AI JOB SWITCH

120 concepts for your AI job switch

8 areas, 15 concepts each. Every concept explained in one line, the order to study each area, and two practice questions to test yourself.

How to use it: one area at a time, tick each concept only when you can explain it out loud in under a minute, then answer the two practice questions without notes.

@chalkesahil · chalkesahil.ai

AREA 01 / 08

Python and maths

Study order: NumPy and pandas first, then the linear algebra and probability you need to read ML code.

01	**NumPy**	Fast arrays and vectorised maths; most ML code is NumPy underneath.
02	**pandas**	Load, filter, join and group tabular data before any model sees it.
03	**Vectors and matrices**	Data, weights and embeddings are all just arrays of numbers.
04	**Dot product**	Multiply and sum; measures how aligned two vectors are.
05	**Derivatives**	How fast an output changes when an input changes.
06	**Gradients**	The direction of steepest increase; training walks the opposite way.
07	**Probability**	Models predict likelihoods, not certainties.
08	**Bayes rule**	Update a belief when new evidence arrives.
09	**Distributions**	Normal, uniform, binomial: the shapes data and noise usually take.
10	**Mean and variance**	The centre of the data and how spread out it is.
11	**Hypothesis testing**	Is a difference real or just noise? p-values and confidence intervals.
12	**Data cleaning**	Missing values, duplicates and outliers decide model quality more than the model.
13	**Plotting**	Look at the data with matplotlib before you trust any number.
14	**Jupyter**	Notebooks for quick experiments; move stable code into modules.
15	**Big O basics**	Know why a loop over a million rows in pure Python is slow.

Practice

1. Compute cosine similarity between two vectors in NumPy without a loop.
2. Explain gradient descent to a non-engineer in 30 seconds.

AREA 02 / 08

Machine learning

Study order: Regression and classification end to end on one dataset, then metrics, then trees and boosting.

01	**Supervised vs unsupervised**	Learning from labelled answers versus finding structure without labels.
02	**Train/test split**	Hold data back so you measure the model on examples it never saw.
03	**Overfitting**	Memorising training data instead of learning patterns that generalise.
04	**Bias vs variance**	Too simple misses the pattern; too complex chases noise.
05	**Linear regression**	Fit a line to predict a number; the baseline for everything.
06	**Logistic regression**	Predicts a probability for yes or no; simple and strong.
07	**Decision trees**	A chain of if-else splits learned from data.
08	**Random forest**	Many trees on random slices of data, voting together.
09	**XGBoost**	Trees added one by one, each fixing the last one's errors; wins on tables.
10	**k-means**	Group points into k clusters around moving centres.
11	**Cross-validation**	Rotate which slice is the test set for a more honest score.
12	**Precision and recall**	Of what you flagged, how much was right, and how much of the truth you caught.
13	**ROC-AUC**	How well a classifier ranks positives above negatives across thresholds.
14	**Feature engineering**	Turning raw data into inputs a model can actually use.
15	**Regularization**	Penalise large weights so the model stays simple: L1 and L2.

Practice

1. Your fraud model is 99% accurate but useless. Why?
2. When would you pick logistic regression over XGBoost?

AREA 03 / 08

Deep learning

Study order: Build a tiny network by hand once, then switch to PyTorch and train on a GPU.

01	**Neurons and layers**	Weighted sums passed through a function, stacked in layers.
02	**Activations**	ReLU, sigmoid, GELU: the non-linearity that lets networks learn curves.
03	**Loss functions**	A single number for how wrong the model is: MSE, cross-entropy.
04	**Backpropagation**	The chain rule, run backwards, to get every weight's gradient.
05	**Gradient descent**	Nudge weights against the gradient, a small step at a time.
06	**Adam**	The default optimiser: gradient descent with momentum and per-weight step sizes.
07	**Dropout**	Randomly switch off neurons in training so the network can't over-rely on any.
08	**Batch norm**	Keep each layer's inputs at a stable scale so training is smoother.
09	**CNNs**	Sliding filters that find edges, then shapes, then objects in images.
10	**RNNs**	Process sequences one step at a time with a running memory; replaced by transformers.
11	**Embeddings**	Words, users or items as vectors where similar things sit close together.
12	**PyTorch**	Tensors, autograd and the training loop: the framework most AI teams use.
13	**GPU training**	Thousands of parallel cores for matrix maths; mind memory and batch size.
14	**Learning rate schedules**	Warm up, then decay the step size so training settles.
15	**Transfer learning**	Start from a pretrained model and fine-tune it on your data.

Practice

1. Write a PyTorch training loop from memory.
2. Your loss is NaN after 200 steps. What do you check?

AREA 04 / 08

Transformers and LLMs

Study order: Read how attention works, then how a model generates one token at a time.

01	**Tokenization**	Text split into sub-word tokens; models see numbers, not letters.
02	**Attention**	Each token decides how much to look at every other token.
03	**Self-attention**	Queries, keys and values all come from the same sequence.
04	**Multi-head attention**	Several attention patterns in parallel, each learning a different relation.
05	**Positional encoding**	Tells the model the order of tokens, which attention alone ignores.
06	**Encoder vs decoder**	Encoders read and understand; decoders generate text left to right.
07	**Pretraining**	Learning language from huge amounts of text before any task.
08	**Next-token prediction**	An LLM's only job: guess the next token, again and again.
09	**Context window**	How many tokens the model can see at once, prompt plus answer.
10	**Temperature and top-p**	Knobs for how random or focused the next-token choice is.
11	**KV cache**	Saves past attention work so each new token is cheaper to generate.
12	**Hallucination**	Fluent, confident output that is false; design around it.
13	**Scaling laws**	More data, parameters and compute give predictably lower loss.
14	**Open vs closed models**	Weights you can run yourself versus models behind an API.
15	**Model sizes**	Parameter count trades quality against cost, speed and memory.

Practice

1. Explain attention without using the word attention.
2. Why does a longer prompt make responses slower and costlier?

AREA 05 / 08

Prompt engineering

Study order: Write prompts against a small test set from day one, so every change is measured.

01	**System prompts**	Standing instructions that set the model's role and rules.
02	**Few-shot**	Show two or three worked examples instead of describing the task.
03	**Chain of thought**	Ask the model to reason step by step before answering.
04	**Structured JSON output**	Make the model return data your code can parse, then validate it.
05	**Prompt templates**	Reusable prompts with slots filled in by code.
06	**Delimiters**	Fence user content with tags so instructions and data don't mix.
07	**Prompt injection**	Text in the input that tries to override your instructions.
08	**Guardrails**	Checks before and after the model call to block unsafe or off-topic output.
09	**Function calling**	The model asks your code to run a tool and uses the result.
10	**Prompt caching**	Reuse a long, unchanging prompt prefix to cut cost and latency.
11	**Token cost**	Input and output tokens are billed; long prompts add up fast.
12	**Latency**	Time to first token and total time; stream to feel faster.
13	**Model choice**	Use the smallest model that passes your tests.
14	**Prompt versioning**	Track prompts like code so you can roll back a bad change.
15	**Eval-driven prompts**	Change one thing, rerun the test set, keep it only if scores rise.

Practice

1. Make an LLM always return valid JSON for an invoice. How?
2. A user pastes 'ignore all instructions'. How do you defend?

AREA 06 / 08

RAG

Study order: Build the simplest pipeline first, measure retrieval, then add hybrid search and reranking.

01	**Chunking**	Split documents into pieces small enough to retrieve and big enough to make sense.
02	**Embedding models**	Turn text into vectors so similar meanings land close together.
03	**Vector databases**	Store embeddings and find the nearest ones fast.
04	**Cosine similarity**	The angle between two vectors; the usual closeness score.
05	**Hybrid search**	Combine keyword search (BM25) with vector search to catch both.
06	**Reranking**	A second model reorders the top results by true relevance.
07	**Metadata filters**	Narrow by date, source or user before the similarity search.
08	**Recall@k**	Did the right chunk show up in the top k results?
09	**Citations**	Show which source each claim came from so users can check.
10	**Query rewriting**	Rephrase the user's question into a better search query.
11	**Document parsing**	Getting clean text out of PDFs, tables and slides is half the work.
12	**Re-indexing**	Keep the index fresh when documents change.
13	**Context limits**	Only so many chunks fit; pick the best, not the most.
14	**RAG vs fine-tuning**	RAG adds knowledge; fine-tuning changes behaviour and style.
15	**GraphRAG**	Retrieve over entities and relations for questions that span documents.

Practice

1. Your RAG bot answers from the wrong document. Walk through debugging it.
2. Design RAG over 50,000 company PDFs that update weekly.

AREA 07 / 08

AI agents

Study order: Write one agent loop by hand before using a framework, then add tools, memory and evals.

01	**Tool calling**	The model picks a function and its arguments; your code runs it.
02	**The agent loop**	Think, call a tool, read the result, repeat until done.
03	**Planning**	Break a goal into steps before acting.
04	**Short and long memory**	The current conversation versus facts saved across sessions.
05	**MCP**	Model Context Protocol: a standard way to plug tools and data into models.
06	**ReAct**	Interleave reasoning and actions so each step is grounded in a result.
07	**Multi-agent**	Several agents with separate roles hand work to each other.
08	**Error recovery**	Retry, try another tool or ask, instead of looping forever.
09	**Human in the loop**	Pause for approval before risky actions like payments or deletes.
10	**Sandboxing**	Run agent code and tools in isolation so mistakes stay contained.
11	**State**	Track what's done and what's pending so a run can resume.
12	**LangGraph**	A framework for agent workflows as graphs of steps.
13	**Cost limits**	Cap steps, tokens and spend per run.
14	**Agent evals**	Score whole tasks, not single replies: did it finish, correctly, how cheaply?
15	**Permissions**	Give each tool the least access it needs.

Practice

1. Build an agent that books a meeting from an email. What tools does it need?
2. Your agent loops forever on one task. How do you stop and debug it?

AREA 08 / 08

Deployment and evals

Study order: Ship one small AI feature end to end with logging and an eval set; that is the portfolio project.

01	**Model APIs**	Calling hosted models: auth, retries, timeouts and errors.
02	**FastAPI serving**	Wrap a model or pipeline in a typed Python API.
03	**Streaming**	Send tokens as they're generated so users see progress.
04	**Rate limits**	Providers cap requests and tokens per minute; queue and back off.
05	**Caching**	Reuse answers for repeated questions to save cost and time.
06	**LoRA fine-tuning**	Train small adapter weights instead of the whole model.
07	**Quantization**	Store weights in fewer bits to run bigger models on less hardware.
08	**vLLM**	A fast open-source server for running LLMs with high throughput.
09	**Tracing**	Log every prompt, tool call and output so failures can be replayed.
10	**LLM-as-judge**	Use a model with a rubric to grade outputs at scale; spot-check it.
11	**Eval datasets**	A fixed set of real inputs with expected results; your regression test.
12	**A/B tests**	Compare two prompts or models on live traffic.
13	**Drift monitoring**	Watch for inputs or quality changing over time.
14	**Cost tracking**	Know the cost per request and per user before it surprises you.
15	**AI system design**	The interview round: design a chatbot, a RAG search or an agent end to end.

Practice

1. Design a customer support bot for 1 million users a day.
2. How do you know a prompt change didn't make things worse?