

ELLYPUNK

Applied

WEEKLY DIGEST / T201

17-23 August 2026



This week

17-23 August 2026

Available archived editions, in their original date order.

TUESDAY / 18 August 2026

AGENTS.md

3

[Open original online](#)

WEDNESDAY / 19 August 2026

SQL Window Functions

8

[Open original online](#)

THURSDAY / 20 August 2026

Idempotent Data Pipelines

13

[Open original online](#)

FRIDAY / 21 August 2026

SCD Type 2: Keep the History

18

[Open original online](#)

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

Copyright Ellypunk. Contains AI-generated artwork.

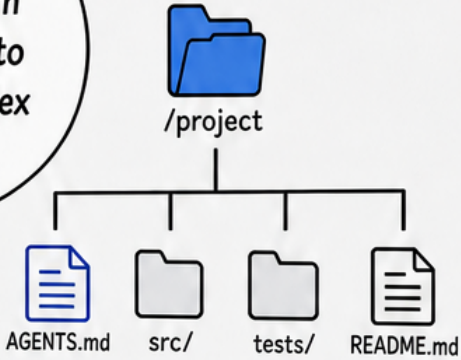
ellypunk.com

AGENTS.md

TEACH CODEX THE RULES



Hi! I'm **Elly**.
Today we'll turn
team habits into
instructions Codex
can reuse.



WITHOUT IT



Repeat setup



Miss conventions

WITH IT



Consistent commands



Safer changes

TODAY'S OBJECTIVE



Give every task the right **project context** before work begins.



Persistent **guidance** beats repeated reminders.

THE INSTRUCTION CHAIN



1

GLOBAL DEFAULTS

~/.codex/AGENTS.md

Reusable preferences
for every project.



2

PROJECT RULES

<repo>/AGENTS.md

Build, test, style,
and safety rules.



3

NESTED OVERRIDE

services/payments/
AGENTS.override.md

Special rules for one area.



Codex reads from
broad to specific.

CLOSER FILES WIN.



WRITE RULES CODEX CAN EXECUTE

Make every rule
clear enough to **act** on
and **check**.

1 CONCRETE COMMAND



Run **npm test** after
JavaScript changes.

2 BOUNDARY



Never edit
generated files.

3 TRIGGER



Before adding
a dependency, **ask**.

4 SCOPE



In **services/payments**,
use **make test-payments**.

GUARDRAILS



Keep rules
concise.

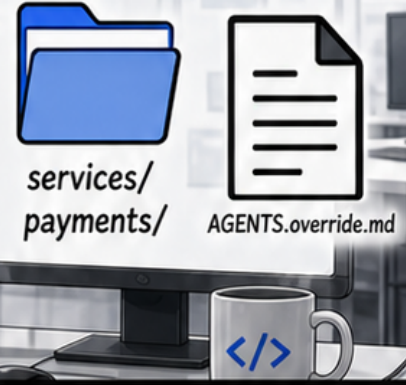
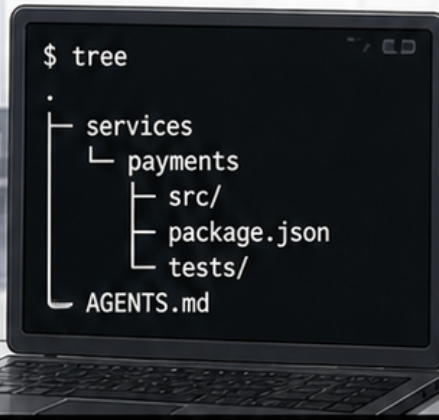


Put each rule in
the **closest**
relevant folder.



Use **CI** for
routine
formatting.

WORKED EXAMPLE: THE WRONG TEST



1 PROBLEM

Codex runs `npm test` in `payments`.
The service needs its *own suite*.



2 ACTION

`services/payments/AGENTS.override.md`

Payments

- After edits, run `make test-payments`.
- Never rotate keys without approval.



3 CHECK

`codex --cd services/payments`
Show active instructions.

```
$ codex --cd services/payments
```

Active instructions (high → low):

- 1 `services/payments/AGENTS.override.md`
- 2 `AGENTS.md` (repo root)
- 3 `/Users/elly/.codex/AGENTS.md`

Global → repo → payments override



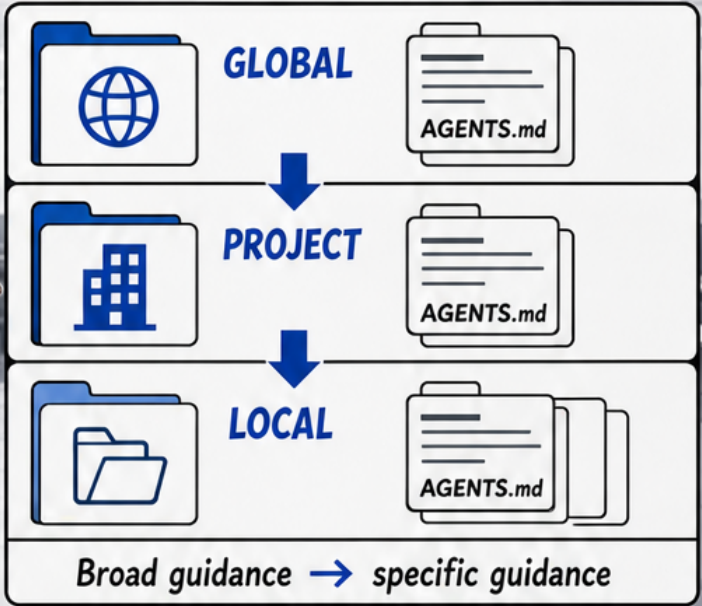
4 IMPROVEMENT

- ✓ Specific command.
- ✓ Clear scope.
- ✓ Safety boundary.



RIGHT RULE. RIGHT PLACE. VERIFIED.

RULES THAT TRAVEL WITH THE WORK



PRACTICAL CHECKLIST

- 1 ☒ Put reusable defaults in **global** AGENTS.md.
- 2 ☒ Put repository rules at the **project** root.
- 3 ☒ Add nested **overrides** only where needed.
- 4 ☒ Name exact **build** and **test** commands.
- 5 ☒ Start a **new run** after instruction changes.

TROUBLESHOOT



If Codex surprises you, **inspect** which instruction file won.

FINAL TIP FROM ELLY

Write rules like a teammate must act on them: **specific, scoped, testable.**



CONSISTENT CONTEXT. SAFER WORK.

WINDOW FUNCTIONS 101

PAGE 1



Hi! I'm **Elly**.
Window functions
add **context** while
keeping **every row**.

SALES

Date	Region	Product	Amount
2025-04-01	East	Laptop	1200
2025-04-01	East	Mouse	25
2025-04-02	East	Keyboard	45
2025-04-01	West	Monitor	300
2025-04-02	West	Laptop	1100
2025-04-02	West	Mouse	25

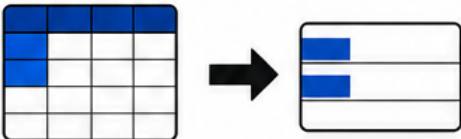


OBJECTIVE:

Compare rows without collapsing them.

GROUP BY

ONE ROW PER GROUP



Example Result

Region	Total Amount
East	1270
West	1425

Rows collapse into one
summary row per group.

VS.

WINDOW FUNCTION

EVERY ROW + CONTEXT



Example Result

Date	Region	Product	Amount	Running Total (Region)
2025-04-01	East	Laptop	1200	1200
2025-04-01	East	Mouse	25	1225
2025-04-02	East	Keyboard	45	1270
2025-04-01	West	Monitor	300	300
2025-04-02	West	Laptop	1100	1400
2025-04-02	West	Mouse	25	1425

Every row stays.
Extra insight is added.

Keep the detail.
Add the insight.



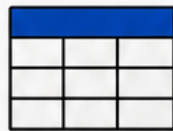
THE OVER CLAUSE



Build a window in **five** clear steps.

```
SELECT rep, sale_date, amount,
       SUM(amount) OVER (
         PARTITION BY rep
         ORDER BY sale_date
       ) AS running_total
FROM sales;
```

1. Choose the rows



2. Pick a function



3. PARTITION BY group



4. ORDER BY sequence



5. Name the result



sales

rep	sale_date	amount
Alice	2025-01-01	120
Alice	2025-01-03	80
Alice	2025-01-07	150
Bob	2025-01-02	200
Bob	2025-01-04	100
Bob	2025-01-06	50

with running_total

rep	sale_date	amount	running_total
Alice	2025-01-01	120	120
Alice	2025-01-03	80	200
Alice	2025-01-07	150	350
Bob	2025-01-02	200	200
Bob	2025-01-04	100	300
Bob	2025-01-06	50	350



Each rep gets a **separate** running total.

RULES THAT PREVENT SURPRISES



PARTITION BY —
resets per group.



ORDER BY —
makes sequence
explicit.



ROWS frame —
defines which
rows count.



TIE-BREAKER —
makes equal
values deterministic.



CHOOSE YOUR RANKING FUNCTION



ROW_NUMBER: unique order



RANK: gaps after ties



DENSE_RANK: no gaps

RANKING WITH TIES: EXAMPLE

	score	ROW_NUMBER	RANK	DENSE_RANK
	98	1	1	1
	95	2	2	2
	95	3	2	2
	90	4	4	3
	90	5	4	3
	88	6	6	4

} Tied scores

} Tied scores

RANK skips numbers after ties.

DENSE_RANK does not skip.



RULE TO REMEMBER

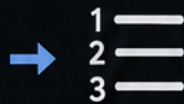
Filter window results
in an outer query.



Test one
small
partition
first.



PARTITION BY



ORDER BY



ROWS frame



TIE-BREAKER



RANKING



FILTER OUTER
QUERY

Follow the
rules.
Avoid
surprises.





WORKED EXAMPLE: LATEST STATUS

Problem.
Action.
Check.
Improve.

PROBLEM

Keep only the latest status for each order.

order_id	event_time	status	event_id
101	09:00	PACKING	3
101	10:30	SHIPPED	4
202	08:45	PACKING	8

ACTION

```
WITH ranked AS (
  SELECT *,
    ROW_NUMBER() OVER (
      PARTITION BY order_id
      ORDER BY event_time DESC,
        event_id DESC
    ) AS rn
  FROM order_events
)
SELECT order_id, status
FROM ranked
WHERE rn = 1;
```

CHECK



101 → SHIPPED



202 → PACKING

IMPROVE

event_id breaks
timestamp ties.





WINDOW FUNCTION CHECKLIST



- ☒ Keep row detail 
- ☒ Choose the partition 
- ☒ Make order explicit 
- ☒ Set the frame when needed 
- ☒ Add a tie-breaker 
- ☒ Test edge cases 

RECAP FORMULA

fx

FUNCTION

+

OVER

OVER

+



PARTITION

+



ORDER

Use **windows** when you need **detail** and **comparison** together.

ELLY'S FINAL TIP

If the answer depends on **order**, make that **order explicit**.





OBJECTIVE



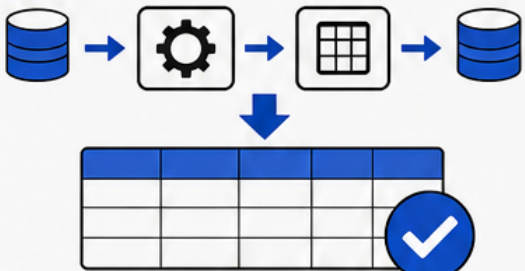
Build a data step that gives the same final state when rerun.



Retries happen. A safe pipeline can run again without duplicating or corrupting data.

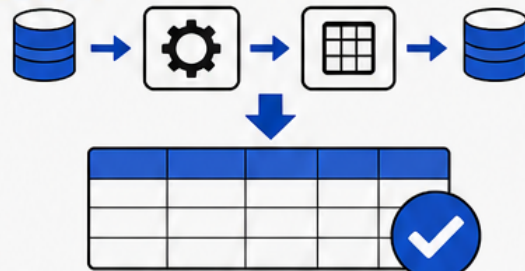


RUN 1



CORRECT TABLE

RUN 2



SAME CORRECT TABLE

KEY IDEA



Same input + same rules
=
same final state.



THE IDEMPOTENT PATTERN

PAGE 2



1 CHOOSE A STABLE KEY



Example: `order_id`

2 READ THE TARGET STATE



What is already loaded?

3 UPSERT, DON'T BLINDLY APPEND



Insert new rows.
Update changed rows.

4 RECORD THE CHECKPOINT



Use a `batch_id`,
source date, or watermark.

5 VERIFY



Check row counts,
duplicate keys, and
rejected rows.

SOURCE

STAGE

DEDUPE

UPSERT

CHECK



Design every retry as
a **normal, expected** event.



FOUR GUARDRAILS FOR SAFE RETRIES

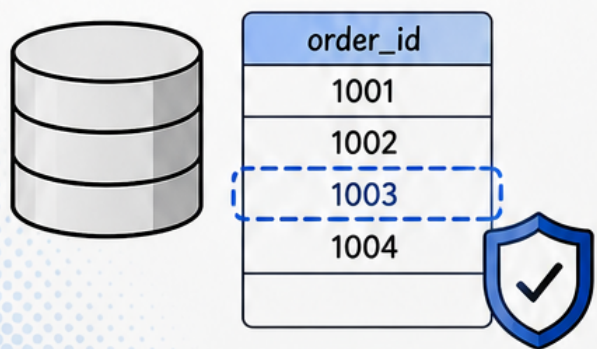


Use all four.



1 UNIQUE KEY

Enforce one row per business key.



2 TRANSACTION

Commit the full unit or roll it back.



3 CHECKPOINT

Advance only after success.



4 AUDIT LOG

Record batch_id, status, row counts, and errors.

batch_id	2025-04-20_01
status	SUCCESS
rows_inserted	12,532
rows_rejected	87
errors	3



RULES

- ✓ Never update the checkpoint before commit.
- ✓ Never hide rejected rows.
- ✓ Never trust row counts alone: check keys.

WORKED EXAMPLE: **RETRY A DAILY LOAD**

PAGE 4

PROBLEM



Source: **B42 (3 rows)**

order_id	customer	amount
1001	Alice	120.00
1002	Bob	75.00
1003	Cara	42.00



First run wrote **2 rows**

order_id	customer	amount
1001	Alice	120.00
1002	Bob	75.00



ACTION



</>

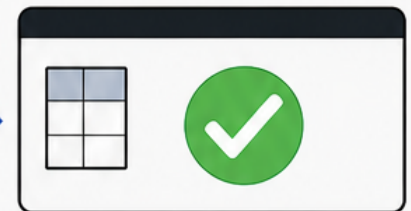
```
MERGE INTO fact_orders t
USING stage_orders s
ON t.order_id = s.order_id
WHEN MATCHED THEN UPDATE
WHEN NOT MATCHED THEN INSERT;
```

CHECK



</>

```
SELECT order_id, COUNT(*)
FROM fact_orders
GROUP BY order_id
HAVING COUNT(*) > 1;
```



RESULT: 0 DUPLICATE KEYS

IMPROVEMENT



After retry: **fact_orders (3 rows)**

order_id	customer	amount
1001	Alice	120.00
1002	Bob	75.00
1003	Cara	42.00



RUN IT TWICE, GET ONE RESULT



RECAP



Retry → same final table



Partial failure → safe recovery



Repeatability → calmer operations

PRACTICAL CHECKLIST

- ☒ Stable business key
- ☒ Stage before writing
- ☒ Dedupe the batch
- ☒ Upsert inside a transaction
- ☒ Advance checkpoint after commit
- ☒ Check duplicate keys and row counts
- ☒ Log batch_id and errors



FINAL TIP FROM ELLY

Make retries boring:
**keys, transactions,
checkpoints, checks.**



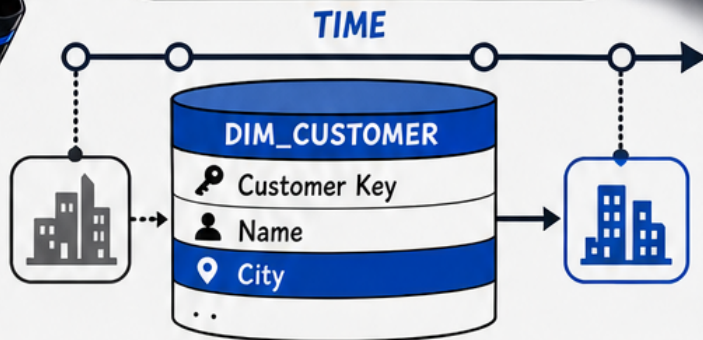
SCD TYPE 2: KEEP THE HISTORY



OBJECTIVE

Record meaningful changes without losing the old values.

SCD means **S**lowly **C**hanging **D**imension.



TYPE 1 OVERWRITES



City: Leeds → York



Old value lost.



DIM_CUSTOMER	
Customer Key	101
Name	Alice
City	York
...	

TYPE 2 ADDS HISTORY



Leeds | past



York | current



DIM_CUSTOMER				
Customer Key	City	Valid From	Valid To	Is Current
101	Leeds	2023-01-01	2023-06-30	N
101	York	2023-07-01	9999-12-31	Y



Ask: what was true on any date?



Keep the past. Mark the present.



THE TYPE 2 CHANGE FLOW



1 DETECT

Compare the incoming row with the current row.

business key

	101
—	
—	
...	

current row

id	name	email	valid_from	valid_to	is_current
5001	Alice	alice@x.com	2024-01-01	9999-12-31	1
...					

changed?



2 CLOSE

Set the old row's valid_to to the change time.

current row

id	name	email	valid_from	valid_to	is_current
5001	Alice	alice@x.com	2024-01-01	9999-12-31	1
...					

close old



id	name	email	valid_from	valid_to	is_current
5001	Alice	alice@x.com	2024-01-01	2025-05-10	0
...					



3 INSERT

Add a new row with a new surrogate key.

insert new



id	name	email	valid_from	valid_to	is_current
5002	Alice	alice_new@x.com	2025-05-10	9999-12-31	1
...					



4 MARK CURRENT

Exactly one row has is_current = 1.

id	name	email	valid_from	valid_to	is_current
5001	Alice	alice@x.com	2024-01-01	2025-05-10	0
5002	Alice	alice_new@x.com	2025-05-10	9999-12-31	1
...					



No change? Do nothing.



COLUMNS AND GUARDRAILS



In SCD Type 2, these columns and guardrails keep your history **accurate, consistent, and trustworthy.**



BUSINESS KEY Stable source ID: <code>customer_id</code>	SURROGATE KEY Warehouse row ID: <code>customer_sk</code>	VALIDITY Use <code>[valid_from, valid_to)</code>: start inclusive, end exclusive.
CURRENT FLAG One <code>is_current = 1</code> per business key.	CHANGE CHECK Compare only tracked business columns.	ATOMIC WRITE Close old + insert new in one transaction.
Reject overlaps.	Ignore unchanged input.	Audit every load.

Good history has no **gaps, overlaps, or double-current** rows.

WORKED EXAMPLE: LEEDS TO YORK

PAGE 4

1 PROBLEM

Customer 42 was in Leeds. Today's feed says York.



customer_sk	customer_id	city	valid_from	valid_to	current
700	42	Leeds	2026-01-01	2026-08-21	0
814	42	York	2026-08-21	9999-12-31	1

2 ACTION

Close the old row and insert the new row in **one transaction**.

Close old row
(set valid_to and
current = 0)



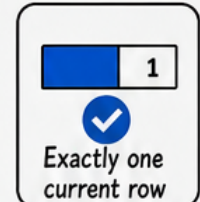
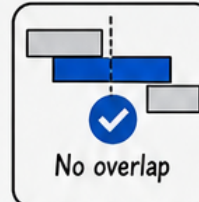
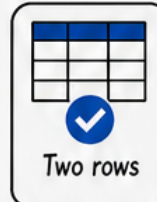
Insert new row
(new valid_from,
current = 1)



customer_sk	customer_id	city	valid_from	valid_to	current
700	42	Leeds	2026-01-01	2026-08-21	0
814	42	York	2026-08-21	9999-12-31	1

3 CHECK

Two rows. **No overlap**.
Exactly one current row.



customer_sk	customer_id	city	valid_from	valid_to	current
700	42	Leeds	2026-01-01	2026-08-21	0
814	42	York	2026-08-21	9999-12-31	1

4 IMPROVEMENT

Add a **unique-current** rule,
then rerun the same feed.



UNIQUE-CURRENT RULE:
For each **customer_id**,
at most one row can have
current = 1.



RERUN:
0 NEW ROWS

The rule prevents duplicates.
Data stays clean.

customer_sk	customer_id	city	valid_from	valid_to	current
700	42	Leeds	2026-01-01	2026-08-21	0
814	42	York	2026-08-21	9999-12-31	1



Problem



Action



Check



Improve

YOUR TYPE 2 CHECKLIST

PAGE 5



RECAP



Detect



Close



Insert



Verify



Track only meaningful columns.



Use stable business keys.



Use half-open validity dates.



Write close + insert atomically.



Prove one current row; rerun unchanged input.



PAST

2024-01-01

Original row inserted



CHANGE DETECTED

2024-04-10

Value changed



CLOSE OLD ROW

2024-04-10

Set end date



INSERT NEW ROW

2024-04-10

Insert new version



CURRENT ROW

2025-05-10

Still active



History preserved. One current row at any time.



History should grow
only when the
business meaning changes.



PAST PRESERVED. PRESENT CLEAR. ✨



Elly's tip: test change, no-change, and late-arriving cases.

