

GOTO **COPENHAGEN**

Craft software with confidence in 2026



HighTech Innovators
YOUR SOFTWARE. BETTER

Spotting Digital Waste in Code Gives You Energy

Wilco Burggraaf



Yesterday I heard
your **goto;** is code.
So...



```
using elem_t = std::pair<float, int64_t>;
std::vector<elem_t> queue(dim_size);

for (int j = 0; j < dim_size; j++) {
    queue[j].first = values[j];
    queue[j].second = j;
}

std::partial_sort(
    queue.begin(), queue.begin() + k,
    queue.end(), compare
);
```

```
</> C++ · simplified TopKImpl.h  
using elem_t = std::pair<float, int64_t>;  
std::vector<elem_t> queue(dim_size);  
  
for (int j = 0; j < dim_size; j++) {  
    queue[j].first = values[j];  
    queue[j].second = j;  
}  
  
std::partial_sort(  
    queue.begin(), queue.begin() + k,  
    queue.end(), compare  
);
```

For every new token it writes, an LLM scores thousands of possible next tokens.

This code helps select the strongest candidates.

```
</> C++ · simplified TopKImpl.h  
using elem_t = std::pair<float, int64_t>;  
std::vector<elem_t> queue(dim_size);  
  
for (int j = 0; j < dim_size; j++) {  
    queue[j].first = values[j];  
    queue[j].second = j;  
}  
  
std::partial_sort(  
    queue.begin(), queue.begin() + k,  
    queue.end(), compare  
);
```

For every new token it writes, an LLM scores thousands of possible next tokens.

This code helps select the strongest candidates.

```
</> C++ · simplified TopKImpl.h

using elem_t = std::pair<float, int64_t>;
std::vector<elem_t> queue(dim_size);

for (int j = 0; j < dim_size; j++) {
    queue[j].first = values[j];
    queue[j].second = j;
}

std::partial_sort(
    queue.begin(), queue.begin() + k,
    queue.end(), compare
);
```

For every new token it writes, an LLM scores thousands of possible next tokens.

This code helps select the strongest candidates.

</> C++ · simplified

TopKImpl.h

```
using elem_t = std::pair<float, int64_t>;
std::vector<elem_t> queue(dim_size);

for (int j = 0; j < dim_size; j++) {
    queue[j].first = values[j];
    queue[j].second = j;
}

std::partial_sort(
    queue.begin(), queue.begin() + k,
    queue.end(), compare
);
```

For every new token it writes, an LLM scores thousands of possible next tokens.

This code helps select the strongest candidates.

</> C++ · simplified

TopKImpl.h

```
using elem_t = std::pair<float, int64_t>;  
std::vector<elem_t> queue(dim_size);  
  
for (int j = 0; j < dim_size; j++) {  
    queue[j].first = values[j];  
    queue[j].second = j;  
}  
  
std::partial_sort(  
    queue.begin(), queue.begin() + k,  
    queue.end(), compare  
);
```

What this code does

```
using elem_t =  
std::pair<float, int64_t>;
```

Store: (score, token_id)

</> C++ · simplified

TopKImpl.h

```
using elem_t = std::pair<float, int64_t>;  
std::vector<elem_t> queue(dim_size);  
  
for (int j = 0; j < dim_size; j++) {  
    queue[j].first = values[j];  
    queue[j].second = j;  
}  
  
std::partial_sort(  
    queue.begin(), queue.begin() + k,  
    queue.end(), compare  
);
```

What this code does

```
std::vector<elem_t>  
queue(dim_size);
```

**Create space for
all 50,257 tokens**

</> C++ · simplified

TopKImpl.h

```

using elem_t = std::pair<float, int64_t>;
std::vector<elem_t> queue(dim_size);

for (int j = 0; j < dim_size; j++) {
    queue[j].first = values[j];
    queue[j].second = j;
}

std::partial_sort(
    queue.begin(), queue.begin() + k,
    queue.end(), compare
);

```

What this code does

```

for (int j = 0;
j < dim_size; j++)

```

**Overwrite all slots,
already zeroed once:
2 × 800 KB written
for every token.**

</> C++ · simplified

TopKImpl.h

```
using elem_t = std::pair<float, int64_t>;
std::vector<elem_t> queue(dim_size);

for (int j = 0; j < dim_size; j++) {
    queue[j].first = values[j];
    queue[j].second = j;
}

std::partial_sort(
    queue.begin(), queue.begin() + k,
    queue.end(), compare
);
```

What this code does

```
queue[j].first =
values[j];
```

Store the token's score

</> C++ · simplified

TopKImpl.h

```
using elem_t = std::pair<float, int64_t>;
std::vector<elem_t> queue(dim_size);

for (int j = 0; j < dim_size; j++) {
    queue[j].first = values[j];
    queue[j].second = j;
}

std::partial_sort(
    queue.begin(), queue.begin() + k,
    queue.end(), compare
);
```

What this code does

```
queue[j].second = j;
```

Store the token's ID

</> C++ · simplified

TopKImpl.h

```
using elem_t = std::pair<float, int64_t>;
std::vector<elem_t> queue(dim_size);
```

```
for (int j = 0; j < dim_size; j++) {
    queue[j].first = values[j];
    queue[j].second = j;
}
```

```
std::partial_sort(
    queue.begin(), queue.begin() + k,
    queue.end(), compare
);
```

What this code does

```
std::partial_sort(...)
```

Move the Top-K highest scores to the front

Tiny example — K = 2

```
values = [3, 9, 2, 7]
```

Pair value with position:

```
[(3,0), (9,1), (2,2), (7,3)]
```

Top 2 after partial_sort:

```
[(9,1), (7,3)]
```

Meaning: score 9 was highest and came from token/index 1.

But somewhere
inside it, we're doing
thousands of operations
we don't actually need.

</> C++ · simplified

TopKImpl.h

```
using elem_t = std::pair<float, int64_t>;
std::vector<elem_t> queue(dim_size);

for (int j = 0; j < dim_size; j++) {
    queue[j].first = values[j];
    queue[j].second = j;
}

std::partial_sort(
    queue.begin(), queue.begin() + k,
    queue.end(), compare
);
```

Can you
spot where
energy is
wasted?

</> C++ · simplified

TopKImpl.h

```

1 using elem_t = std::pair<float, int64_t>;
  std::vector<elem_t> queue(dim_size);

  for (int j = 0; j < dim_size; j++) {
2     queue[j].first = values[j];
     queue[j].second = j;
  }

  std::partial_sort(
3     queue.begin(), queue.begin() + k,
     queue.end(), compare
  );

```

Is it...

1 the allocation?

2 the loop?

3 the sorting?

Before I tell you,
we need to understand
how we got to these
few lines of **PyTorch**.

Two years ago,



I met
Paull Young,



Head of
Sustainability



at GitHub,



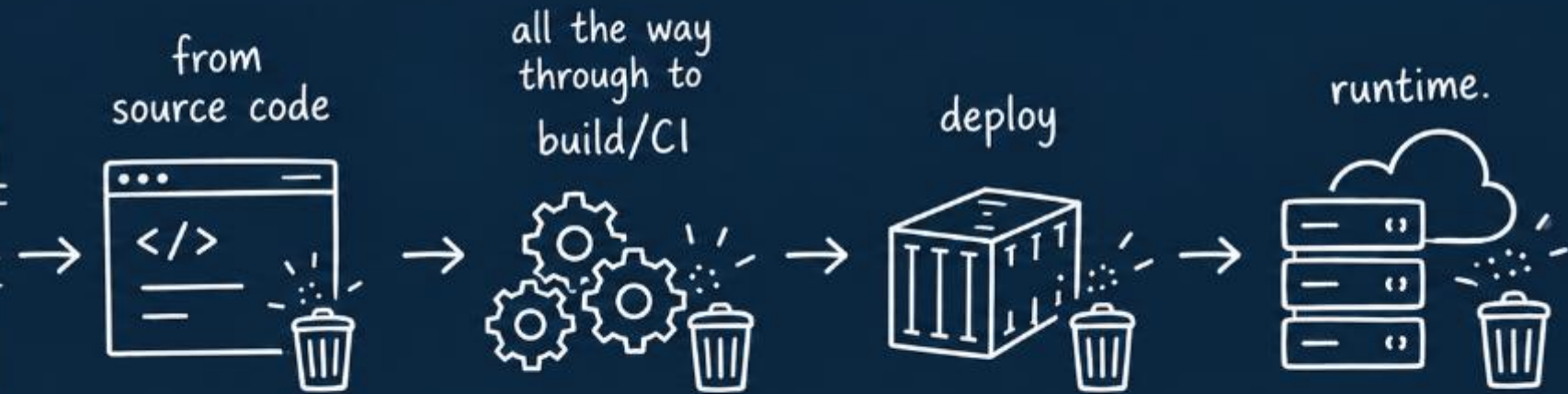
at Green 10
London.



So I started
explaining an idea I had



for how, in theory, we could
detect digital waste across a system,



Last December in 2025
(he's from California, remote)



My team of Hightech Innovators
in the Netherlands



together with
Microsoft



teamed up and join on
building an open-source project
to make this happen.



SAY HI TO

CodeWeave

Open source. Detect digital waste across a system,
from source code to build, deploy and runtime.

HighTech Innovators × Microsoft

github.com/HighTech-Innovators/CodeWeave



Scan → star the repo ★



So I was thinking...
if we can analyse, in theory,
any source code in the world,



it would make sense
to take a really beefy
code base,



existing for
many years,

where everyone
understands some serious
maintenance is done
on it.



And still show
you can find
digital waste
in it.



And also a OSS project
that has a lot of impact
on the world.



I chose **PyTorch**. Why?

Maybe a bit over-ambitious. But so many of today's AI models trace back to it.



One open-source project. **Massive impact.**

Where does this code actually run?

PyTorch on PyPI

2.1 B

lifetime downloads

Every one of those downloads ships the same **CPU** TopK kernel. A waste pattern in one open-source file is multiplied across all of them.

IoT devices by 2030

39 B

connected, forecast

Edge AI runs mostly on **CPUs**: 88.8% of edge AI hardware volume in 2024. IDC expects half of enterprise AI inference to run at the edge by 2030.

One kernel. Billions of downloads. Tens of billions of devices.

So we set up a complete
Autonomous GenAI CLI
with Copilot.



We knew nothing
about PyTorch.



The AI reverse engineered
the documentation to
understand why it exists,
how it's built, what the
architecture is, everything.



Then it integration
tested it on energy
related patterns
completely on its own.



And it...
actually worked.



From zero knowledge to real insights.
Autonomously. With AI. On PyTorch.
And it actually worked. ♥

SAY HI TO

CodeWeave-PyTorch

CodeWeave pointed at PyTorch: an autonomous energy analysis of a massive, well-maintained codebase.

Built with the GitHub Copilot CLI

github.com/HighTech-Innovators/CodeWeave-PyTorch



Scan → star the repo ★

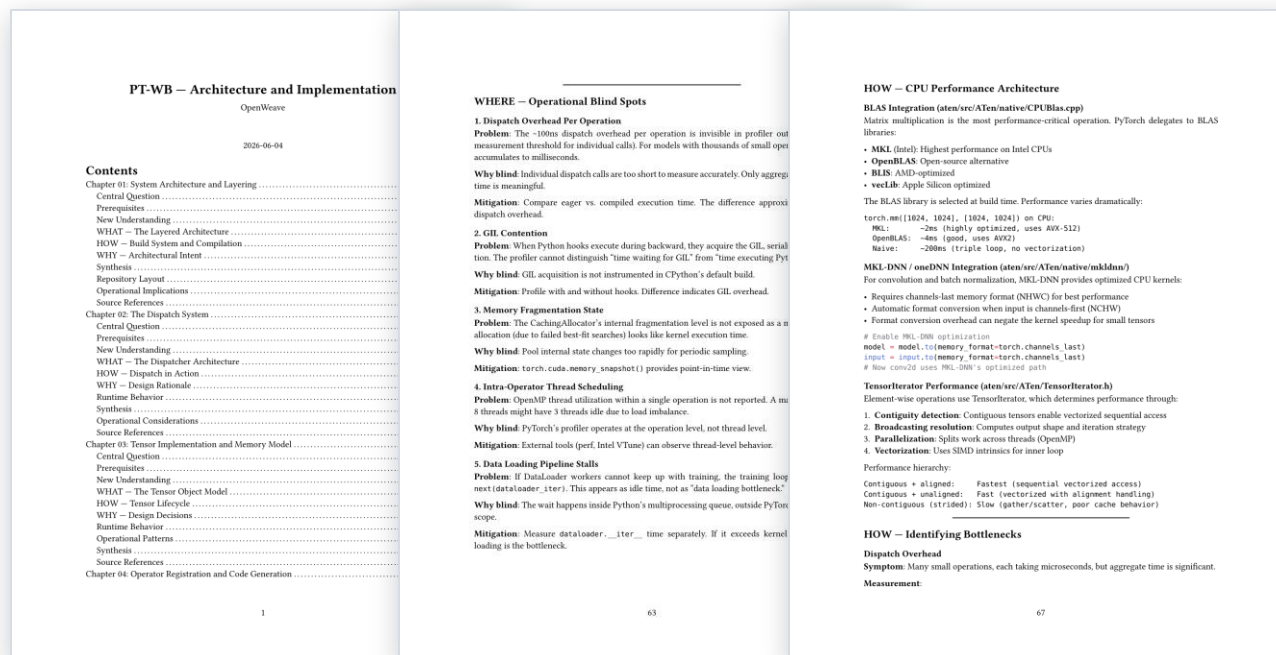


What CodeWeave writes: the book

The book

13

Markdown files · one per chapter
Each chapter.md asks one central question and answers it as WHAT, HOW and WHY.
Rendered to PDF only to make it easier to read.



Read like a senior engineer. Written down.

What CodeWeave writes: the ADRs

The ADRs

23

Markdown files · one per component

Each ADR.md covers one component, from `aten/native/cpu` to `torch/_dynamo`. Rendered to PDF only to make it easier to read.

PT-WB – Architecture Decision Records

OpenWeave

2020-06-04

Contents

aten

aten/src/ATen

aten/src/ATen/core

aten/src/ATen/native

aten/src/ATen/native/cpu

aten/src/ATen/native/cuda

aten/src/ATen/native/ort

aten/src/ATen/native/quantized/cpu

aten/src/ATen/native/quantized/ort

Key Files

File

Purpose

aten/src/ATen/native/cpu/Loops.h

Provides `cpu_kernel(...)`, the common-wise loop helpers used across CPU.

aten/src/ATen/native/cpu/BinaryOpKernel.cpp

Registers CPU implementations for and comparison stubs such as `div_true_stub` and `add_clamp`.

aten/src/ATen/native/cpu/CopyKernel.cpp

Implements type-converting and precision copy kernels, including `sd` and `float` copy paths.

aten/src/ATen/native/cpu/ReduceOpKernel.cpp

Implements CPU reduction scans such as `cumsum_cpu`, `cumprod_cpu`, `std_var_cpu`, and `norm_kernel_tensor_iterate`.

aten/src/ATen/native/cpu/SoftMaxKernel.cpp

Implements vectorized softmax and max forward and backward kernels, `AOXS12` registrations.

aten/src/ATen/native/cpu/BlasKernel.cpp

Supplies CPU fallback kernels for `gather` stubs when the higher-level patch path selects in-tree implementation.

Public Interface

This directory is primarily an implementation detail rather than a API surface. The symbols other components consume are registered kernel points such as `mul_kernel`, `div_true_kernel`, `copy_kernel`, `softmax_lastdim_kernel`, and `log_softmax_lastdim_kernel_impl`. shared helpers `cpu_kernel(cpu_kernel_vec...)`; callers normally reach them through stubs like `mul_stub`, `copy_softmax_lastdim_kernel` rather than by direct linkage.

Dependencies

aten/src/ATen/native

aten/src/ATen/core

c10/core

Dependencies

Component

Direction

Nature

aten/src/ATen/native

mutual

The parent directory declares dispatch stubs such as `mul_stub`, `copy_stub`, and `softmax_lastdim_kernel`, and this directory binds concrete CPU kernels to them with `REGISTER_DISPATCH` and `ALSO_REGISTER_AOXS12_DISPATCH`.

aten/src/ATen

depends-on

CPU kernels use `Dispatch.h`, `Parallel.h`, `TensorIterator`, and `ATen/cpu/vec`. SMD helpers from the broader ATen runtime.

aten/src/ATen/core

depends-on

Files such as `CopyKernel.cpp` and `ReduceOpsKernel.cpp` use `ATen/core/Tensor.h` and `TensorBase` accessors to read metadata and data pointers.

c10/core

depends-on

Kernel code depends on `ScalarType`, `DeviceType`, `DispatchKey`, and low-level buffer or stride metadata supplied by `c10`.

Runtime Behaviour

Most files in this directory compile inline `namespace CPU, CAPABILITY` and are entered through a stub registration rather than a direct call `BinaryOpKernel.cpp` registers dozens of CPU kernels with `REGISTER_DISPATCH(...)`, while `SoftMaxKernel.cpp` additionally uses `ALSO_REGISTER_AOXS12_DISPATCH(...)` so the same logical operator can pick an `AOXS12`-specialized implementation when available.

The kernels explicitly manage work partitioning and temporary storage. `CopyKernel.cpp` uses `parallel_forb`, `iter_numel()`, `grain_size`, ... around reduced-dot conversion loops. `ReduceOpsKernel.cpp` builds `TensorIteratorConfig` objects and early-returns for empty or scalar inputs, and `SoftMaxKernel.cpp` allocates thread-local buffers with `std::make_unique` inside its `parallel_for` regions for reduced-type softmax paths.

Performance Profile

This directory is optimized around SIMD and contiguous-memory fast paths. `Loops.h` documents that `cpu_kernel_vec(...)` uses explicit SIMD intrinsics, and `BinaryOpKernel.cpp` uses

1

16

17

Confusing, but true:
we use AI (**CodeWeave**)
to improve **PyTorch**,
the code that runs AI.

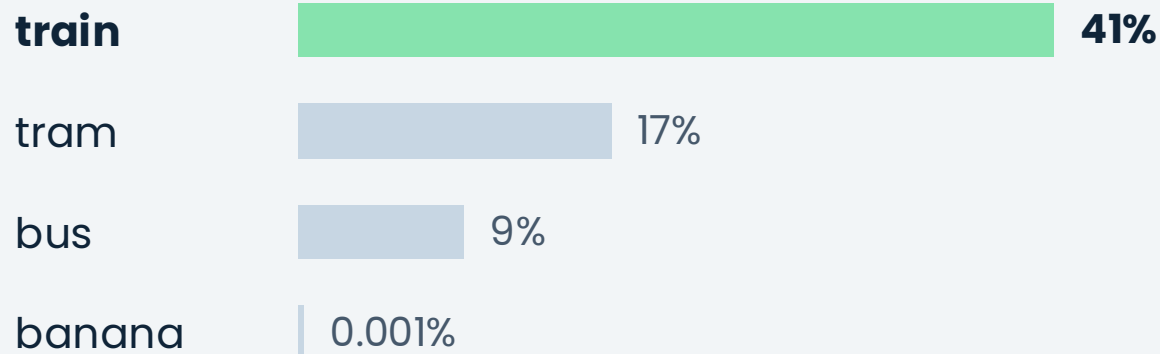
Meet distilgpt2: autocomplete on a diet

This is the model we run on PyTorch, to test and improve PyTorch.
Its one job: guess the next word. Then do it again. And again.

The prompt so far

The curator boards the _ _ _ _ _

Its guesses (illustrative), out of 50,257 tokens



Keep the top 50 (`top_k`), pick one, append, repeat.

The whole model

~330 MB

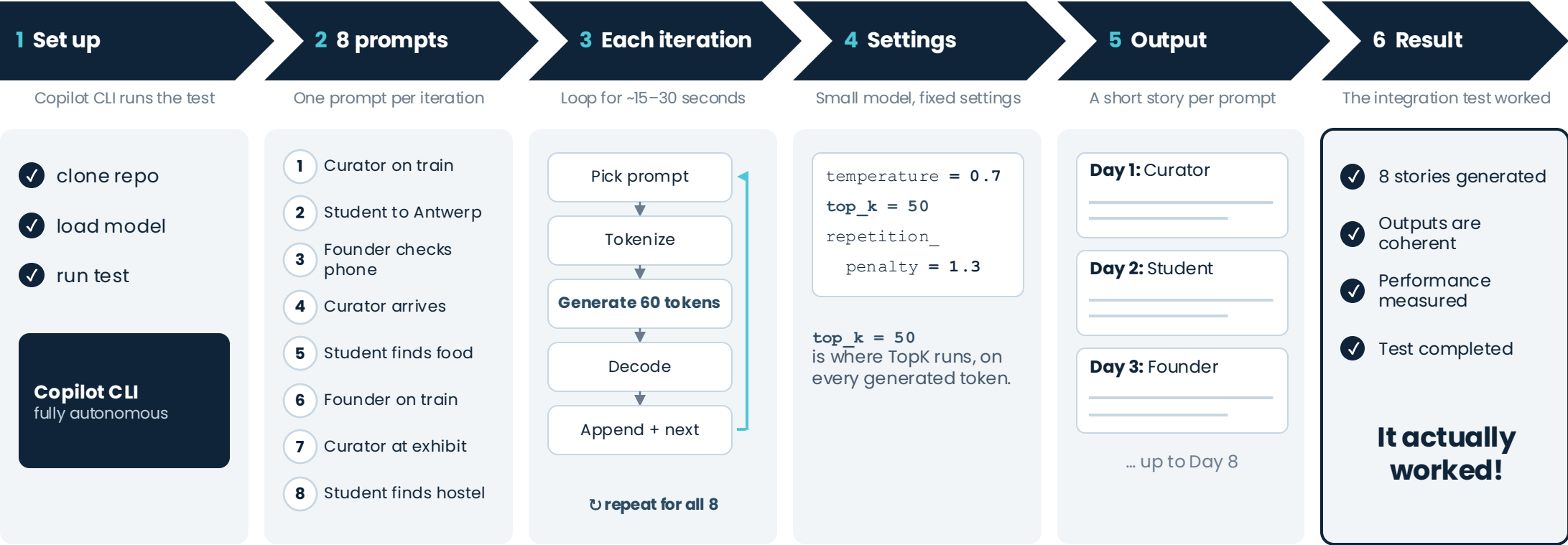
82M params × 4 bytes

≈ 400× smaller than
Llama 3 70B (~140 GB).

Small enough for a phone, a
laptop or a Raspberry Pi:
the edge.

Use case: story generation

8 prompts + a small AI model on PyTorch: **distilgpt2** (82M params).



A steady 15–30 s load on PyTorch makes inefficiencies easier to spot.
But you don't need that: it even works on 1–2 second API calls.

</> C++ · simplified

TopKImpl.h

```

1 using elem_t = std::pair<float, int64_t>;
  std::vector<elem_t> queue(dim_size);

  for (int j = 0; j < dim_size; j++) {
2     queue[j].first = values[j];
     queue[j].second = j;
  }

  std::partial_sort(
3     queue.begin(), queue.begin() + k,
     queue.end(), compare
  );

```

Is it...

1 the allocation?

2 the loop?

3 the sorting?

Our energy model: wrong, but useful

Too rough to measure with. Good enough to decide with.

1 Why it's wrong

Language runtimes, the operating system and the CPU itself reorder, merge and skip μ ops.

The real count is always different.

2 The model

`1  $\mu$ op/s = 1 Hz`

`1 active core`

`@ 1.2 V`

Energy = runtime $\times$ ~6 W per busy core

3 Why it's useful

The same simple rules for every line of code.

So you can compare lines and spot the hot spots.



Prompt context: give this model to any AI, and it writes energy ranges into your code comments.

</> C++ · simplified

TopKImpl.h

Energy model (prompt context):

1 μop/s = 1 Hz · 1 active core · @ 1.2 V

```

using elem_t = std::pair<float, int64_t>;           // 0 mWh (compile-time only)
std::vector<elem_t> queue(dim_size);               // ~5-15 mWh / 1M generated tokens

for (int j = 0; j < dim_size; j++) {               // ~80-105 mWh / 1M generated tokens
    queue[j].first = values[j];
    queue[j].second = j;
}

std::partial_sort(                                // ~70-95 mWh / 1M generated tokens
    queue.begin(),
    queue.begin() + k,
    queue.end(),
    compare
);

// TOTAL ≈ 185 mWh / 1M generated tokens (measured)

```

Why a range, not one number?

1 Never off

25 W+

Idle, a CPU package already burns this.

2 Speed costs

↑ **GHz** = ↑ **V**

A faster core needs more voltage. Every step costs more.

3 Same code

≠ mWh

Different speed, different energy bill.



That's why we write: // ~80–105 mWh / 1M generated tokens

Your code changes the speed

Async and parallel work across cores changes how fast each core runs.

1 core busy



That core runs fast.

4 cores busy



Each core runs slower.



How exactly? Ask me over a beer.

Where is the waste in TopK?

Three suspects: the allocation, the loop, the sort. We measured each one.

1 Allocation?

Small cost

tmp =

--	--	--	--

 ...

↓

RAM

Temporary buffer only.

Not the main issue.

2 The loop?

Main waste

values		(value, index)
0.4	→	(0.4, 0)
2.1	→	(2.1, 1)
5.3	→	(5.3, 2)
...	→	...

Copies all **50,257** values.
Needs only the top **50**.

3 The sorting?

Some cost

(9.1, 42)	partial sort	(9.1, 42)
(7.8, 17)		(7.8, 17)
(3.2, 99)		...
...		

Moves bigger items.
Second biggest cost.

4 Real answer

Sort indices, not pairs

Before · (value, index) · 16 B

804 KB per token

After · index only · 8 B

402 KB

- Half the bytes to write
- Values are never copied
- Traffic ~2.6 → ~1.4 MB

Main waste = copying + extra data movement

Before		110,834 ns
After		65,196 ns

1.70×
speedup

41.18%
lower latency

Where is the waste in TopK?

Three suspects: the allocation, the loop, the sort. We measured each one.

1 Allocation?

Small cost

tmp =

--	--	--	--

 ...

↓

RAM

Temporary buffer only.

Not the main issue.

2 The loop?

Main waste

values		(value, index)
0.4	→	(0.4, 0)
2.1	→	(2.1, 1)
5.3	→	(5.3, 2)
...	→	...

Copies all **50,257** values.
Needs only the top **50**.

3 The sorting?

Some cost

(9.1, 42)	partial sort	(9.1, 42)
(7.8, 17)		(7.8, 17)
(3.2, 99)		...
...		

Moves bigger items.
Second biggest cost.

4 Real answer

Sort indices, not pairs

Before · (value, index) · 16 B

804 KB per token

After · index only · 8 B

402 KB

- Half the bytes to write
- Values are never copied
- Traffic ~2.6 → ~1.4 MB

Main waste = copying + extra data movement

Before		110,834 ns
After		65,196 ns

1.70×
speedup

41.18%
lower latency

`</> C++` · simplified, NaN handling omitted

`TopKImpl.h`

```
- using elem_t = std::pair<float, int64_t>;
- std::vector<elem_t> queue(dim_size);
+ std::vector<int64_t> indices(dim_size);

- for (int j = 0; j < dim_size; j++) {
-     queue[j].first = values[j];
-     queue[j].second = j;
- }
+ std::iota(indices.begin(), indices.end(), 0);

std::partial_sort(
-     queue.begin(), queue.begin() + k, queue.end(),
-     [](auto& a, auto& b) {
-         return a.first > b.first; }
+     indices.begin(), indices.begin() + k,
+     indices.end(),
+     [&](int64_t a, int64_t b) {
+         return values[a] > values[b]; }
);
```

Why is it faster?

	Old	New
Values copied	50,257	0
Buffer size	804 KB	402 KB
Memory traffic	~2.6 MB	~1.4 MB

Less data to move.
Less time. Less energy.

per token, 50,257 candidates

</> C++ · simplified, NaN handling omitted

TopKImpl.h

```
- using elem_t = std::pair<float, int64_t>;
- std::vector<elem_t> queue(dim_size);
+ std::vector<int64_t> indices(dim_size);

- for (int j = 0; j < dim_size; j++) {
-     queue[j].first = values[j];
-     queue[j].second = j;
- }
+ std::iota(indices.begin(), indices.end(), 0);

std::partial_sort(
-     queue.begin(), queue.begin() + k, queue.end(),
-     [](auto& a, auto& b) {
-         return a.first > b.first; }
+     indices.begin(), indices.begin() + k,
+     indices.end(),
+     [&](int64_t a, int64_t b) {
+         return values[a] > values[b]; }
);
```

Same answer

values: never copied



indices: sorted by their value



The 8 B index moves, not the 16 B pair.
Top 2 = index 1 and 0.

</> C++ · simplified, NaN handling omitted

TopKImpl.h

```
- using elem_t = std::pair<float, int64_t>;
- std::vector<elem_t> queue(dim_size);
+ std::vector<int64_t> indices(dim_size);

- for (int j = 0; j < dim_size; j++) {
-     queue[j].first = values[j];
-     queue[j].second = j;
- }
+ std::iota(indices.begin(), indices.end(), 0);

std::partial_sort(
-     queue.begin(), queue.begin() + k, queue.end(),
-     [](auto& a, auto& b) {
-         return a.first > b.first; }
+     indices.begin(), indices.begin() + k,
+     indices.end(),
+     [&](int64_t a, int64_t b) {
+         return values[a] > values[b]; }
);
```

TopK microbenchmark

110,834 ns

→ 65,196 ns

41.18% less time

</> C++ · simplified, NaN handling omitted

TopKImpl.h

```
- using elem_t = std::pair<float, int64_t>;
- std::vector<elem_t> queue(dim_size);
+ std::vector<int64_t> indices(dim_size);

- for (int j = 0; j < dim_size; j++) {
-     queue[j].first = values[j];
-     queue[j].second = j;
- }
+ std::iota(indices.begin(), indices.end(), 0);

std::partial_sort(
-     queue.begin(), queue.begin() + k, queue.end(),
-     [](auto& a, auto& b) {
-         return a.first > b.first; }
+     indices.begin(), indices.begin() + k,
+     indices.end(),
+     [&](int64_t a, int64_t b) {
+         return values[a] > values[b]; }
);
// 110,834 ns -> 65,196 ns (-41.18%)
// 185 -> 109 mWh per 1M generated tokens
```

A busy core burns power the whole time it runs.

energy per million generated tokens

before

185 mWh

after

109 mWh

76 mWh less

every million generated tokens.

Each token: TopK picks 50 of 50,257.

assumes ~6 W per busy core

TopK is **41.18%** faster.
So how much faster
is **DistilGPT-2**?

And the whole model?

0.80%

faster

0.63%

less energy

One kernel down. Thousands to go.

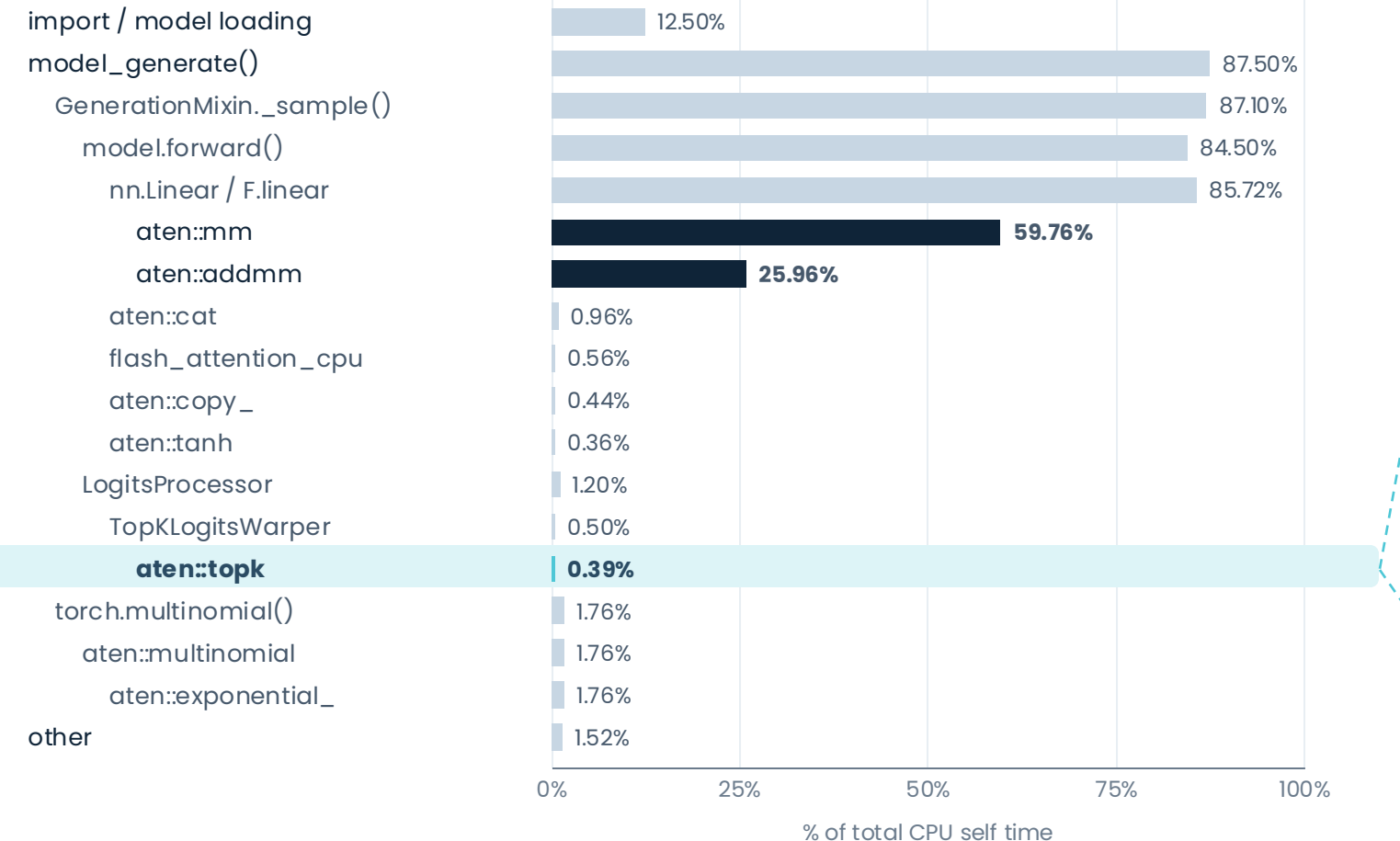
Could you have
predicted this **only**
based on static code?

NO!

Where does the CPU time go?

PyTorch CPU inference flamegraph (self time). **85% is matrix multiply** — the waste we fixed hides in a 0.39% sliver.

PyTorch CPU inference



Zoom: **aten::topk**

0.39%

of total CPU self time

- 1 Allocate temporary vector
- 2 **Copy 50,257 values + indices**
- 3 partial_sort (k = 50)

Tiny slice. Fixed: **1.70× faster**

Does this go beyond code?



YES!



Three ways hardware wastes energy

1

Underutilization

The hardware could do more, but we never hand it enough work.

2

Waiting

The hardware sits blocked — on data, on a lock, on something else.

3

Bottlenecks

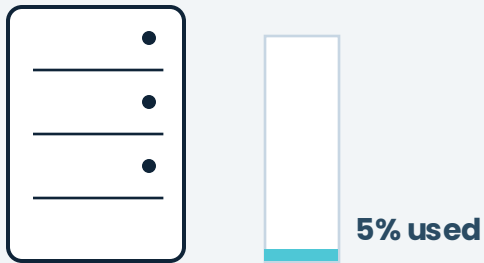
The hardware is working. It just spends too long on one step: redundant work, overhead, data it never needed to move.

TopK was the third kind. **The CPU wasn't waiting — it was busy doing work we didn't need.**



Underutilization

Oversized VMs



Paying for capacity nobody uses

Idle clusters



Machines awake with nothing to do

Always-on dev



Environments that never sleep



Resources are on, but barely used.

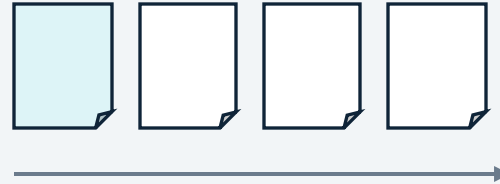
Waiting

I/O waits



The CPU spins on disk or network

Queues



Jobs pile up behind each other

Slow dependencies



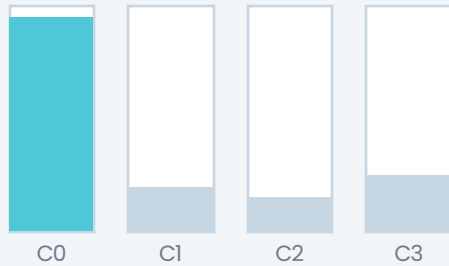
Every call waits on someone else



Work is blocked, not progressing.

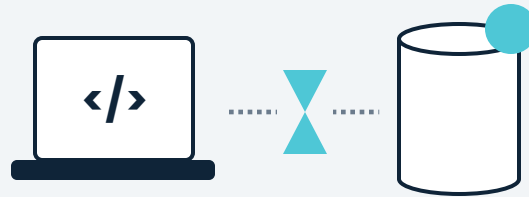
Bottlenecks

Hot spot CPU



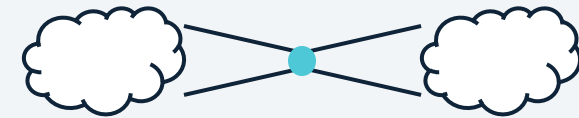
One core limits the rest

Slow database



Queries hold everything up

Network choke point



Transfer becomes the limit

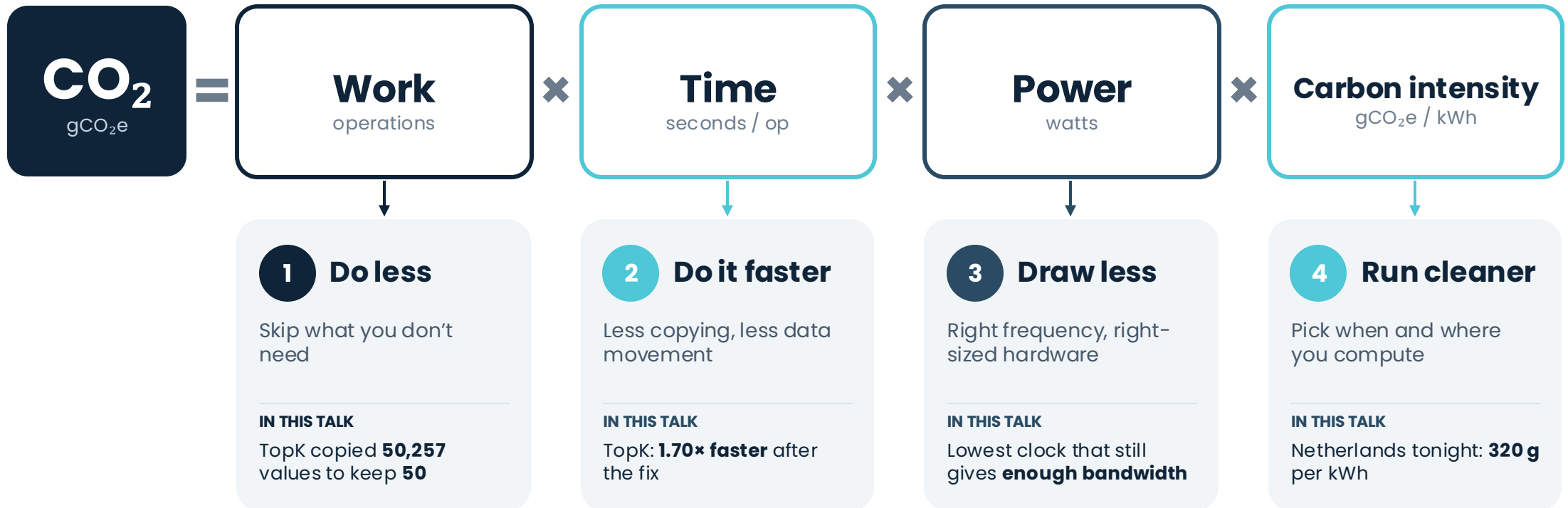


One slow step sets the pace for all work.

So, we spotted digital waste,
how does this relate to CO₂?

Four levers between your code and CO₂

Emissions are a product. Shrink any factor and the whole result shrinks with it.



Your code controls work, time and power. **Where and when** you run it controls the fourth.

Sep 7, 2026

9:15 PM GMT+2

15 min



Netherlands

Sep 7, 2026, 9:15 PM GMT+2

Preliminary

38%

gas

320

gCO₂eq/kWh

207.78

€/MWh

47%

Carbon-free

44%

Renewable

Dominant source Carbon intensity Electricity price Carbon-free Renewable

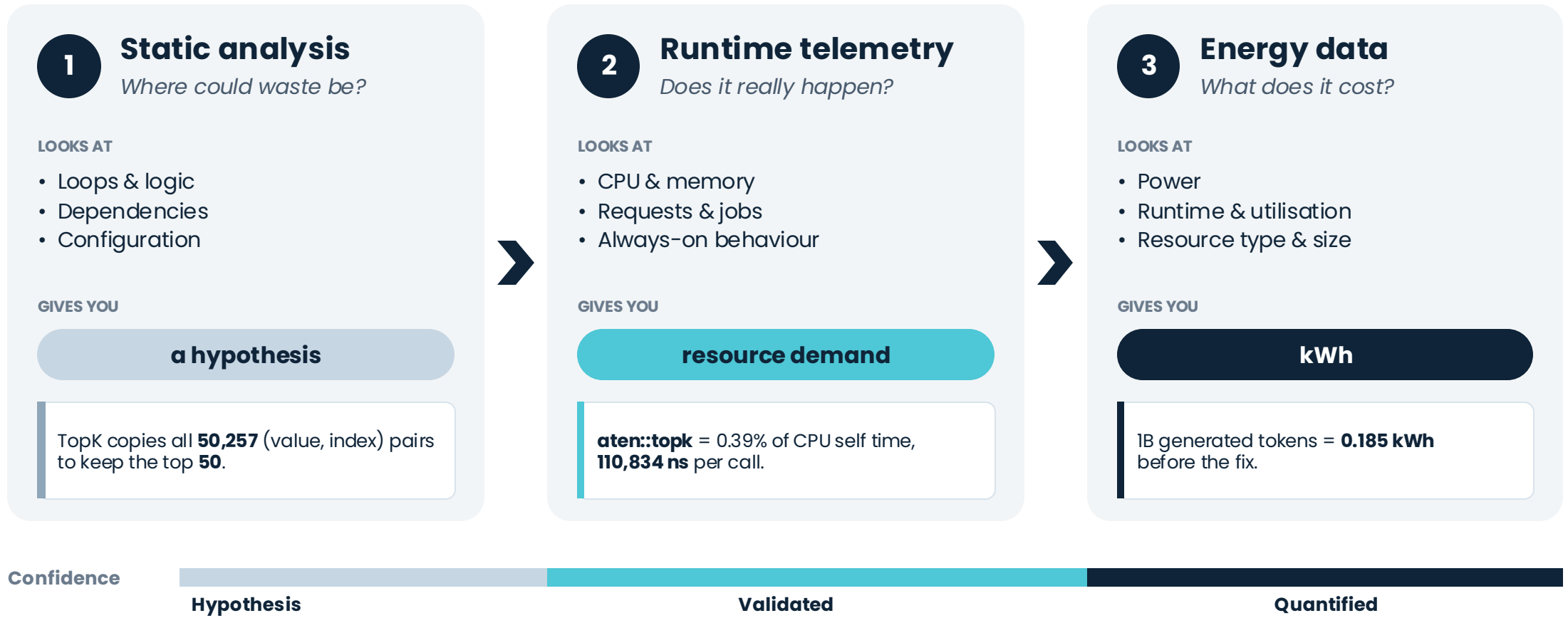
Carbon intensity

gCO₂eq/kWh



No single source is enough

Code gives you the hypothesis. Runtime confirms it. Energy puts a number on it.



SCI for TopK

Carbon per unit of useful work: electricity ($E \times I$) plus the carbon of building the hardware (M).

E 0.185 kWh

TopK energy before optimisation

I 320 gCO₂e/kWh

carbon intensity (NL grid)

M 0

embodied, ignored here

R 1B tokens

generated: our useful work

$$\text{SCI} = \frac{(E \times I) + M}{R}$$

$$(0.185 \times 320 + 0) / 1\text{B}$$

$$= 59.2 \text{ gCO}_2\text{e per billion generated tokens}$$



TopK before: 1B generated tokens = 0.185 kWh = 59.2 gCO₂e

What if the edge ran this fix?

Not a forecast. Just a sense of scale.

before **0.0185** → after **0.0109**
mWh per 100 generated tokens
calculated: runtime × 6 W, not a direct measurement

0.0076 mWh
less per 100 generated tokens

By 2030: 39 billion connected IoT devices (IoT Analytics, as on slide 25)

IF THIS SHARE USES THE FIXED TOPK CODE ...

1% 390 million devices

5% 1.95 billion devices

10% 3.9 billion devices

25% 9.75 billion devices

... IT SAVES THE ELECTRICITY OF

11 households 30 MWh

57 households 148 MWh

115 households 296 MWh

287 households 741 MWh

Assumptions: 1 million generated tokens per device per year · shares are chosen, not predicted · 1 household = 2,580 kWh/year (CBS 2025)

This counts only electricity. Building the hardware (M in SCI) comes on top.

Demand observability shows how usage really looks

Observability data reveals the patterns behind resource demands — so we see peaks, waiting times, and real usage in action.



Peaks

See when demand spikes and by how much.



Waiting times

Understand delays and how they impact performance.



Real usage

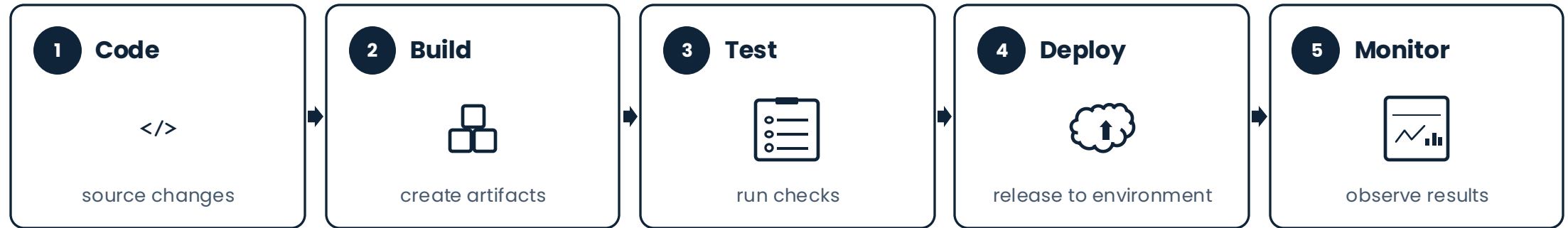
See actual patterns, not assumptions, to right-size resources.



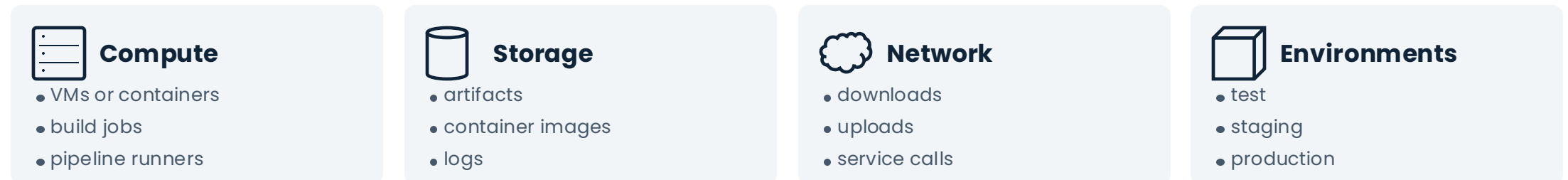
Why it matters: Observability turns raw data into insight, so we can plan capacity, improve performance, and reduce waste.

CI/CD is a cloud application

A CI/CD pipeline runs in the cloud and uses multiple resources to build, test and deploy software.



Cloud resources behind the pipeline



Every pipeline run consumes cloud resources. Better visibility helps reduce waste.



Key idea: CI/CD is not just code. It drives compute, storage, network and deployment activity in the cloud.

Why all this effort?



WATER BILL

Meter Reader Id:		Month:		
Cust No	Inv No	Date	Time	Trf
	62158		09:47	MD
Mtr Location:				
Meter Reading				
Meter No.	Previous	Present	Units	
	365	373	8	
Charge This Month				
Units	Rate	Total		





Act

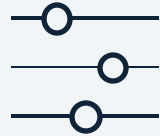
Use the insight to prioritise improvements.

Use the results to decide what to optimise, resize, or retire.



Optimise

Improve efficiency of underused resources.



Resize

Match capacity to real demand.



Retire

Remove resources no longer needed.

Priority list

- 1 Unused test environments
- 2 Oversized databases
- 3 Right-size app servers
- 4 Old storage & backups

Focus on high impact, low effort first.



Key idea

Measure first. Then optimise, resize, and retire.

Who knows an application
still running in your organisation
that should have been
decommissioned ages ago?

Show some hands.

QUESTIONS?

ASK ME IN THE APP



RATE MY TALK

GET REWARDED