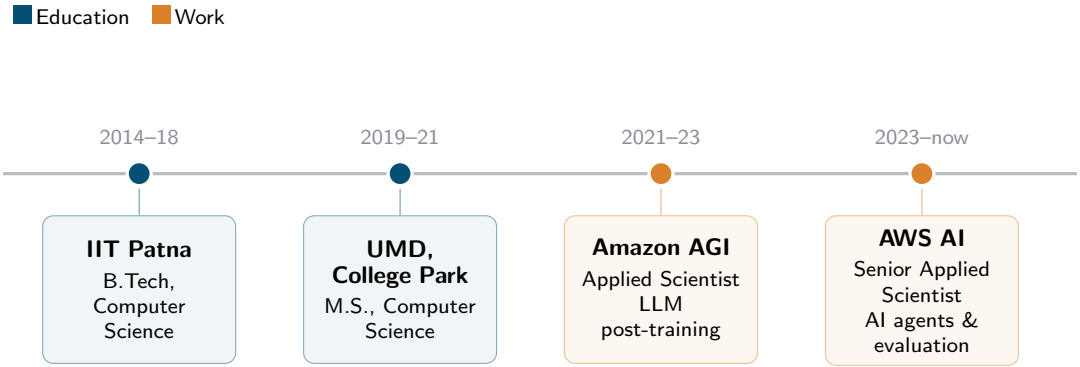
From Notebook to the Real World

Real-world constraints, responsible deployment, and accountability in ML

Abhishek Kumar
Senior Applied Scientist, AWS AI
Guest lecture, IIIT Delhi

About me



From NLP research, to training data for large language models, to building and testing AI agents.

What I've worked on: the same ideas as this course, at larger scale

Alexa AI • 2020

Voice assistant.
Helping Alexa remember context across separate conversations.

Amazon AGI • 2021–23

Data for large language models.
Improving the quality of the data used to train LLMs.

AWS AI • 2023–now

AI agents and how to test them.

- Open-source tools that find *why* an agent failed
- Simulated users that test agents before real users do

Today's plan

Part 1
Real-world constraints

Is it fast and small enough?

- Latency: chatbot, coding agent, offline jobs
- Memory: cloud vs. on-device
- Making models smaller

Part 2
Responsible deployment

How do we launch it safely?

- Running example: a Q&A assistant
- Shadow testing
- A/B testing

Part 3
Accountability

Is it still working, for everyone?

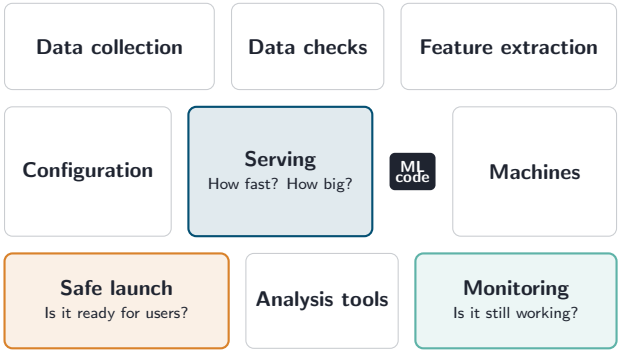
- Monitoring dashboards
- Success without labels
- Fairness by group

In a notebook, `accuracy = 0.93` is the end. In a product, it's the start.

```
model = XGBClassifier()
model.fit(X_train, y_train)

acc = model.score(X_test, y_test)
print(acc) # 0.93
```

The model is the small black box.
Everything around it decides whether real users are well served.



After Sculley et al., "Hidden Technical Debt in Machine Learning Systems", NeurIPS 2015.

Part 1

Real-world constraints

Accuracy is not the only number that matters. Every product also has a **time budget** and a **memory budget**.

1. Real-world constraints
2. Responsible deployment
3. Accountability: monitoring

Latency: where does the time go in one request?

Latency is the time from a user asking to the user getting an answer.



Network **60 ms** Queue **40 ms** Pre-process **20 ms** Model **420 ms** Safety + post-process **60 ms**

Total ≈ 600 ms (illustrative numbers)

Averages hide the slow requests

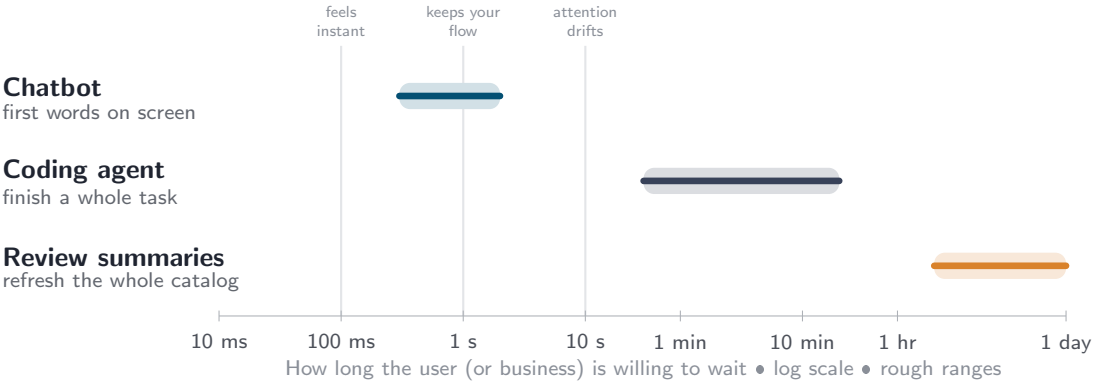
p99 = the slowest 1% of requests. If one user action makes n model calls,
$P(\text{at least one slow}) = 1 - 0.99^{n}$

n calls	1	10	100
hit a slow call	1%	10%	**63%**

How teams cut latency

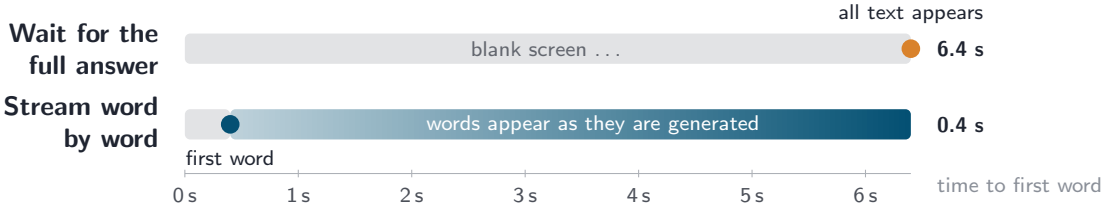
- **Move closer:** run servers near the users.
- **Avoid queues:** plan for the busiest hour.
- **Shrink the model:** smaller models run faster.

Different products, very different time budgets



Low latency: a chatbot's first word has to come fast

Same model, same total time (6.4 s). What the user sees on screen:



Illustrative numbers.

Stream the answer

Show words as they are generated.

Right-size the model

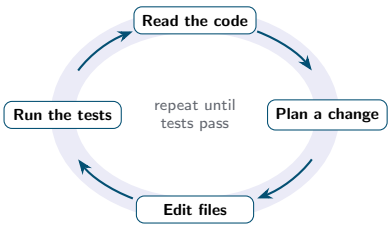
A small, fast model for easy questions.

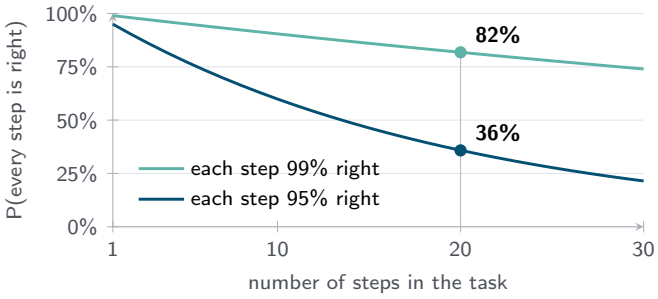
Cache

Reuse answers to popular questions.

Medium latency: for coding agents, slow is fine but wrong is not

Tools like Claude Code take a task and loop: read, plan, edit, test. One task can take minutes.

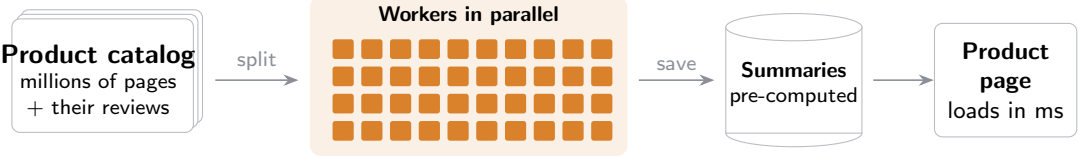




Small errors compound over many steps, so coding agents spend the time on **accuracy**: bigger models, running the tests, double-checking.

High latency (offline): for review summaries, nobody is waiting

Amazon shows AI-written review highlights on its product pages. There are millions of pages, and a summary only has to be ready by the time someone opens one.



10M products × 2 s each, split over w workers:

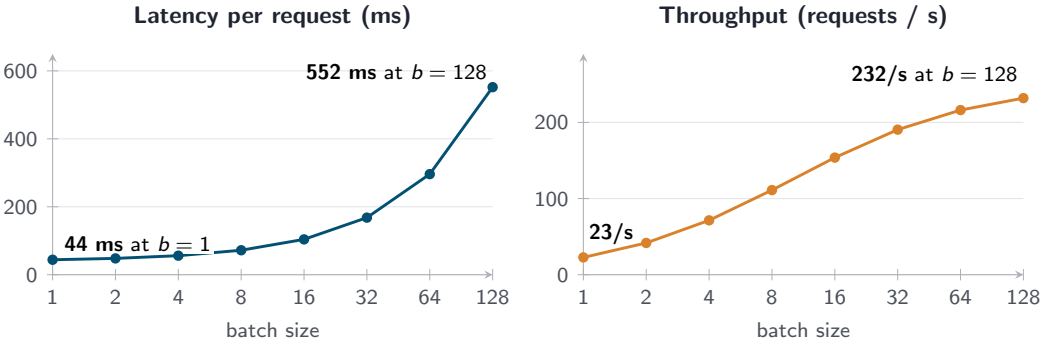
workers w	time to finish
1	231 days
100	2.3 days
1,000	**5.6 hours** (overnight)

Illustrative numbers.

- Measure **throughput** and **cost**, not latency.
- Each product is independent, so **split the work**.
- Pre-compute at night, then serve in milliseconds.

Batching: more throughput, but each request waits longer

A GPU can work on many requests at the same time.



Chat: small batches, low latency

Overnight job: big batches, 10× the throughput

Toy model: one pass on b requests takes $40 + 4b$ ms.

Memory: how much does a model need?

Rough rule: **number of parameters** $\times$ **bytes per parameter** (weights only). Logistic regression: a few KB. KNN: the whole training set. LLMs: billions of weights.

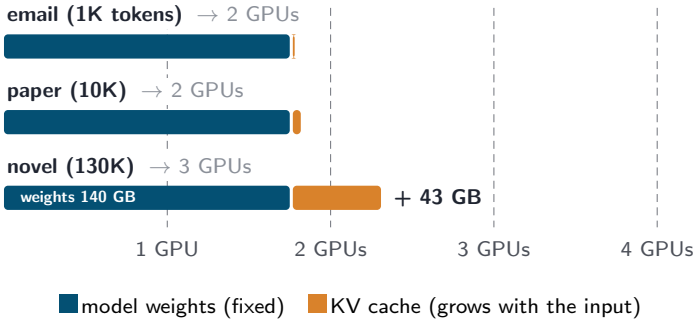
Parameters	FP32 (4 bytes)	FP16 (2 bytes)	INT8 (1 byte)	INT4 (0.5 byte)
1 billion	4 GB	2 GB	1 GB	0.5 GB
3 billion	12 GB	6 GB	3 GB	1.5 GB
8 billion	32 GB	16 GB	8 GB	4 GB
70 billion	**280 GB**	**140 GB**	70 GB	35 GB
405 billion	**1,620 GB**	**810 GB**	**405 GB**	**203 GB**

fits a phone ($\leq$4 GB free of 8) fits a laptop (16 GB) fits one big GPU (80 GB) needs several GPUs

Cloud: question answering over a whole novel

Say a student uploads a whole novel and asks why the narrator leaves home in chapter 3.

- Read the **whole book**: ∼130K tokens
- **Reason** across chapters, so a big model
- Keep a memory of every token read (the **KV cache**)



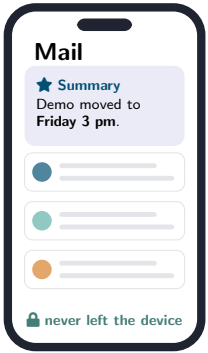
Big model + long input ≈ 183 GB → **several data-center GPUs**. A phone can't do this.

On-device: summarizing your email can happen on your phone

- **Privacy:** your email never leaves the phone
- **Speed:** no round trip to a server
- **Works offline:** on a plane, in a tunnel
- **Cheap:** runs on the user's own chip

But a phone is a tight box

~8 GB of memory **shared with everything else**, plus battery and heat limits. So: small models, compressed to a few bits per weight.



Making models fit: quantization

Same weights, stored with fewer bits:

PRECISION	BITS FOR ONE WEIGHT	STORED VALUE	8B MODEL	FITS A PHONE?
FP32		`0.70710677`	**32 GB**	✗
FP16		`0.7071`	**16 GB**	✗
INT8		`≈ 0.71`	**8 GB**	✗
INT4		`≈ 0.7`	**4 GB**	✓

The idea

Store each weight with fewer bits by rounding it to one of a few levels (4 bits = 16 levels). FP16 → INT4 makes the model **4× smaller**.

The catch

Rounding loses a little accuracy. Always re-check on your test set before you ship.

Part 2

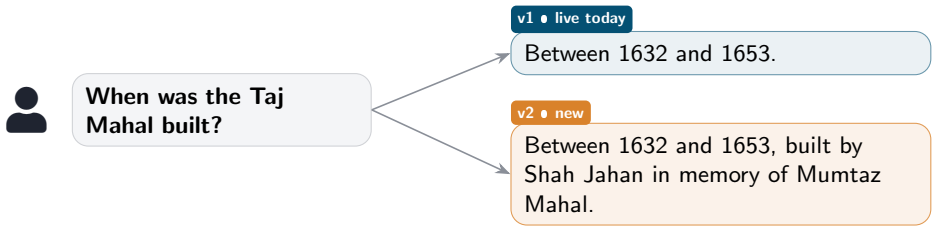
Responsible deployment

You trained a model that beats the old one on your test set. How do you launch it **without hurting real users**?

1. Real-world constraints
2. Responsible deployment
3. Accountability: monitoring

Running example: a question-answering assistant

Our assistant runs **v1** today. We have trained **v2**, and it does better on the test set.



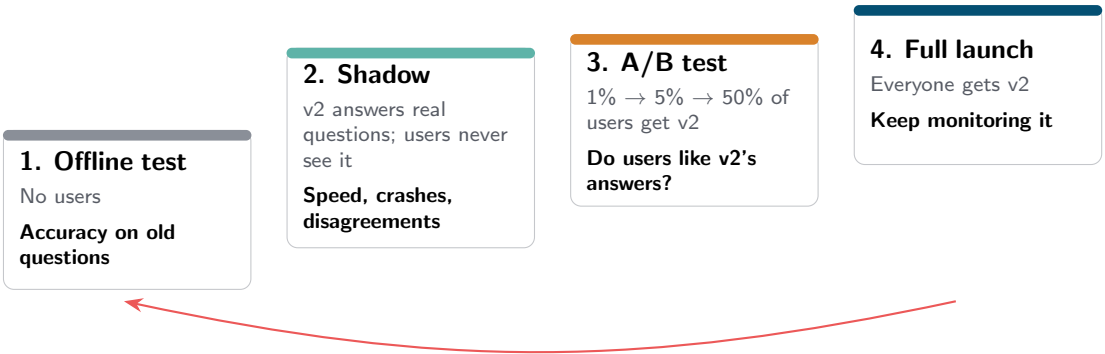
	v1	v2
Accuracy on the test set	88%	**91%**

Illustrative numbers.

The test set is made of past questions, though, and real users keep asking new ones.

The launch ladder: never jump straight to 100% of users

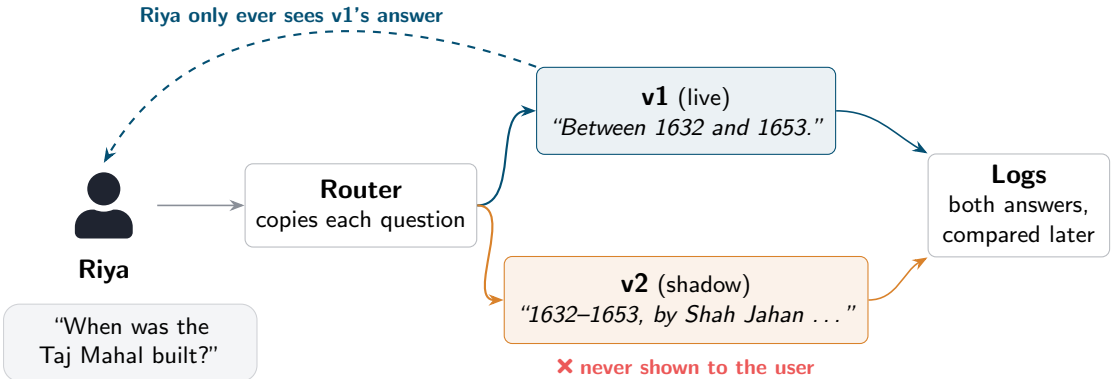
Each step shows v2 to more people, and only after the previous step looked fine.



Something looks wrong at any stage? **Roll back** to v1 in one click.

Shadow testing: real questions, zero risk

Every real question is also sent to v2. Users only ever see v1's answer.



Shadow testing has no labels. So what do we compare?

Logged question	v1 (live)	v2 (shadow)	Judge says
When was the Taj Mahal built?	1632–1653	1632–1653	same answer: skip
Who designed the Taj Mahal?	Shah Jahan	Ustad Ahmad Lahauri	**v2 better**
Which city is it in?	Agra	New Delhi	**v1 better** (v2 is wrong)



100 logged questions (illustrative)

88 identical: low risk, skip

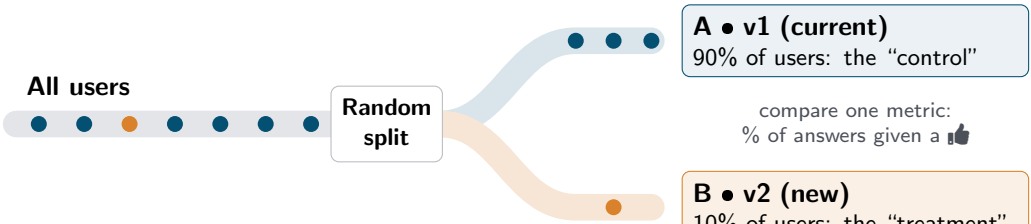
12 different: a judge labels *only these*

7 v2 better **3** v1 better **2** tie

Users never see v2, so we can't tell if they *like* it.

That's what A/B testing is for.

A/B testing: let a small, random slice of users try v2



Random

The two groups are alike in every way except the model.

At the same time

Weekends and outages hit both groups equally.

Metric chosen first

Decide what "better" means **before** looking.

A/B assignment: why not flip a coin for every question?

Riya asks three questions in a row, and a coin decides which model answers each one:

When was the Taj Mahal built?

v1 Between 1632 and 1653.

Who designed it?

v2 Ustad Ahmad Lahauri, for the emperor Shah Jahan.

Which city is it in?

v1 Agra.

Confusing for the user

The answers switch style halfway through the conversation.

An unclear result

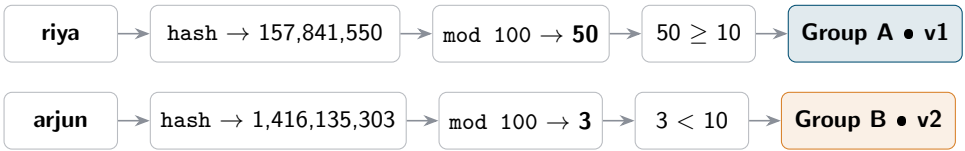
If Riya is unhappy, we can't tell which model caused it.

Instead

Pick the group **once per user** and keep it.

The fix: hash the user ID to pick the group

Each server can work out a user's group on its own, with no shared database:



Rule: `bucket = hash("qa-v2-test:" + user_id) mod 100`; buckets 0–9 (10% of users) get v2.

Sticky

Same user → same group, on every server, every time.

Even

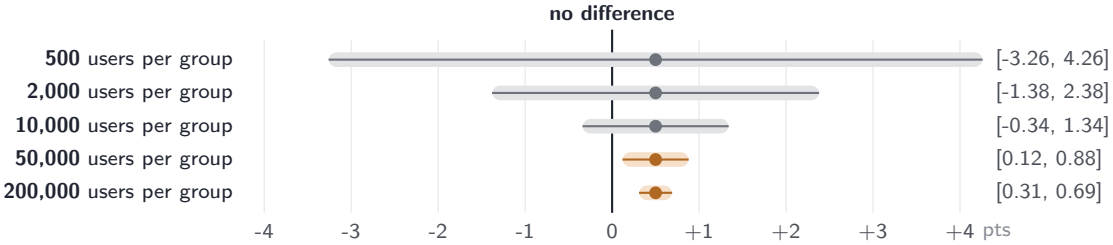
Users spread evenly over buckets 0–99, like a fair draw.

Fresh for each test

A new test name reshuffles: in "qa-v3-test", Riya gets group B.

v2 got 10.5% 👍, v1 got 10.0%. Real, or just luck?

Same idea as estimating a mean from samples: the more users, the smaller the uncertainty.



95% confidence interval for (v2 − v1), illustrative.

Read the interval

If it **includes 0**, you can't tell yet: collect more users.

Set the rules first

Fix the metric and test length up front.
Peeking every hour finds a "winner" by luck.

Part 3

Accountability: monitoring

Launch day is not the finish line. Models break **silently**: the data changes, users change, and nobody tells you.

1. Real-world constraints
2. Responsible deployment
3. Accountability: monitoring

A model dashboard: is everything OK right now?

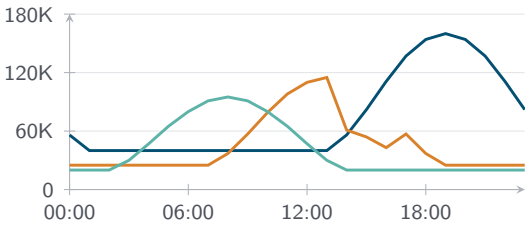
Model calls today
3.81M
+3% vs last week

p99 latency
1.8 s
budget 2.0 s

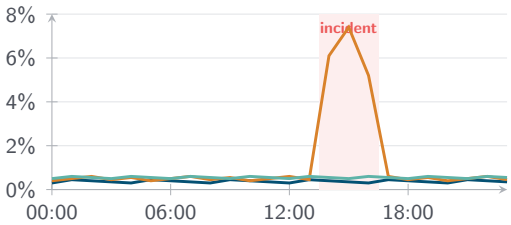
Error rate, Europe
7.4%
peak at 15:00

Task success
91.2%
−1.1 pts this week

Model calls per hour



Error rate



US East Europe Asia Pacific

Illustrative numbers.

No labels in production. So how do we know it worked?

Explicit feedback

👍 / 👎, ratings, "report a problem".

Problem: few people use it, mostly when upset.

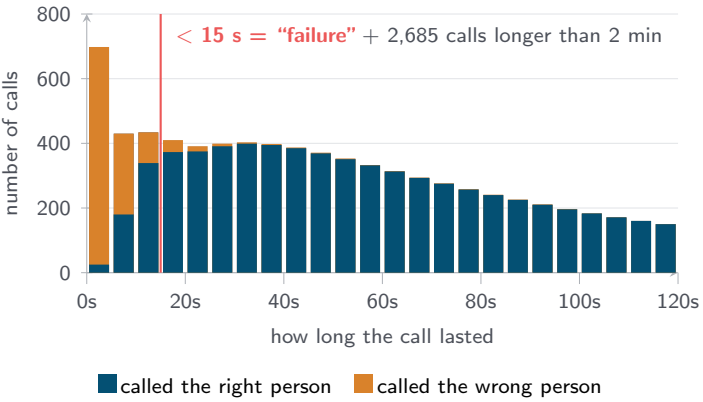
Implicit signals: what users *do* next

Product	**Likely success**	**Likely failure**
Q&A assistant	Copies the answer, moves on	Asks the same thing again in other words
"Alexa, call Mom"	Call lasts a few minutes	Hangs up within seconds
Coding agent	Code is kept, tests pass	Changes are undone

Implicit signals cover **every** interaction and cost nothing to collect. But they are guesses, so check them against a few real labels.

“Alexa, call Mom”: is a short call a failure?

We’d like a success rate for “Alexa, call Mom” without listening to anyone’s calls. One option is to count very short calls as failures.



Illustrative numbers.

Success rate by this rule

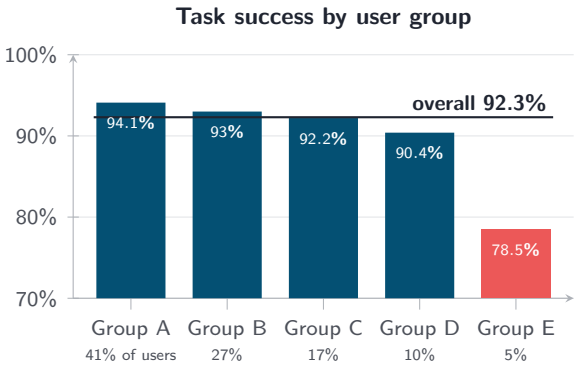
84.9%

True success rate: **89.4%**

	who was actually called: wrong	right
rule: failure	1,018	544
rule: success	83	8,728

The rule is a proxy. It has its own **confusion matrix**.

Fairness & transparency: an average can hide a group that's badly served



Illustrative numbers.

Slice every metric

By language, region, device, age group.
One confusion matrix **per group**.

Be transparent

Write down what it is for, its data, and where it fails ("model cards").

Three things to remember

1 • Fit the budget

Pick the best model that meets the product's **time** and **memory** budget. Chat, agents and batch jobs need different choices.

2 • Launch in steps

Offline → shadow → A/B → full, with rollback ready. One group per user (hash the ID). Choose the metric first.

3 • Keep watching

Dashboards, implicit signals (checked against labels), and every metric sliced by group.

The model is the small box. The system around it is the product.

Thank you! Questions?

Abhishek Kumar
Senior Applied Scientist, AWS AI

iyamabhishek.github.io linkedin.com/in/iyamabhishek