How Small Transformer Models Actually Work

The linear algebra, the transformations, the training, and the hardware — built up from first principles, not hand-waved.

GROUND-UP EXPLAINER · A FACILITIES AI FOR SENSORS AND LANGUAGE



The same machine — whether the input is a wristband's accelerometer stream or a line of text.

By the end you will be able to explain, precisely:

- **Foundations** — the neuron and the multilayer perceptron — the atom of every model
- **Representation** — how a word — or a sensor window — becomes a vector, and why geometry is meaning
- **Attention** — the four matrices Q, K, V, O and the one formula the whole model is built around
- **The block** — residuals, normalisation and the feed-forward net — and why depth helps
- **Training** — the loss, back-propagation, the optimiser, and fine-tuning on your own data
- **Infrastructure** — what sits in memory, why generation is slow, and what it costs to run
- **Two builds** — the wrist-sensor cleaning-assurance system, and a language side for tickets & manuals

Prepared for Riddhish · 4 July 2026

Read start-to-finish, or jump in by part.

Notation is defined as it appears, and gathered into a glossary at the end.

ORIENTATION

What a transformer is, before any mathematics

A transformer is a machine that reads a **sequence** of things — words, sub-words, or short slices of sensor data — and, for every position in that sequence, produces a fresh set of numbers that summarises what that position *means in the context of everything around it*. That is the whole trick. It does not look things up in a table of rules; it re-computes the meaning of each item from scratch every time, taking its neighbours into account. A one-second burst of wrist motion means something different in the middle of a long mopping rhythm than it does on its own; the word *"leaking"* means something different next to *"coolant"* than next to *"confidential"*. The transformer is built to capture exactly that kind of context — which is why the same machine serves both halves of your operation.

Everything a transformer does is built from one humble operation: multiplying a list of numbers by a grid of numbers to get a new list of numbers. That operation — the **matrix multiplication** — is repeated, with different grids, thousands of times. The grids are the model's **parameters** (also called **weights**): they start as random noise and are slowly tuned during training until the arithmetic they perform happens to produce useful answers. There is no database inside, no rules written by a person. There is only a very large pile of numbers and a fixed recipe for combining them.

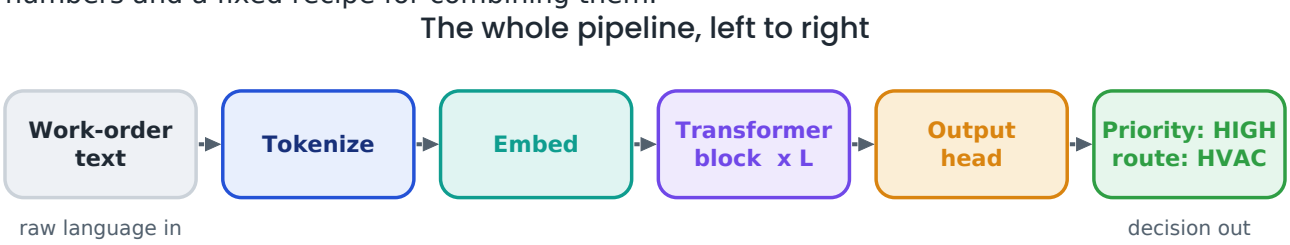


Figure 1. The end-to-end pipeline. This document opens up each arrow into the exact matrix operations underneath. Read left to right: a sequence in, a decision out.

The load-bearing ideas

Three things to hold in your head for the whole document. (1) **Everything is a vector.** A word, a position, a one-second window of accelerometer readings — all become fixed-length lists of numbers. (2) **Learning = tuning grids.** The model's skill lives entirely in the numbers inside its weight matrices; training is the process of nudging those numbers. (3) **Meaning is geometry.** Similar things end up as nearby vectors, and the model exploits distance and direction in that space to reason.

“Small” is worth pinning down. The famous chat models have tens or hundreds of *billions* of parameters and are trained on much of the public internet. A **small** transformer — the kind sensible for focused tools like yours — might have anywhere from a hundred thousand to a few million parameters (for the sensor model) or up to a few hundred million (for a text model), trained on a far narrower, cleaner body of data: your own sensor logs, tickets and manuals. Smaller models are cheaper to run, easy to keep private on your own hardware, and — because your problems are narrow — usually entirely good enough. The mathematics on the following pages is *identical* at every size; only the dimensions of the grids change.

> **One engine, two builds — sensors (Part 7) and language (Part 8)**
> This report teaches the machinery using **language** — words and sentences — because it is the most concrete way to see every step. It then applies that machinery to **two facilities builds** that share one engine. **Part 7** is your core project: wrist-sensor data for cleaning assurance, where a “token” is a short *window* of accelerometer, gyroscope and compass readings. **Part 8** is the language side — the tickets, notes and manuals your operation already generates — where the tokens really are words. Everything between here and there (embedding, attention, blocks, training) is the same arithmetic for both.

FOUNDATIONS

Neurons and the multilayer perceptron

Before the transformer's clever parts, there is the plain building block that every neural network — including yours — is made of: the **multilayer perceptron** (MLP), also just called a feed-forward network or a “fully-connected” network. If you understand this one page, the transformer's feed-forward sub-layer (Part 3) and its output head (Part 4) are already familiar, and — directly useful for your project — you have the strongest *simple* model for recognising cleaning activities from sensor data.

F.1 The neuron: a weighted sum, then a bend

A single artificial **neuron** does two things. First it takes a weighted sum of its inputs and adds a **bias** — that is just a dot product, exactly the operation from Part 1. Then it passes the result through a non-linear **activation** function φ:

$$a = \varphi(w_1 x_1 + w_2 x_2 + \ldots + w_n x_n + b) = \varphi(w \cdot x + b)$$

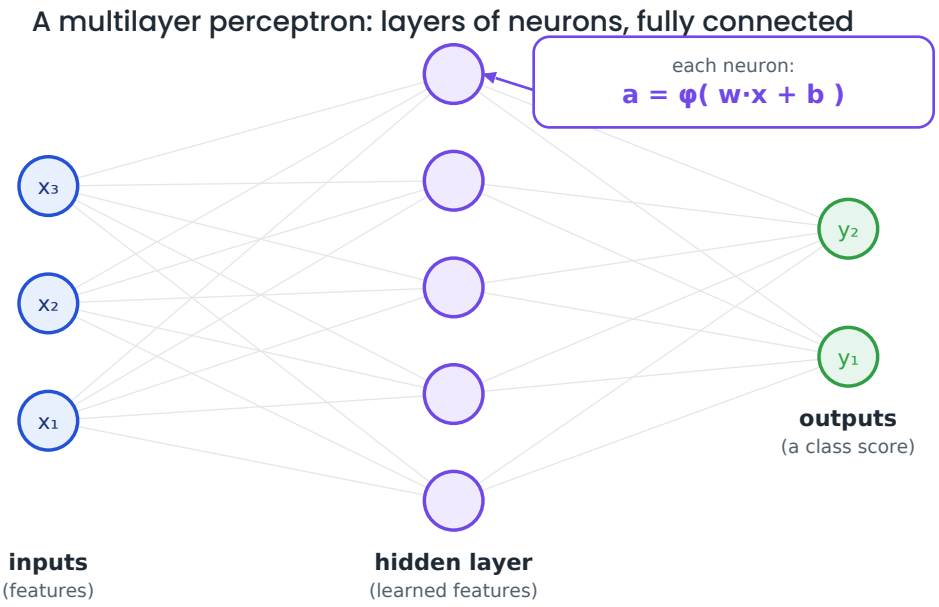
The weights w say how much each input matters; the bias b shifts the threshold at which the neuron “fires”; φ decides the final output. A quick worked neuron: inputs x = [2, 0, 1], weights w = [0.5, −1, 2], bias b = −0.5. The sum is 0.5·2 + (−1)·0 + 2·1 − 0.5 = **2.5**. If φ is ReLU (keep positives, zero out negatives) the output is 2.5; if φ is the sigmoid (squash to a 0–1 “probability”) it is $1/(1 + e^{-2.5}) \approx$ **0.92**. That is the entire computation of one neuron.

F.2 A layer is a matrix multiply; a stack is an MLP

Put many neurons side by side, each with its own weights, all reading the same inputs, and you have a **layer**. Stacking their weight rows into a matrix W turns the whole layer into a single matrix–vector multiply plus a bias, then φ applied element-wise — the same shape of operation you have seen throughout:

$$h = \varphi(W x + b) \text{ (W has one row per neuron)}$$

Feed one layer's output into the next and you get a **multilayer perceptron**: an input layer, one or more **hidden layers** that build up increasingly abstract combinations of the inputs, and an output layer that produces the answer — a class score, a probability, or a number.



Every line is a learned weight; every neuron takes a weighted sum, adds a bias, and bends it with φ. Stack more hidden layers to go deeper.

Figure F. A multilayer perceptron. Inputs on the left, a hidden layer of neurons in the middle, outputs on the right; every connection is a learned weight. Add more hidden layers to make it 'deeper'.

F.3 Why the bend matters

The activation φ is not decoration — it is the whole reason depth helps. Without it, each layer would be a plain linear map, and a stack of linear maps is just *one* linear map: no matter how many layers you piled up, the model could only draw straight-line boundaries and could never capture a curved or conditional relationship. Inserting a non-linearity between the linear layers is what lets the network bend, fold and carve the input space into arbitrarily complex regions. The common choices:

$$\text{ReLU}(z) = \max(0, z) \quad \text{sigmoid}(z) = 1 / (1 + e^{-z}) \quad \text{GELU} \approx \text{a smooth ReLU}$$

In fact, a single hidden layer with enough neurons can approximate essentially any continuous function — the **universal approximation** result. That is a statement about what an MLP *can* represent, not a promise that training will find it easily; in practice several

moderate layers, trained well, beat one impossibly wide one. But it is why this humble structure is so powerful.

F.4 How it learns — and where you will meet it again

An MLP is trained exactly the way Part 5 will describe: run inputs forward to a prediction, score the error with a loss, use back-propagation to find each weight's gradient, and step downhill with an optimiser. Nothing new is needed. Keep this in your pocket, because the MLP shows up three times in what follows:

- **Inside every transformer block** — the feed-forward sub-layer of Part 3 is precisely an MLP, applied to each token on its own.
- **As the output head** — turning a final vector into class scores (Part 4) is a one- or two-layer MLP.
- **As your baseline model** — for the wrist-sensor system in Part 7, computing a handful of features from each window of accelerometer and gyroscope data and feeding them to a small MLP is the simplest classifier that actually works for recognising cleaning activities. Start there; reach for the transformer when you need to model long stretches of time and fuse many sensors.

> **The atom of the whole thing**
>
> The transformer is not a rival to the MLP — it is *built out of* MLPs, with attention bolted on to let tokens share information first. Master the MLP and you have the atom; the rest of this document is about how those atoms are arranged, fed, and trained.

PART 1

Turning a sequence into numbers

A neural network can only add and multiply numbers, so the first job is to convert the input into numbers in a way that preserves meaning. For language this happens in two steps: **tokenisation** (chop the text into pieces and give each piece an ID) and **embedding** (turn each ID into a vector). A third step, **positional encoding**, records *where* each piece sat in the sequence. For sensor data the very same three steps apply — the "pieces" are short time windows instead of words — which we make explicit in Part 7.

1.1 Tokenisation — from raw input to IDs

Text is first broken into **tokens**: common words become a single token, while rarer or compound words are split into sub-word pieces. A fixed dictionary — the **vocabulary**, often 30,000–50,000 entries — maps every possible token to an integer ID. Sub-word tokenisation means the model is never stumped by a word it has not seen: *"deionised"* can arrive as *"de"* + *"ionis"* + *"ed"*. Nothing is understood yet; these are just row-numbers. (The sensor analogue, in Part 7, is to slice the continuous signal into short overlapping *windows* — each window is one "token".)



Figure 2. A sentence split into sub-word tokens, each mapped to its integer ID in the vocabulary. The IDs carry no meaning on their own — they are addresses.

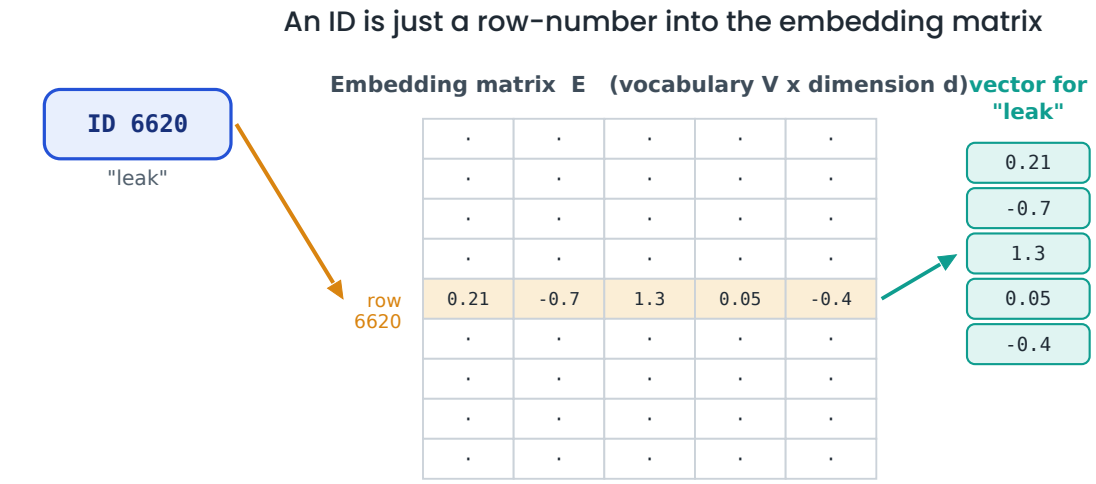
1.2 Embeddings — from IDs to vectors

An integer ID is meaningless arithmetically (ID 6620 is not "twice" ID 3310). So each ID is used to look up a row in a giant table called the **embedding matrix**, written **E**. If the vocabulary has V tokens and we choose a model width of d numbers per token, then E is a grid with V rows and d columns. Token ID *i* simply copies out row *i* of E — a vector of d numbers. That vector is the token's starting representation.

$$x_i = E[\, i \,] \text{ (E has shape } V \times d \text{ ; the vector } x_i \text{ has d numbers)}$$

The crucial point: **the numbers in E are learned**. They begin random, and training gradually arranges them so that tokens used in similar ways acquire similar vectors. Nobody tells the model that *"chiller"* and *"AHU"* are both HVAC equipment — it discovers

this because those words appear in similar contexts, and so their rows in E get pulled into a similar region of the space. (For sensor windows there is no lookup table; instead a small linear layer — an MLP — turns each window straight into its vector. Same idea, computed rather than looked up.)



No arithmetic - the model simply copies out row 6620. Those d numbers ARE the word, as far as the network is concerned.

Figure 3. Embedding is a pure table lookup: the ID selects a row of E and copies out its d numbers. Those numbers are the token, as far as the rest of the network is concerned.

Because every token is now a point in a d-dimensional space, **meaning becomes geometry**. Direction and distance carry information. The standard way to measure "how similar" two vectors are is the **cosine** of the angle between them — effectively the dot product after adjusting for length. Two vectors pointing the same way score near 1 (very similar); at right angles, near 0 (unrelated).

$$\text{similarity}(\, a \,,\, b \,) = (\, a \cdot b \,) \,/\, (\, \|a\| \, \|b\| \,) \text{ where } a \cdot b = \Sigma_k \, a_k \, b_k$$

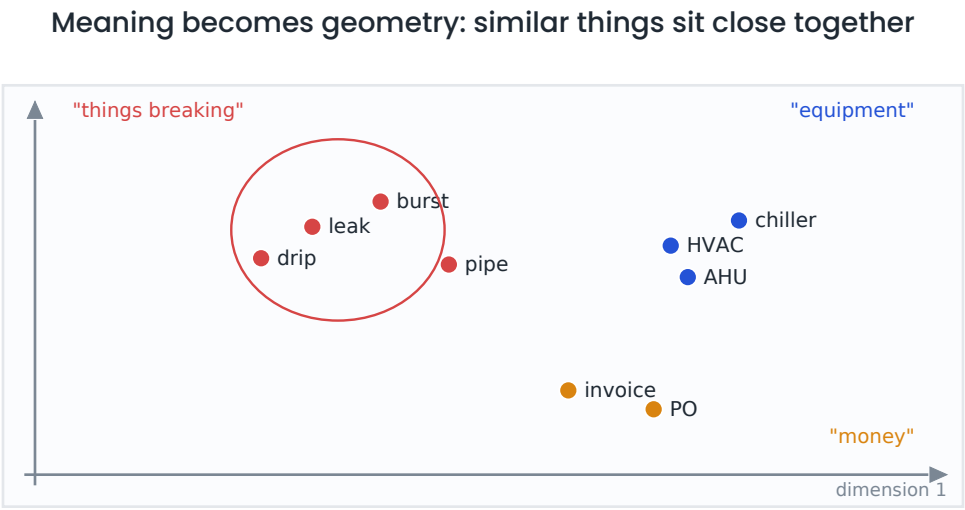


Figure 4. A cartoon of the embedding space (real ones have hundreds of dimensions). Items that play similar roles cluster; unrelated ones sit far apart. The model reasons by moving around in this space — and, usefully for you, two sensor windows of the same cleaning motion will likewise land close together.

1.3 Positional encoding — putting order back in

There is a subtlety. The attention mechanism we are about to meet treats its input as a *set* — it has no built-in notion of first, second, third. But *"valve closed, then pressure rose"* and *"pressure rose, then valve closed"* mean very different things — and, for you, the order of motions is what distinguishes one cleaning action from another. So we must inject position explicitly. The classic method adds a **positional encoding** vector to each token's embedding: a fixed pattern of sine and cosine waves of many different wavelengths, sampled at that token's position.

$$\mathrm{PE}(\mathrm{pos}, 2k) = \sin\left(\mathrm{pos} / 10000^{2k/d}\right) \quad \mathrm{PE}(\mathrm{pos}, 2k+1) = \cos\left(\mathrm{pos} / 10000^{2k/d}\right)$$

Each position gets a unique fingerprint across those waves, and — because the sine and cosine of a shifted angle can be written in terms of the originals — the pattern makes *relative* distances easy for the network to compute. Modern models often use variants (learned position vectors, or "rotary" encodings that rotate the vectors instead of adding to them), but the purpose is unchanged: **tell each position where it sits.** The input to the first transformer block is therefore embedding + position, one vector per token.

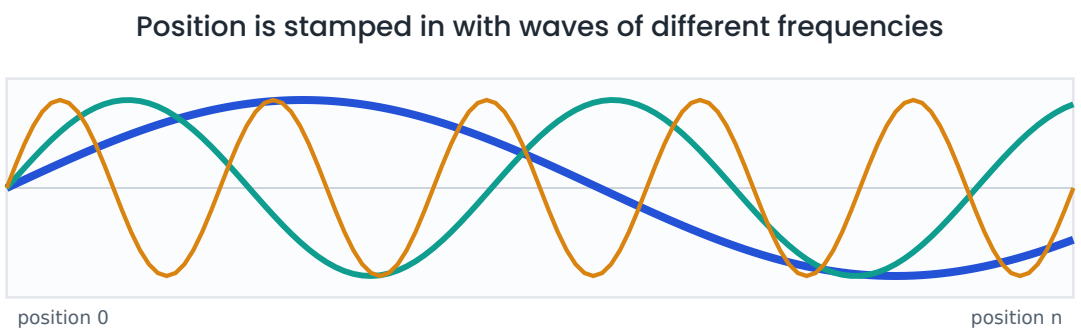


Figure 5. Position is encoded as samples from waves of different frequencies. Low frequencies separate far-apart positions; high frequencies distinguish neighbours.

PART 2

Attention — the heart of the machine

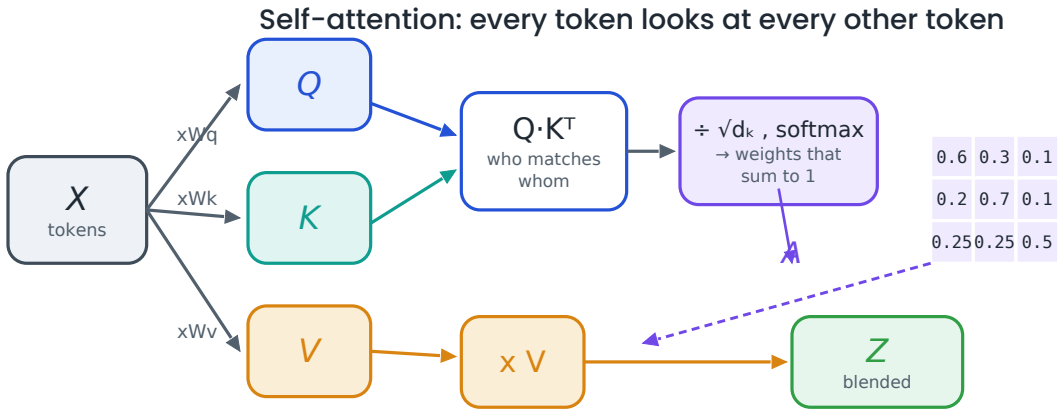
Here is the problem attention solves. After Part 1 we have one vector per token, but each vector still only knows about *its own* word. The vector for *"it"* is identical whether *"it"* refers to a pump or to a policy. To understand language, each token must be allowed to look at the other tokens and pull in the information it needs. **Attention is the mechanism that lets every token gather context from every other token**, and it is the one genuinely new idea that made transformers work.

2.1 Queries, Keys and Values

The metaphor is a library search. Each token issues a **Query** ("what am I looking for?"), and every token also advertises a **Key** ("here is what I am about"). A query is compared against all keys to decide relevance; the token then collects a blend of everybody's **Value** ("here is what I actually contribute"), weighted by that relevance. Query, Key and Value are each produced by multiplying the input X by a *learned* weight matrix — these three grids, plus a fourth (W_o) used later, are the only things attention learns:

$$Q = X\,W_q \quad K = X\,W_k \quad V = X\,W_v$$

If X holds n tokens each of width d (shape n × d), then W_q, W_k, W_v have shape d × d_k, and Q, K, V come out as n × d_k. In words: the same three grids are applied to every token, projecting each token's vector into a "what I want", a "what I offer as a label", and a "what I carry" version of itself.



Each output row is a weighted blend of all the value vectors - the weights say how much attention to pay.

Only four small matrices are learned: Wq, Wk, Wv (the projections) and Wo (mixing the result back). Everything else is fixed arithmetic.

Figure 6. Self-attention end to end. X is projected into Q, K, V; queries are matched against keys to produce a grid of weights A; those weights blend the values into the output Z.

2.2 The one formula the model is built around

With Q, K and V in hand, attention is a single expression. It is worth memorising, because a transformer is essentially this formula stacked dozens of times:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(Q\,K^{T} / \sqrt{d_k} \right) \cdot V$$

Read it inside-out. **Q K^{T}** is every query dotted with every key: a score for each ordered pair of tokens, saying how well token *i*'s query matches token *j*'s key. **÷ √d$_k$** is a scaling constant we justify in a moment. **softmax** turns each row of scores into positive weights that sum to 1 — a genuine set of attention percentages. Multiplying those weights by **V** blends the value vectors: each token's output is a weighted average of what every token contributes, with the weights chosen by relevance. The result Z has the same shape as the input — one refreshed vector per token — so the operation can be repeated.

2.3 The same thing, in actual numbers

Abstractions slide past the eye, so here is the entire calculation on a tiny example you can check by hand: three tokens — *“pump”, “is”, “leaking”* — with width d = 4 and projection size d$_k$ = 2. The input X and the three learned grids are:

X =

1	0	1	0
0	2	0	1
2	1	0	1

W$_q$=

1	0
0	1
1	0
0	1

W$_k$=

0	1
1	0
1	1
1	0

W$_v$=

1	0
0	1
1	1
0	1

Each row of X is one token; each column of a weight grid is one output feature.

Multiplying (row by column, the ordinary way) gives the queries, keys and values — three tables, one row per token:

Q =

2	0
0	3
2	2

K =

1	2
3	0
2	2

V =

2	1
0	3
2	2

For instance the query for “pump” = [1,0,1,0]·W$_q$ = [1·1+1·1 , 1·0+1·0] = [2, 0].

Now the scores S = Q K^{T}: dot every query with every key. Scale each by 1/√d$_k$ = 1/√2, then apply softmax along each row to get the attention weights A (each row sums to 1):

S = Q K^{T} =

2	6	4
6	0	6
6	6	8

÷√2, softmax →

A =

0.05	0.77	0.19
0.50	0.01	0.50
0.16	0.16	0.67

Worked cell: “leaking” vs “leaking” scores Q$_2$·K$_2$ = 2·2 + 2·2 = 8; scaled 8/√2 ≈ 5.66; after the row softmax that becomes [0.16, 0.16, 0.67] — here “leaking” attends mostly to itself.

Finally, multiply the weights by the values, Z = A · V. Each output row is that token's personalised blend of all three value vectors:

$Z = A \cdot V =$

0.46	2.72
1.99	1.51
1.67	2

← one refreshed vector per token

That is the complete operation — projections, a similarity grid, a softmax, and a weighted sum. Scale d to 512 and n to a few thousand and you have the real thing, unchanged in structure. Notice there is *no loop over words writing rules*; the "reasoning" lives entirely in which numbers ended up in W_q, W_k, W_v during training.

2.4 Softmax, and why we divide by $\sqrt{d_k}$

Softmax converts a row of arbitrary scores z into probabilities: exponentiate each (making it positive and exaggerating differences), then divide by the total so they sum to 1.

$$\text{softmax}(z)_i = e^{z_i} / \Sigma_j e^{z_j}$$

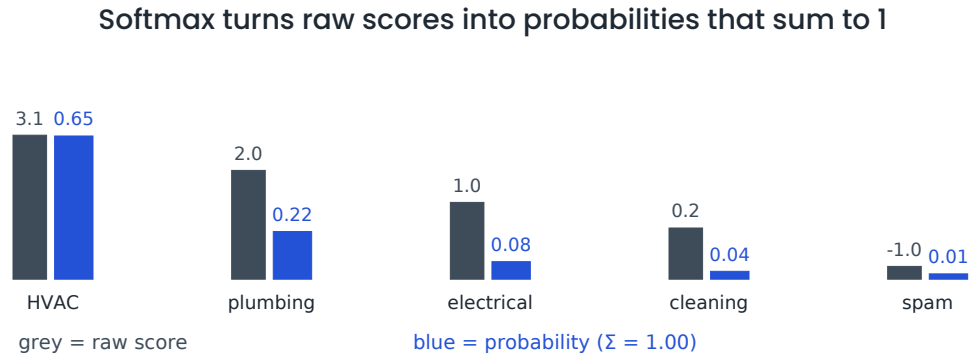


Figure 7. Softmax turns raw scores (grey) into a probability distribution (blue) that sums to 1. Larger scores win a disproportionate share — it is a soft "pick the best".

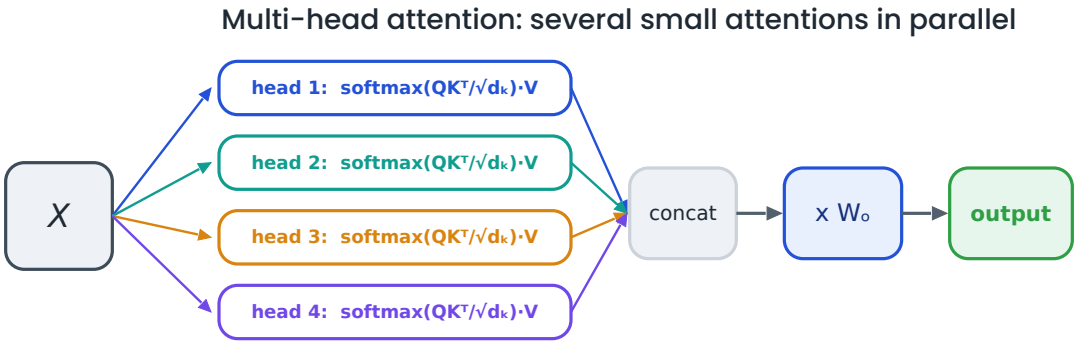
Why ÷ √d — the variance argument

A score is a dot product of two d_k-dimensional vectors: $Q_i \cdot K_j = \Sigma_k\, q_k k_k$, a sum of d_k terms. If the entries are roughly independent with variance 1, the variance of that sum is about d_k, so a typical score grows like $\sqrt{d_k}$. Left unchecked, with $d_k = 64$ the scores swing by ±8 or more; the softmax of such large numbers saturates — one weight becomes ≈ 1 and the rest ≈ 0 — and where the output is flat, the **gradient is nearly zero and learning stalls**. Dividing by $\sqrt{d_k}$ rescales the scores back to variance ≈ 1, keeping softmax in its responsive range. One symbol; a real numerical fix.

2.5 Multi-head attention

One attention calculation forces every token to compress all its relationships into a single set of weights. That is limiting: "which asset is this about?" is a different question from "what is the fault verb?" which is different again from "where is it?". So we run several attentions in parallel, called **heads**. The width d is split into h slices of size d/h; each head gets its own small W_q, W_k, W_v and attends independently. Their outputs are concatenated back to width d and passed through the final learned grid W_o that mixes them:

$$\text{MultiHead} = \text{Concat}(\text{head}_1, \ldots, \text{head}_h) \cdot W_o \quad \text{head}_i = \text{Attention}(Q_i, K_i, V_i)$$



Each head can specialise - one tracks the equipment ID, another the fault word, another the location.

Splitting d into h heads costs almost nothing: per-head width is d/h, so total work is unchanged.

Figure 8. Multi-head attention. Several small attentions run in parallel, each free to specialise, then their results are combined. Because each head is narrower (d/h), the total cost is about the same as one full-width head.

Facilities-Management lens

On a work-order thread — a customer message, three technician replies, a parts note — attention is what lets the model connect *"it is still doing it"* in the last line back to *"the AHU-3 fan is surging"* in the first. One head can learn to track the equipment reference across the whole thread while another tracks the escalating urgency. That cross-referencing is exactly what keyword rules and template systems cannot do.

2.6 A note on masking (for models that generate)

When a model must *generate* text left to right — drafting a reply, say — it must not peek at words it has not written yet. This is enforced by **causal masking**: before the softmax, every score for a "future" position is set to $-\infty$, so its weight becomes exactly 0. Same machinery, one small change — each token may attend only to itself and to earlier tokens. A model used purely to *classify* or *extract* from a finished document usually skips the mask and lets every token see the whole text.

PART 3

Assembling the block, and stacking it deep

Attention is one sub-layer. A real **transformer block** wraps it together with a second sub-layer and two supporting tricks — **residual connections** and **layer normalisation** — that are what make deep stacks trainable at all. Then the whole block is repeated L times. Depth is where the model's capacity to reason in stages comes from.

3.1 Residual connections — do not replace, refine

Every sub-layer adds its output back onto its input rather than replacing it:

$$\text{output} = x + \text{Sublayer}(x)$$

This tiny **+ x** matters enormously. It means each sub-layer only has to learn a *correction* to the running representation, not rebuild it from scratch. It also creates an uninterrupted path — the **residual stream** — along which information (and, during training, the error signal) can flow straight through dozens of layers without being diluted. Without residuals, deep transformers simply do not train: the signal degrades layer by layer until nothing useful survives. With them, you can stack layers freely.

3.2 Layer normalisation — keeping the numbers civilised

As vectors pass through many layers, their magnitudes can drift — some components exploding, others vanishing — which destabilises training. **Layer normalisation** resets each vector before a sub-layer: it subtracts the vector's own mean and divides by its own standard deviation (computed across that token's d components), then rescales by two *learned* vectors γ and β so the network can still choose the scale it wants.

$$\text{LN}(x) = \gamma \odot (x - \mu) / \sqrt{\sigma^2 + \varepsilon} + \beta \quad (\mu, \sigma \text{ over the d components ; } \odot \text{ is element-wise})$$

The effect is that every sub-layer receives inputs in a predictable numerical range, which lets you use a healthy learning rate without the training diverging. Modern models place the normalisation *before* each sub-layer ("pre-norm"), which is what the block diagram shows.

3.3 The feed-forward network — where most of the knowing lives

After attention has let a token gather context, the second sub-layer lets it *think* about what it gathered. This is a small two-layer network applied to **each token independently**: expand the vector from width d up to 4d, apply a non-linearity, then contract back to d.

$$\text{FFN}(x) = \text{GELU}(x W_1 + b_1) W_2 + b_2 \quad (W_1: d \times 4d,\ W_2: 4d \times d)$$

The non-linearity — **GELU**, a smooth cousin of the "pass positives, block negatives" function ReLU(x)=max(0,x) — is essential: without a bend somewhere, stacking linear layers would

collapse into a single linear layer and the model could only draw straight lines. The expansion to 4d gives the layer room to compute many intermediate features before compressing back. Because W_1 and W_2 are large ($8d^2$ parameters versus attention's $4d^2$), **roughly two-thirds of a transformer's parameters live in these feed-forward layers** — they are the model's main store of learned knowledge.

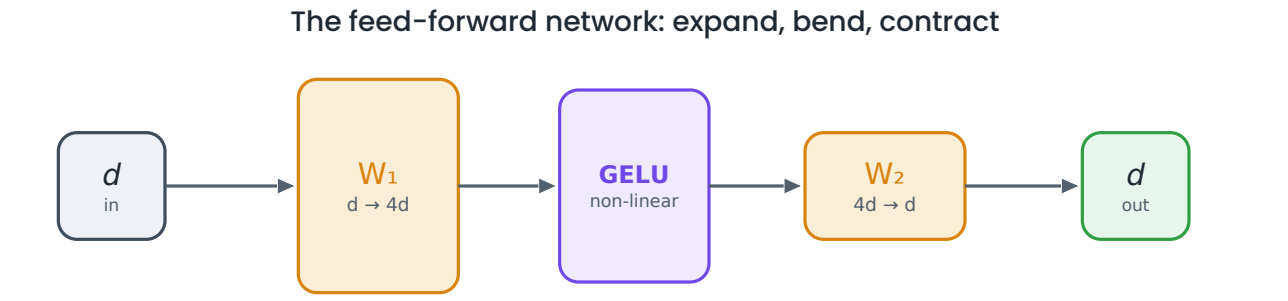


Figure 9. The feed-forward sub-layer: widen to 4d, bend with a non-linearity, narrow back to d. Applied to each token on its own, in parallel.

3.4 The block, and why depth helps

Put it together: normalise, attend, add back; normalise, feed-forward, add back. That is one block. Its output is a set of vectors the same shape as its input, so blocks chain trivially — the output of block 1 is the input of block 2, and so on for L blocks.

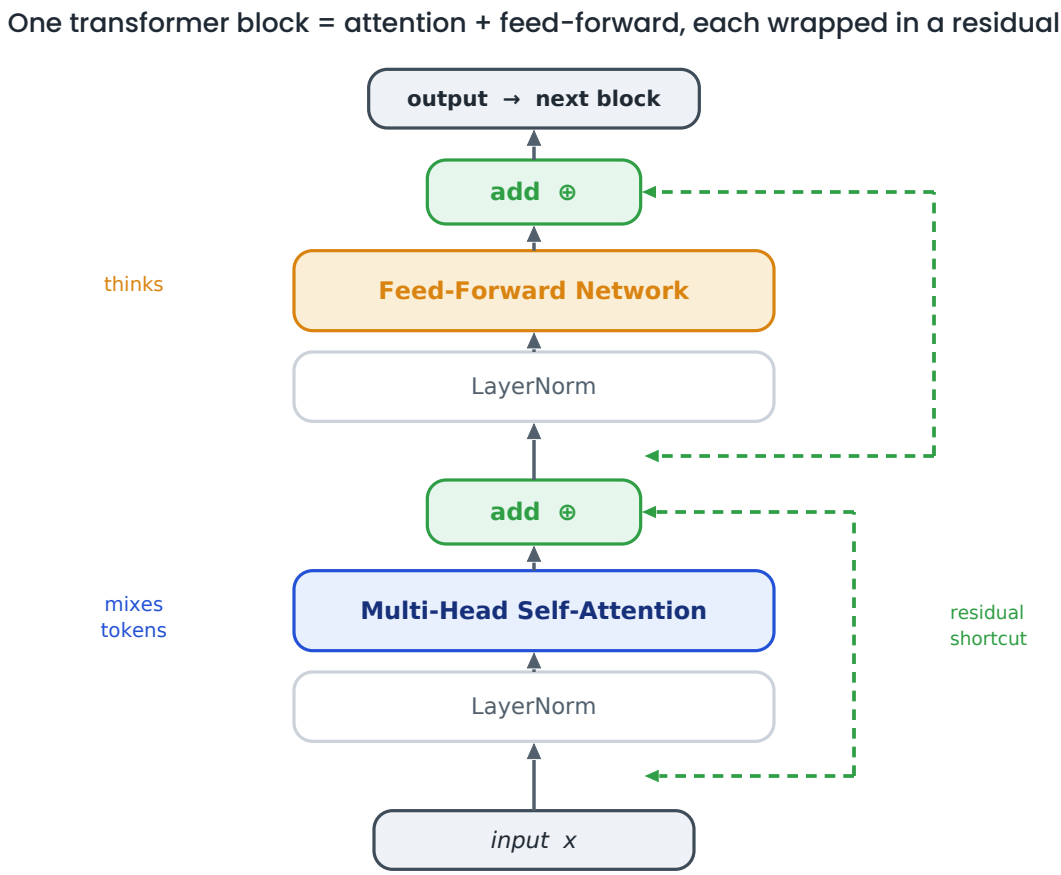


Figure 10. One complete block. Two sub-layers (attention, then feed-forward), each normalised on the way in and added back via a residual shortcut. Stack L of these.

Why does stacking help? Because each block can build on the last. A useful — if simplified — picture is that early blocks resolve local, surface structure (word boundaries, grammar, which token an asset tag attaches to), middle blocks assemble phrases and relationships, and later blocks operate on task-level meaning (this ticket describes an urgent HVAC failure). Nobody hand-assigns these roles; they emerge from training. More depth means more stages of refinement — which is why, for a fixed width, deeper models are generally more capable, up to the point where you run out of data or patience.

Two sub-layers, two jobs

A handy way to hold the two sub-layers apart: **attention moves information *between* tokens** (it is the only place tokens talk to each other), while the **feed-forward network processes each token on its own** (it is the only place with room to store learned facts). Alternating them — mix, think, mix, think — is the entire rhythm of a transformer.

PART 4

From the final layer to an answer

After the last block we still have one vector per token — but now each vector is richly context-aware. The final step converts these vectors into whatever the task actually needs: the next word, a category, an extracted field, or a number.

4.1 The output head — turning a vector back into words

To predict the next token, take the final vector for the last position and multiply it by an **un-embedding** matrix W^U of shape d × V. This produces V numbers — one **logit** (raw score) per vocabulary entry. A softmax over those V logits gives a probability for every possible next token. (Often W^U is simply the transpose of the input embedding E, reused to save parameters — "weight tying".)

$$\text{logits} = h_{\text{final}} \cdot W^U \quad P(\text{next token}) = \text{softmax}(\text{logits}) \quad (\text{V numbers, summing to 1})$$

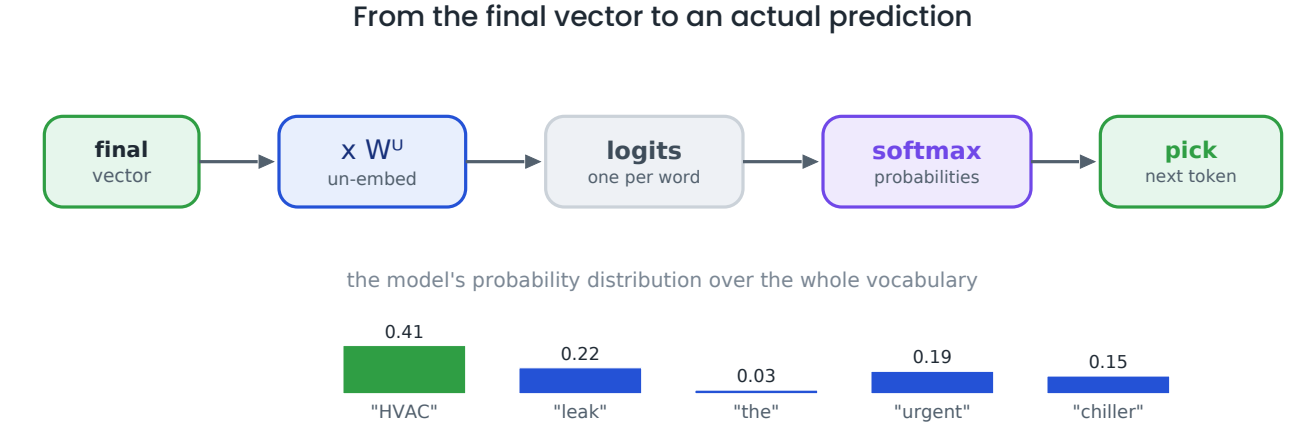


Figure 11. The output head. The final vector is projected onto the whole vocabulary to get one score per word, then softmax turns those into a probability distribution.

4.2 Generating text, one token at a time

A model writes text **autoregressively**: predict a distribution over the next token, choose one, append it to the input, and run the whole network again to predict the token after that. Repeat until a stop token appears. How the token is *chosen* shapes the output:

Greedy — always take the highest-probability token: deterministic, safe, sometimes flat. **Sampling** — draw randomly in proportion to the probabilities: more varied and natural. **Temperature** T rescales the logits before softmax (divide by T): T < 1 sharpens toward the favourite (more predictable), T > 1 flattens (more adventurous). **Top-p** restricts the draw to the smallest set of tokens whose probabilities add up to p, trimming the long tail of nonsense. For a facilities tool drafting dispatch notes you would keep temperature low — you want correct and boring, not creative.

Why writing is slower than reading

Autoregression is also why generation is inherently sequential and therefore *slow*: token 200 cannot be computed until tokens 1–199 exist. Reading or classifying a finished document, by contrast, processes every token at once. Keep this asymmetry in mind — it is the root of the performance discussion in Part 6, and it argues for using the smallest model that does the job.

4.3 Not everything is a chatbot

For facilities work the "answer" is often not free text at all. The same stack of blocks — the **backbone** — can be finished with different **heads**:

- **A classification head** for triage. Pool the final vectors into one summary vector, pass it through a small learned matrix down to your categories, softmax → a probability over, say, {Plumbing, HVAC, Electrical, Cleaning} or {P1, P2, P3, P4} priority. The answer is a label with a confidence, not a sentence.
- **An extraction / tagging head** that labels each token — this span is an asset tag, that one is a location, that one a date — to pull structured fields out of a messy free-text ticket.
- **No head at all — just the vectors.** The context vectors themselves are high-quality *embeddings*; two tickets describing the same fault land close together, which powers semantic search, de-duplication, and clustering of your work-order history.

$$\text{classify: } p = \text{softmax}(\ \text{pool}(h_{1..n}) \cdot W_{cls}\ )\ (\ W_{cls}\text{: } d \times \text{number-of-classes}\ )$$

Facilities-Management lens

This is the most important practical point for your project: **one fine-tuned backbone can serve several heads at once**. Triage (classification), field extraction (tagging), thread summarisation and reply drafting (generation), and "find similar past jobs" (embeddings) can all ride on the same small model. You are not building four AIs — you are building one, and reading it four ways.

PART 5

How it learns: training

Everything so far assumes the weight matrices already hold good numbers. Training is how they get there. It is, at heart, a staggeringly repetitive process: show the model an example, measure how wrong its prediction was, work out how each weight contributed to that error, and nudge every weight a hair in the direction that would have made the error smaller. Do this hundreds of millions of times.

5.1 The objective: predict the next token

The elegant trick behind language models is that the training signal is **free**. You do not need humans to label anything — ordinary text labels itself. Take any sentence, hide the next word, and ask the model to predict it; the true next word (which you can see) is the correct answer. This is called **self-supervised** learning, and it is why models can be trained on enormous piles of raw text. To do next-word prediction well across millions of examples, a model is quietly forced to learn grammar, facts, and a good deal of reasoning — because all of those help it guess better.

5.2 The loss: measuring wrongness with cross-entropy

We need a single number that says how wrong a prediction was. The model output is a probability distribution over the vocabulary; the truth is "it was token *t*". The **cross-entropy loss** is simply the negative logarithm of the probability the model assigned to the correct token:

$$L = -\sum_i y_i \log p_i = -\log p_{(\text{correct token})} \quad (\ y \text{ is 1 for the true token, 0 elsewhere}\ )$$

It behaves exactly as a good error score should. If the model gave the correct token probability 0.9, the loss is $-\log(0.9) \approx 0.11$ — small, barely wrong. If it gave it only 0.1, the loss is $-\log(0.1) \approx 2.30$ — large, very wrong. Confident and correct is cheap; confident and wrong is punished heavily. Averaged over a whole dataset, this single number is what training drives downward. (Exponentiate it and you get **perplexity**, e^{L} — loosely, "how many words the model was effectively choosing between"; lower is better.)

5.3 Gradient descent: rolling downhill

Think of the loss as a landscape stretched over the model's millions of weights: every possible setting of the weights gives some average loss, and we want the lowest valley. We cannot see the whole landscape, but at our current point we *can* compute the **gradient** — the direction of steepest increase, and its slope, for every weight. Step a little the opposite way (downhill) and the loss drops. Repeat.

$$W \leftarrow W - \eta \cdot \nabla_W L \quad (\ \eta = \text{learning rate}\ ;\ \nabla = \text{gradient, the slope in every direction}\ )$$

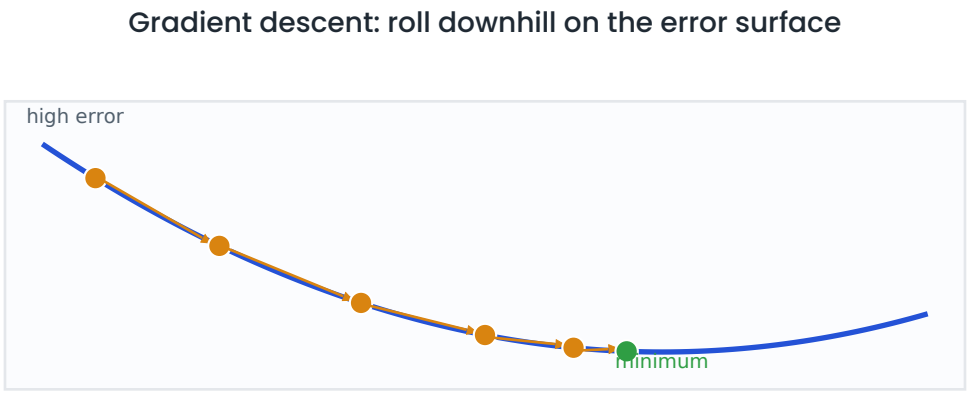


Figure 12. Gradient descent. Each step moves every weight a small amount (set by the learning rate η) in the downhill direction. Too large a step overshoots; too small and it crawls.

The **learning rate** η is the single most important training knob: too big and the steps overshoot the valley and the loss diverges; too small and training takes forever. In practice it is scheduled — warmed up, then slowly decayed.

5.4 Backpropagation: the chain rule at scale

How do we get the gradient for millions of weights buried deep in the network? With the **chain rule** from calculus, applied mechanically. The idea: if you know how sensitive the loss is to a layer's output, you can work out how sensitive it is to that layer's inputs and weights, by multiplying by the layer's own local slope. So you compute the error at the very end and pass it *backwards*, layer by layer, each layer multiplying the incoming sensitivity by its local derivative. Hence **back-propagation**.

A one-line example makes it concrete. Suppose a single weight does $y = w \cdot x$, and the loss is $L = \tfrac{1}{2}(y - t)^2$ for a target t. Then the local slopes multiply:

$$\partial L/\partial w = (\partial L/\partial y) \cdot (\partial y/\partial w) = (\, y - t \,) \cdot x$$

So this weight's gradient is “how wrong the output was” times “how much this weight influenced it”. Every weight in a transformer gets its gradient the same way — just with longer chains of multiplications. Modern frameworks (PyTorch, JAX) do this bookkeeping automatically (**automatic differentiation**): you write only the forward calculation, and the backward pass is generated for you.

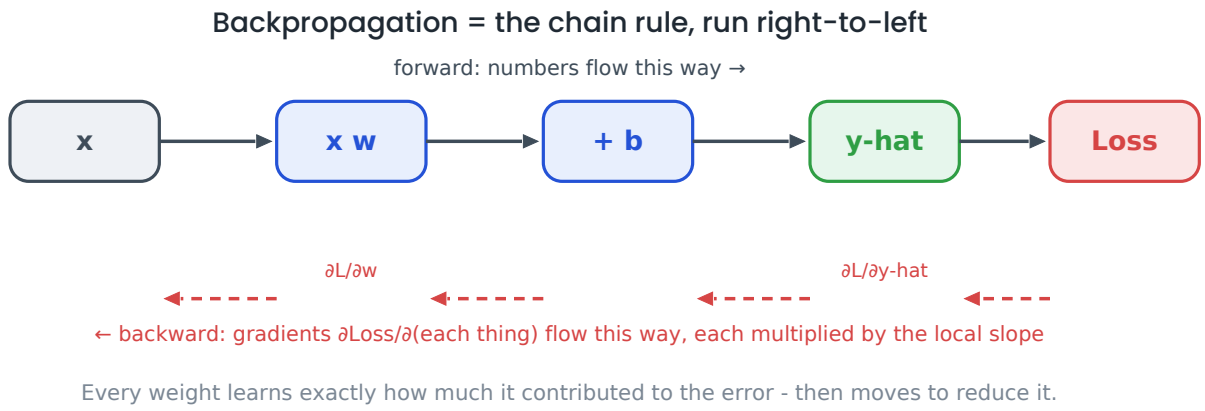


Figure 13. Forward, the numbers flow left to right to produce the loss. Backward, the gradient of the loss with respect to each quantity flows right to left, each step multiplied by a local slope. Every weight learns its share of the blame.

5.5 The optimiser: Adam

Plain gradient descent works but is twitchy — the gradient from one small batch is a noisy estimate, and a single learning rate suits some weights better than others. The **Adam** optimiser, used almost universally, fixes both. It keeps a running average of recent gradients (**momentum** m, which smooths the noise and builds speed in consistent directions) and a running average of their squared size (v, which gives each weight its own adaptive step — small where gradients are wild, large where they are gentle):

$$m \leftarrow \beta_1 m + (1-\beta_1) g \quad v \leftarrow \beta_2 v + (1-\beta_2) g^2 \quad W \leftarrow W - \eta \cdot \hat{m} \,/\, (\sqrt{\hat{v}} + \varepsilon)$$

(g is the gradient; $\hat{m}$, $\hat{v}$ are the averages with a small start-up correction.) You do not need to memorise this — the point is that Adam turns a raw, noisy gradient into a stable, per-weight update, and it is why deep networks train reliably rather than thrashing.

5.6 The loop, in full

Assemble the pieces and the entire training procedure is a short cycle, repeated relentlessly. Data is fed in **batches** (a few dozen to a few thousand examples at once, for efficiency and a less noisy gradient); one full pass over the dataset is an **epoch**.

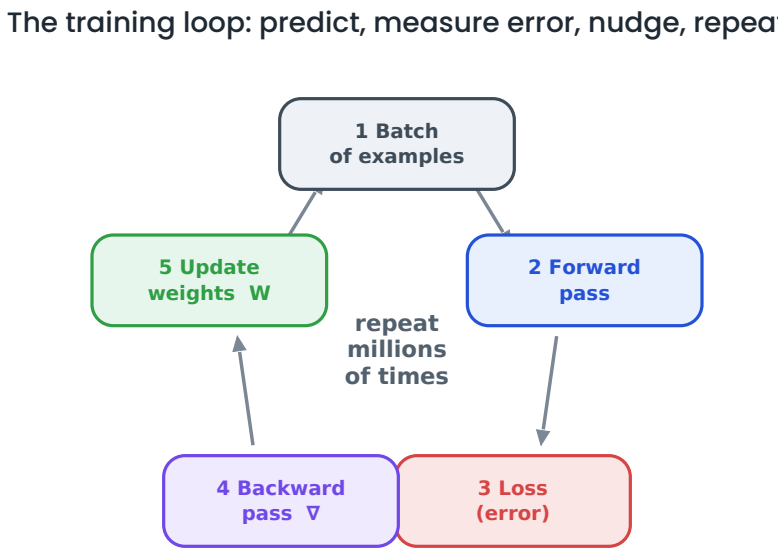


Figure 14. The training loop. Forward pass to a prediction, cross-entropy loss, backward pass for the gradients, an Adam step to update the weights — then the next batch. Millions of laps, and the pile of numbers slowly becomes competent.

5.7 You will fine-tune, not train from scratch

Training a capable model from random noise needs trillions of words and a fortune in compute — out of reach and unnecessary for a focused tool. The standard, sane path has three tiers:

- **Pre-training** (done once, by someone with a data centre): a general model learns language from a huge generic corpus. You download the result — an open base model — rather than doing this yourself.
- **Fine-tuning** (done by you, cheaply): continue training that base model for a while on *your* corpus — work orders, manuals, resolved tickets with their categories — so it adopts your vocabulary, tone, and task. Hours on a single GPU, not months on a cluster.
- **Parameter-efficient fine-tuning (LoRA)**: cheaper still — freeze the big model entirely and train only a tiny set of added weights.

LoRA deserves a line of maths because it is so useful at your scale. Instead of updating a large weight matrix W (say 512 × 512 = 262,144 numbers), you freeze it and learn a low-rank *correction* ΔW = B A, where B is 512 × r and A is r × 512 for a small rank like r = 8:

$$W_{\text{effective}} = W + B\,A \quad \text{trainable: } r(d+k) = 8\times1024 \approx 8\text{k vs } d\cdot k = 262\text{k (about 3\%)}$$

Because you train only these slim factors — often well under 1% of the model's parameters — a laptop-class GPU can fine-tune a model whose full weights it could never afford to update, and you can keep dozens of small LoRA “adapters” (one per task) beside a single shared base model.

5.8 How much data and compute?

A useful rule of thumb: training compute in floating-point operations is roughly **C ≈ 6 · N · D**, where N is the parameter count and D the number of training tokens (the 6 covers one forward and one backward pass over each parameter). Research on *compute-optimal* training suggests a base model wants on the order of **20 tokens of data per parameter** — so a 100-million-parameter base model is well-matched to a couple of billion tokens of pre-training. That is the pre-training budget you are *avoiding* by starting from an open model.

> **The realistic training budget for your project**
>
> For fine-tuning the numbers are pleasantly small. Adapting an open small model to facilities work might need thousands to tens of thousands of good examples (labelled tickets, cleaned notes), a few hours on one rented GPU, and a budget in the tens to low hundreds of dollars — repeatable whenever your data grows. The expensive, planet-scale part was done once by whoever released the base model; you inherit it for free and specialise cheaply.

PART 6

Running it: the infrastructure

A trained model is just a file of numbers; running it means doing the arithmetic of Parts 1–4 fast enough to be useful. Two very different jobs live here. **Training** happens rarely, is enormously compute-heavy, and can take its time. **Inference** — actually using the model to answer — happens constantly, must feel instant, and is where day-to-day cost and engineering effort go. The two have different bottlenecks, so we treat them separately.

6.1 Why this runs on GPUs

Look back at every operation in this document and it is the same shape: multiply a vector by a matrix, which is thousands of independent multiply-then-add sums. A CPU has a handful of powerful cores that do such sums more or less one after another. A **GPU** has thousands of small cores plus dedicated matrix-multiply units ("tensor cores") that do them all at once, and — just as importantly — very high memory **bandwidth** to feed those cores. Matrix multiplication is embarrassingly parallel, so a GPU runs it one to two orders of magnitude faster than a CPU. That single fact is why the whole field is built on them.

6.2 What has to sit in memory

The main constraint on a GPU is not speed but **memory** (tens of gigabytes of fast on-board HBM). Three things must fit. First, the **weights**. Their count is dominated by the blocks and embeddings:

$$N \approx 12 \cdot L \cdot d^2 + V \cdot d \quad (\text{per block: } 4d^2 \text{ attention} + 8d^2 \text{ feed-forward} = 12d^2)$$

Worked example — a genuinely small model with width d = 768, depth L = 12, vocabulary V = 32,000: the blocks hold $12 \times 12 \times 768^2 \approx 85$ million parameters and the embeddings $V \cdot d \approx 25$ million, so $N \approx 110$ million. The memory they occupy is just N times the bytes per number:

$$\text{weight memory} = N \times \text{bytes/param} \quad 110\text{M} \times 2 \text{ bytes (bf16)} \approx 220 \text{ MB}$$

Second, the **KV cache**. During generation the Keys and Values of every token already processed are kept so they need not be recomputed each step. That store grows with the conversation:

$$\text{KV cache per token} = 2 \cdot L \cdot d \cdot \text{bytes} \quad 12 \cdot 768 \text{ model, bf16: } 2 \cdot 12 \cdot 768 \cdot 2 \approx 36 \text{ KB / token}$$

So a 4,000-token context adds roughly 140 MB per active conversation — and it grows every token, which is why long contexts and many simultaneous users cost memory, not just compute. Third, transient **activations** — scratch space for the current step, scaling with batch size × sequence length. Weights + KV cache + activations + framework

overhead must all fit in the card.

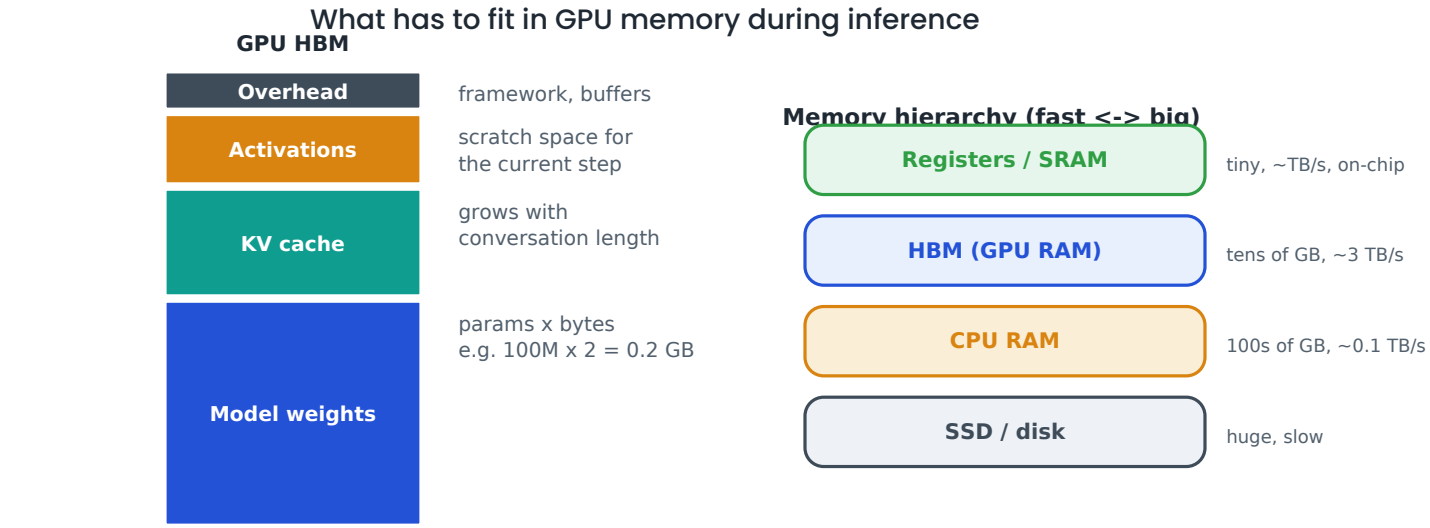


Figure 15. The inference memory budget, and the memory hierarchy behind it. Speed comes from keeping the numbers you need in the fast, small tiers; you pay dearly whenever you must reach down to a slower, bigger one.

6.3 Quantisation: trading precision for room and speed

Weights are trained in 32- or 16-bit precision, but for *inference* you rarely need that many significant figures. **Quantisation** stores each number in fewer bits — halving or quartering both the memory footprint and the amount of data that must be read from memory (which, as the roofline shows, is the real bottleneck). The quality cost is usually small down to 8 bits, and often acceptable at 4:

Precision	Bytes / param	Our 110M model	Quality
FP32 — full	4	440 MB	reference; rarely needed at inference
BF16 / FP16	2	220 MB	essentially identical — the default
INT8	1	110 MB	small, usually invisible loss
INT4	0.5	55 MB	noticeable but frequently fine for narrow tasks

For a facilities model this is the difference between needing a data-centre card and running comfortably on a modest on-site GPU — or even a CPU.

6.4 Prefill vs decode, and the roofline

Inference has two phases with opposite characters. **Prefill** reads your prompt: all its tokens are processed together, one big efficient matrix multiply — the GPU's cores are busy, it is **compute-bound**. **Decode** then generates the answer one token at a time, and each new token requires reading the *entire* set of weights from memory just to produce that single token. Here the cores sit mostly idle, waiting on memory — it is **memory-bound**. The roofline picture captures the divide:

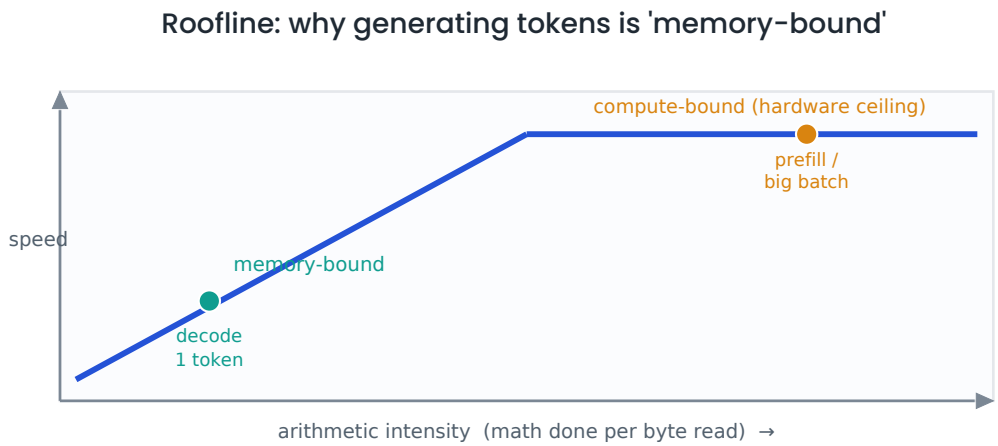


Figure 16. The roofline. Below the knee you are limited by memory bandwidth; above it, by raw compute. One-token-at-a-time decode lives far down the memory-bound slope.

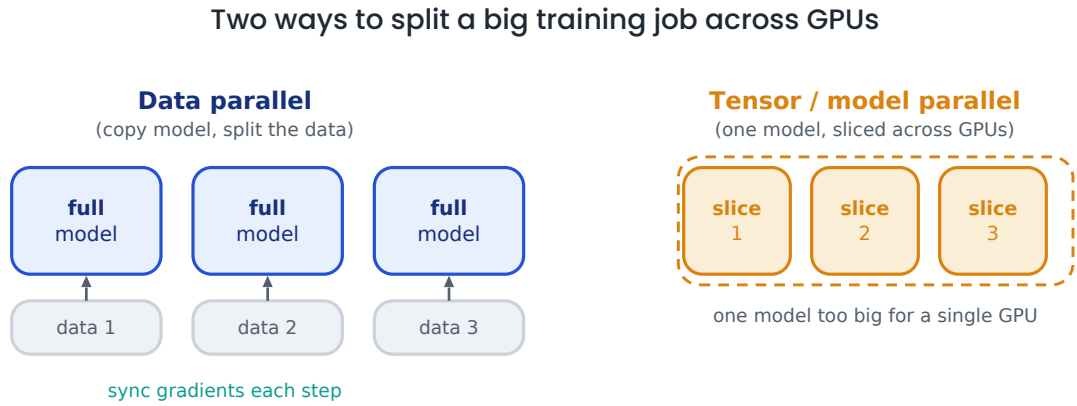
> **Why generation is slow — the arithmetic**
>
> Put numbers on it, for our 110M model at bf16 (220 MB of weights) on a card with ~3.3 TB/s of bandwidth and ~1 PFLOP/s of compute. **Memory** to emit one token: read 0.22 GB ÷ 3,300 GB/s ≈ **66 µs**. **Compute** for one token: about 2N = 220 million operations ÷ 10^{15}/s ≈ **0.2 µs**. The chip spends ~66 µs moving weights and ~0.2 µs doing maths — **over 300× more time waiting on memory than computing**. That is what "memory-bound" means, and it caps single-stream speed at roughly 1 ÷ 66 µs ≈ **15,000 tokens/second** before any batching. The fix is to batch many requests together so each expensive weight-read is reused across dozens of tokens — pushing you rightward toward the compute ceiling and multiplying total throughput.

6.5 Throughput versus latency

This is the central serving trade-off. **Latency** is how fast one user gets their answer; **throughput** is how many users you serve per second per GPU. Because decode is memory-bound, processing requests in **batches** costs barely more time than one request but serves many — a huge throughput win — at the price of a little latency while the batch fills. Modern servers use **continuous batching** (slotting new requests in as others finish) to get most of the throughput with little latency penalty. For an internal facilities tool with bursty, modest traffic, one GPU with sensible batching serves a whole team comfortably.

6.6 Scaling training across GPUs (only if you must)

When a model or its training batch will not fit on one card, the work is split. **Data parallelism** puts a full copy of the model on each GPU, feeds each a different slice of the batch, and averages the gradients every step. **Tensor / model parallelism** slices a single model across GPUs when even one copy is too big. These techniques are how billion- and trillion-parameter models are trained — and, reassuringly, they are things you will almost never need.



Small FM models rarely need this - one GPU is plenty. It matters only when the model or batch won't fit.

Figure 17. Two ways to spread training across GPUs. A small facilities model fine-tunes on a single card, so this is background knowledge rather than a task on your plate.

6.7 Where it runs, and what it costs

What you are doing	Hardware	Rough cost	Notes
Fine-tune a small model	1 rented GPU, a few hours	tens–low hundreds $	repeat whenever data grows
Serve inference (100M–1B)	1 modest GPU	hundreds $/mo, or owned	one GPU serves a whole team
Tiny model at the edge	CPU / small on-site box	≈ hardware you own	fully private, very low latency

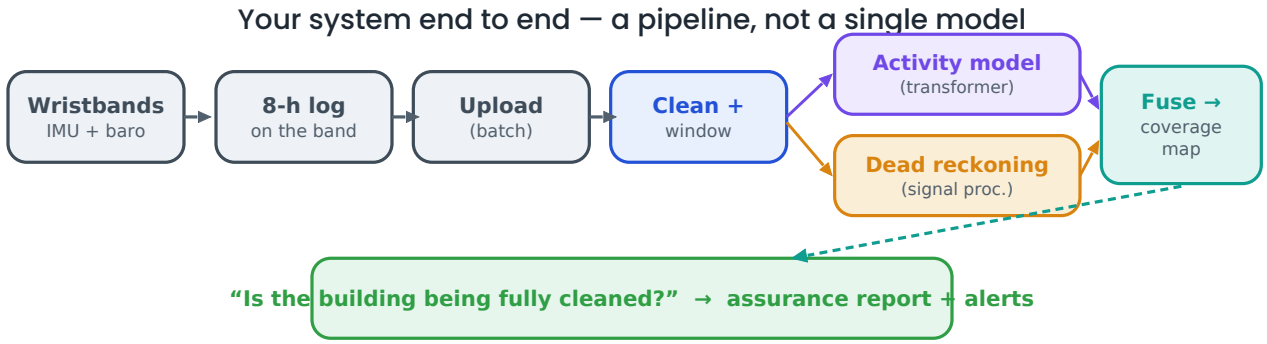
Facilities-Management lens

For facilities management the small-and-local option is not a compromise — it is an advantage. A quantised small model runs on hardware you already own, so tenant data, floor plans and contracts **never leave your network**; there is no per-call cloud bill; and responses are fast because there is no round-trip. Privacy, cost and latency all point the same way: the smallest model that does the job, run on your own metal.

PART 7

Your system: from wrists to cleaning assurance

Now the machine meets your project. The goal: wristbands worn by cleaners stream motion data for a full shift; that data is uploaded and a model must work out **what** cleaning was done, **how thoroughly**, and **where** in the building — and roll all of that up into an honest answer to "are these large, multi-level buildings actually being cleaned properly?". This is very achievable, but it is important to be clear-eyed from the outset: it is **a pipeline of several components**, and the transformer is one powerful part of it, not the whole thing.



The transformer of this document is one box — the activity model. The other boxes are classical signal processing and analytics. A good system uses the right tool for each job (see section 7.7).

Figure 12. The end-to-end system. Sensor logs are cleaned and windowed, then read two ways — an activity model (the transformer) and a dead-reckoning module (signal processing) — and fused into a spatio-temporal coverage map that drives the assurance report.

7.1 The data your bands produce

Each band carries an **inertial measurement unit** (IMU): a 3-axis **accelerometer** (linear motion and gravity), a 3-axis **gyroscope** (rotation rate), and a 3-axis **magnetometer** (compass heading) — nine channels in all. One addition is worth insisting on: a **barometer** (air-pressure sensor). It is cheap, and it is by far the most reliable way to tell *which floor* someone is on in a multi-level building, because pressure changes measurably as they take a lift or stairs. Sampling at 25–50 Hz captures all human motion; going higher just drains battery and storage.

The data volume is comfortably small. At 50 Hz, ten channels, over an eight-hour shift, one band produces $50 \times 10 \times 8 \times 3600 \approx$ **14 million samples** — about **29 MB** at two bytes each. Even hundreds of cleaners over months is a modest database, not a big-data problem.

> Offline batch processing is a huge advantage
>
> **Your setup removes the hardest performance constraint in this whole document.** Because data is uploaded *after* the shift and analysed offline, there is no real-time deadline. The "generation is slow / memory-bound" problem of Part 6 simply does not apply — you are not producing text token by token, you are analysing a finished, fixed-length log. You can therefore afford a thorough model and give it the whole shift as context, and run it on ordinary hardware overnight. Batch, offline processing is a gift; spend it on accuracy.

7.2 From sensor stream to "tokens": windowing

A transformer needs a sequence of vectors. Your sensor data is a continuous multi-channel signal, so we slice it into short, overlapping **windows** — a common choice is about 2.5 seconds (128 samples at 50 Hz) with 50% overlap. **Each window is one token.** Turning a window into its vector can be done two ways, and you will likely use both at different stages:

- **Hand-crafted features** — compute a handful of numbers per window: means and variances per axis, signal energy, dominant frequency (from a quick Fourier transform), correlations between axes, jerk. This gives an interpretable vector that a small MLP (Foundations) or gradient-boosted trees can classify immediately. It is your fast, strong baseline.
- **Learned patch-embedding** — feed the raw window through a small linear layer or 1-D convolution, exactly analogous to word embedding, letting the model discover its own features. This is what you feed a transformer.

Either way, add a positional encoding so the order of windows across the shift is preserved, and you now have precisely the input Parts 2–3 expect — a sequence of token vectors. Everything you have already read about attention, blocks and depth now applies verbatim, with windows in the role of words.

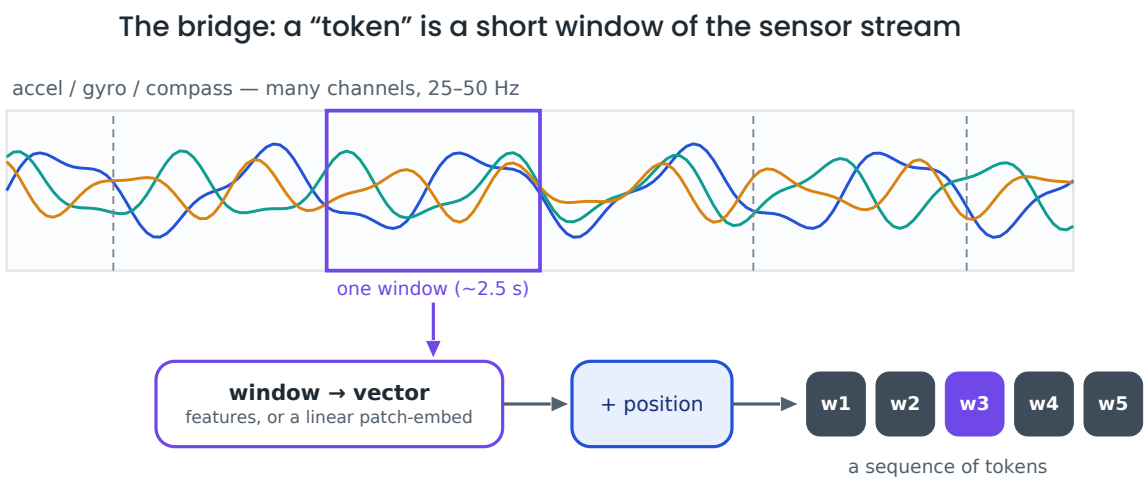


Figure 13. The one idea that connects this whole document to your project: a token is a short window of the sensor stream. Featurise or linearly embed each window, add its position, and feed the sequence to the transformer.

$$\text{token}_t = \text{features}(\ \text{window}_t\ ) \text{ or } \text{Linear}(\ \text{window}_t\ ) \text{ then add positional encoding, exactly as in Part 1}$$

7.3 Task one — recognising the cleaning activity

This is **human activity recognition** (HAR), and it is the part a transformer does genuinely well. Think of it in a hierarchy. At the bottom, each ~2.5-second window is one **micro-activity**: a wiping stroke, a spray, a scrub, a mop pass, a vacuum push, dusting, lifting a bin, walking, standing idle, climbing stairs. Runs of micro-activities make up a **task** (“clean this restroom”, “vacuum the corridor”), and tasks fill a **shift**. The model classifies the windows; simple logic and the temporal model assemble them upward.

Reach for models in order of complexity, and stop when accuracy is good enough:

- **Features → MLP or gradient-boosted trees** — the baseline from Foundations. Often startlingly good for distinct motions, cheap, and interpretable. Build this first; it sets the bar.
- **1-D convolutional network** — learns its own window features from the raw signal; a reliable step up.
- **Transformer** — self-attention over the sequence of windows. This is where it pays to have read Parts 2–3: attention lets the model relate a motion now to the rhythm of the last two minutes, and fuse all nine sensor channels, which is exactly what separates a genuine mopping bout from an incidental arm swing. Use it when long-range context and multi-sensor fusion matter — and to unlock the trick below.

The one dependency that decides everything is labels. Supervised HAR needs ground truth: examples where you know which activity occurred when. You have three ways to get it, best combined:

- **A labelling campaign** — have cleaners perform known activities to a script, or record a handful of shifts with synchronised video and annotate them. A few well-labelled shifts go a long way.
- **Weak labels** — cleaning schedules, area check-ins, or supervisor logs give approximate "this hour was restroom cleaning" signals you can learn from.
- **Self-supervised pre-training** — train the transformer on your *unlabelled* logs first by masking out windows and making it reconstruct them (the sensor version of the next-token game in Part 5). It learns the structure of cleaning motion for free, then only needs a small labelled set to specialise.

> Turn your unlabelled logs into the model
>
> **This is the highest-leverage move in your build.** You will have a mountain of unlabelled sensor logs and only a little labelled data — which is precisely the situation self-supervised pre-training was made for. Pre-train on everything (mask-and-reconstruct sensor windows), then fine-tune on a few carefully labelled shifts. It is the same paradigm that trains the big language models, applied to your wristbands, and it is what lets a small model reach strong accuracy without an enormous labelling budget.

One practical note: a single window's prediction is noisy, so smooth the label stream over time — either let the sequence model see the context (its natural job) or apply a light temporal smoothing pass — before you count a genuine change of activity.

7.4 Task two — how thoroughly (the honest part)

Here rigour demands candour: **no sensor measures "clean".** A wristband records motion, not hygiene. Thoroughness therefore cannot be read off directly; it must be *inferred* from behaviour through proxies — and you must treat it as a proxy. Reasonable signals include time-on-task per area, the intensity and amount of cleaning motion, whether coverage looks *systematic* (methodical passes) or *cursory* (a token wipe), repetition counts, and how a cleaner compares to their own norm and to peers on the same kind of space.

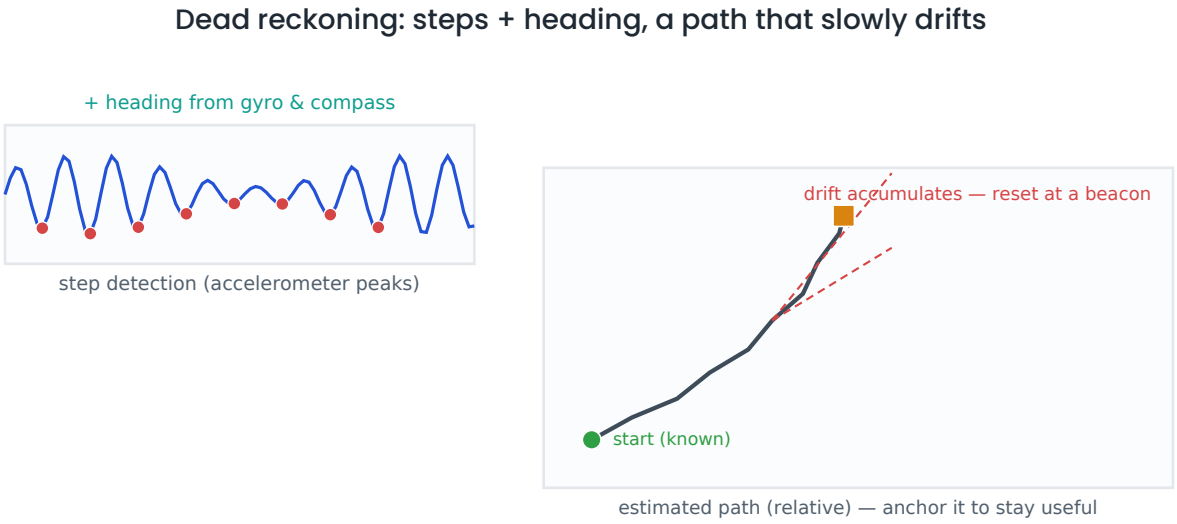
Because these are proxies, they only become trustworthy once **calibrated against ground truth**. That ground truth comes from periodic audits you already can run — inspection scores, supervisor checks, or ATP hygiene swab tests. Those audit outcomes are the labels that tell you which behavioural patterns actually correspond to a clean result; you then train a model to predict the audit outcome from the behaviour, or at minimum validate that your proxy tracks reality. Without that loop, a "thoroughness score" is a confident guess.

> **Start with ‘what’s unusual’, not ‘what’s clean’**
>
> The pragmatic first win needs *no* cleanliness labels at all: **anomaly detection**. You do not need to know what “clean” looks like to notice that a restroom which normally receives six minutes of vigorous activity got forty seconds today, or that an entire scheduled zone shows no cleaning motion. Flagging deviations from the normal pattern catches the cases that matter most — skipped and rushed areas — and is largely unsupervised. Start there; layer calibrated thoroughness scoring on top once audits give you ground truth.

And a firm line: a thoroughness figure should be **decision-support and an early warning, never an automatic verdict** on a person. Surface it to a supervisor, keep a human in the loop, and validate it continuously against audits — both because it is fairer and because it is the only way the number stays honest.

7.5 Task three — where they went (dead reckoning)

To know *where* cleaning happened without a GPS signal indoors, you use **pedestrian dead reckoning**: detect each step from accelerometer peaks, estimate its length, take a heading from the gyroscope (integrated) fused with the compass, and add up the steps into a running position. A barometer tells you the floor. It is elegant — and it must be handled with honesty, because it has two hard limits.



Wrist mounting is the hard case: the arm swings and does the cleaning, so a precise metre-accurate map from a wristband alone is not realistic. Aim for floor (barometer) + coarse zone (beacons) + relative motion.

Figure 14. Dead reckoning turns steps and headings into a path, but small errors accumulate: heading drifts, and position error compounds over time. Periodic anchors are what keep it usable.

First, drift. Every estimate carries a small error, and because you are continually adding steps and integrating rotation, those errors accumulate — heading error grows roughly in proportion to time, and position error faster still. Over a few minutes it is metres; over an eight-hour shift, uncorrected, it is hopeless. **Second, the wrist is the worst place to do this.** The wrist swings independently of the body and is the very limb performing the

cleaning, so extracting clean “steps” and a stable heading from it is far harder than from a phone in a pocket or a foot-mounted sensor. **A precise, metre-accurate map from a wristband alone is not realistic — do not promise one.**

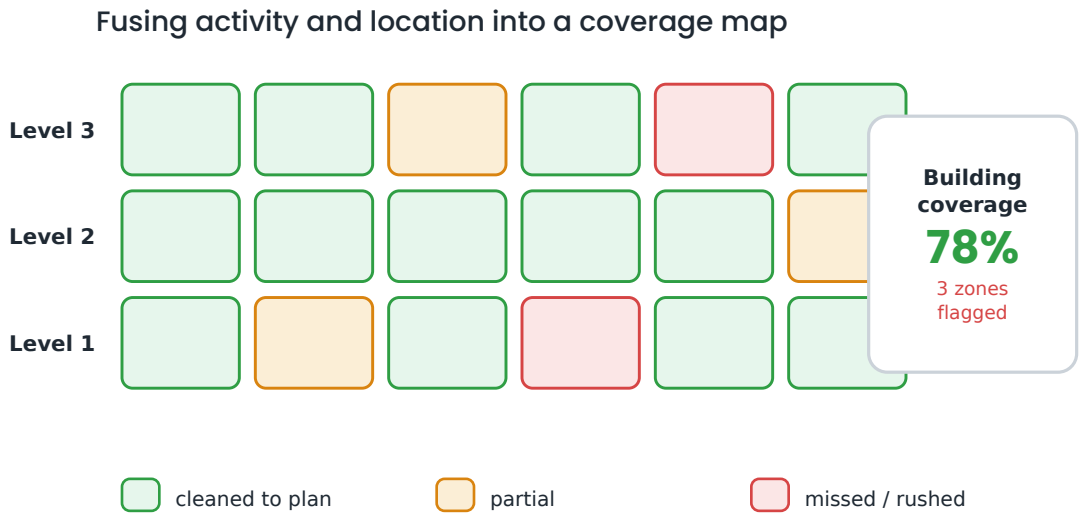
What *is* realistic, and enough for your purpose, is coarser and anchored:

- **Floor** — from the barometer, this is reliable and immediately valuable in a multi-level building.
- **Coarse zone** — resettable to a known location whenever the cleaner passes an anchor: a few cheap Bluetooth beacons at entrances, lifts and supply rooms, or Wi-Fi signal-strength fingerprinting, periodically correct the drift. A known start point (the supply closet) helps too.
- **Relative movement** — even without absolute position, “how much ground was covered” and “was this area visited at all” are informative.

This module is mostly classical **signal processing and sensor fusion** (step detection, complementary or Kalman filters), not a transformer. Learned inertial odometry — a neural network that maps raw IMU to motion — is a legitimate upgrade path, but the wrist makes it hard; treat it as a later experiment, not the foundation.

7.6 Bringing it together — coverage and assurance

The payoff is the fusion. Lay the activity timeline (what cleaning, how much, how thoroughly — section 7.3–7.4) alongside the location timeline (which floor and zone, when — section 7.5), and you get a **spatio-temporal coverage map**: for every zone of every floor, which cleaning it received, for how long, and when — set against the **expected cleaning plan**. Now the headline question has an answer you can defend.



Each cell is a zone × the cleaning it received vs. what the schedule expected. The rollup is analytics, not a neural net.

Figure 15. The deliverable. Activity fused with location, per zone, compared to the schedule, rolled up to a building coverage figure with the problem areas flagged.

From that map fall out the things a facilities manager actually wants: a per-zone coverage status, a list of missed or rushed areas, rollups by shift, by cleaner and by building, trends over weeks, and automatic flags when a scheduled area shows no activity. Note that this final layer is ordinary **analytics and rules** — comparing observed coverage to the plan, computing percentages, detecting gaps — not another neural network. The clever learning happened upstream, in recognising the activity; the assurance report is disciplined bookkeeping on top of it, and is all the more trustworthy for being transparent.

7.7 Is a transformer even the right tool here?

Rigour means asking this directly rather than reaching for the fashionable answer. The truthful verdict is **"yes, for one part; no, for the rest — and that is fine."** A transformer is genuinely strong for the activity-recognition core: it models long stretches of time, fuses many sensor channels, and — crucially — lets you pre-train on your abundant unlabelled logs. But the other jobs are better done by methods built for them, and forcing everything through one end-to-end network would be worse, not cleverer.

Component	Best approach	Why
Recognise the activity	transformer (baseline: feature+MLP / 1-D CNN)	sequence + multi-sensor fusion; pre-train on unlabelled logs
Thoroughness score	features + model, calibrated to audits	a proxy — only valid once tied to real cleanliness checks
Location / dead reckoning	signal processing + sensor fusion	physics-based; learned odometry optional, wrist is hard
Coverage & assurance	analytics + rules + anomaly detection	transparent bookkeeping against the schedule

Two honest caveats worth stating plainly. Simpler models often match a transformer on basic activity recognition with a fraction of the data and compute — so **benchmark the baseline first** and adopt the transformer only where its strengths earn their keep. And for the numeric, physics-driven parts (localisation), classical estimation remains the backbone; deep learning is a garnish, not the main course. Use each tool for what it is best at — the same principle whether you are building for language or for wristbands.

7.8 A staged build plan

Sequence the work so each phase ships something useful and de-risks the next:

- **Phase 0 — foundations.** Fix the sensor set (insist on a barometer), the wrist placement, and a sampling rate; define the activity taxonomy and a machine-readable cleaning schedule per building; and run a small labelling campaign — a few shifts recorded with synchronised video, or scripted activities — so you have honest ground truth.
- **Phase 1 — activity + floor.** Build the features→MLP (or gradient-boosted) HAR baseline and the barometer floor detector. Ship the first real capability: *what* cleaning

happened, on *which floor*. Measure it properly before going further.

- **Phase 2 — transformer + coarse location + coverage.** Self-supervised pre-train on the unlabelled logs, fine-tune the transformer, add beacons for coarse zones, fuse into a coverage map against the schedule, and stand up the dashboard with anomaly flags for skipped and rushed areas.
- **Phase 3 — thoroughness + assurance.** Calibrate a thoroughness score against periodic audits, roll up to per-building assurance, add alerting, and monitor for drift as new sites, equipment and staff appear.

The evaluation discipline that keeps you honest

Measure like a statistician, or you will fool yourself. Report macro-F1 and a full confusion matrix for activity recognition (accuracy alone hides trouble when some activities are rare); floor accuracy, zone accuracy and drift for localisation; and agreement with manual audits for coverage and thoroughness. Above all, **split your train and test sets by cleaner and by building, never at random** — different people and sites are exactly what the model must generalise to, and a random split leaks that away and inflates every number. Test on people and buildings the model has never seen.

Privacy, fairness and trust are part of the design

This system watches workers, so treat it as such. Be transparent with staff about what is collected and why; aim it at **building-level assurance, not individual policing**; keep a human in every consequential decision; mind the labour and privacy law that applies to you; and keep the data on your own infrastructure. Watch for fairness too — motion models can be skewed by body type, handedness or disability, so validate accuracy across your whole workforce, not just on average. The technology is only as good as the trust it is deployed with.

PART 8

The other build: the language side — tickets, manuals, knowledge

Your wristband system (Part 7) is the hard, novel project. But the very same transformer machinery powers a second, complementary facilities capability that is far more well-trodden and quick to stand up: reading and writing the **text** your operation already drowns in — work orders, service tickets, technician notes, and the manuals and SOPs on the shelf. Here the tokens really are words (Part 1), so nothing needs reinterpreting; it is the mainstream use of these models. If it is useful to you, this is what it looks like — and it can ride on the same on-prem hardware as the sensor pipeline.

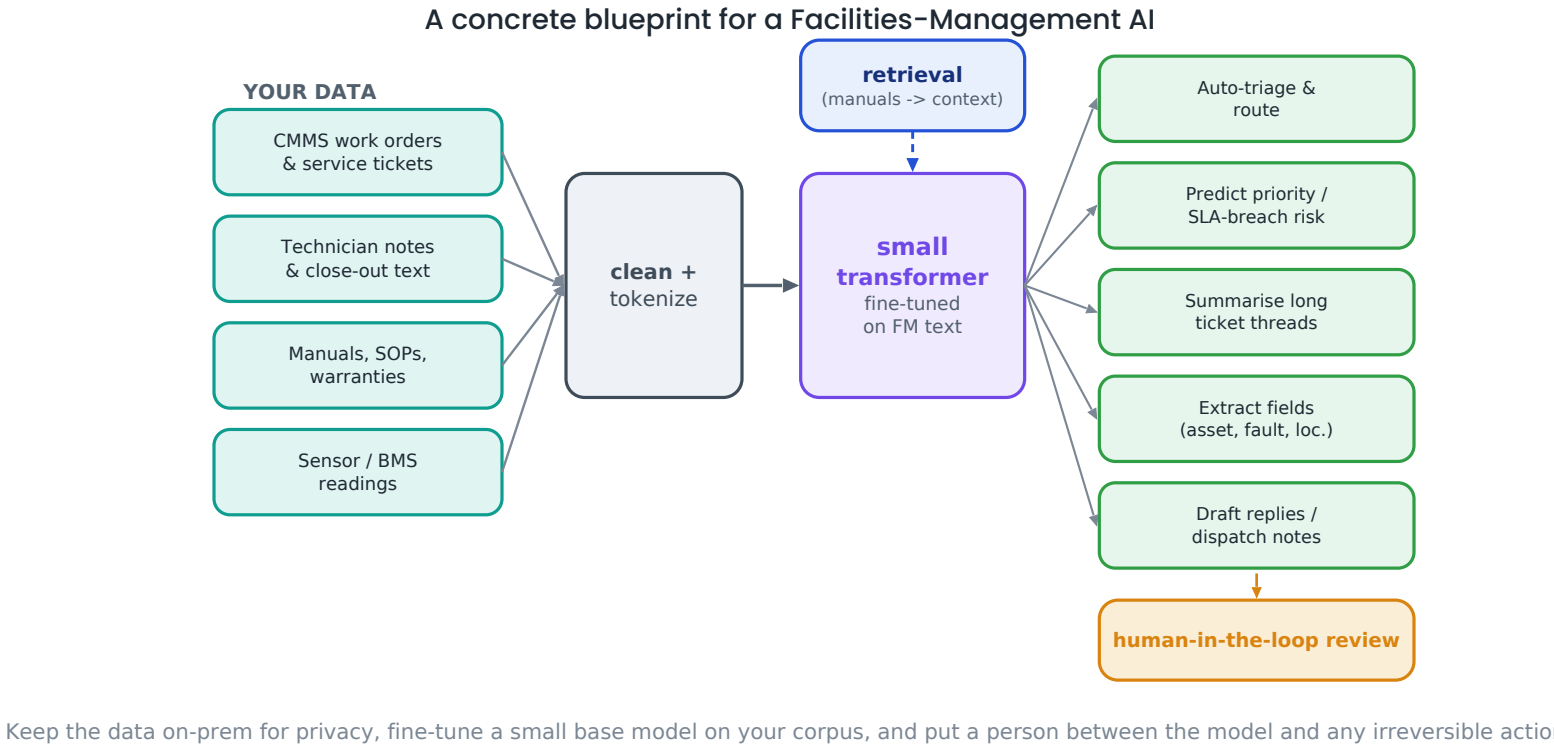


Figure 16. The language-side blueprint. One small fine-tuned backbone, read several ways, grounded in your manuals by retrieval, with a human reviewing anything irreversible.

8.1 The tasks worth building

These are the jobs a small language model does well on facilities text, and how each one reads the same backbone (Part 4 explained the heads):

Task	How the model is read	What comes out
Triage & routing	classification head	P(category) → send to the right trade
Priority / SLA-breach risk	classification or regression head	P(P1…P4), or a 0–1 risk score
Field extraction	per-token tagging head	asset tag, location, fault, dates
Summarise a ticket thread	generation (decoder)	a short, faithful summary
Draft reply / dispatch note	generation, low temperature	suggested text for a human to send
Answer questions on manuals	retrieval + generation (RAG)	a grounded, cited answer

8.2 Getting the data right

As with the sensor build, **data quality dominates**. Your assets are the CMMS work-order history, resolved tickets (whose closed category is itself the training label), technician notes, and the manuals and SOPs. Clean and de-duplicate; normalise asset tags and trade names so the tokeniser sees them consistently; strip or mask personal data before training; and hold out a genuine test set the model never sees, so accuracy numbers are honest. A few thousand well-labelled tickets beat a model twice the size trained on mush.

8.3 Fine-tune, retrieve, or just prompt?

Three ways to inject your knowledge, best used together — and the choice between them is the main design decision on the language side:

- **Fine-tuning** teaches *skill and style* — how to classify your tickets, your tone in a reply, your field formats. Use it for the classification and drafting tasks. It bakes knowledge into the weights, so it suits things that do not change often.
- **Retrieval-augmented generation (RAG)** supplies *facts that change*. Embed every manual and SOP into a vector index (using the model's own embeddings, section 4.3); at question time, fetch the few most relevant passages and paste them into the prompt as context, so the model answers *from your documents* and can cite them. When a warranty or procedure is updated, you re-index — no retraining. This is how you stop a model confidently inventing a torque spec.
- **Prompting** a capable base model with clear instructions is the zero-training way to prototype and to handle one-off asks. Start here to prove value, then fine-tune what recurs.

8.4 Measure it, and keep a human in the loop

The discipline mirrors Part 7. For classification (triage, priority) report **precision, recall and F1**, not bare accuracy, and test on tickets and sites the model never trained on. For generation (summaries, drafts) there is no substitute for structured human review. Then act automatically only above a confidence threshold, route the uncertain cases to a person, and never let the model take an irreversible action — dispatch a crew, close a safety ticket, send an external email — without sign-off. The point where a transformer is the *wrong* tool is the same as in section 7.7: hand the *numbers* (sensor streams, schedules, costs) to methods built for them, and let the language model do the *words*.

A pragmatic language-side first version

A sensible language-side first build: fine-tune one small open model on your closed-ticket history for **triage + priority** (highest volume, clearest labels, easily measured), add **RAG over your manuals** for grounded Q&A, keep it on the same on-prem box as the sensor pipeline, and route low-confidence cases to a coordinator. Ship that, measure it against the human baseline, and expand into extraction and drafting once it earns trust. The two builds — sensors and language — reinforce each other: a flagged coverage gap (Part 7) can open a ticket the language model then triages.

PART 9

The whole machine, in one breath

Every idea in this document, in order, as a single chain — from a neuron to two working facilities systems:

- **Foundations.** A neuron is a weighted sum plus a bend; stack neurons into layers and you have a multilayer perceptron — the atom every other part is built from.
- **Represent.** Turn each word — or each short **window** of sensor data — into a vector that carries meaning, and add a positional signal so order survives. (Part 1)
- **Attend.** In each block, attention lets every token gather context — project to Q, K, V, score with QK^T, scale by $\sqrt{d_k}$, softmax, blend the values; several heads in parallel. (Part 2)
- **Refine.** Residual shortcuts and layer-norm keep the deep stack trainable; a per-token MLP does the heavy thinking; stack the block L times. (Part 3)
- **Answer.** An output head turns the final vectors into what you need — a next token, a category, extracted fields, or reusable embeddings. (Part 4)
- **Train.** Predict the hidden piece, score the miss with cross-entropy, assign blame by back-propagation, step with Adam — and for you, **self-supervised pre-training** then a small fine-tune. (Part 5)
- **Run it.** Inference is bounded by memory — and your **offline, batch** sensor job sidesteps the slow-generation problem entirely, so use a thorough small model on ordinary hardware. (Part 6)
- **Build one — sensors.** Window the wrist-sensor stream, recognise the activity with the transformer, locate coarsely with dead reckoning plus a barometer, fuse into a coverage map against the schedule, validate against audits, and keep a human in the loop. (Part 7)
- **Build two — language.** Fine-tune the same kind of model on your tickets to triage, prioritise, extract and draft, and ground it in your manuals with retrieval. The two builds feed each other. (Part 8)

The honest summary

There is no magic in the box — only vectors, a handful of learned matrices, one attention formula, and a great deal of disciplined arithmetic tuned by feedback. For your work, that one engine powers two things: it reads *motion* to verify cleaning, and it reads *text* to run the ticket-and-manual side — while classical methods handle the geometry and transparent analytics turn both into answers a manager can trust. Understanding it at this level is exactly what lets you make sound calls about model size, data, sensors, hardware and where to trust it — which is the real work of building an AI for your industry.

APPENDIX

Notation and terms at a glance

Core transformer terms

Term / symbol	Meaning
vector / embedding	a fixed-length list of numbers representing a token; d of them
token	the atomic unit the model reads — a word, or a window of sensor data
d (width) · L (depth)	numbers per token vector · how many blocks are stacked
V (vocabulary)	number of distinct tokens a language model knows (~30–50k)
E	embedding matrix (V × d); row i is token i's starting vector
Q, K, V	Query, Key, Value — three learned projections of the input
W_q, W_k, W_v, W_o	the four learned weight matrices of an attention layer
softmax	turns a row of scores into positive weights that sum to 1
d_k	width of the Q/K vectors; scores are divided by $\sqrt{d_k}$
head	one of several parallel attentions, each on a slice of d
residual	adding a sub-layer's input back to its output (x + Sublayer(x))
layer-norm	rescales each vector to a stable range before a sub-layer
FFN	per-token feed-forward network (an MLP); holds most parameters
logits	raw pre-softmax scores, one per class or vocabulary entry
parameters / weights	the model's tunable numbers; N of them (≈ $12Ld^2 + Vd$)
loss / cross-entropy	−log(probability of the correct answer); the wrongness score
gradient / back-prop	the slope of the loss for each weight, found by the chain rule
learning rate η · Adam	step size · the optimiser (momentum + adaptive steps)
fine-tuning / LoRA	cheaply specialising a pre-trained model on your own data
quantisation	storing weights in fewer bits (bf16 / int8 / int4) to save memory
prefill / decode	reading the input (fast, parallel) / writing output (slow, sequential)
RAG	retrieval-augmented generation — answering grounded in fetched documents
semantic search	finding similar items by the nearness of their embedding vectors
classification head	a small MLP on the pooled vector that outputs class scores

Foundations & your sensor system

Term / symbol	Meaning
neuron · activation φ	a weighted sum + bias, passed through a non-linearity (ReLU/GELU)
MLP	multilayer perceptron — stacked fully-connected layers; the base model
IMU	inertial measurement unit — the motion sensor package in the band
accelerometer	measures linear acceleration (and gravity), 3 axes
gyroscope	measures rotation rate, 3 axes
magnetometer / compass	measures heading relative to magnetic north, 3 axes
barometer	air-pressure sensor; the reliable way to detect floor changes
window	a short slice of the sensor stream (~2.5 s); your ‘token’
HAR	human activity recognition — classifying the motion in each window
self-supervised pre-training	learning from unlabelled data by masking and reconstructing windows
dead reckoning (PDR)	estimating position from steps + heading, with no GPS
drift	the accumulating error that makes dead reckoning degrade over time
sensor fusion / Kalman filter	combining noisy sensors into one best estimate
anomaly detection	flagging shifts/areas that deviate from the normal pattern
coverage map	activity × location vs. schedule — the assurance deliverable
macro-F1 / confusion matrix	honest accuracy measures for imbalanced classes

Prepared as a ground-up technical explainer for building small-transformer AI in the facilities-management industry — one engine, two builds: wrist-worn sensor data for cleaning assurance, and language models for tickets and manuals. Every figure was drawn from the same equations it describes, and the worked attention example (Part 2) was computed, not asserted.