Virtual Metrology 2:
Deploying the Model

ECE 8803-AI4 · AI for Semiconductor

Asif Khan
Associate Professor
School of Electrical and Computer Engineering
School of Materials Science and Engineering
Georgia Institute of Technology
akhan40@gatech.edu

Virtual Metrology 1:
Building the Model

ECE 8803-AI4 · AI for Semiconductor Manufacturing

https://aikhan123.github.io/khan-lab-website/ai-semiconductors-course.html

Asif Khan
Associate Professor
School of Electrical and Computer Engineering
School of Materials Science and Engineering
Georgia Institute of Technology
akhan40@gatech.edu

What you should be able to do by the end of today

Name two ways a random split leaks information, and split the data so it cannot.

Derive the error floor a model faces when its training labels are measurements.

Decide what a virtual metrology model is allowed to do, given how accurate it is.

Tune a run-to-run controller for a signal that is noisier than metrology.

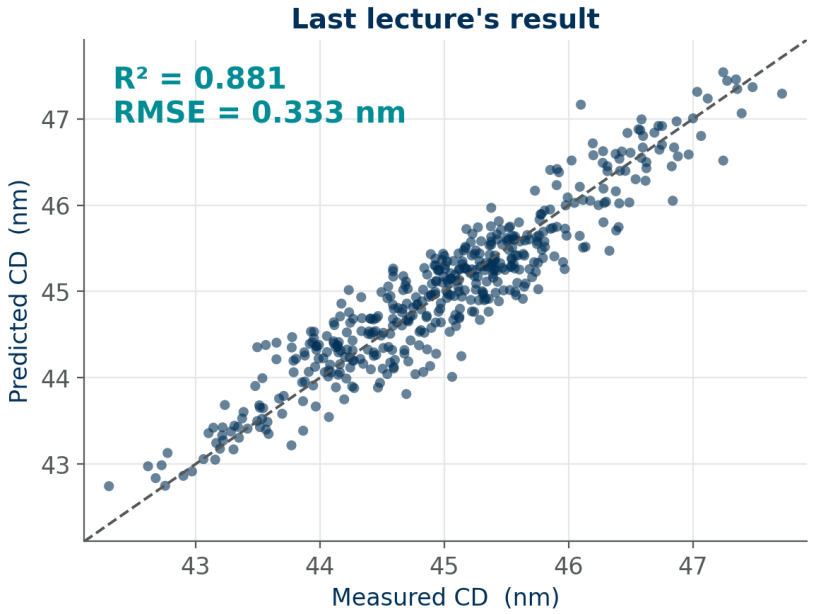
Explain why closing the loop destroys the data the model was trained on.

Design the control chart, the retraining trigger and the rollback plan a deployed model needs.

One sentence holds the lecture together: a virtual metrology model is a metrology tool made of software.

Where we left off

PLS, 8 components, 465 labelled wafers, five-fold cross-validation



The result

R² of 0.881 and RMSE of 0.333 nm, against a gauge noise floor of 0.220 nm. Cross-validated, on real held-out folds.

The plan for today

We change nothing about the model. Same features, same algorithm, same data. We only change how we ask whether it works — and the answer changes.

Everything that follows is a consequence of one decision made in one line of code: shuffle=True.

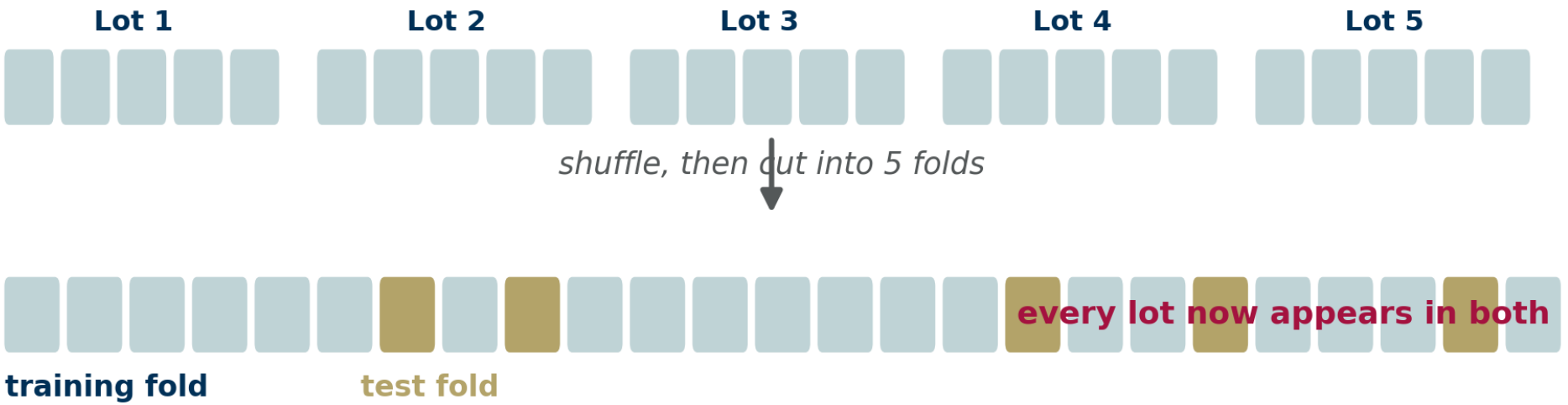
SEGMENT 1 OF 5

Validation done right

The model was fine. The question we asked it was not.

What five-fold cross-validation actually did

Take 465 wafers in run order, shuffle, cut into five folds



The shuffle is the default in every library and in every tutorial. On wafer data it destroys two things at once.

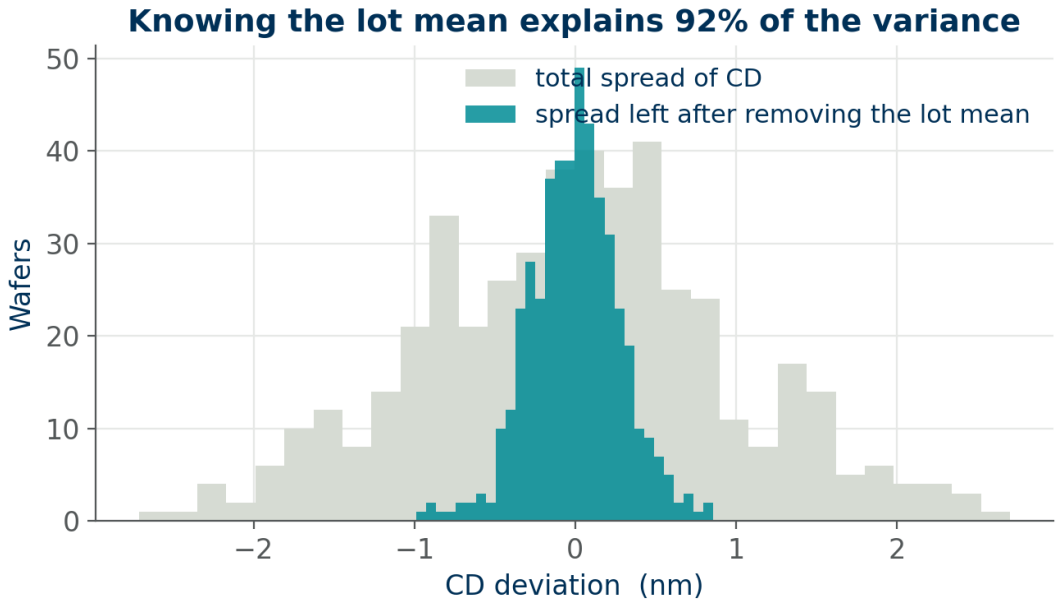
Leak one: the same lot on both sides

25 wafers share a carrier, a queue time and an incoming measurement

One lot, 25 wafers

train	train	test	train	train
test	train	test	train	train
test	train	train	train	train
train	train	test	train	train
train	train	train	train	train

The lot-level features are identical across all 25 of these.



Knowing which lot a wafer came from explains 92% of the variance in CD. A random split hands the model that answer.

You already saw the evidence

Last lecture, slide 34 — and I asked you to hold on to it

What the loadings showed

The first PLS component leaned hardest on the lot-level block, f040 to f059. Seven of the ten largest lasso coefficients came from the same twenty columns.

Those columns are identical for all 25 wafers in a lot.

What that means

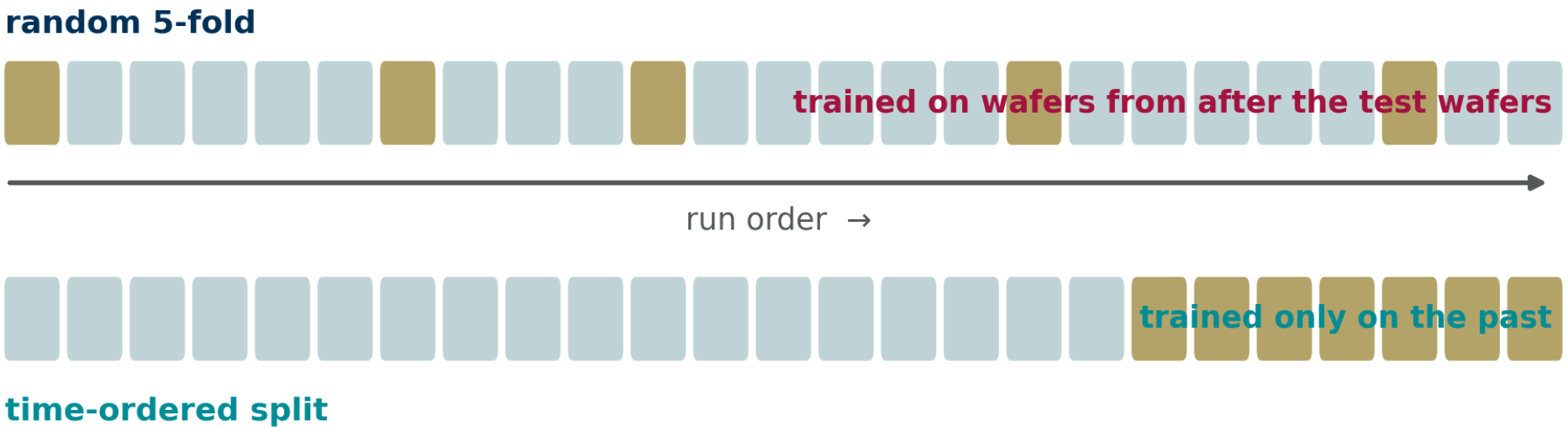
A lot-level feature carries no information about an individual wafer. It identifies the lot.

Under a random split, the other wafers from that lot are in the training set with their measurements attached. The model is doing a lookup.

The feature that made the model look good is the feature that made the evaluation wrong. That is the usual pattern.

Leak two: training on the future

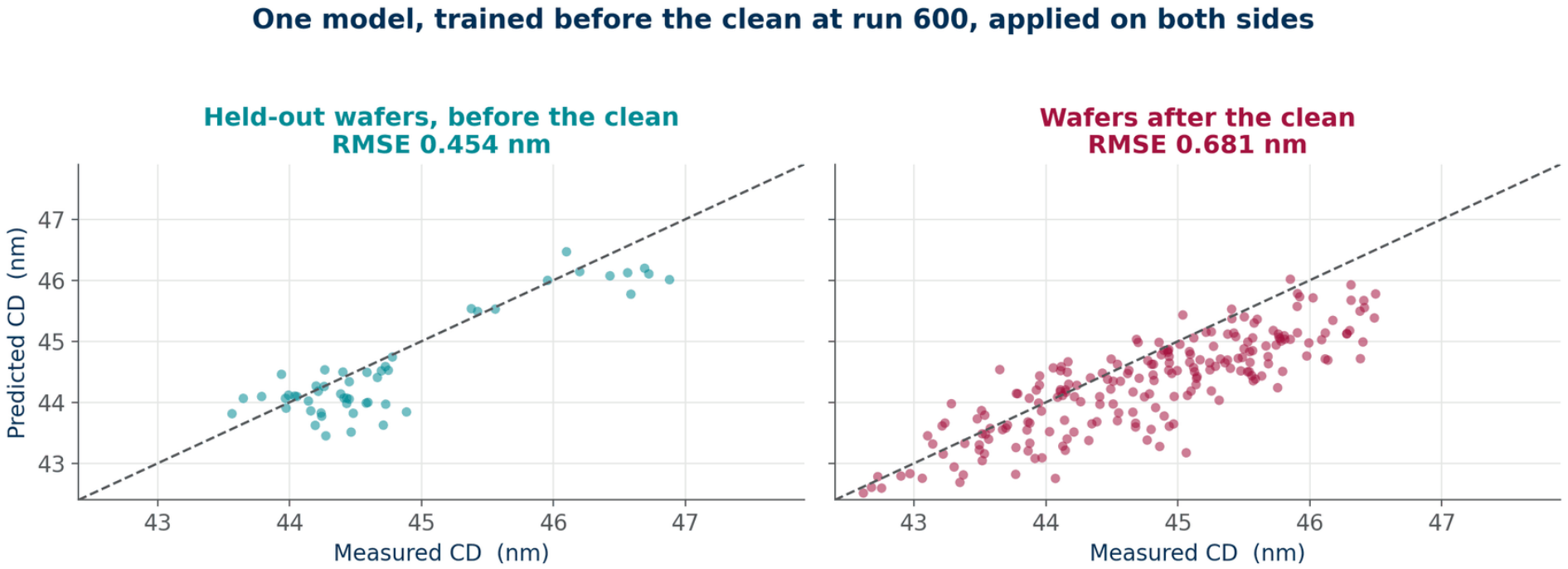
A shuffled fold contains wafers processed after the wafers it predicts



In deployment, every wafer you predict is in the future. No shuffled evaluation ever tests that.

And the process did change

Trained on wafers before the chamber clean at run 600



Two splits that cannot leak

Both are two lines of code, and both should be your default

Group by lot

GroupKFold with lot_id as the group. Every wafer of a lot lands in the same fold, so the model never sees a lot it is then asked about.

Use this whenever rows share anything — a lot, a carrier, a wafer, a patient, a user.

Order by time

Train on runs 1 to k, test on runs after k. Then advance k and repeat — forward-chaining, or rolling-origin evaluation.

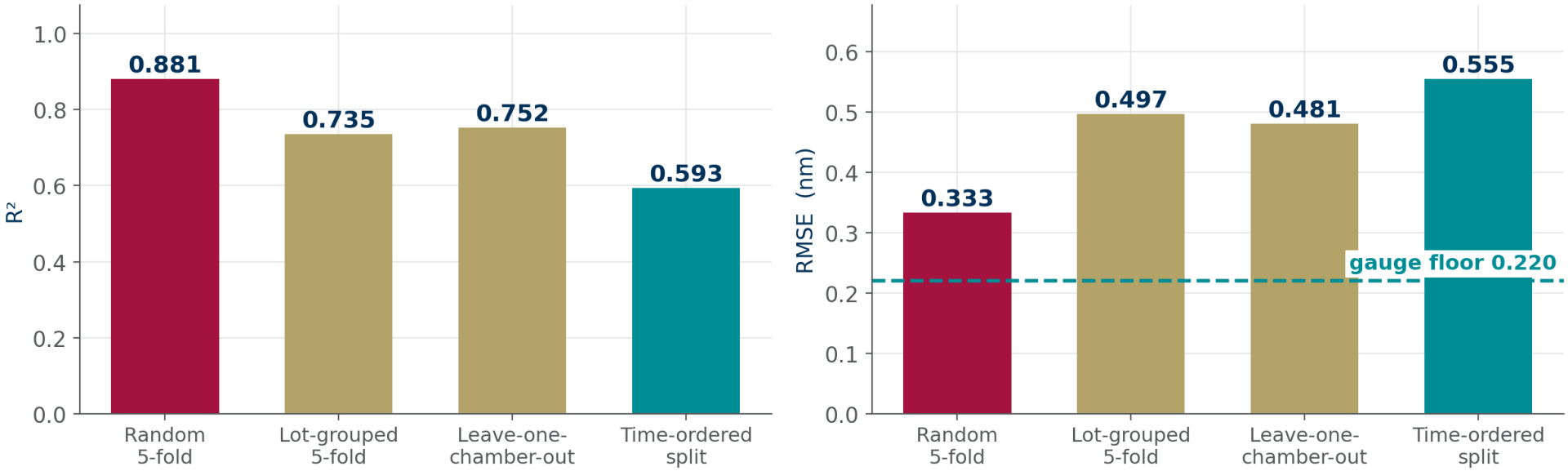
This is the only scheme that answers the question deployment asks: given everything up to now, how well will it do next?

When rows are grouped and ordered — which fab data always is — do both: group by lot and order by time.

The same model, evaluated four ways

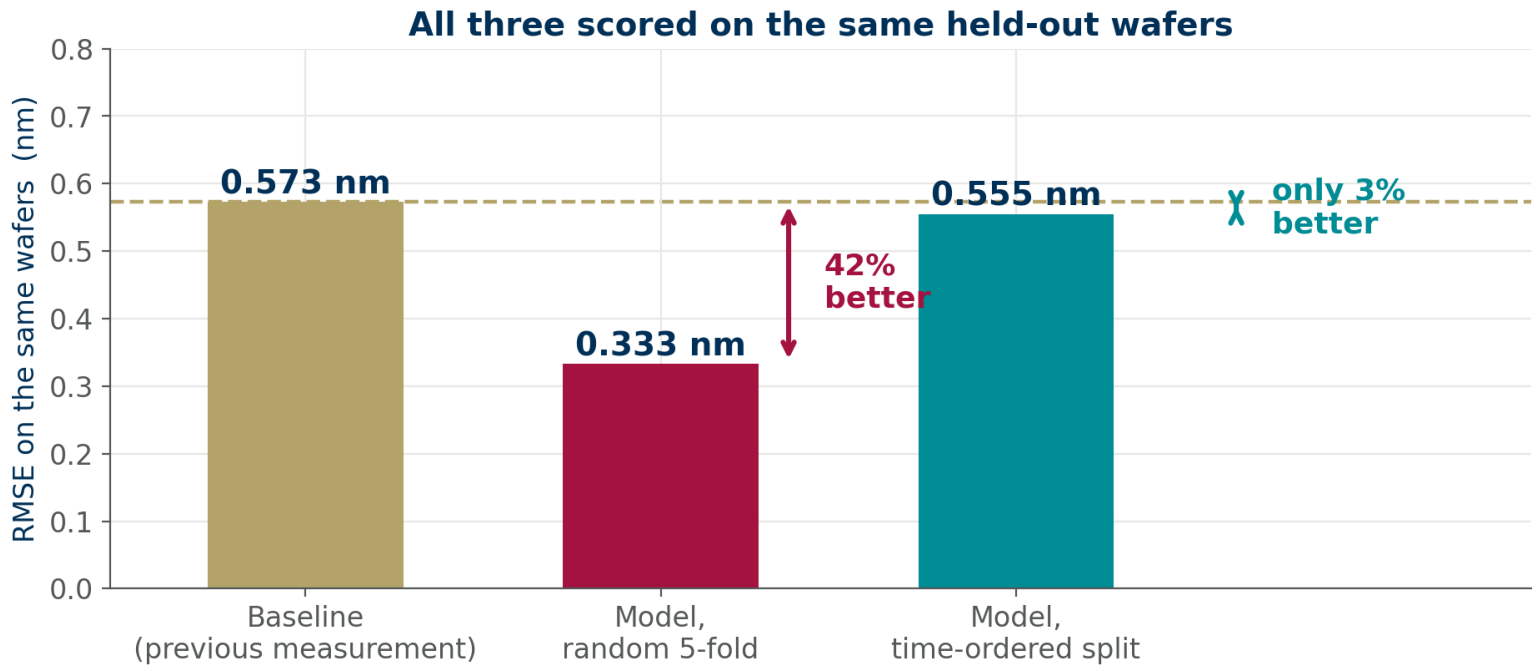
Nothing about the model changed. Only the question changed.

Same model, same data, same features — four ways of asking whether it works



Now put it back next to the baseline

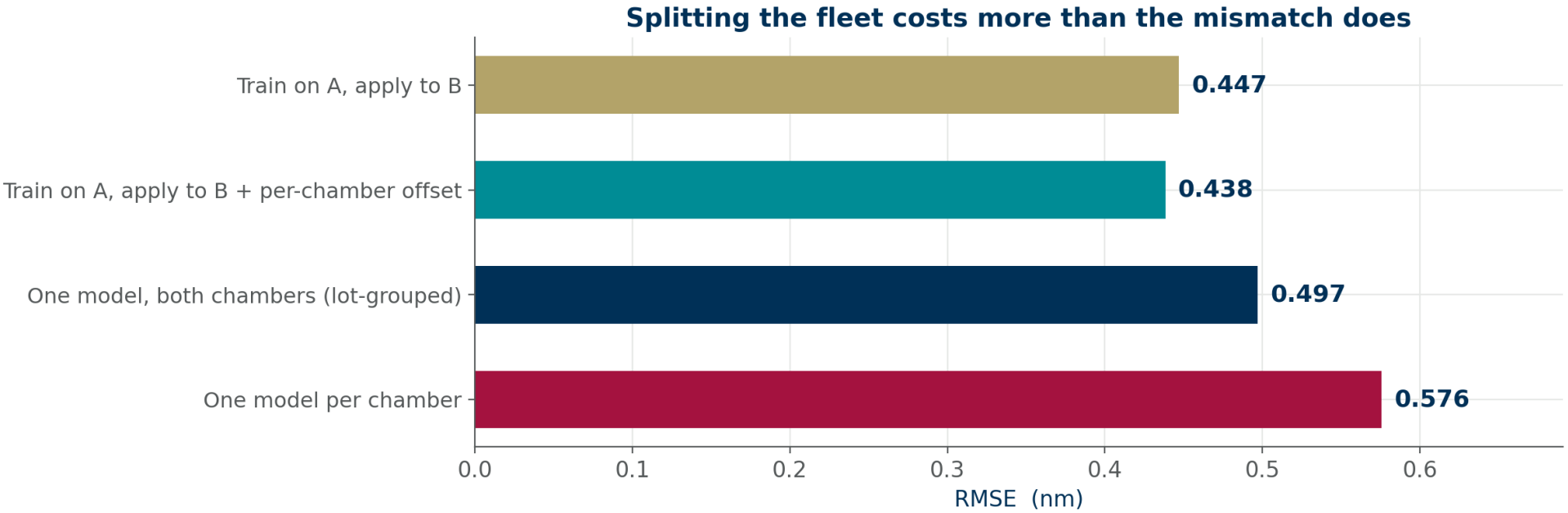
All three scored on exactly the same held-out wafers



The advantage over one line of code falls from 42% to 3%. That is the real result of last lecture's work.

And one more question deployment will ask

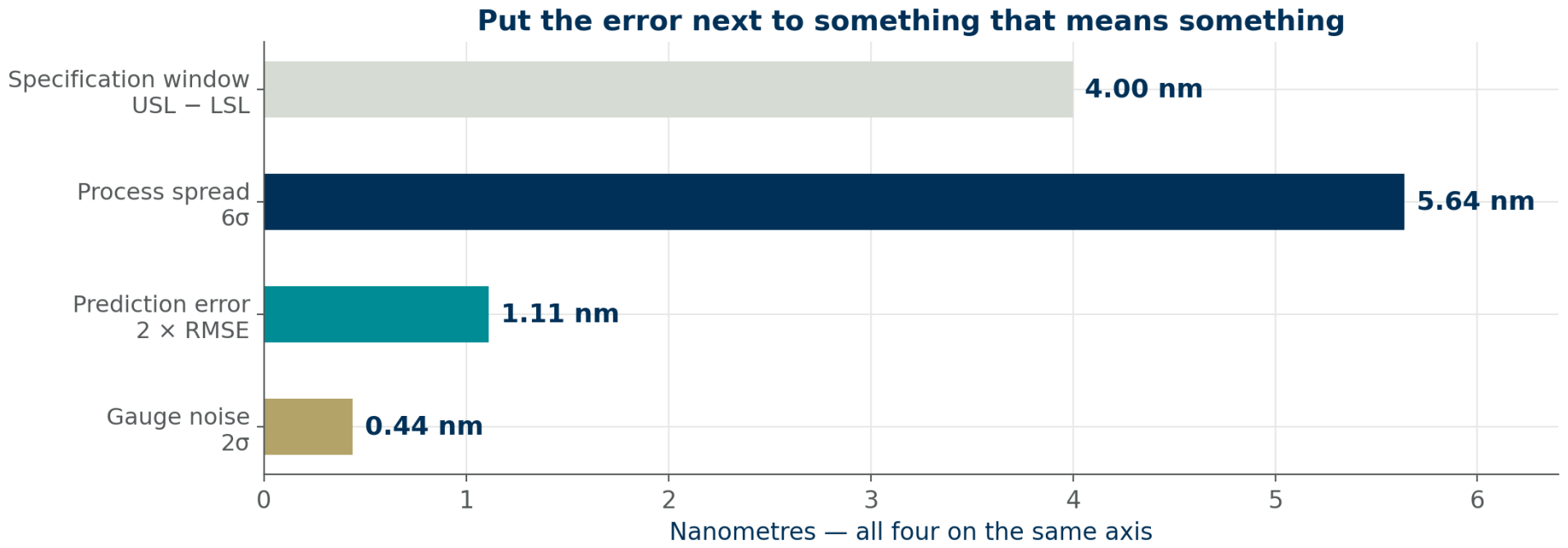
Which model do you run on a fleet of chambers?



One model per chamber sounds safer and scores worst — it halves the training data. Measure before you split the fleet.

Report the error next to something that means something

R² hides all of this; nanometres do not



Two numbers to quote every time: error as a fraction of the specification window, and error as a multiple of the gauge noise.

A splitting checklist

Five questions to ask before you trust any number

What do rows share?

If any two rows share a lot, a wafer, a carrier or a maintenance cycle, they must not straddle the split. Group on the coarsest shared unit.

Does time run through it?

If the data has an order and the process drifts, evaluate forward. A shuffled estimate is an upper bound you will never see again.

What did you fit before splitting?

Scalers, imputers, feature selection, PCA. Every one of them must be fitted inside the fold. Fitting on all the data leaks the test set.

Will deployment look like the test set?

New chamber, new product, new quarter. If deployment differs, build a split that mimics that difference and report it.

And the fifth: if the honest number is disappointing, report the honest number. It is the one that will happen.

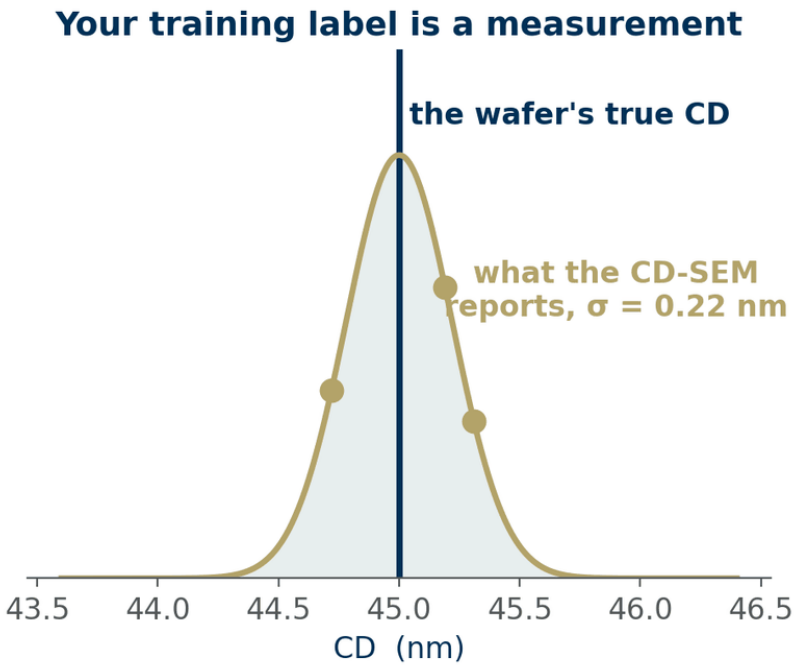
SEGMENT 2 OF 5

The noise floor

There is a number below which no model can go, and it has nothing to do with the model.

Your training labels are measurements too

$\sigma^2_{observed} = \sigma^2_{process} + \sigma^2_{measurement}$ — applied to the target column



$$RMSE_{min} = \sigma_{gauge} = 0.220 \text{ nm}$$

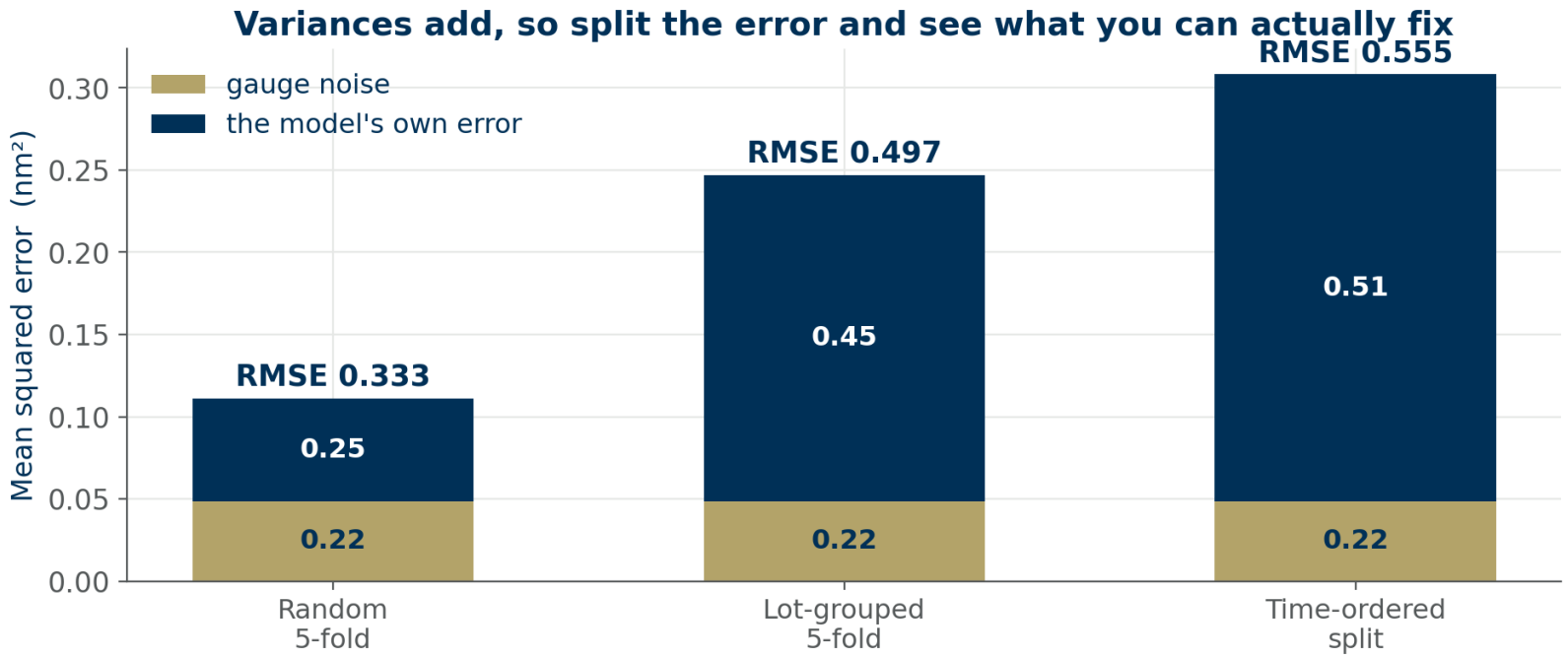
A perfect model still misses every label by the amount the gauge missed it.

$$R^2_{max} = 1 - \sigma^2_{gauge} / \sigma^2_{observed} = 0.948$$

Lecture 3 used that equation on the process. Point it at the label column and it gives you the ceiling on every model you will ever fit here.

Split the error and see what you can actually fix

Variances add, so the two contributions separate cleanly



Under the honest split, 0.220 nm is the gauge and 0.510 nm is the model. Modelling effort can only touch the second number.

What each level of accuracy lets you do

Accuracy is not a score. It is a permission slip.

Worse than the gauge

Use it for monitoring and for ranking wafers. Chart it, look for shifts, decide which wafers to measure. Do not put a number from it on a disposition record.

Comparable to the gauge

Now it can feed a run-to-run controller with a discounted gain, and it can drive an adaptive sampling plan. Most deployed virtual metrology sits here.

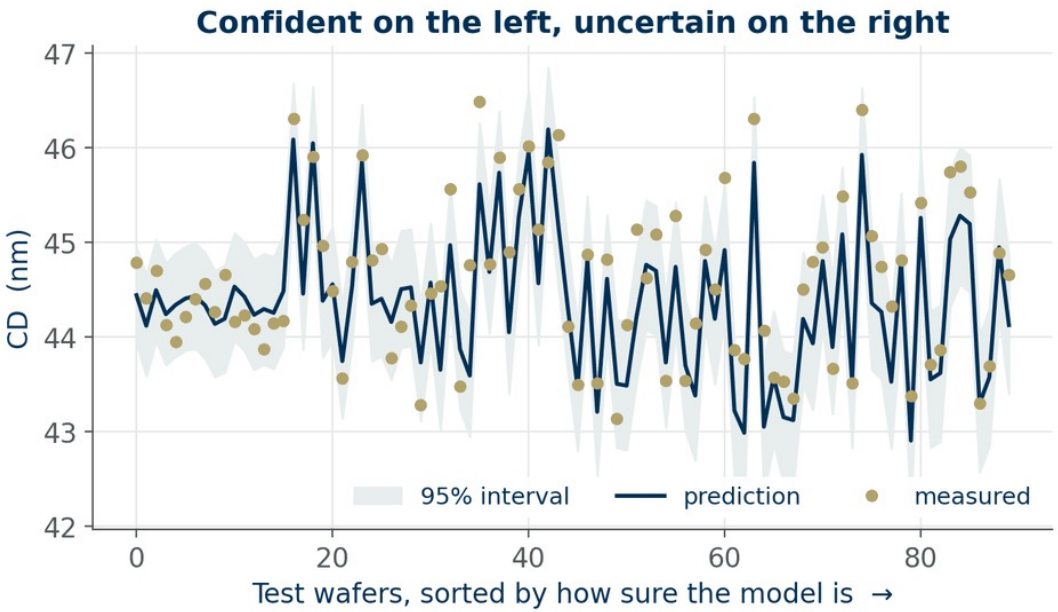
Better than the gauge

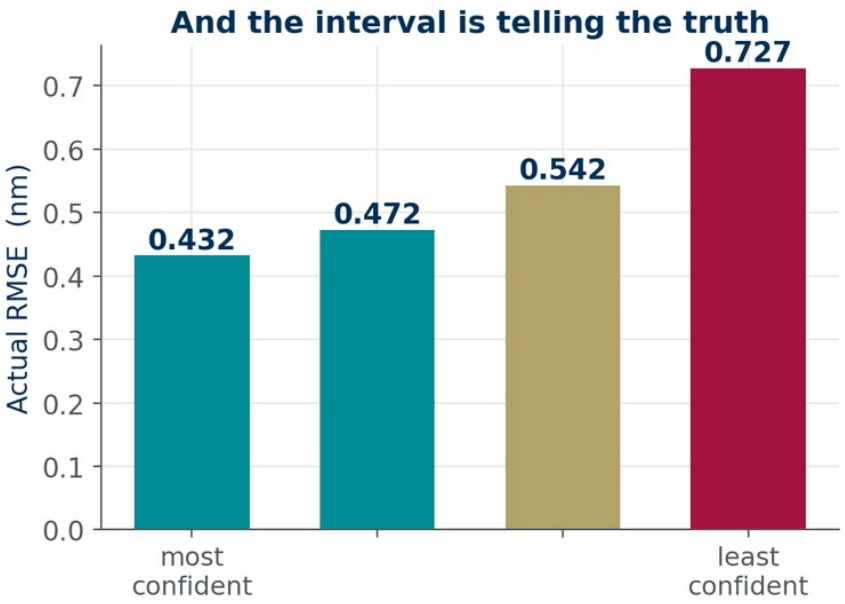
Rare, and worth pausing over — usually it means the model is averaging away metrology noise the gauge adds. Now skipping measurements becomes arguable, with a monitoring sample retained.

So decide what the model is allowed to do from what it measurably is, rather than from what it was built to be.

A prediction with an interval on it

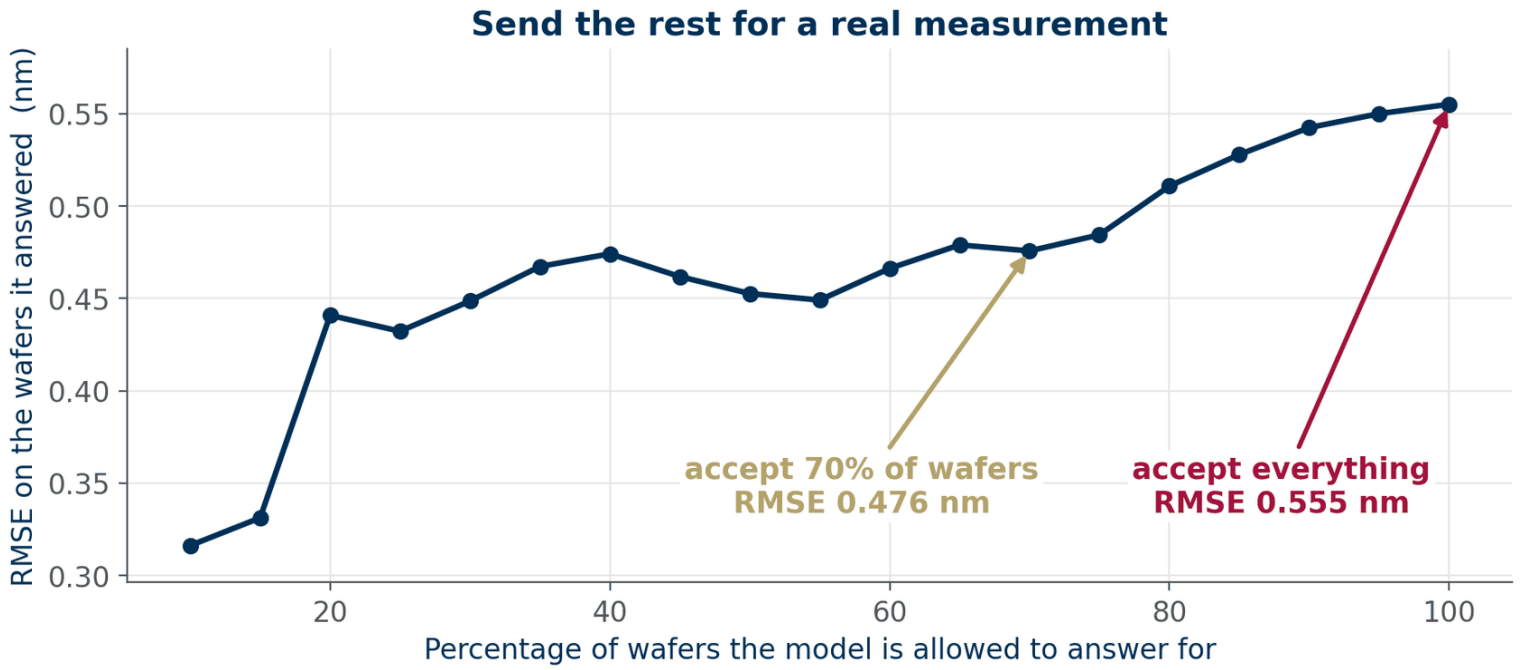
And the interval turns out to be honest





So let the model decline to answer

Accept the confident predictions; send the rest for a real measurement



Answering for 70% of wafers gives 0.476 nm instead of 0.555 nm. The model got better by doing less.

SEGMENT 3 OF 5

Virtual metrology inside the loop

Four things you can do with a prediction, in increasing order of what it costs to be wrong.

Four uses, increasing risk

Deploy in this order — each one earns the right to the next

1 · Monitor

Chart the prediction alongside the measurements you still take. Nothing downstream depends on it. This is where every deployment should start, and it costs nothing to be wrong.

2 · Feed the controller

The prediction becomes the feedback signal on unmeasured lots. Now a bad prediction moves the recipe, so the gain has to be discounted for the extra noise.

3 · Choose what to measure

Use the model's uncertainty to pick which wafers go to metrology. Wrong predictions cost you sampling efficiency and nothing else — the best value for the risk.

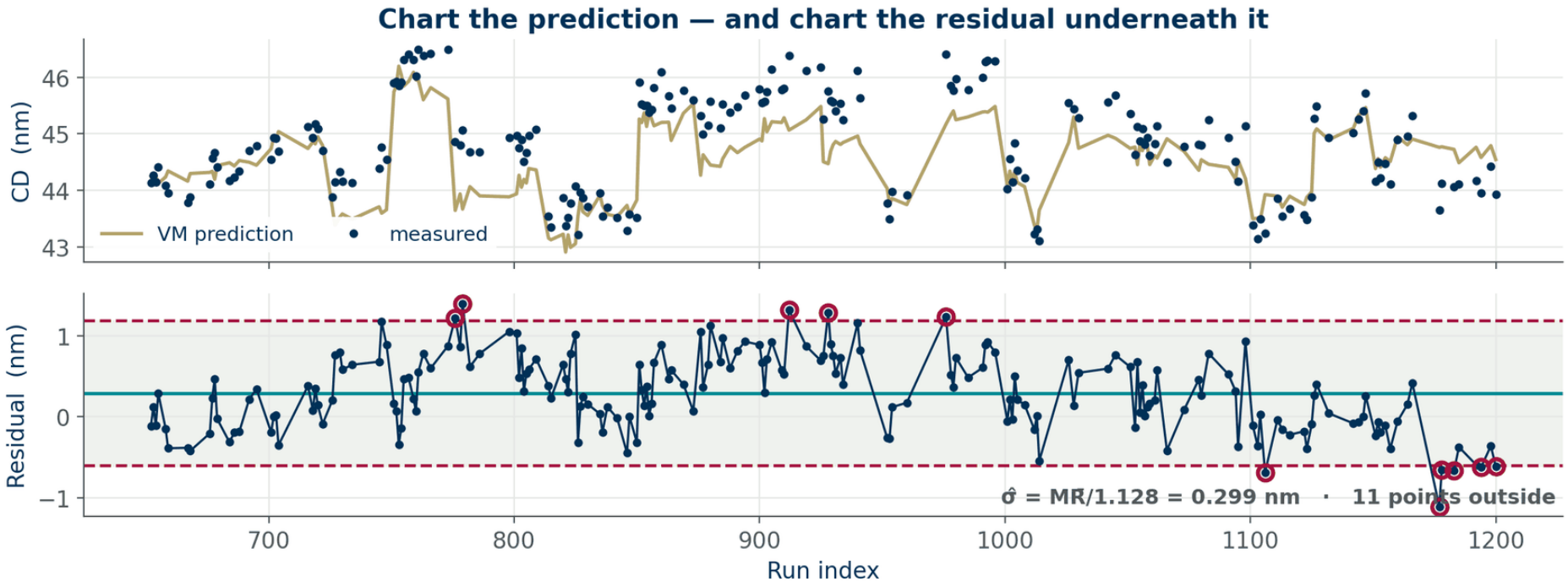
4 · Disposition wafers

Release or hold material on a prediction, with no measurement. Highest value, and the only one where being wrong ships bad product.

Most deployed virtual metrology lives at levels one to three. Level four needs a much stronger case than a good R^2.

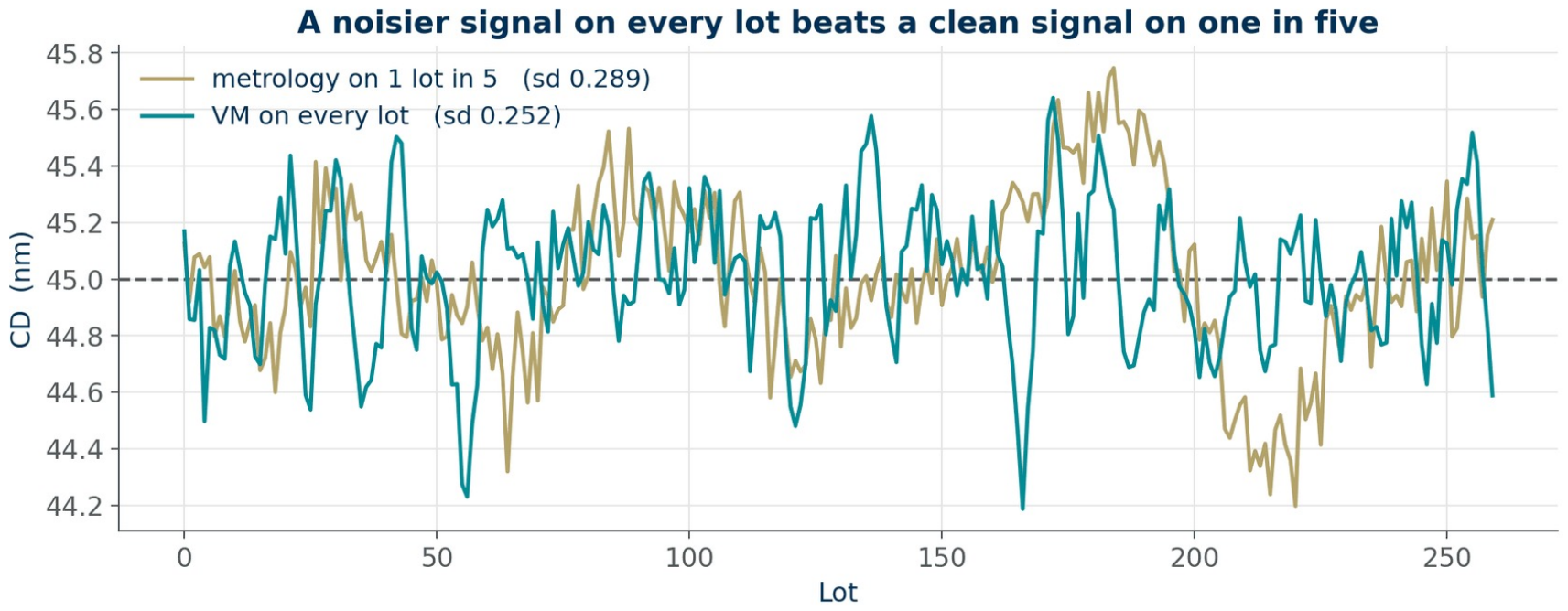
Use one: chart the prediction, and chart the residual

The residual chart is the model's own control chart



Use two: feeding the run-to-run controller

A noisier signal every lot, against a clean signal one lot in five

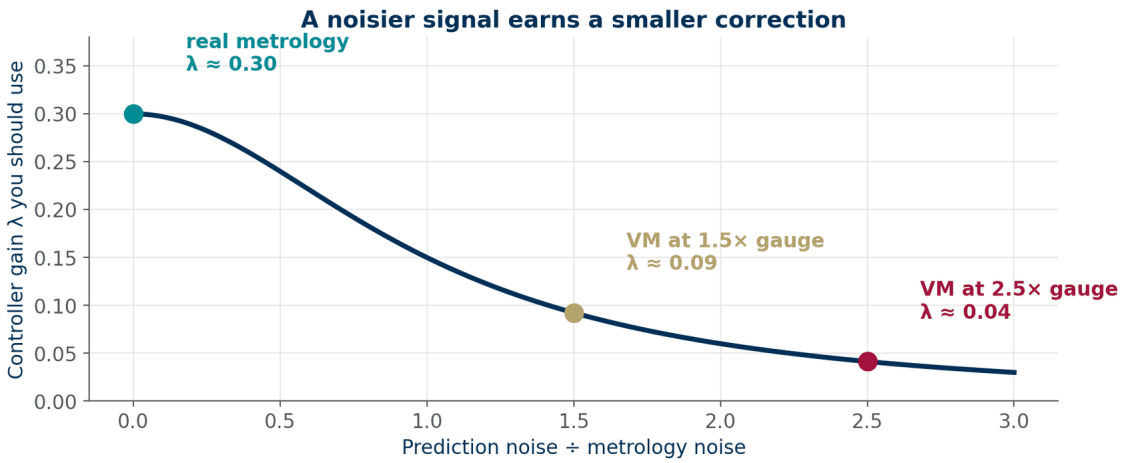


More frequent feedback beat cleaner feedback here — 0.252 nm against 0.289 nm. That is the operational case for virtual metrology.

But discount the gain for the extra noise

$$\lambda_{VM} = \lambda_{metrology} \; / \; (1 + (\sigma_{VM} / \sigma_{gauge})^2)$$

a noisier signal earns a smaller correction



Why it has this shape

The EWMA from Lecture 3 already weighs the newest observation against the history. A noisier observation deserves less weight, and the optimal weight falls with the square of the noise ratio.

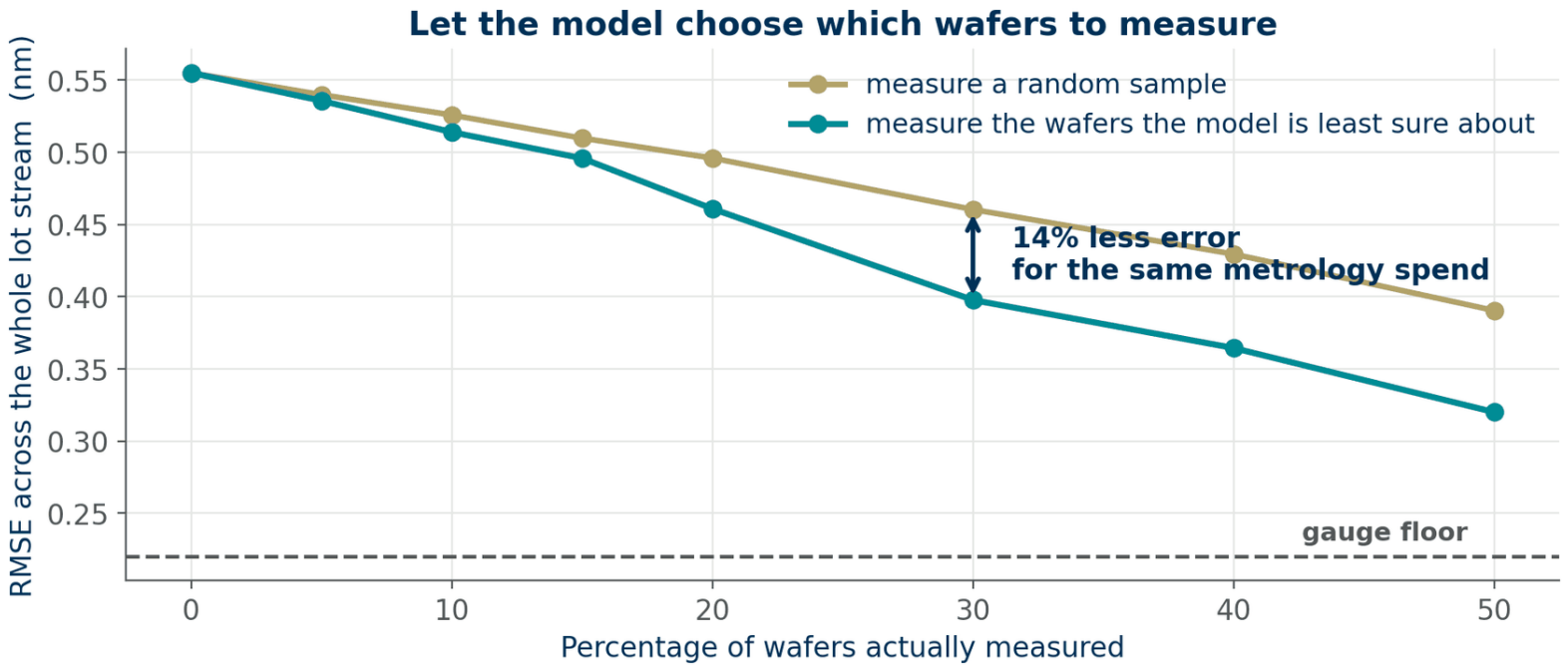
What happens if you skip it

The controller chases prediction noise and injects variation into a process that was fine. That is the tampering experiment from Lecture 1, with a model playing the part of the person.

The controller does not know the signal changed source. You have to tell it, by changing λ.

Use three: let the model choose what to measure

Same metrology budget, spent on the wafers the model is least sure about



At a 30% sampling rate this is 14% less error for exactly the same metrology spend. No new equipment, no extra risk.

That has a name, and the name is useful

Uncertainty-driven sampling is active learning

The immediate gain

Measuring where the model is least sure reduces error most per measurement. A fixed plan spends its budget on wafers the model already predicts well.

The compounding gain

Those measurements become tomorrow's training labels — and they are the informative ones, from exactly the regions the model handles worst. The sampling plan improves the model.

The trap to avoid

Sample only where the model is unsure and you stop observing the regions it thinks it understands, so you cannot tell when it stops. Keep a random sample alongside, always.

Roughly 80% chosen by uncertainty, 20% chosen at random. The random part is what keeps your evaluation honest.

Use four: releasing material on a prediction

Possible, occasionally justified, and it needs a different conversation

What the case has to contain

The honest error, under a time-ordered split, on the product and chamber in question.

The cost of a wrong release, multiplied by how often the error exceeds the margin to the spec limit.

A monitoring sample large enough to detect the model going wrong before the material ships.

What usually happens instead

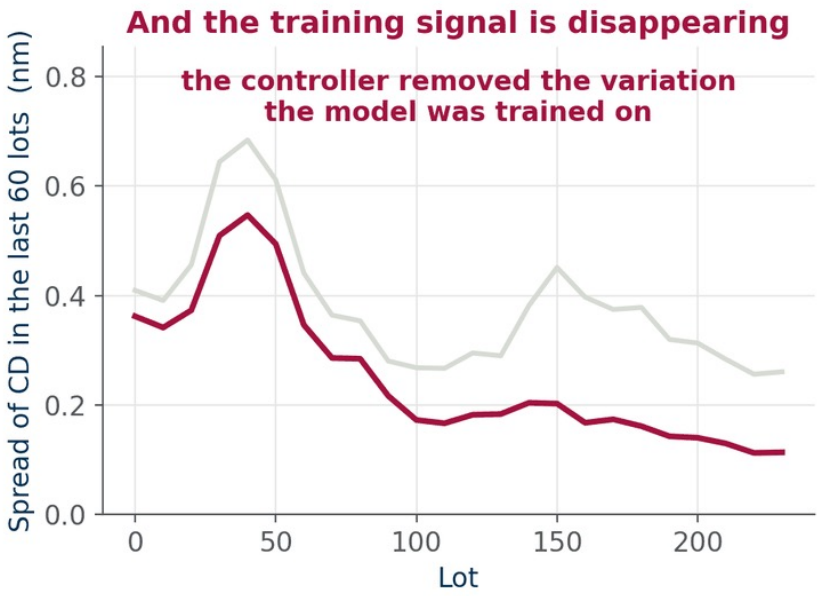
A hybrid. The model dispositions wafers that are comfortably inside the specification, and anything near a limit — or anything the model is unsure about — goes to metrology.

You keep most of the saving and give up almost none of the safety.

Set the abstain threshold from the cost of being wrong, and revisit it when the product mix changes.

The trap that catches good teams

What happens to your training data once the controller starts working



The controller removes the variation the model learned from. The chart goes flat, the model decays, and the flatness looks like success.

So pay for the data you need

Three ways, in ascending order of how much anyone will argue about them

Keep an uncompensated sample

A small fraction of lots run without the controller's correction, or with it logged and reversible. They cost a little variation and they are the only lots that still show you the raw process.

Log the correction

Every controller output, stored per lot. Then the uncompensated value is reconstructable — you add the correction back — and the model can be trained on it. Costs nothing except discipline.

Deliberate dithering

Small, planned perturbations of the recipe, on a schedule. Standard in process identification, and unpopular in a fab. Reach for it when the first two are not enough.

Logging the correction is nearly free and solves most of it. Do that before the loop is closed, not after.

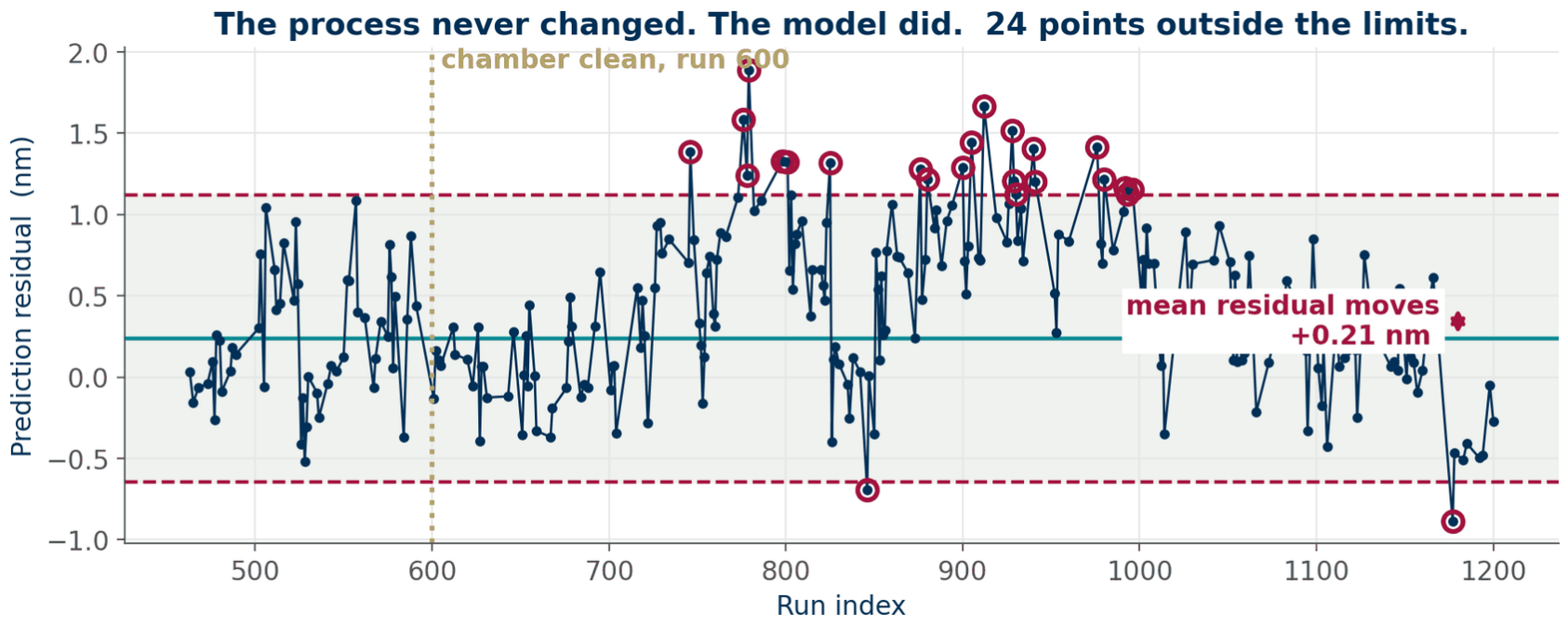
SEGMENT 4 OF 5

Keeping it alive

The model was correct on the day it was trained. That is the only day you can be sure about.

What model drift looks like on a chart

Trained before the chamber clean, then run forward through it



The residual mean moves 0.21 nm and 24 points break the limits. Nothing about the process was out of control.

Retrain on events, not on a calendar

The process does not drift on the first of the month

Chamber events

Cleans, preventive maintenance, part replacement, chamber rebuilds. Each one can move the sensor-to-CD mapping, and each one is already timestamped in the maintenance log.

Recipe and product events

Any recipe edit, any new product or layer running through the same chamber. The model was never trained on that condition and has no way to say so.

Statistical triggers

The residual chart from the previous slide. A run rule firing on the residual is the model asking to be retrained, and it needs no calendar.

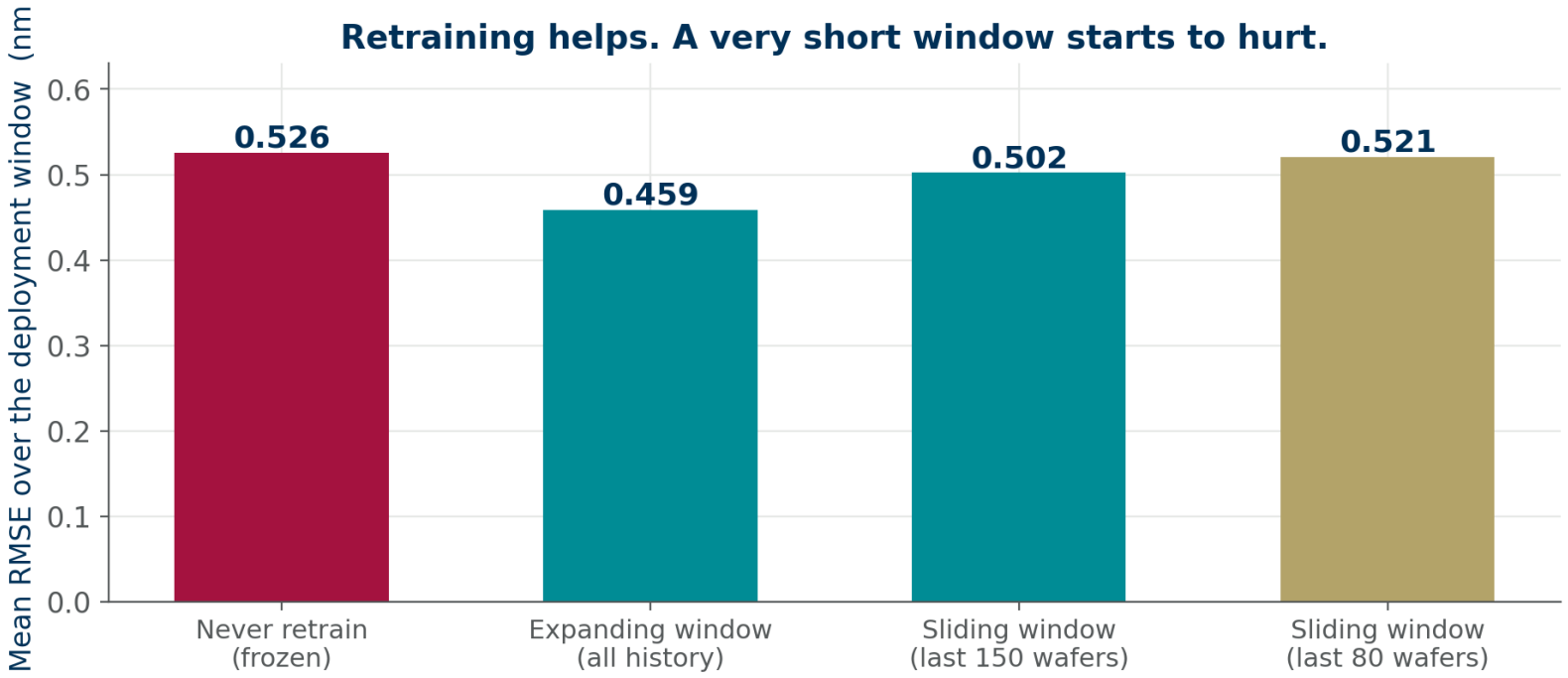
Scheduled refresh

A slow background retrain as a safety net, for the drift too gradual to trip a rule. Monthly is typical. Treat it as the floor, not the plan.

The maintenance log is a retraining schedule that somebody else is already maintaining for you.

How much history should a retrain use?

Measured on our own data, rolling forward through the deployment window



Retraining on all history beat every sliding window here, because 465 labels is too few to throw any away. Measure this on your own data.

One model, or one per chamber?

Three answers, and the middle one is usually right

A model per chamber

Matches each chamber exactly, and divides your labels by the number of chambers. On our data it scored worst — 0.576 nm against 0.497 nm for a single shared model.

One shared model

All the labels, and it averages over chamber differences it cannot see. Fine when chambers are well matched, and it degrades quietly when one chamber drifts away from the fleet.

Shared model, per-chamber offset

One model on all the data, plus a small correction fitted per chamber and refreshed often. Keeps the labels, absorbs the mismatch. 0.438 nm here, the best of the three.

This is transfer learning at its simplest — a shared representation and a cheap per-target adjustment. It scales to a fleet.

The part nobody writes a paper about

Everything between a working notebook and a running system

Shadow mode first

Run the model in production, log its predictions, act on none of them. Weeks of this. You learn the true drift rate, the true latency and the true failure modes before anything depends on the answer.

Version the whole thing

Model weights, feature code, the training set, the preprocessing. A prediction you cannot reproduce is a prediction you cannot defend when somebody asks why a lot was released.

Rollback in minutes

The previous model, one command away. And a defined fallback — usually reverting to the fixed sampling plan — so a bad model degrades service rather than stopping the line.

Name an owner

When the residual chart alarms at three in the morning, somebody is paged. If that person does not exist, the alarm is decoration and the model is unmonitored.

A model with no owner and no rollback is a prototype, whatever it is running on.

SEGMENT 5 OF 5

From one model to a digital twin

You have spent two lectures building one component of something larger. Here is what it is a component of.

What a digital twin actually is

ISO 23247-1:2021 — the manufacturing definition, and it is a careful one

"a **fit for purpose** *digital representation of an* **observable manufacturing element** *with* **synchronization** *between the element and its digital representation"*

"fit for purpose"

The twin models what the decision needs and nothing more. A twin for run-to-run control needs CD and chamber state. It does not need the plasma chemistry solved from first principles.

Fidelity is a cost, so you buy only what a decision will use.

"observable manufacturing element"

The thing being twinned: a chamber, a tool, a lot, a recipe, an operator, a whole facility.

Observable is the operative word. If you cannot sense it, you cannot synchronise to it, and you have a simulation instead.

"synchronization"

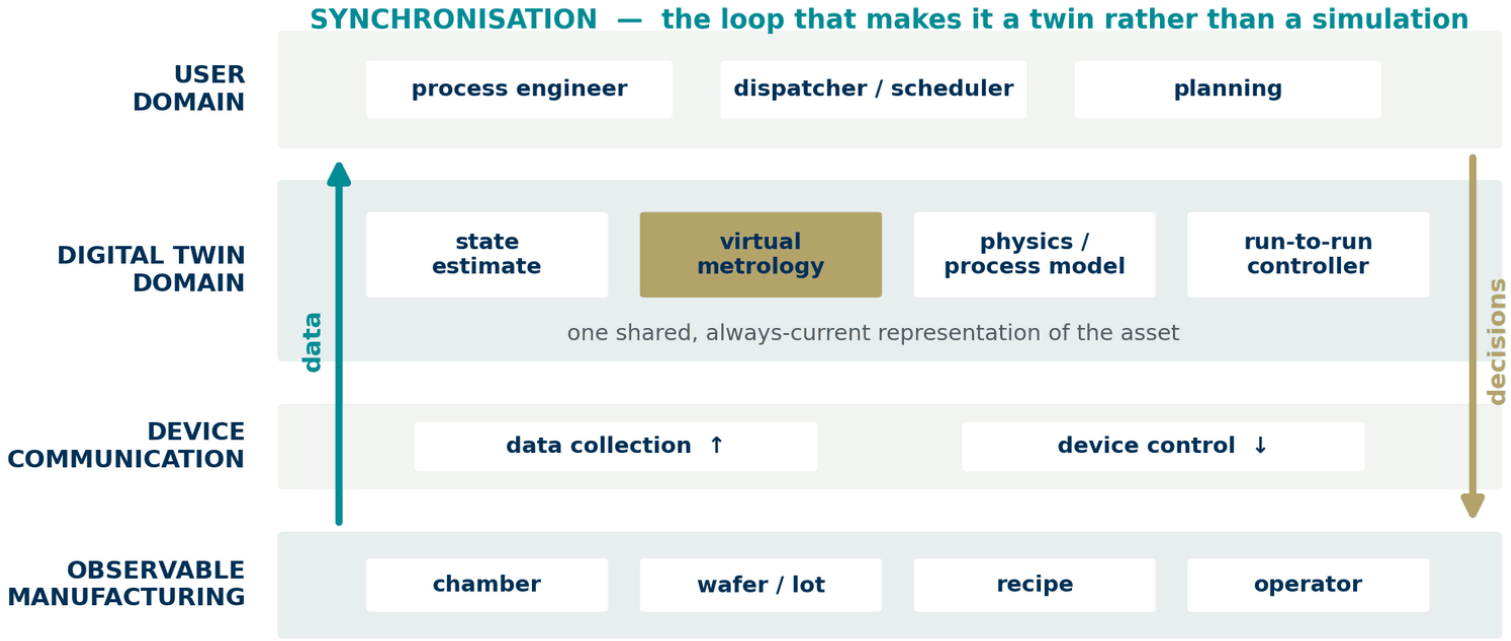
The twin updates as the asset changes, and it can send something back. This is the clause that does the work.

A model that is not kept in sync with its asset is a simulation, however detailed it is.

A simulation answers what would happen to a chamber like this one. A twin answers what is happening to this chamber, now.

Where your model sits inside it

The ISO 23247 reference architecture, with Lecture 1's model in gold

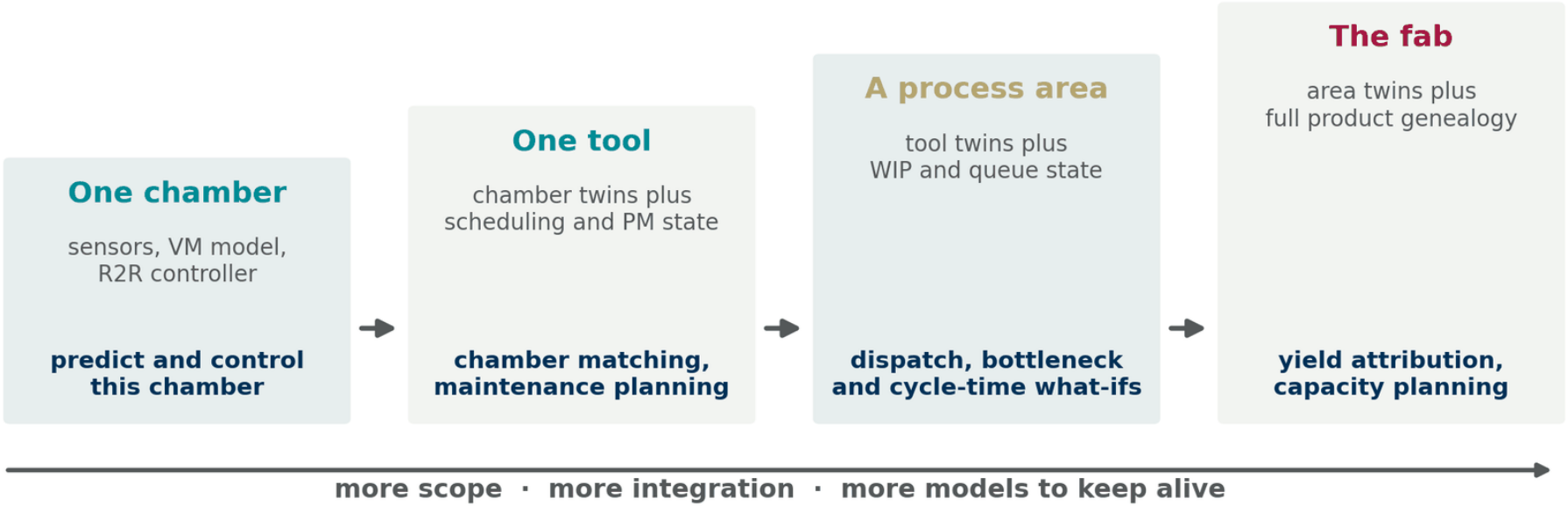


Everything gold is what you built. The twin is the box it plugs into, and the loop around the outside.

How far does the twin extend?

Each rung needs everything to its left, kept alive, in sync

Almost every fab digital twin in production today lives in the first two boxes.



Scope multiplies the maintenance burden rather than adding to it. Earn each rung before you claim the next.

What it adds, and what it costs

Both halves are real, and the second half is where the projects die

It can answer questions you did not ask

A model predicts one number. A synchronised twin can be re-run with a changed input — a different recipe, a delayed PM, a re-ordered queue — and give you the counterfactual before you commit.

It gives the fleet one shared state

Chamber matching, dispatch and scheduling all need the same current picture of every tool. A twin is that picture, maintained once instead of reconstructed by each application.

Every component inherits today's four problems

Leakage, a noise floor, drift and ownership. A twin with forty models has forty residual charts to watch and forty retraining triggers — and it is as current as its most neglected component.

The integration is the hard part

One shared state needs one shared vocabulary across tools from different vendors, and that metadata problem is bigger than any of the models. It is why standards like ISO 23247 exist at all.

A digital twin is a maintenance commitment before it is a capability. Start with one chamber and keep it honest.

One sentence for the whole unit

A virtual metrology model is a metrology tool made of software.

So it gets a gauge study

You would never accept a CD-SEM without knowing its repeatability. Characterise the model's error the same way — honestly split, in nanometres, against the gauge it is imitating.

So it gets a control chart

Every metrology tool in a fab has one. The model's is the residual chart: a small sample of real measurements, charted against the prediction, with limits and run rules.

So it gets a maintenance schedule

Tools are recalibrated after events and on a schedule. So are models. Retrain on chamber events, recipe changes and residual alarms, with a slow refresh underneath.

Everything in these two lectures follows from taking that sentence literally.

Homework and project prompt

Same datasets · due as posted

- **1 Redo it honestly.** Take your Lecture 1 model unchanged. Report R^2 and RMSE under random 5-fold, lot-grouped 5-fold, leave-one-chamber-out, and a time-ordered split at 58%. Explain each gap.
- **2 Score against the baseline.** Compute the previous-measurement baseline on the time-ordered test window only. State the model's advantage over it, honestly.
- **3 Find the break.** Train on runs before 600, apply forward, and plot the residual against run index with limits from held-out pre-clean wafers. When would it have alarmed?
- **4 Budget the error.** Split your honest RMSE into the gauge contribution and the model's own. How much of the remaining error is worth chasing?
- **5 Tune the controller.** Given your model's RMSE and a gauge σ of 0.22 nm, what λ should a run-to-run controller use on virtual measurements if it uses 0.30 on real ones?
- **6 Project.** Propose a deployment: which of the four uses, what abstain threshold, what the residual chart looks like, what triggers a retrain, and what rollback means. Two pages.

References, and where this goes next

Leakage. Kaufman, Rosset and Perlich, *Leakage in Data Mining* (2011). The canonical treatment, and every example in it has a fab equivalent.

Evaluating time series models. Bergmeir and Benítez on cross-validation for time-dependent data — why forward-chaining is the right default when order matters.

Active learning. Settles, *Active Learning Literature Survey*. Uncertainty sampling is chapter one, and it is what slide 27 was doing.

Conformal prediction. A distribution-free way to get the intervals from slide 20 out of any model, including gradient boosting. Increasingly the practical default.

Concept drift. Gama and colleagues, *A Survey on Concept Drift Adaptation* — detection, windowing and retraining strategies, with the vocabulary the literature uses.

Where this goes next. Everything after this in the course — defect classification, yield prediction, learned control — inherits the same four problems: leakage, a noise floor, drift, and nobody owning the model at three in the morning.

Georgia Tech