

ELLYPUNK

Classic Training

WEEKLY DIGEST

14-20 September 2026



This week

14-20 September 2026

Available archived editions, in their original date order.

MONDAY / 14 September 2026

Elly Explains: What Is Agile?

3

[Open original online](#)

MONDAY / 14 September 2026

Fact Table Grain: One Row, One Promise

8

[Open original online](#)

TUESDAY / 15 September 2026

Elly Explains: What Is Python?

13

[Open original online](#)

TUESDAY / 15 September 2026

SQL Anti-Joins: Find What Is Missing

18

[Open original online](#)

WEDNESDAY / 16 September 2026

Docker Multi-Stage Builds: Ship Only What Runs

23

[Open original online](#)

Only retained editions are included; gaps have not been filled.

Copyright Ellypunk. Contains AI-generated artwork.

ellypunk.com

Elly Explains:

WHAT IS AGILE?

Agile helps a team build software in **small, useful steps**—then **learn and adapt**.

WHY CARE?

Instead of waiting months to discover the wrong thing, a team can show a small working slice **sooner**.

BUILD A LITTLE



SHOW IT



LEARN WHAT'S NEXT

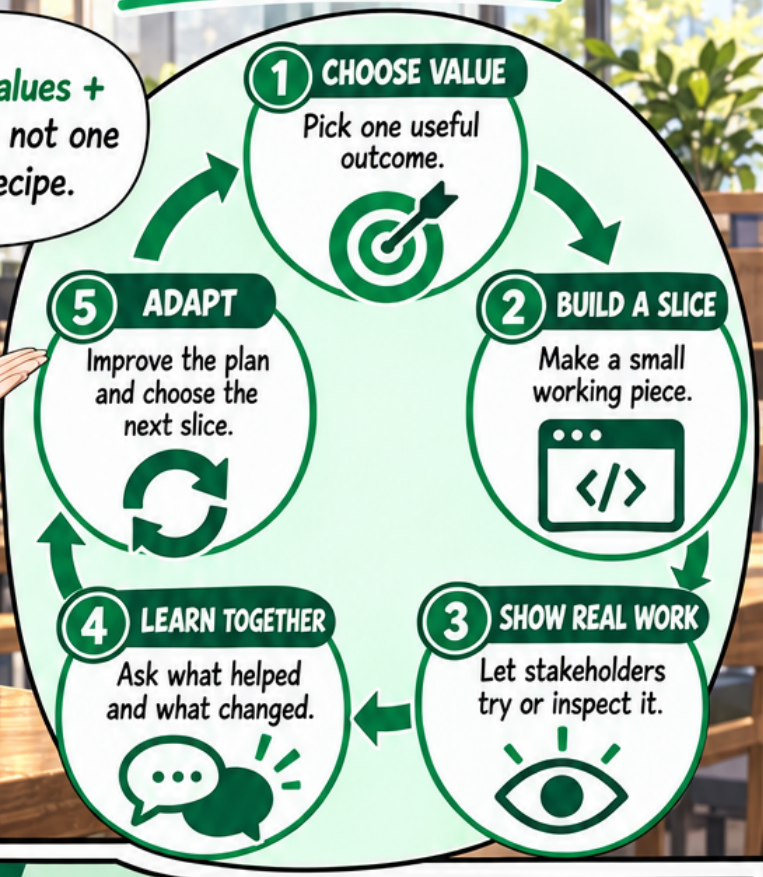


NEXT: THE FEEDBACK LOOP →

Elly Explains:

THE AGILE FEEDBACK LOOP

Agile is **values + principles**, not one fixed recipe.



AGILE VALUES



PEOPLE & INTERACTIONS



WORKING SOFTWARE



CUSTOMER COLLABORATION



RESPONDING TO CHANGE

Plans, tools, contracts and documents still matter.
Agile values the left side more.

NEXT: A TINY EXAMPLE

Elly Explains: A TINY WEATHER APP

FICTIONAL EXAMPLE



GOAL



Help commuters
prepare for rain.



BACKLOG

1

TODAY'S RAIN CHANCE



2

SAVE A HOME CITY



3

HOURLY RAIN CHART



FIRST SLICE



FEEDBACK



ADAPT



SMALL SLICE → REAL FEEDBACK → BETTER NEXT CHOICE

NEXT: USE IT WISELY →

Elly Explains:

AGILE IN PRACTICE

TEAMS OFTEN...



Choose a valuable next slice.



Check quality as they build.



Show usable work to stakeholders.

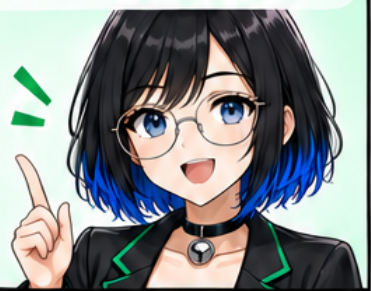


Reflect and improve how they work.

MYTH: AGILE MEANS NO PLAN.



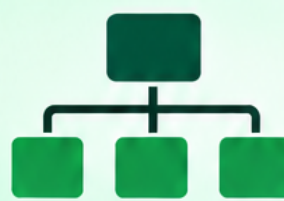
REALITY: Plans evolve as the team learns.



MYTH: AGILE IS SCRUM.



REALITY: Scrum is one framework that can support agile work.



MYTH: AGILE GUARANTEES SPEED.



REALITY: Agile aims for earlier value and learning; speed is not guaranteed.



NEXT: YOUR AGILE TOOLKIT





RECAP

1 VALUE



Pick a useful outcome.

2 SLICE



Build a small working piece.

3 FEEDBACK



Learn from real use.

4 ADAPT



Change the next step.

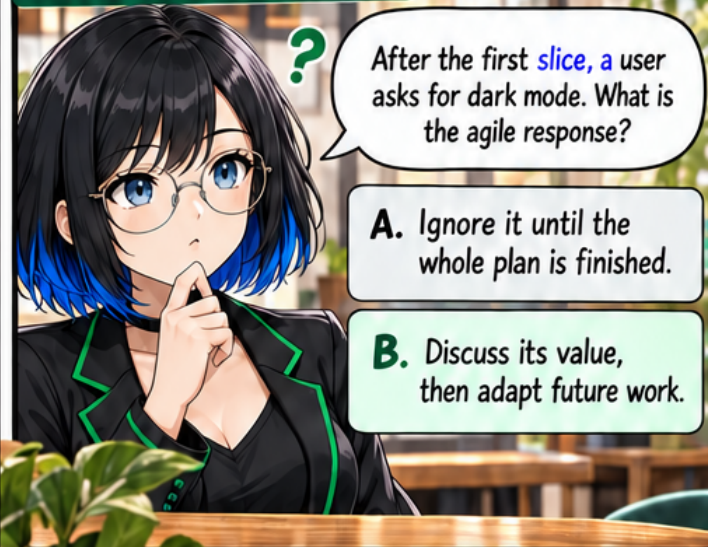
TINY GLOSSARY

Increment: a usable step toward a product goal.

Stakeholder: a person affected by or interested in the product.

Retrospective: a regular look at how a team can improve.

QUICK CHECK



After the first **slice**, a user asks for dark mode. What is the agile response?

A. Ignore it until the whole plan is finished.

B. Discuss its value, then adapt future work.



NEXT: WHAT IS PYTHON? →

FACT TABLE GRAIN: ONE ROW, ONE PROMISE

OBJECTIVE

Define **exactly** what each fact row measures **before** choosing columns.

ONE ROW

A grain declaration says what **one row** represents.



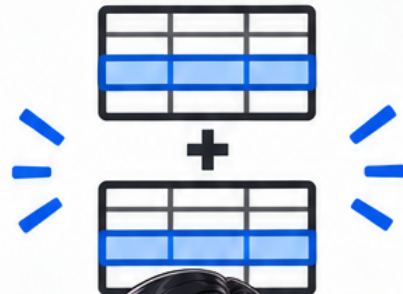
ONE LEVEL

Every dimension and fact must match **that level**.



ONE RISK

Mixed grain can silently **double** your totals.



One row, one promise.
Keep it true.



DESIGN IN THE RIGHT ORDER

1 SELECT PROCESS

Retail sale.



2 DECLARE GRAIN

One product line on one completed receipt.



3 CHOOSE DIMENSIONS

Date, Store, Product, Customer, Receipt.



4 CHOOSE FACTS

Quantity, Line Net Amount, Discount Amount.



Grain first makes every later choice testable.



KEEP EVERY COLUMN TRUE TO THE ROW



BELONGS ON A SALES LINE

Product Key	Store Key	Quantity	Line Net Amount
			
			

DOES NOT BELONG

Receipt Total - repeats across lines.	Monthly Target - month-store grain.	Customer Lifetime Value - changes independently.
		
		

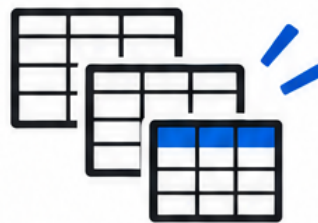
ONE FACT TABLE, ONE GRAIN



START WITH ATOMIC DETAIL



SEPARATE DIFFERENT GRAINS



Ask of every column:
is it true for this
exact row?



WORKED EXAMPLE: THE DOUBLED RECEIPT



PROBLEM



Receipt R104

Tea	4.00
Mug	8.00

True receipt total: 12.00

BAD MIXED GRAIN



Receipt Total 12.00 is repeated on both line rows.

Tea	12.00
Mug	12.00

↓

SUM = 24.00 WRONG

ACTION



Keep line amounts:
4.00 + 8.00

Tea	4.00
Mug	8.00

CHECK



Two rows.

↓

SUM = 12.00

No repeated receipt-level measure.

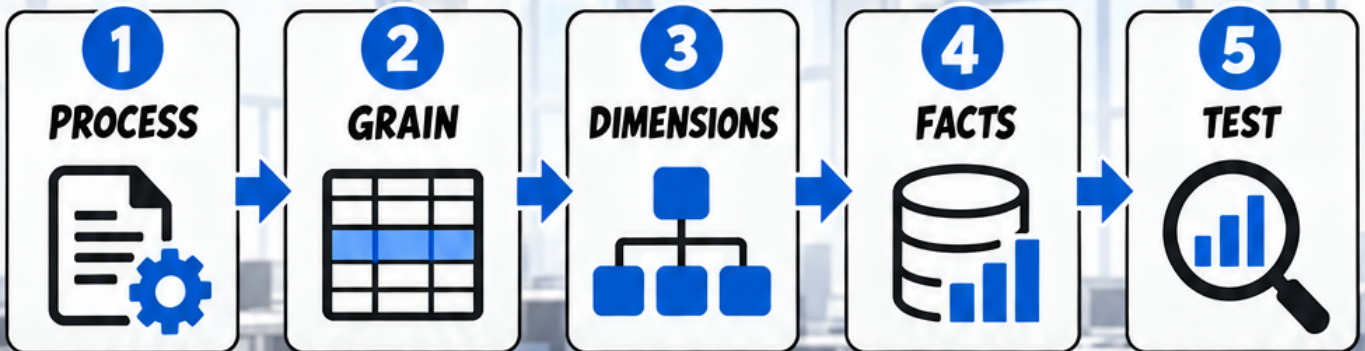
Store measures at their natural grain; calculate totals when needed.

YOUR GRAIN-FIRST CHECKLIST



- ☒ Write: One row represents...
- ☒ Confirm the source records that event.
- ☒ Choose dimensions that describe that exact row.
- ☒ Keep facts measured at the same level.
- ☒ Separate every different grain.
- ☒ Test row counts and sums with a tiny example.

RECAP



FINAL TIP FROM ELLY

If you cannot finish
‘one row represents...’,
do not design
the columns yet.



ELLY EXPLAINS: WHAT IS PYTHON?

PAGE 1
15 SEPTEMBER 2026



WHY CARE?



YOU WRITE CODE



PYTHON RUNS IT

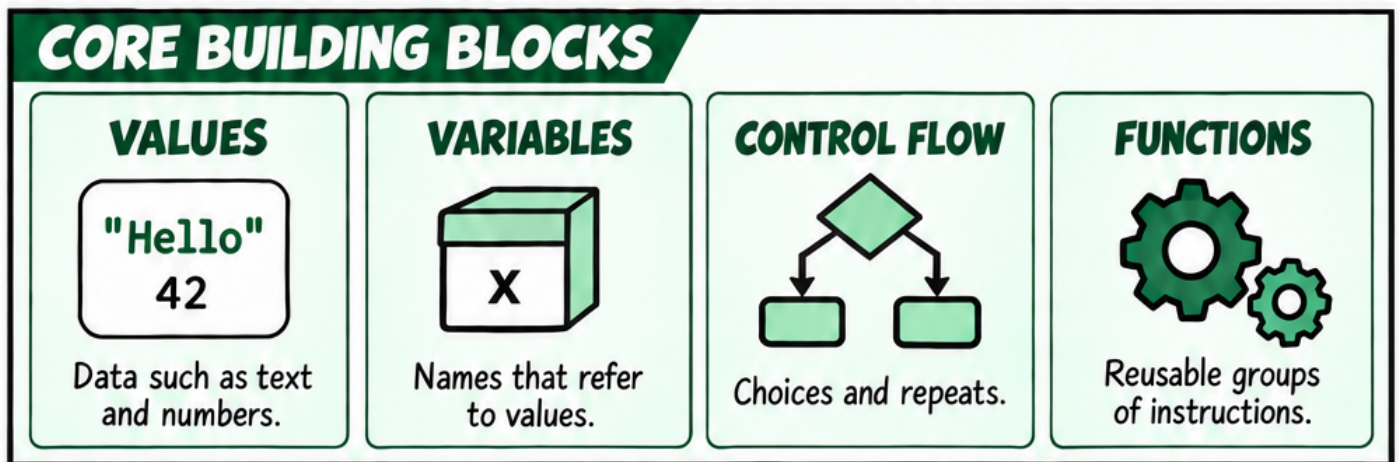


THE COMPUTER ACTS





ELLY EXPLAINS: FROM CODE TO RESULT



Python is the language.
A Python interpreter is the program that runs Python code.

15 SEPTEMBER 2026

ELLY EXPLAINS: A TINY PYTHON PROGRAM

GOAL:
GREET EACH
NAME

PYTHON CODE

```
1 names = ["Mina", "Kai"]
2 for name in names:
3     print("Hello", name)
```

OUTPUT

```
Hello Mina
Hello Kai
```

LIST

`names` holds
two text
values.

```
names = ["Mina", "Kai"]
```

LOOP

`for` visits
each value
once.

```
for name in names:
```

FUNCTION

`print` shows
each greeting.

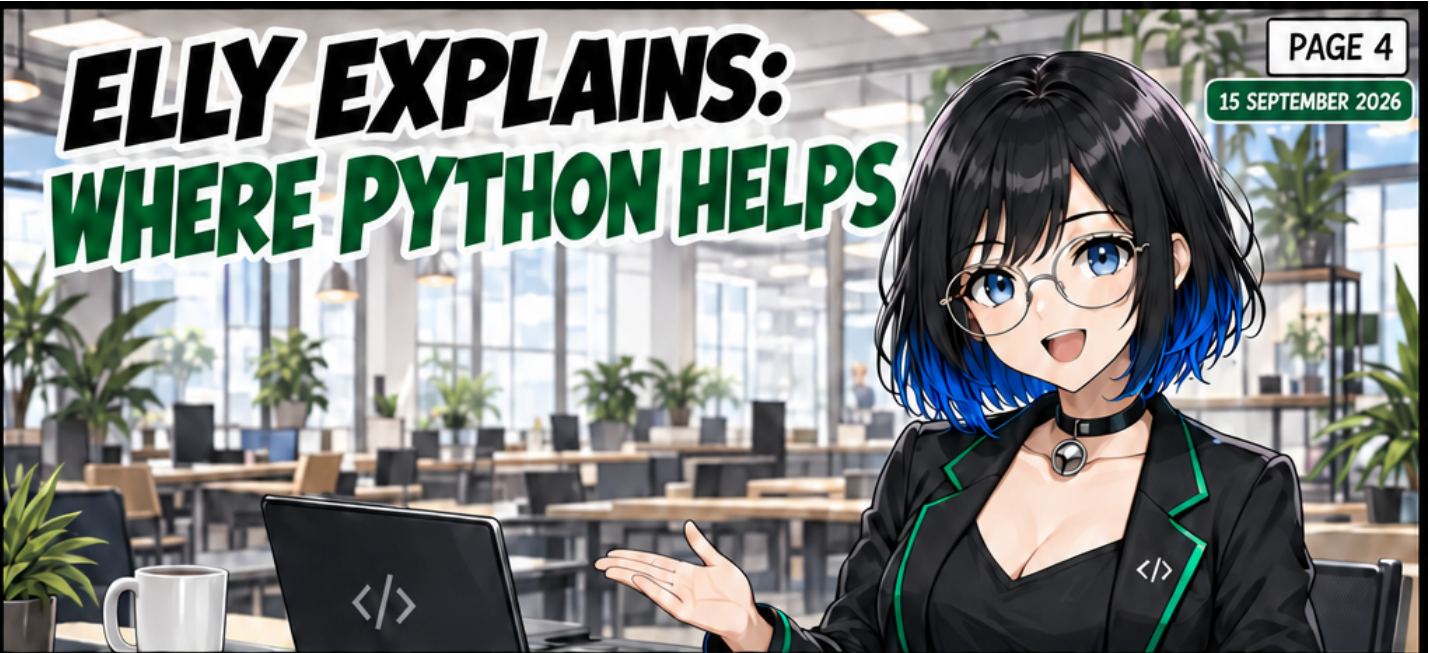
```
print("Hello", name)
```

Indentation groups
the line inside the loop.

NEXT: WHERE PYTHON HELPS →



ELLY EXPLAINS: WHERE PYTHON HELPS



AUTOMATION



Repeat useful
computer tasks.



DATA



Explore, clean
and analyse
information.



WEB



Build services
and back-end logic.



SCIENCE & AI



Use specialist
libraries and tools.



MYTH: PYTHON IS ONLY FOR BEGINNERS.



REALITY:

It is approachable,
but also used for
substantial real-world
systems.

MYTH: PYTHON IS AN APP.



REALITY:

Python is a language;
interpreters and tools
let you write and
run it.

NEXT: RECAP AND CHECK →

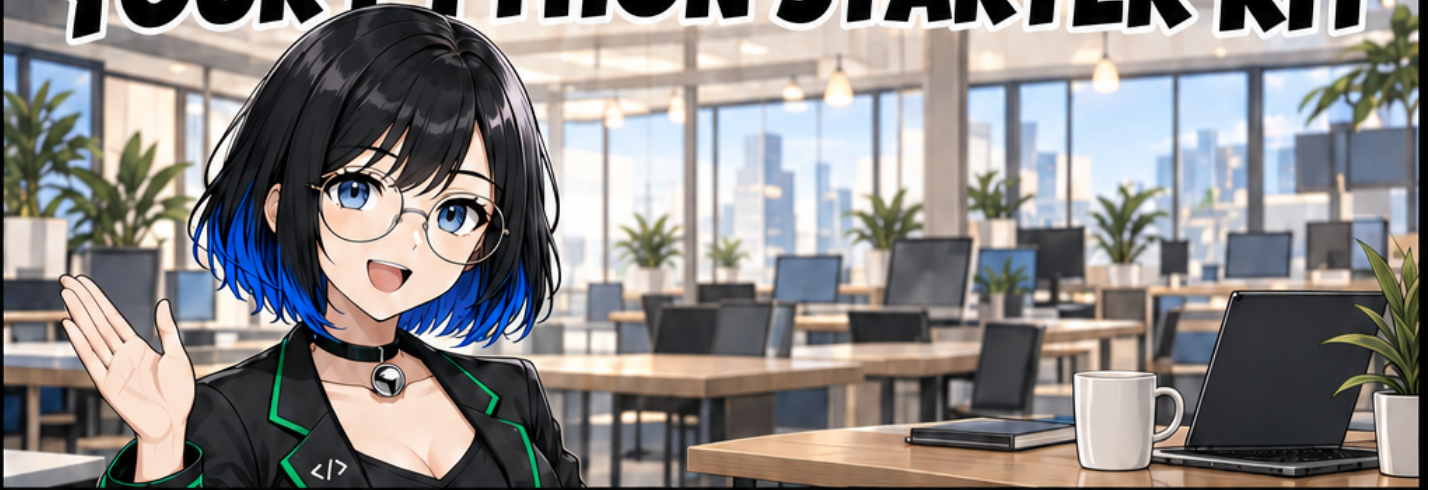


ELLY EXPLAINS:

PAGE 5

15 SEPTEMBER 2026

YOUR PYTHON STARTER KIT



RECAP

1 WRITE



Express instructions as code.

2 RUN



An interpreter executes the code.

3 BUILD



Combine data, control flow and functions.

TINY GLOSSARY



Variable: a name that refers to a value.



Loop: a way to repeat instructions.



Function: a reusable group of instructions.



QUICK CHECK



What happens inside the loop on Page 3?

A Each name is greeted once.

B Only the first name is greeted.

ANSWER:



A — the loop visits both names.

NEXT: KEEP EXPLORING →

SQL ANTI-JOINS

Find rows in one table that have no matching row in another.

An anti-join answers a practical question:
what is missing?

Customers with no orders



Devices with no heartbeat



Claims with no payment



Start with the
**population you want
to keep**, then test for
absence.



ASK: DOES A MATCH EXIST?

1 For each customer *c*:

customers <i>c</i>	
	
	
	

2 Look in orders *o*.

orders <i>o</i>	
	
	
	

3 Match
o.customer_id =
c.customer_id.

orders <i>o</i>	customers <i>c</i>
	
	
	



customers <i>c</i>			orders <i>o</i>
			
			
			

3. **NOT EXISTS** keeps *c* only when that search returns no row.

customers <i>c</i>	
	

Keep this left row when no matching right row exists.

THE SAFE PATTERN

PAGE 3



```
SELECT c.customer_id, c.name
FROM customers AS c
WHERE NOT EXISTS (
  SELECT 1
  FROM orders AS o
  WHERE o.customer_id = c.customer_id
);
```

1

Correlate the exact
business key.



2

SELECT 1 means
only existence matters.



3

Put right-table
match conditions
inside the subquery.



4

Prefer this clear
absence test when
NULL values may exist.



WARNING: NOT IN can return
no rows when its subquery
contains NULL.



WORKED EXAMPLE

1. PROBLEM

Find customers with *no orders*.

customers		orders	
customer_id	name	customer_id	
101	Ava	101	
102	Ben	103	
103	Cy	NULL	

2. ACTION

```
SELECT c.customer_id, c.name
FROM customers c
WHERE NOT EXISTS (
  SELECT 1
  FROM orders o
  WHERE o.customer_id = c.customer_id
);
```

Run the *NOT EXISTS* query.

3. CHECK

customer_id	name
102	Ben

101 and 103 have matches.
NULL matches nobody.
Only Ben remains.

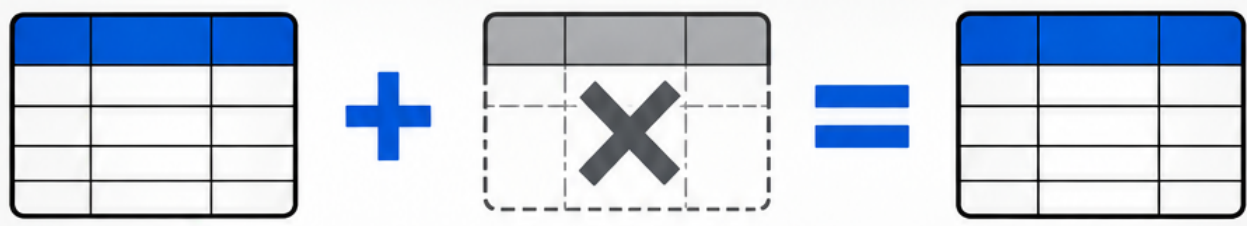
4. IMPROVEMENT

On large tables, inspect the query plan and index the match key *only when evidence supports it*.

ANTI-JOIN CHECKLIST



- 1**  Name the **left-side** population.
- 2**  Write the **exact** match **rule**.
- 3**  Use **NOT EXISTS** for the absence test.
- 4**  Test zero, one, many, and **NULL** matches.
- 5**  Check row counts and inspect the **plan**.



Left rows + no right-side match = **anti-join result**.

FINAL TIP FROM ELLY

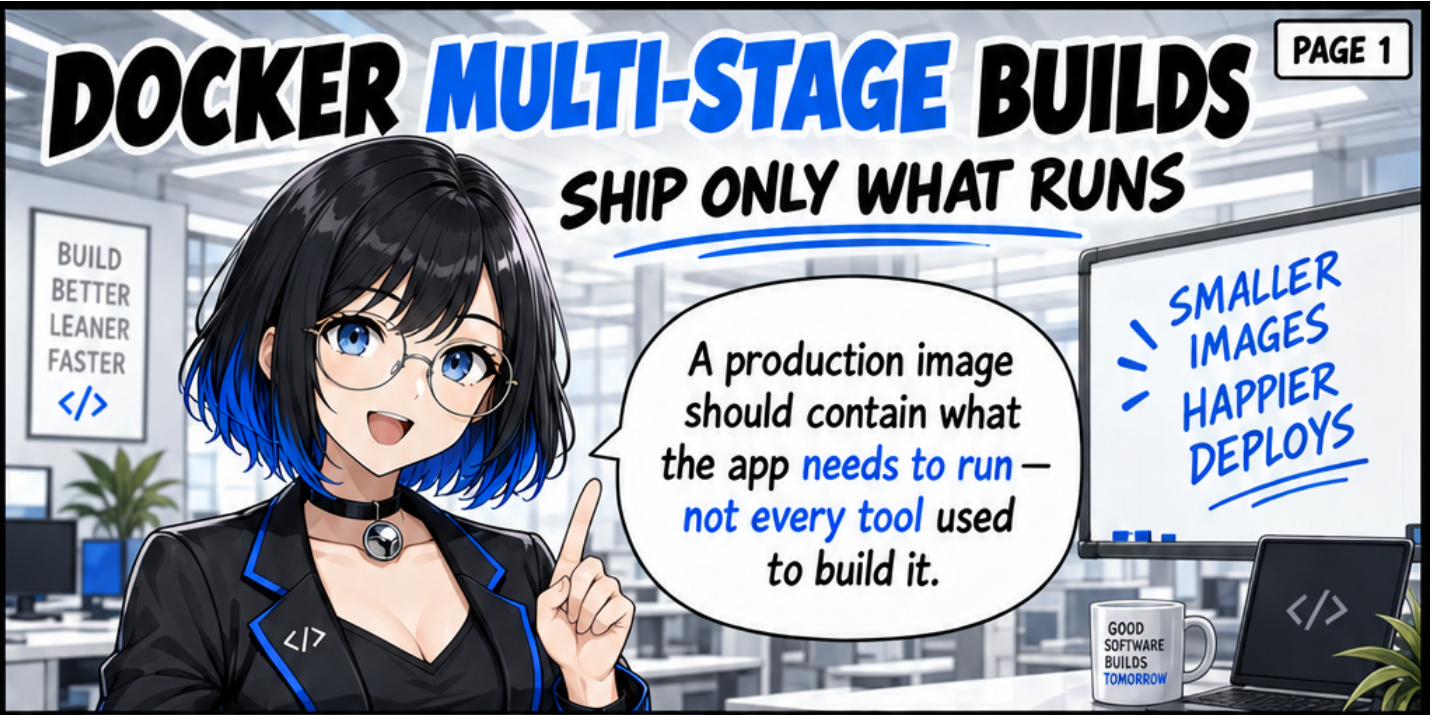
An anti-join asks about **absence**—not inequality.



DOCKER MULTI-STAGE BUILDS

SHIP ONLY WHAT RUNS

PAGE 1



TWO STAGES, DIFFERENT PURPOSES

BUILD STAGE



Build tools stay behind.

RUNTIME STAGE



Runtime gets only required files.

KEY IDEA



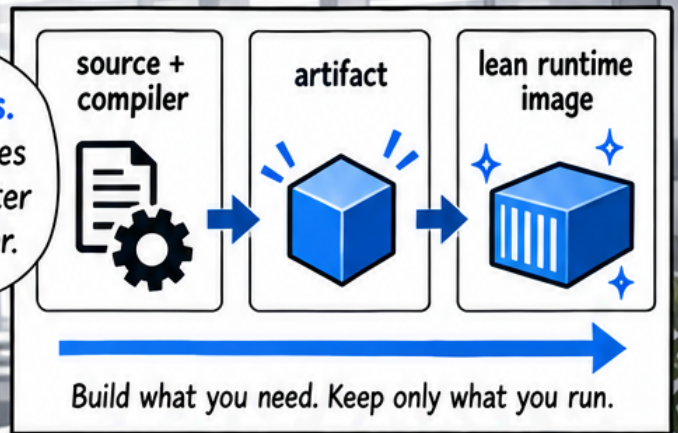
Several **FROM** stages. Copy only the **finished artifact** forward.



Leave the build tools behind. Deliver a clean, minimal image.



THE MENTAL MODEL



1. A **FROM** starts a new build stage.



A **FROM** instruction starts a new stage with its own base image.

2. The build stage may **compile**, **bundle**, or **test**.



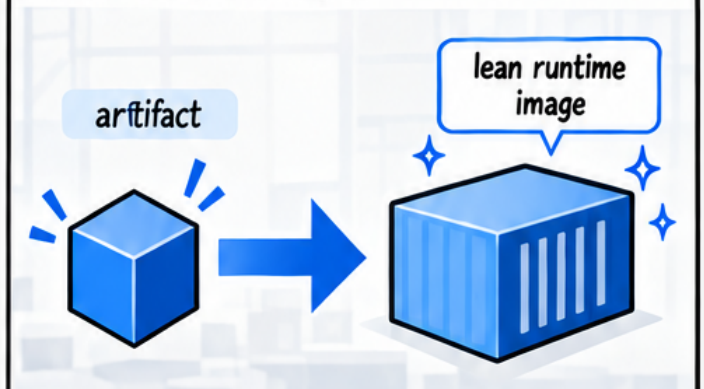
In the build stage, you can compile code, bundle assets, run tests, and create the output your application needs.

3. The runtime stage **starts clean**.



The runtime stage starts from a fresh base image and doesn't include the build tools, dependencies, or files from the build stage.

4. **COPY --from** transfers only chosen output.



COPY --from lets you copy only the specific files or folders you need from a previous stage into the runtime image, keeping the final image small and clean.



Same code. Smaller images. **Bigger possibilities.**

NEXT: Real-world example! →

WHAT CROSSES THE BOUNDARY?

A multi-stage build creates a boundary between the build stage and the final runtime image. We *only copy what the application needs to run*.

BUILDER (STAGE 1)

COPY WHAT RUNS

RUNTIME (FINAL IMAGE)

CARRY FORWARD

These go into the final runtime image.

- built application
- production dependencies
- needed runtime assets

LEAVE BEHIND

These stay in the build stage and do not end up in the runtime image.

- compilers
- package-manager caches
- source tests
- temporary files
- debugging tools

KEY GUARDRAIL

Small is useful only if the app still has every *runtime library* and *certificate* it requires.

Lean is *deliberate*, never guesswork.

SMALLER IMAGES STRONGER DEPLOYMENTS

WORKED EXAMPLE: FIX A BLOATED WEB SERVICE



1) PROBLEM

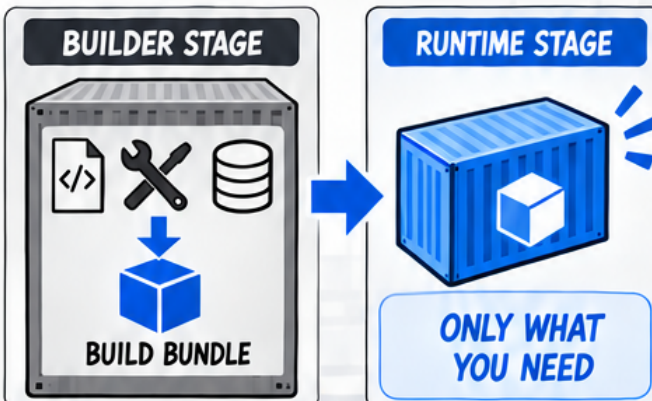
One image contains **source**, **test tools**, and **build cache**.



- Larger size
- Longer build times
- More to maintain
- Higher risk

2) ACTION

Build the web bundle in a **builder stage**. Copy the bundle into a separate **runtime stage**.



3) CHECK

Run the same health check. Inspect runtime contents. Compare size and vulnerability scan results.



Health check passes



Inspect runtime contents



Compare size and scan results



4) IMPROVE

Add a **debug target** for investigation instead of shipping debug tools in production.



SHIP A LEANER RUNTIME



Recap:
Build in one stage.
Run from another.
Transfer only the intended output.



LEAN IMAGES
BRIGHTER
FUTURES

SMALLER
CONTAINERS
HAPPIER
DEPLOYMENTS

SMALLER
LEANER
SAFER
FASTER

CHECKLIST

Follow these 5 practices for a leaner, safer runtime.

- 1** Name stages


- 2** Copy only intended output


- 3** Use a trusted minimal base


- 4** Exclude irrelevant build context


- 5** Rebuild regularly and test the final runtime stage





Optimise the **image** you **deploy**, not merely the stage that compiles.

A LEANER RUNTIME MEANS...

-  Smaller images
-  Lower risk
-  Faster deployments
-  A cleaner, simpler production environment

REAL IMPACT ☆

 **Small, clear, tested — ready to ship.** ———→