push accuracy further on structured, table-like data without heavy feature engineering, **Two-Class Boosted Decision Tree** trains trees in sequence, each one learning from the previous one's mistakes; in practice, this often gives you a few extra points of performance without a big increase in serving cost. Finally, **Two-Class Neural Network** can capture subtle, tangled patterns if you have ample data and are willing to tune. Consider high-frequency sensor streams from industrial equipment where failures are rare and early faint signals matter; a small, carefully regularized network can tease out those patterns.

**b) Multiclass Classification (more than two outcomes)**

When the outcome is one of several categories—say, auto-assigning a support ticket as *Billing*, *Login*, or *Cancellation*—you can extend the same ideas. **Multiclass Logistic Regression** remains the fast, explainable baseline: it fits a straight-line rule for each category and picks the most confident. If your classes are uneven (some are rare, others common) or numerous, you can wrap a binary learner with strategies that scale better. **One-vs-All Multiclass** trains one two-class model per category ("is it class A or not?"). This is straightforward and effective when you have a handful of categories. **One-vs-One Multiclass** trains a tiny duel between every pair of classes, which can help when some classes are easily confused with only a subset of others; this is handy for handwriting recognition where, say, "1" versus "7" is the real knife fight.

Tree ensembles translate naturally here. **Multiclass Decision Forest** provides a rugged first pass on mixed data without extensive preprocessing. **Multiclass Boosted Decision Tree** tends to become the workhorse when you need extra accuracy with sensible training time. And when your features are high-dimensional (for example, embeddings from text or images), **Multiclass Neural Network** can carry more nuance. A concrete example: routing inbound customer emails into one of twelve workflow queues. Start with **Multiclass Logistic Regression** on simple text-derived features; graduate to **Multiclass Boosted Decision Tree** on richer engineered signals; escalate to **Multiclass Neural Network** if those gains justify the added complexity.

**c) Regression (predict a number)**

If your target is a quantity—delivery time in minutes, price in dollars, energy consumed in kilowatt-hours—go with a baseline you can explain and then iterate. **Linear Regression** estimates a straight-line relationship between your inputs and the number you care about. If adding one bedroom tends to add a fixed amount to a house price, this model captures that cleanly and tells you approximately how much that bedroom is worth, all else equal. When data is scarce and you also need honest uncertainty, **Bayesian Linear Regression** gives you a best-guess line and a reasonable range around it, which is invaluable when reporting to finance: "We expect $1.2M, likely between $1.0M and $1.4M given what we know."

If the real relationship bends and twists, **Decision Forest Regression** averages many decision trees to capture interactions and thresholds ("after 25°C, consumption rises faster") without delicate hand-tuning. **Boosted Decision Tree Regression** stacks small trees so each one corrects the previous errors; this often yields top accuracy on structured business data with modest training time. When the signal is complex (for example, mapping a wide set of machine sensors to a quality score), **Neural Network Regression** can approximate the curve as closely as data allows. For counts—integers like "tickets per day" that are zero or small on quiet days and larger on busy days—**Poisson Regression** is designed for that use: it models how the average count grows with the inputs, then uses that to predict a likely integer. And when the business question is about *ranges* rather than single-point guesses, **Fast Forest Quantile Regression** estimates percentiles directly. If operations needs a promise like, "ninety percent of deliveries arrive within X minutes," this model gives you a defensible X that lines up with your service targets.

**d) Anomaly Detection (find the weird stuff)**

Sometimes you don't have labels; you just want to catch the oddballs. **PCA-Based Anomaly Detection** learns the main ways your data tends to vary and then tries to reconstruct each example from those patterns. If an example cannot be reconstructed well, it is probably unusual. Imagine network traffic with regular daily rhythms; sharp deviations in that reconstruction error flag suspicious activity. **One-Class SVM** takes a different angle: it learns the outer boundary of "normal" and marks anything outside as an outlier. This is practical for predictive maintenance when you only have sensors from healthy machines; the model wraps an envelope around that healthy region and raises a hand when new readings fall outside.

**e) Clustering (discover structure without labels)**

When you have no labels and you want to organize your data into sensible groups, **K-Means** is a strong place to start. It places a fixed number of "centers" and assigns each data point to the closest center, then refines both assignments and centers until things settle. Use it for customer segmentation: perhaps six clusters emerge such as "loyal value shoppers" and "new seasonal buyers." Those groups are not given to you—the algorithm discovers them—and downstream you tailor messaging, site experience, and offers accordingly.

**f) Image Classification (what is in this picture?)**

If you have labeled images and need to classify them, modern deep networks offer a pragmatic route via fine-tuning. **ResNet** uses shortcut connections that help train much deeper models reliably; you load a version already trained on a massive image set and adjust it to your domain with your labeled examples. A retailer can classify photos into *tops*, *bottoms*, and *shoes* with a weekend's worth of careful labeling and training. **DenseNet** connects each layer to many others, enabling efficient feature reuse; it often achieves strong accuracy with fewer parameters. In manufacturing, a fine-tuned **DenseNet** can detect cosmetic defects (scratches, dents) on high-resolution product images under varied lighting, where tiny texture differences matter.

**g) Text Analytics (turn words into usable signals)** Text needs a pipeline: clean it, turn words into numbers, then model. **Preprocess Text** removes obvious clutter like repeated punctuation, common stop-words ("the", "and"), and normalizes casing so "Azure" and "azure" are treated consistently. Next, **Extract N-Gram Features from Text** counts single words and short phrases; this simple counting works shockingly well as a baseline. If your vocabulary explodes into millions of unique phrases, **Feature Hashing** compresses it to a manageable size by mapping phrases into a fixed set of bins; different phrases may occasionally share a bin, but you gain speed and memory efficiency. For unsupervised exploration, **Latent Dirichlet Allocation** groups documents by topic without labels, surfacing themes like "shipping delays" or "billing complaints" across thousands of reviews. And if you want to capture semantic similarity—that "refund" and "reimbursement" are close in meaning—**Word2Vec** learns numerical representations of words so words used in similar contexts land near each other in that learned space. In practice, you often stack these: preprocess, create n-gram features or word vectors, and then train a straightforward classifier like **Two-Class Logistic Regression** for spam detection or **Multiclass Logistic Regression** for intent routing.

**h) Recommenders (what should we show next?)**

Recommendations are about ranking items for each user. **SVD Recommender** starts from a large table where rows are users, columns are items, and entries are preferences or interactions (clicks, ratings, views). It then compresses that table into a smaller set of hidden factors that explain most of the behavior, which helps predict what a given user is likely to enjoy next, even when the original table is very sparse. This is a natural fit for suggesting courses on a learning platform: after a few enrollments and ratings, **SVD Recommender** can surface high-value follow-ons. When you also have rich item and user features (descriptions, tags, demographics), **Wide & Deep Recommender** blends two ideas: a "wide" part that memorizes useful rules like "users who liked A also liked B," and a "deep" part that generalizes by understanding content. A news app, for example, can recommend fresh articles to a brand-new reader by leaning on content features, while still exploiting association rules for returning readers.

**Practical Selection Playbook**

Put simply: name the problem type; ship the simplest plausible model first; measure whether it meets the bar; scale only if the business case improves. For a two-class decision with explainability needs, start with **Two-Class Logistic Regression**. For multiclass on structured data, begin with **Multiclass Logistic Regression** and switch to **Multiclass Boosted Decision Tree** if you need more punch. For a number, begin with **Linear Regression**; move to **Boosted Decision Tree Regression** when relationships bend. For outliers, try **PCA-Based Anomaly Detection**, then **One-Class SVM** if the envelope of normal behavior is better defined. For unlabeled grouping, **K-Means**. For images, fine-tune **ResNet** or **DenseNet**. For text, clean, featurize, then a simple classifier. For recommendations, **SVD Recommender** first, and if you have content features and scale, consider **Wide & Deep Recommender**. This is not romantic; it is operational. Start with the fastest credible option, protect your timeline, and only complicate the stack when metrics and dollars agree.

With the ML part out of the way, it is time to start with the question of how we partition our documents, turn these into vectors, and ground an agent with these and *only these* documents. That is, it is time to build a RAG agent.

## Part 15: RAG in Microsoft Azure

In this part, I provide a complete guide to implementing a RAG assistant in Microsoft Teams using Microsoft Azure.

### Overall idea

We assemble a conversational system where a large language model (LLM) answers with citations grounded in your own documents. The core ingredients are **Azure Blob Storage** for documents, **Azure OpenAI Service** for the LLM and embeddings, **Azure AI Search** for hybrid retrieval (vector and keyword), an **Azure App Service** that hosts the bot logic as a web app, and **Azure Bot Service** to connect that app to **Microsoft Teams**. We proceed in phases: prepare resources in a single region, ingest data with vectors, validate in the chat playground, deploy the web app, and register the Teams bot.

### Introduction

**Purpose:** The aim is to furnish a step–by–step, technical yet readable path to a Teams chatbot based on RAG, fully on Azure.

**Scope:** We cover creation of cloud resources, configuration of the data and index, validation of answers with citations, and deployment to Teams.

**Architecture:** The solution consists of the following collaborating services:

- **Azure Blob Storage**: persistent store for source documents.
- **Azure OpenAI Service**: access to the chat LLM and the embeddings model.
- **Azure AI Search**: hybrid retrieval over vectors and keywords.
- **Azure App Service**: hosts the web app that exposes the bot endpoint.
- **Azure Bot Service**: registers the bot and handles the Teams channel.

### Prerequisites

You will need an Azure subscription with sufficient permissions (Owner or Contributor), enabled access to **Azure OpenAI Service**, a **Microsoft Teams** account for tests, and a collection of documents (PDF, DOCX, TXT, etc.) to serve as the knowledge base. For latency and cost coherence, keep all resources in the same *Region*.

## Phase 1: Provision the Azure infrastructure

Begin in the **Azure Portal**. Create a resource group, storage, search, and Azure OpenAI in the same region. Then deploy two models in **Azure AI Studio**: a chat LLM and an embeddings model. Let's go through this step by step.

**Create a Resource Group.** Open the **Azure Portal**, choose **Resource groups** and click **Create**. Provide **Subscription**, **Name** (e.g. `RAG-KnowledgeBot-DEV`), and **Region**. Click **Review + create** and then **Create**.

**Create a Storage Account (Blob Storage).** Search **Storage accounts** and click **Create**. Select your **Resource group**, choose a globally unique lowercase **Storage account name** (e.g. `stknowledgebotdev`), keep the **Region** consistent, leave defaults, and click **Review + create** → **Create**. We will later add a container for the documents.

**Create Azure AI Search.** Search for **Azure AI Search**, then **Create**. Select the **Resource group**, supply a unique **Service name** (e.g. `search-knowledgebot-dev`), choose the same **Region**, and pick an appropriate **Pricing tier** (**Basic** is sufficient to start; **Free** is limited). Click **Review + create** → **Create**.

**Create Azure OpenAI Service.** Search **Azure OpenAI**, click **Create**, attach to the same **Resource group** and **Region**, provide a unique **Name** (e.g. `openai-knowledgebot-dev`), choose **Standard S0**, and **Create**.

**Deploy models in Azure AI Studio.** Navigate to the new **Azure OpenAI Service**. In the left menu, choose **Model deployments** and click **Go to Azure AI Studio**. Inside **Azure AI Studio**, open **Deployments** and create two deployments:

- **Chat LLM:** click **+ Create** and choose a chat model (e.g. `gpt-35-turbo-16k` or `gpt-4`), name it (e.g. `gpt-35-turbo-chat`), and confirm.
- **Embeddings:** click **+ Create** again, choose `text-embedding-ada-002` (or current equivalent), name it (e.g. `text-embedding-ada`), and confirm.

## Phase 2: RAG configuration and data ingestion

Upload your documents to **Azure Blob Storage**; then, from the **Chat Playground** in **Azure AI Studio**, attach your data, configure the **Azure AI Search** index with vector search, and validate retrieval quality.

**Upload documents to Blob Storage.** Open the **Storage account** (`stknowledgebotdev`). In **Data storage**, click **Containers**, then **+ Container**, and name it `knowledge-base`. Open the container and click **Upload** to add your PDFs/DOCX/TXT.

**Connect data in Azure AI Studio (Playground).** Return to **Azure AI Studio** bound to your **Azure OpenAI Service**. Open the **Chat Playground**. In the right–hand configuration panel, click **+ Add your data**. Choose **Azure Blob Storage** as the source, select your **Subscription**, the **Storage account** `stknowledgebotdev`, and the **Container** `knowledge-base`. For search, select your **Azure AI Search** service `search-knowledgebot-dev`. When prompted for the index, provide a name such as `idx-knowledge-base`. Check **Add vector search to this search resource**. Select the embeddings deployment (**text-embedding-ada**) and acknowledge any embedding–related costs. Choose **Search type: Hybrid**. Complete the wizard by clicking **Next**, then **Create** or the offered confirmation buttons until indexing begins. Depending on data volume, ingestion can take several minutes.

**Validate in the Chat Playground.** Once indexing completes, the **Chat Playground** is connected to your knowledge base. Ask questions whose answers should be in your documents; confirm that citations refer to expected passages. Iterate until retrieval + LLM behavior is coherent.

### Phase 3: Deploy the web application

Package the playground configuration into a web app on **Azure App Service**.

In the top–right of the **Chat Playground**, click **Deploy to** and select **A new web app. . .** . Provide a globally unique **App name** (e.g. `app-knowledgebot-dev`), confirm **Subscription** and **Resource group**, choose the same **Location**, and either create or select an **App Service plan** (a **Basic (B1)** plan is a reasonable starting point). Enable **Enable chat history** if desired for diagnostics. Click **Deploy**. This provisions **Azure App Service** and publishes the code.

**Verify the App Service.** Open the created **App Service** (`app-knowledgebot-dev`) in the portal and copy its default **URL**; we shall use it when configuring the bot endpoint.

### Phase 4 — Create the bot and integrate with Microsoft Teams

Create an **Azure Bot**, point it to your web app's messaging endpoint, enable the **Microsoft Teams** channel, and test inside Teams.

**Create the Azure Bot resource.** Search for **Azure Bot** and click **Create**. Choose a **Bot handle** (e.g. `KnowledgeBot-PROD`), select the same **Subscription** and **Resource group**, and set **Application type: Multi Tenant**. Click **Review + create → Create**.

**Configure the Messaging endpoint.** Open the **Azure Bot** resource and choose **Configuration**. In **Messaging endpoint**, paste the App Service URL from Phase 3 and append `/api/messages`. The final shape should be:

```
https://app-knowledgebot-dev.azurewebsites.net/api/messages
```

Click **Save**.

**Enable the Microsoft Teams channel.** In the bot's **Channels**, click the **Microsoft Teams** icon, accept defaults, and **Save**. The Teams channel becomes active.

**Test in Teams.** From the Teams channel blade, click **Open in Teams** and follow the prompts to add the bot to your client. Initiate a conversation and repeat your playground tests to verify end–to–end behavior and that citations remain intact.

### Conceptual note on indexing and retrieval

In **Azure AI Search**, searchable content resides in a *search index* hosted by your service. The index does not store entire documents (like whole PDFs) but fields representing the content, optimized for fast queries. In a RAG pattern, a user's question is forwarded both to the search engine and to the LLM. Search results are then provided to the LLM, which composes the final answer. There is no search mode—neither semantic nor vector—that generates novel answers by itself; only the LLM provides generative output. Hybrid retrieval improves the grounding material supplied to that LLM.