DATA ACCESS CONCIERGE · GUIDE

Show an auditor who approved each person's data access, and why

A script that matches every data access to a request, an approver, a reason and an end date, and names each gap.

Published 2026-11-04 · Online version and downloads: www.datameaning.com/guides/prove-who-approved-each-data-access/

A Python script reads five files: exports from your data platform and your Microsoft Entra tenant, and a role list you keep. For each access, it writes who asked and why, who approved and why, and when it ends. Where that record is missing, it names the gap. A person still reviews the result. The demo data is generated.

What you will need

- Python 3 and the duckdb module (we used DuckDB 1.5.6).
- The script, who_approved_each_grant.py.
- Five CSV files (step 1). Without them, run the script with --demo to see it work on generated records; --demo also needs to download Microsoft's Graph metadata file once (8,200,681 bytes).

Step 1. Put the five lists in five CSV files

Microsoft Entra entitlement management keeps a request for each access package assignment. The request carries the requester's justification (accessPackageAssignmentRequest (read 2026-11-04)). The approval is a separate record whose stages carry the reviewer, the result (NotReviewed, Approved or Denied), the time and the reviewer's justification (approvalStage (read 2026-11-04)). The assignment carries a schedule whose expiration holds an end date or says there is none (expirationPattern (read 2026-11-04)). Stages are listed per request by List approval stages (read

2026-11-04), and a single request is read with Get accessPackageAssignmentRequest (read 2026-11-04).

The script takes five files. Nested Graph properties are flattened by joining the names with an underscore, so requestor.objectId becomes requestor_objectId and assignment.target.principalName becomes assignment_target_principalName.

File	Columns the script reads	Where they come from
grants.csv	grantee_name, role, created_on	Your platform's current grants. Snowflake's GRANTS_TO_USERS view has these three columns (read 2026-11-04). If your platform exports these three, map them in COLUMN_MAP. The script expects the grantee to be the sign-in name that Entra calls principalName.
role_packages.csv	role, access_package_id	A list you keep: which Entra access package gives which role.
requests.csv	id, requestType, state, createdDateTime, completedDateTime, justification, requestor_objectId, assignment_id, assignment_target_principalName, accessPackage_id	accessPackageAssignmentRequest (read 2026-11-04), with its assignment expanded (it supports $expand): the assignment the request creates or, for a removal, the one it ends. The requester can be someone else: for an admin's request it is the admin.
approvals.csv	request_id, reviewedBy_id, reviewedBy_displayName, reviewResult, reviewedDateTime, justification	approvalStage (read 2026-11-04), one row per stage.
assignments.csv	id, accessPackage_id, target_principalName, state, schedule_startDateTime, schedule_expiration_type, schedule_expiration_endDateTime	accessPackageAssignment (read 2026-11-04)

With --demo the script downloads Microsoft's published Graph metadata at a pinned commit (microsoftgraph/msgraph-metadata at commit c4b98b7b), stops if the file's size or SHA-256 differs from the pinned one, and stops if any of the 22 columns it takes from Graph is not a property in it.

Step 2. Let the input checks run first

The script runs eight checks on the files before it looks at a single grant: the date, that every file and column is there, that every date and time is readable, that every grant names a grantee and a role, that every request id appears once, that every request state, request type, result, expiration type and assignment state is one Microsoft documents, that every role in the role map holds a grant, and

that every role maps to one access package. If any fails, the run says INCOMPLETE and computes nothing, because a report built on files it could not read would look clean for the wrong reason.

Step 3. Match each grant to its request, its approvals and its end date

For each grant the script finds the delivered request that assigned that person the access package the role maps to, whether the person asked or an admin assigned it, preferring the one completed just before the grant. Then it counts that request's approvals by a named reviewer, how many of them are by the requester, how many have no reason, and when the last one was recorded. This is the matching query, copied from the script:

```
WITH cand AS (
  SELECT g.grant_no, r.id AS request_id, r.oid, r.created, r.justification AS req_reason,
         row_number() OVER (PARTITION BY g.grant_no ORDER BY coalesce(r.completed > g.granted_at,
TRUE),
                            abs(epoch(r.completed) - epoch(g.granted_at)), r.id) AS rk
  FROM g JOIN pkg p ON p.role = g.role JOIN req r ON r.principal = g.grantee AND r.package =
p.package)
SELECT c.grant_no, c.request_id, c.oid, c.created, c.req_reason,
       (SELECT count(*) FROM apr a WHERE a.request_id = c.request_id AND a.reviewer IS NOT NULL) AS
n_appr,
       (SELECT count(*) FROM apr a WHERE a.request_id = c.request_id AND a.reviewer = c.oid) AS
n_self,
       (SELECT count(*) FROM apr a WHERE a.request_id = c.request_id AND a.reviewer IS NOT NULL
          AND a.justification IS NULL) AS n_blank,
       (SELECT max(a.at) FROM apr a WHERE a.request_id = c.request_id AND a.reviewer IS NOT NULL)
AS last_appr,
       (SELECT string_agg(DISTINCT coalesce(a.reviewer_name, a.reviewer), '; ') FROM apr a
          WHERE a.request_id = c.request_id AND a.reviewer IS NOT NULL) AS approvers,
       (SELECT string_agg(coalesce(a.justification, '(none)'), ' | ') FROM apr a
          WHERE a.request_id = c.request_id AND a.reviewer IS NOT NULL) AS approver_reasons
FROM cand c WHERE c.rk = 1
```

Each finding is one line of a second query. Here are its first lines and the last one (the full set is in the download):

```
SELECT grant_no, 'UNMAPPED_ROLE' AS code, 'role ' || role_shown || ' is not in role_packages' AS
detail
  FROM m WHERE NOT mapped
UNION ALL SELECT grant_no, 'NO_REQUEST', 'no delivered request for this person and package' FROM m
  WHERE mapped AND NOT has_req
UNION ALL SELECT grant_no, 'NO_APPROVAL', 'request ' || request_id || ' has no approval by a named
reviewer'
  FROM m WHERE has_req AND n_appr = 0
UNION ALL SELECT grant_no, 'SELF_APPROVAL', 'request ' || request_id || ' was approved by the
person who asked'
  FROM m WHERE has_req AND n_self > 0
  ...
UNION ALL SELECT grant_no, 'PAST_END_DATE', 'ended ' || strftime(ends_at, '%Y-%m-%d') || ', grant
still present'
  FROM m WHERE mapped AND ends_at < {asof}
```

An approval with a blank reviewer counts as no approval: an approval that does not say who gave it cannot answer the auditor's question. A grant whose role is not in the role map gets only UNMAPPED_ROLE, because without a package the script has no request to look for. Reason fields count as blank after trimming spaces.

Finding	What it says	What a person does with it
NO_REQUEST	No delivered request that assigned this person the access package.	Find out how the grant was made, then request it properly or remove it. This is the finding for grants that did not come through a delivered request.
NO_APPROVAL	The request has no approval stage with result Approved and a named reviewer.	Ask the owner to decide on the record. Entra can assign access without an approval: the request's parameters can bypass the approval requirement (accessPackageAssignmentRequest (read 2026-11-04)), so a request that skipped approval shows up here, and an assignment with no request shows as NO_REQUEST.
SELF_APPROVAL	A reviewer's id equals the requester's object id.	Have a second person review the access.
NO_REASON	The requester or an approver left the justification blank.	Ask for the reason now and file it with the request.
GRANT_BEFORE_APPROVAL	The grant was created before the last approval was recorded.	Check for a side-door grant that was approved afterwards.
NO_END_DATE	The person's most recent assignment for this package holds no end date, or there is none.	Set an end date, or record why this access has none.
REMOVED_IN_ENTRA	The person's most recent assignment for this package is not active in Entra: its state is something other than delivered (expired, for example), or a delivered removal request names it. The grant is still there.	Check the assignment in Entra. If it expired or was removed, remove the grant on the platform, because the access ended in Entra but not where the data is.
PAST_END_DATE	The end date is before AS_OF and the grant is still there.	Remove the grant or renew it through a new request.

Finding	What it says	What a person does with it
UNMAPPED_ROLE	The role is not in role_packages.csv.	Add the role to the map, or treat it as a grant with no request.

The code

The complete download, who_approved_each_grant.py, exactly as published at https://www.datameaning.com/guides/prove-who-approved-each-data-access/who_approved_each_grant.py:

```
#!/usr/bin/env python3
"""Match every current data-access grant to a request, a named approver, a reason and an end date,
in DuckDB.

    python3 who_approved_each_grant.py --demo                  # clean generated records
    python3 who_approved_each_grant.py --demo --plant-error    # generated records with 9 grants
that have a gap
    python3 who_approved_each_grant.py --demo --plant-input    # generated records with planted
input faults
    python3 who_approved_each_grant.py                         # your exports: set DATA_DIR and the
maps below first
    python3 who_approved_each_grant.py --strict                # exit 1 if any grant has a finding
    add  --out <folder>  to write every file under that folder (default: ./who-approved-out)

It reads five CSV files (see COLUMN_MAP) and, for each grant in grants.csv, writes the evidence an
auditor asks for
(who requested it, who approved it, the reason each gave, when the access ends) to
grant_evidence.csv, prints each
finding and prints three matched grants as examples. The findings:
  NO_REQUEST            no delivered request that assigned this person the access package
  NO_APPROVAL           the request has no approval by a named reviewer
  SELF_APPROVAL         the approver is the person who asked
  NO_REASON             the request or the approval carries no justification
  GRANT_BEFORE_APPROVAL the grant was created before the last approval was recorded
  NO_END_DATE           the person's most recent assignment for this package holds no end date, or
there is none
                        (an assignment that carries only a duration and no end date is reported
here too)
  REMOVED_IN_ENTRA      the person's assignment is not active in Entra (its state is not delivered,
or a delivered
                        removal request names it) and the grant is still there
  PAST_END_DATE         the end date has passed and the grant is still there
  UNMAPPED_ROLE         the role is not in role_packages.csv, so no request can be looked for
Input checks run first. If one fails, no findings are computed and the run says INCOMPLETE.
The script reads files; it never connects to a platform and never writes to one.

The demo records are GENERATED (seed 124): not a company's data. With --demo the script downloads
Microsoft's
published Graph metadata at a pinned commit and fails if a column it takes from Graph is not in it:
  https://github.com/microsoftgraph/msgraph-metadata/tree/c4b98b7b7808ee5f4e7d752378339d70ef472154
Column names come from:
  Graph   https://learn.microsoft.com/en-us/graph/api/resources/accesspackageassignmentrequest
```

```
          https://learn.microsoft.com/en-us/graph/api/resources/accesspackageassignment
          https://learn.microsoft.com/en-us/graph/api/resources/approvalstage
  Snowflake (grants.csv)  https://docs.snowflake.com/en/sql-reference/account-usage/grants_to_users
"""
import csv
import datetime as dt
import hashlib
import pathlib
import random
import re
import sys
import urllib.request
import xml.etree.ElementTree as ET

import duckdb

# ---- CHANGE THESE FOR YOUR OWN DATA -------------------------------------------------------
AS_OF = "2026-11-04"          # the day you took the exports (UTC, YYYY-MM-DD); an end date before
it has passed
DATA_DIR = None               # folder holding the five files named in FILE_NAMES
FILE_NAMES = {"grants": "grants.csv", "role_packages": "role_packages.csv", "requests":
"requests.csv",
              "approvals": "approvals.csv", "assignments": "assignments.csv"}
# For each file: the script's column name (left, do not change) = the column name in your file
(right).
# Names are matched without regard to case or surrounding spaces.
COLUMN_MAP = {
    "grants": {"grantee_name": "grantee_name", "role": "role", "created_on": "created_on"},
    "role_packages": {"role": "role", "access_package_id": "access_package_id"},
    "requests": {"id": "id", "requestType": "requestType", "state": "state", "createdDateTime":
"createdDateTime",
                 "completedDateTime": "completedDateTime", "justification": "justification",
                 "requestor_objectId": "requestor_objectId", "assignment_id": "assignment_id",
                 "assignment_target_principalName": "assignment_target_principalName",
                 "accessPackage_id": "accessPackage_id"},
    "approvals": {"request_id": "request_id", "reviewedBy_id": "reviewedBy_id",
                  "reviewedBy_displayName": "reviewedBy_displayName", "reviewResult":
"reviewResult",
                  "reviewedDateTime": "reviewedDateTime", "justification": "justification"},
    "assignments": {"id": "id", "accessPackage_id": "accessPackage_id", "target_principalName":
"target_principalName",
                    "state": "state", "schedule_startDateTime": "schedule_startDateTime",
                    "schedule_expiration_type": "schedule_expiration_type",
                    "schedule_expiration_endDateTime": "schedule_expiration_endDateTime"},
}
# -------------------------------------------------------------------------------------------

# Where each script column comes from. A Graph path is checked against Microsoft's metadata with --
demo.
# None = a key or list this script adds (a Snowflake column, or a join key), not a Graph property.
GRAPH_PATHS = {
    "requests": {"id": "accessPackageAssignmentRequest.id", "requestType":
"accessPackageAssignmentRequest.requestType",
                 "state": "accessPackageAssignmentRequest.state",
                 "createdDateTime": "accessPackageAssignmentRequest.createdDateTime",
                 "completedDateTime": "accessPackageAssignmentRequest.completedDateTime",
                 "justification": "accessPackageAssignmentRequest.justification",
                 "requestor_objectId": "accessPackageAssignmentRequest.requestor.objectId",
```

```
                "assignment_id": "accessPackageAssignmentRequest.assignment.id",
                "assignment_target_principalName":
"accessPackageAssignmentRequest.assignment.target.principalName",
                "accessPackage_id": "accessPackageAssignmentRequest.accessPackage.id"},
    "approvals": {"request_id": None, "reviewedBy_id": "approvalStage.reviewedBy.id",
                  "reviewedBy_displayName": "approvalStage.reviewedBy.displayName",
                  "reviewResult": "approvalStage.reviewResult", "reviewedDateTime":
"approvalStage.reviewedDateTime",
                  "justification": "approvalStage.justification"},
    "assignments": {"id": "accessPackageAssignment.id", "accessPackage_id":
"accessPackageAssignment.accessPackage.id",
                    "target_principalName": "accessPackageAssignment.target.principalName",
                    "state": "accessPackageAssignment.state",
                    "schedule_startDateTime": "accessPackageAssignment.schedule.startDateTime",
                    "schedule_expiration_type": "accessPackageAssignment.schedule.expiration.type",
                    "schedule_expiration_endDateTime":
"accessPackageAssignment.schedule.expiration.endDateTime"},
    "grants": {"grantee_name": None, "role": None, "created_on": None},
    "role_packages": {"role": None, "access_package_id": None},
}
GRAPH_URL = ("https://raw.githubusercontent.com/microsoftgraph/msgraph-metadata/"

"c4b98b7b7808ee5f4e7d752378339d70ef472154/clean_v10_metadata/cleanMetadataWithDescriptionsv1.0.xml"
)
GRAPH_BYTES = 8200681
GRAPH_SHA256 = "ff4f707d562dd1aa1dfaa591f4e17f4e6f50d517dc65b0278d216fb4fc4e547b"

# Values Microsoft documents (read 2026-11-04); a value outside these lists fails an input check.
KNOWN_STATES = ["submitted", "pendingApproval", "delivering", "delivered", "deliveryFailed",
"denied", "scheduled",
                "canceled", "partiallyDelivered", "unknownFutureValue"]
KNOWN_REQUEST_TYPES = ["notSpecified", "userAdd", "userUpdate", "userRemove", "adminAdd",
"adminUpdate", "adminRemove",
                       "systemAdd", "systemUpdate", "systemRemove", "onBehalfAdd",
"unknownFutureValue", "approverRemove"]
KNOWN_REVIEW_RESULTS = ["NotReviewed", "Approved", "Denied"]
KNOWN_EXPIRATION_TYPES = ["notSpecified", "noExpiration", "afterDateTime", "afterDuration"]
KNOWN_ASSIGNMENT_STATES = ["delivering", "partiallyDelivered", "delivered", "expired",
"deliveryFailed",
                           "unknownFutureValue"]
GRANT_REQUEST_TYPES = ["userAdd", "adminAdd", "onBehalfAdd"]   # requests that create an assignment
(v1.0 marks
                                                               # onBehalfAdd not supported; it is
counted if present)
REMOVE_REQUEST_TYPES = ["userRemove", "adminRemove", "systemRemove", "approverRemove"]   # requests
that end one
ZERO_GUID = "00000000-0000-0000-0000-000000000000"   # reviewedBy.id of a stage nobody has reviewed
CODES = ["NO_REQUEST", "NO_APPROVAL", "SELF_APPROVAL", "NO_REASON", "GRANT_BEFORE_APPROVAL",
"NO_END_DATE",
         "REMOVED_IN_ENTRA", "PAST_END_DATE", "UNMAPPED_ROLE"]
TS_RE = "\\d{4}-\\d{2}-\\d{2}[ T]\\d{2}:\\d{2}:\\d{2}(\\.\\d+)?( ?(Z|\\+00:00|\\+0000))?"
RUN_AT = dt.datetime.now(dt.timezone.utc).strftime("%Y-%m-%d %H:%M:%S UTC")


def ts(col):
    """A timestamp read strictly: YYYY-MM-DD[ T]HH:MM:SS taken as UTC (a Z, +00:00 or +0000 suffix
is accepted, any
    other shape is NULL and counted by an input check)."""
```

```
    return (f"CASE WHEN regexp_full_match(trim({col}), '{TS_RE}') THEN "
            f"try_strptime(substr(replace(trim({col}), 'T', ' '), 1, 19), '%Y-%m-%d %H:%M:%S')
END")


def blank(col):
    return f"nullif(trim({col}), '')"


def quote_list(items):
    return ", ".join("'" + str(i).replace("'", "''") + "'" for i in items)


def cell(v):
    """A CSV cell a spreadsheet will not run: text starting with = + - @, a tab or a carriage
return gets a ' first."""
    return "'" + v if isinstance(v, str) and v[:1] in ("=", "+", "-", "@", "\t", "\r") else v


# ---------------------------------------------------------------------------------------------
# Demo records: GENERATED (seed 124), with every column name checked against Microsoft's Graph
metadata.
# ---------------------------------------------------------------------------------------------
def check_graph_names(folder):
    """Download Graph's metadata at the pinned commit and fail if a column taken from it is not
there."""
    path = folder / "graph-v1.0-metadata.xml"
    folder.mkdir(parents=True, exist_ok=True)
    cached = path.is_file()
    if cached:
        data = path.read_bytes()
    else:
        req = urllib.request.Request(GRAPH_URL, headers={
            "User-Agent": "who_approved_each_grant.py (https://www.datameaning.com/guides/prove-
who-approved-each-data-access/; "
                          "info@datameaning.com)"})
        data = urllib.request.urlopen(req, timeout=60).read()
    sha = hashlib.sha256(data).hexdigest()
    print(f"  Graph metadata at the pinned commit: {len(data)} bytes, SHA-256 {sha[:16]}...")
    if len(data) != GRAPH_BYTES or sha != GRAPH_SHA256:
        if cached:
            path.unlink()
        sys.exit("The Graph metadata file is not the pinned one (size or SHA-256 differs):
stopping."
                 + (f" Deleted the saved copy {path}; the next run downloads it again." if cached
else ""))
    if not cached:
        path.write_bytes(data)
    types = {}
    for e in ET.fromstring(data).iter():
        if e.tag.endswith("}EntityType") or e.tag.endswith("}ComplexType"):
            types[e.get("Name")] = e

    def member(tname, prop):
        t = types.get(tname)
        while t is not None:
            for p in t:
                if p.get("Name") == prop and p.tag.split("}")[1] in ("Property",
```

```
"NavigationProperty"):
                    return re.sub(r"^Collection\((.*)\)$", r"\1",
p.get("Type")).removeprefix("graph.")
            t = types.get((t.get("BaseType") or "").removeprefix("graph."))
        return None

    missing, n = [], 0
    for fname, cols in GRAPH_PATHS.items():
        for col, gpath in cols.items():
            if gpath is None:
                continue
            n += 1
            parts = gpath.split(".")
            tname = parts[0]
            for prop in parts[1:]:
                tname = member(tname, prop)
                if tname is None:
                    missing.append(f"{fname}.{col} <- {gpath}")
                    break
    print(f"  Checked {n} columns against Graph's own property names; not found: {len(missing)}")
    if missing:
        sys.exit("Columns not in Microsoft Graph's metadata: " + "; ".join(missing))


def generate(folder, plant_error, plant_input):
    rnd = random.Random(124)
    folder.mkdir(parents=True, exist_ok=True)
    first = ["ada", "ben", "chloe", "dev", "elena", "farid", "gwen", "hugo", "iris", "jonas",
"kira", "luca",
             "maya", "noor", "omar", "petra", "quinn", "rosa", "sven", "tara"]
    people = [(f"{n}.{'smith' if i % 2 else 'jones'}@example.com", f"00000000-0000-4000-8000-{i +
1:012d}")
              for i, n in enumerate(first)]
    owners = [(f"{n}.owner@example.com", f"00000000-0000-4000-9000-{i + 1:012d}", n.title() + "
Owner")
              for i, n in enumerate(["sam", "tess", "uma", "vic", "wes"])]
    admin = "00000000-0000-4000-a000-000000000001"     # an Entra admin who assigns access
(adminAdd)
    roles = ["FINANCE_READER", "SALES_READER", "HR_ANALYST", "OPS_READER", "MARKETING_READER",
"SUPPLY_READER"]
    why = ["Quarter-end close reconciliation", "Pipeline review for the regional plan",
           "Headcount analysis for the annual plan", "Throughput study for the new site",
           "Campaign attribution rebuild", "Supplier lead-time analysis"]
    approved_why = ["Approved: the project named in the request needs this role",
                    "Approved: role matches the stated task, review in 180 days",
                    "Approved: owner confirmed the need with the team lead"]
    grants, requests, approvals, assignments, rolemap = [], [], [], [], []
    base = dt.datetime(2026, 8, 3, 9, 0, 0)
    z = lambda d: d.strftime("%Y-%m-%dT%H:%M:%SZ")
    for r in roles:
        rolemap.append({"role": r, "access_package_id": "pkg-" + r.lower().replace("_", "-")})
    for i in range(1, 31):
        email, oid = people[(i - 1) % 20]
        role = roles[(i - 1) % 6]
        pkg = "pkg-" + role.lower().replace("_", "-")
        own = owners[i % 5]
        asked = base + dt.timedelta(days=2 * i, minutes=rnd.randrange(240))
        approved = asked + dt.timedelta(days=1, hours=rnd.randrange(6), minutes=rnd.randrange(60))
```

```
        done = approved + dt.timedelta(minutes=10)
        rid, aid = f"req-{i:03d}", f"asg-{i:03d}"
        # 6 requests made by an admin: the requester is not the person who gets the access
        rtype, asker = ("adminAdd", admin) if i % 5 == 4 else ("userAdd", oid)
        requests.append({"id": rid, "requestType": rtype, "state": "delivered", "createdDateTime":
z(asked),
                         "completedDateTime": z(done), "justification": why[(i - 1) % 6],
                         "requestor_objectId": asker, "assignment_id": aid,
"assignment_target_principalName": email,
                         "accessPackage_id": pkg})
        approvals.append({"request_id": rid, "reviewedBy_id": own[1], "reviewedBy_displayName":
own[2],
                          "reviewResult": "Approved", "reviewedDateTime": z(approved),
                          "justification": approved_why[i % 3]})
        assignments.append({"id": aid, "accessPackage_id": pkg, "target_principalName": email,
"state": "delivered",
                            "schedule_startDateTime": z(done), "schedule_expiration_type":
"afterDateTime",
                            "schedule_expiration_endDateTime": z(done + dt.timedelta(days=180))})
        grants.append({"grantee_name": email.upper() if i % 4 == 0 else email, "role": role,
                       "created_on": z(done + dt.timedelta(minutes=30))})
    clean_values = {"LEGACY_ADMIN", "noExpiration", "forever", "OLD_ROLE", "expired",
"adminRemove"}
    seen = ({g["role"] for g in grants} | {a["schedule_expiration_type"] for a in assignments}
            | {a["state"] for a in assignments} | {r["requestType"] for r in requests})
    if clean_values & seen:
        sys.exit("A planted value already occurs in the clean records: stopping.")

    def row(rows, key, value):
        hit = [r for r in rows if r[key] == value]
        if len(hit) != 1:
            sys.exit(f"Could not plant a fault: {key}={value} matched {len(hit)} rows.")
        return hit[0]

    if plant_error:
        g = lambda n: grants[n - 1]
        rq = lambda n: row(requests, "id", f"req-{n:03d}")
        # 3: no request at all (and so no approval and no assignment)
        requests.remove(rq(3))
        approvals[:] = [a for a in approvals if a["request_id"] != "req-003"]
        assignments.remove(row(assignments, "id", "asg-003"))
        approvals[:] = [a for a in approvals if a["request_id"] != "req-007"]            # 7:
request, no approval
        a11 = row(approvals, "request_id", "req-011")                                    # 11: the
requester approved it
        a11["reviewedBy_id"], a11["reviewedBy_displayName"] = rq(11)["requestor_objectId"], "Self"
        row(approvals, "request_id", "req-015")["justification"] = ""                    # 15:
approval with no reason
        rq(16)["justification"] = ""                                                     # 16:
request with no reason
        ap19 = row(approvals, "request_id", "req-019")["reviewedDateTime"]               # 19:
granted two days early
        grants[18]["created_on"] = z(dt.datetime.strptime(ap19, "%Y-%m-%dT%H:%M:%SZ") -
dt.timedelta(days=2))
        a22 = row(assignments, "id", "asg-022")                                          # 22: no
end date
        a22["schedule_expiration_type"], a22["schedule_expiration_endDateTime"] = "noExpiration",
""
```

```
        row(assignments, "id", "asg-025")["schedule_expiration_endDateTime"] = "2026-10-
15T00:00:00Z"   # 25: ended
        r9, a9 = rq(9), row(assignments, "id", "asg-009")          # 9: removed in Entra, grant
left on the platform
        a9["state"] = "expired"
        requests.append({**r9, "id": "req-031", "requestType": "adminRemove", "requestor_objectId":
admin,
                         "justification": "Project closed", "createdDateTime": "2026-10-
20T09:00:00Z",
                         "completedDateTime": "2026-10-20T09:05:00Z"})
        g(28)["role"] = "LEGACY_ADMIN"                                                  # 28: role
not in the map
    header_fault = False
    if plant_input:
        grants[4]["created_on"] = "03/04/26"                                  # not a timestamp
        grants[5]["grantee_name"] = ""                                        # no grantee
        requests.append(dict(requests[9]))                                    # the same request id
twice
        requests[11]["state"] = "approved"                                    # not a state Graph
defines
        assignments[13]["schedule_expiration_type"] = "forever"               # not an expiration
type Graph defines
        rolemap.append({"role": "OLD_ROLE", "access_package_id": "pkg-old"})  # matches no grant
        rolemap.append(dict(rolemap[0]))                                      # one role mapped
twice
        header_fault = True                                                   # approvals.csv:
justification -> reason
    tables = {"grants": grants, "role_packages": rolemap, "requests": requests, "approvals":
approvals,
              "assignments": assignments}
    for name, rows in tables.items():
        cols = list(COLUMN_MAP[name])
        with open(folder / FILE_NAMES[name], "w", newline="") as f:
            w = csv.writer(f)
            w.writerow(cols)
            w.writerows([[r[c] for c in cols] for r in rows])
    if header_fault:
        p = folder / FILE_NAMES["approvals"]
        p.write_text(p.read_text().replace("reviewedDateTime,justification",
"reviewedDateTime,reason", 1))


# ---------------------------------------------------------------------------------------------
# ENGINE
# ---------------------------------------------------------------------------------------------
def main():
    args = sys.argv[1:]
    global DATA_DIR
    out = pathlib.Path(args[args.index("--out") + 1]) if "--out" in args else pathlib.Path("who-
approved-out")
    plant_error, plant_input = "--plant-error" in args, "--plant-input" in args
    suffix = "-planted-error" if plant_error else "-planted-input" if plant_input else ""
    if "--demo" in args:
        print("Demo: checking the generated records' column names against Microsoft Graph's
metadata")
        check_graph_names(out)
        gen = out / f"generated{suffix}"
        generate(gen, plant_error, plant_input)
```

```
        DATA_DIR = str(gen)
        print(f"Generated records written to {gen}/ (seed 124; not a company's data).")
    if DATA_DIR is None:
        print("Set DATA_DIR at the top of the script (or run with --demo).")
        return 2
    folder = pathlib.Path(DATA_DIR)
    out.mkdir(parents=True, exist_ok=True)
    problems_found = []

    def check(name, bad, detail=""):
        print(f"  [{'FAIL' if bad else 'ok'}] {name}" + (f": {detail}" if bad and detail else ""))
        if bad:
            problems_found.append(name)

    con = duckdb.connect()
    print(f"\nRun at {RUN_AT}. Exports taken {AS_OF} (UTC).")
    print("\nInput checks")
    bad_date = not (isinstance(AS_OF, str) and re.fullmatch(r"\d{4}-\d{2}-\d{2}", AS_OF))
    if not bad_date:
        try:
            dt.date.fromisoformat(AS_OF)
        except ValueError:
            bad_date = True
    check("AS_OF is a real date written YYYY-MM-DD", bad_date, f"AS_OF is {AS_OF!r}")
    if bad_date:
        return 2

    # --- load every file; a missing file or column is a failure, the table is then empty
    bad_files = []
    for name, cmap in COLUMN_MAP.items():
        path = folder / FILE_NAMES[name]
        have = []
        if path.is_file():
            have = [d[0] for d in con.execute("SELECT * FROM read_csv(?, header=true,
all_varchar=true) LIMIT 0",
                                              [str(path)]).description]
        lowered = {h.strip().lower(): h for h in have}
        missing = [theirs for theirs in cmap.values() if theirs.strip().lower() not in lowered]
        if not path.is_file():
            bad_files.append(f"{FILE_NAMES[name]} is missing")
        elif missing:
            bad_files.append(f"{FILE_NAMES[name]} has no column {', '.join(missing)}")
        if path.is_file() and not missing:
            sel = ", ".join(f'"{lowered[theirs.strip().lower()]}" AS "{mine}"' for mine, theirs in
cmap.items())
            con.execute(f"CREATE TABLE t_{name} AS SELECT {sel} FROM read_csv(?, header=true,
all_varchar=true)",
                        [str(path)])
        else:
            con.execute(f"CREATE TABLE t_{name} ({', '.join(chr(34) + c + chr(34) + ' VARCHAR' for
c in cmap)})")
    check("every file is there with every column the script reads", bad_files, ";
".join(bad_files))
    counts = {n: con.execute(f"SELECT count(*) FROM t_{n}").fetchone()[0] for n in COLUMN_MAP}
    for n, c in counts.items():
        print(f"      read {c} rows from {FILE_NAMES[n]}")

    # --- unreadable values are counted, never skipped
```

```
    unreadable = []
    # need: TRUE = every row must have it; otherwise a condition on the row (an Approved stage
needs its time)
    for tbl, col, need in [("grants", "created_on", "TRUE"), ("requests", "createdDateTime",
"TRUE"),
                           ("requests", "completedDateTime", None),
                           ("approvals", "reviewedDateTime", "trim(reviewResult) = 'Approved'"),
                           ("assignments", "schedule_startDateTime", None),
                           ("assignments", "schedule_expiration_endDateTime", None)]:
        given = blank(chr(34) + col + chr(34)) + " IS NOT NULL"
        n = con.execute(f"SELECT count(*) FROM t_{tbl} WHERE ({ts(chr(34) + col + chr(34))}) IS
NULL AND "
                        f"({given if need is None else given + ' OR ' + need})").fetchone()[0]
        if n:
            unreadable.append(f"{FILE_NAMES[tbl]}.{col}: {n}")
    check("every date and time is written as YYYY-MM-DD[T]HH:MM:SS (UTC)", unreadable,
          "not readable: " + ", ".join(unreadable))
    n = con.execute(f"SELECT count(*) FROM t_grants WHERE {blank('grantee_name')} IS NULL OR
{blank('role')} IS NULL").fetchone()[0]
    check("every grant names a grantee and a role", n, f"{n} grant row(s) with no grantee or no
role")
    n = con.execute(f"SELECT count(*) - count(DISTINCT lower(trim(id))) + count(*) FILTER (WHERE
{blank('id')} IS NULL) "
                    f"FROM t_requests").fetchone()[0]
    check("every request id is there once", n, f"{n} request row(s) repeat an id or have none")
    odd = []
    for tbl, col, known in [("requests", "state", KNOWN_STATES), ("requests", "requestType",
KNOWN_REQUEST_TYPES),
                            ("approvals", "reviewResult", KNOWN_REVIEW_RESULTS),
                            ("assignments", "schedule_expiration_type", KNOWN_EXPIRATION_TYPES),
                            ("assignments", "state", KNOWN_ASSIGNMENT_STATES)]:
        bad = con.execute(f"SELECT count(*) FROM t_{tbl} WHERE trim({col}) NOT IN
({quote_list(known)}) "
                          f"AND {blank(col)} IS NOT NULL").fetchone()[0]
        if bad:
            odd.append(f"{FILE_NAMES[tbl]}.{col}: {bad}")
    check("every state, type and result is one Microsoft documents", odd, "unknown values: " + ",
".join(odd))
    n = con.execute("SELECT count(*) FROM t_role_packages m WHERE NOT EXISTS (SELECT 1 FROM
t_grants g "
                    "WHERE lower(trim(g.role)) = lower(trim(m.role)))").fetchone()[0]
    check("every role in role_packages holds at least one grant", n, f"{n} role(s) match no grant:
a typo or a stale row")
    n = con.execute("SELECT count(*) - count(DISTINCT lower(trim(role))) FROM
t_role_packages").fetchone()[0]
    check("every role maps to one access package", n, f"{n} role(s) listed more than once")
    if problems_found:
        print(f"\nINCOMPLETE: {len(problems_found)} input check(s) failed, so no findings are
computed. Fix the input.")
        return 2

    # --- match every grant to its request, approvals and assignment
    asof = f"TIMESTAMP '{AS_OF} 00:00:00'"
    con.execute(f"""CREATE TEMP TABLE g AS
        SELECT row_number() OVER () AS grant_no, lower(trim(grantee_name)) AS grantee,
lower(trim(role)) AS role,
               trim(role) AS role_shown, ({ts('created_on')}) AS granted_at FROM t_grants""")
    con.execute("""CREATE TEMP TABLE pkg AS SELECT lower(trim(role)) AS role,
```

```
lower(trim(access_package_id)) AS package
                   FROM t_role_packages""")
    con.execute(f"""CREATE TEMP TABLE req AS
        SELECT lower(trim(id)) AS id, lower(trim(assignment_target_principalName)) AS principal,
               lower(trim(accessPackage_id)) AS package, lower(trim(requestor_objectId)) AS oid,
               ({ts('completedDateTime')}) AS completed, ({ts('createdDateTime')}) AS created,
               {blank('justification')} AS justification
        FROM t_requests WHERE trim(state) = 'delivered' AND trim(requestType) IN
({quote_list(GRANT_REQUEST_TYPES)})""")
    con.execute(f"""CREATE TEMP TABLE rem AS
        SELECT lower(trim(id)) AS id, trim(requestType) AS rtype, lower(trim(assignment_id)) AS
assignment_id
        FROM t_requests WHERE trim(state) = 'delivered' AND trim(requestType) IN
({quote_list(REMOVE_REQUEST_TYPES)})""")
    con.execute(f"""CREATE TEMP TABLE apr AS
        SELECT lower(trim(request_id)) AS request_id,
               CASE WHEN lower({blank('reviewedBy_id')}) = '{ZERO_GUID}' THEN NULL ELSE
lower({blank('reviewedBy_id')}) END AS reviewer,
               {blank('reviewedBy_displayName')} AS reviewer_name, ({ts('reviewedDateTime')}) AS
at,
               {blank('justification')} AS justification
        FROM t_approvals WHERE trim(reviewResult) = 'Approved'""")
    con.execute(f"""CREATE TEMP TABLE asg AS
        SELECT lower(trim(id)) AS id, lower(trim(target_principalName)) AS principal,
               lower(trim(accessPackage_id)) AS package, trim(state) AS state,
trim(schedule_expiration_type) AS etype, ({ts('schedule_expiration_endDateTime')}) AS ends_at,
               row_number() OVER (PARTITION BY lower(trim(target_principalName)),
lower(trim(accessPackage_id))
                                  ORDER BY ({ts('schedule_startDateTime')}) DESC NULLS LAST, id) AS
rk
        FROM t_assignments""")
    con.execute("""CREATE TEMP TABLE ev AS
        WITH cand AS (
          SELECT g.grant_no, r.id AS request_id, r.oid, r.created, r.justification AS req_reason,
                 row_number() OVER (PARTITION BY g.grant_no ORDER BY coalesce(r.completed >
g.granted_at, TRUE),
                                    abs(epoch(r.completed) - epoch(g.granted_at)), r.id) AS rk
          FROM g JOIN pkg p ON p.role = g.role JOIN req r ON r.principal = g.grantee AND r.package
= p.package)
        SELECT c.grant_no, c.request_id, c.oid, c.created, c.req_reason,
               (SELECT count(*) FROM apr a WHERE a.request_id = c.request_id AND a.reviewer IS NOT
NULL) AS n_appr,
               (SELECT count(*) FROM apr a WHERE a.request_id = c.request_id AND a.reviewer =
c.oid) AS n_self,
               (SELECT count(*) FROM apr a WHERE a.request_id = c.request_id AND a.reviewer IS NOT
NULL
                  AND a.justification IS NULL) AS n_blank,
               (SELECT max(a.at) FROM apr a WHERE a.request_id = c.request_id AND a.reviewer IS NOT
NULL) AS last_appr,
               (SELECT string_agg(DISTINCT coalesce(a.reviewer_name, a.reviewer), '; ') FROM apr a
                  WHERE a.request_id = c.request_id AND a.reviewer IS NOT NULL) AS approvers,
               (SELECT string_agg(coalesce(a.justification, '(none)'), ' | ') FROM apr a
                  WHERE a.request_id = c.request_id AND a.reviewer IS NOT NULL) AS approver_reasons
        FROM cand c WHERE c.rk = 1""")
    con.execute(f"""CREATE TEMP TABLE found AS
        WITH m AS (SELECT g.*, p.package, e.request_id, e.req_reason, e.n_appr, e.n_self,
e.n_blank, e.last_appr,
                          s.id AS asg_id, s.state AS asg_state, s.etype, s.ends_at, s.principal IS
```

```
NOT NULL AS has_asg,
                          (SELECT string_agg(x.id || ' (' || x.rtype || ')', ', ' ORDER BY x.id)
FROM rem x
                             WHERE x.assignment_id = s.id) AS removed_by,
                          p.role IS NOT NULL AS mapped, e.request_id IS NOT NULL AS has_req
                   FROM g LEFT JOIN pkg p ON p.role = g.role LEFT JOIN ev e ON e.grant_no =
g.grant_no
                   LEFT JOIN asg s ON s.principal = g.grantee AND s.package = p.package AND s.rk =
1)
        SELECT grant_no, 'UNMAPPED_ROLE' AS code, 'role ' || role_shown || ' is not in
role_packages' AS detail
          FROM m WHERE NOT mapped
        UNION ALL SELECT grant_no, 'NO_REQUEST', 'no delivered request for this person and package'
FROM m
          WHERE mapped AND NOT has_req
        UNION ALL SELECT grant_no, 'NO_APPROVAL', 'request ' || request_id || ' has no approval by
a named reviewer'
          FROM m WHERE has_req AND n_appr = 0
        UNION ALL SELECT grant_no, 'SELF_APPROVAL', 'request ' || request_id || ' was approved by
the person who asked'
          FROM m WHERE has_req AND n_self > 0
        UNION ALL SELECT grant_no, 'NO_REASON',
               'request ' || request_id || ': ' || CASE WHEN req_reason IS NULL AND n_blank > 0
THEN 'no reason from the requester or the approver'
                 WHEN req_reason IS NULL THEN 'no reason from the requester' ELSE 'no reason from
the approver' END
          FROM m WHERE has_req AND (req_reason IS NULL OR n_blank > 0)
        UNION ALL SELECT grant_no, 'GRANT_BEFORE_APPROVAL', 'granted ' || strftime(granted_at, '%Y-
%m-%d %H:%M')
               || ', last approval ' || strftime(last_appr, '%Y-%m-%d %H:%M')
          FROM m WHERE has_req AND n_appr > 0 AND last_appr > granted_at
        UNION ALL SELECT grant_no, 'NO_END_DATE', CASE WHEN NOT has_asg THEN 'no assignment for
this person and package'
                 ELSE 'assignment type ' || etype || ' holds no end date' END
          FROM m WHERE mapped AND ends_at IS NULL
        UNION ALL SELECT grant_no, 'REMOVED_IN_ENTRA', 'assignment ' || asg_id || ' is '
               || coalesce(asg_state, '(blank)') || coalesce('; removed by request ' || removed_by,
'')
          FROM m WHERE has_asg AND (coalesce(asg_state, '') <> 'delivered' OR removed_by IS NOT
NULL)
        UNION ALL SELECT grant_no, 'PAST_END_DATE', 'ended ' || strftime(ends_at, '%Y-%m-%d') || ',
grant still present'
          FROM m WHERE mapped AND ends_at < {asof}""")
    n_grants = con.execute("SELECT count(*) FROM g").fetchone()[0]
    if n_grants != counts["grants"]:   # a guard on the script itself, not an input check: it
counts the grants it numbered
        sys.exit(f"Internal error: {counts['grants']} grants read but {n_grants} matched.")

    print("\nFindings (one line per gap; a grant can have more than one)")
    rows = con.execute("""SELECT f.grant_no, g.grantee, g.role_shown, f.code, f.detail FROM found f
JOIN g USING (grant_no)
                          ORDER BY f.grant_no, f.code""").fetchall()
    for no, who, role, code, detail in rows:
        print(f"  grant {no:>2}  {who:<28} {role:<16} {code:<21} {detail}")
    if not rows:
        print("  none")
    flagged = con.execute("SELECT count(DISTINCT grant_no) FROM found").fetchone()[0]
    print("\nCounts by finding")
```

```
    by = dict(con.execute("SELECT code, count(*) FROM found GROUP BY 1").fetchall())
    for c in CODES:
        print(f"  {c:<21} {by.get(c, 0)}")
    print(f"\n{n_grants} grants read: {n_grants - flagged} matched to a request, an approver, a
reason and an end date; "
          f"{flagged} with at least one finding.")

    # --- the evidence file: one row per grant
    evidence = out / f"grant_evidence{suffix}.csv"
    cur = con.execute("""SELECT g.grant_no, g.grantee, g.role_shown AS role, strftime(g.granted_at,
'%Y-%m-%d %H:%M:%S') AS granted_at,
              p.package AS access_package, e.request_id, strftime(e.created, '%Y-%m-%d %H:%M:%S')
AS requested_at,
              e.req_reason AS requester_reason, e.approvers, e.approver_reasons,
              strftime(e.last_appr, '%Y-%m-%d %H:%M:%S') AS approved_at, strftime(s.ends_at, '%Y-
%m-%d') AS access_ends,
              coalesce((SELECT string_agg(f.code, ' ' ORDER BY f.code) FROM found f WHERE
f.grant_no = g.grant_no),
                       'MATCHED') AS verdict
        FROM g LEFT JOIN pkg p ON p.role = g.role LEFT JOIN ev e ON e.grant_no = g.grant_no
        LEFT JOIN asg s ON s.principal = g.grantee AND s.package = p.package AND s.rk = 1 ORDER BY
g.grant_no""")
    cols, data = [d[0] for d in cur.description], cur.fetchall()
    with open(evidence, "w", newline="") as f:
        w = csv.writer(f)
        w.writerow(cols)
        w.writerows([[cell(v) for v in r] for r in data])
    print(f"Evidence for every grant written to {evidence} ({len(data)} rows).")
    print("\nThree matched grants, as an auditor would read them")
    shown = 0
    for r in data:
        d = dict(zip(cols, r))
        if d["verdict"] == "MATCHED" and shown < 3:
            shown += 1
            print(f"  grant {d['grant_no']}: {d['grantee']} holds {d['role']}; asked
{d['requested_at']} "
                  f"({d['requester_reason']}); approved {d['approved_at']} by {d['approvers']}
({d['approver_reasons']}); "
                  f"access ends {d['access_ends']}")
    return 1 if ("--strict" in args and rows) else 0


if __name__ == "__main__":
    sys.exit(main())
```

The runs

Run 1: ten grants with a gap, planted

We ran the script on 2026-11-04 with DuckDB 1.5.6. --plant-error changes ten of the 30 generated grants so that each of the nine findings has something to find: grant 3 has no request, 7 no approval, 9 an assignment that Entra ended (its state is expired and an admin's removal request names it) while the grant stayed, 11 an approver who is the requester, 15 an approver with no reason,

16 a requester with no reason, 19 a grant two days before its approval, 22 no end date, 25 an end date that has passed, and 28 a role missing from the map. Grant 3 has two findings: it has no request, and so no assignment either. The script stops before planting if a planted value already occurs in the clean records, or if a request or assignment row it changes or removes, or an approval row it edits, is not found exactly once.

```
python3 guides/prove-who-approved-each-data-access/who_approved_each_grant.py --demo --plant-error
--out loop-out/data/wa
```

```
Demo: checking the generated records' column names against Microsoft Graph's metadata
  Graph metadata at the pinned commit: 8200681 bytes, SHA-256 ff4f707d562dd1aa...
  Checked 22 columns against Graph's own property names; not found: 0
Generated records written to loop-out/data/wa/generated-planted-error/ (seed 124; not a company's
data).

Run at 2026-11-04 10:48:01 UTC. Exports taken 2026-11-04 (UTC).

Input checks
  [ok] AS_OF is a real date written YYYY-MM-DD
  [ok] every file is there with every column the script reads
      read 30 rows from grants.csv
      read 6 rows from role_packages.csv
      read 30 rows from requests.csv
      read 28 rows from approvals.csv
      read 29 rows from assignments.csv
  [ok] every date and time is written as YYYY-MM-DD[T]HH:MM:SS (UTC)
  [ok] every grant names a grantee and a role
  [ok] every request id is there once
  [ok] every state, type and result is one Microsoft documents
  [ok] every role in role_packages holds at least one grant
  [ok] every role maps to one access package

Findings (one line per gap; a grant can have more than one)
  grant  3  chloe.jones@example.com      HR_ANALYST       NO_END_DATE           no assignment for
this person and package
  grant  3  chloe.jones@example.com      HR_ANALYST       NO_REQUEST            no delivered
request for this person and package
  grant  7  gwen.jones@example.com       FINANCE_READER   NO_APPROVAL           request req-007 has
no approval by a named reviewer
  grant  9  iris.jones@example.com       HR_ANALYST       REMOVED_IN_ENTRA      assignment asg-009
is expired; removed by request req-031 (adminRemove)
  grant 11  kira.jones@example.com       MARKETING_READER SELF_APPROVAL         request req-011 was
approved by the person who asked
  grant 15  omar.jones@example.com       HR_ANALYST       NO_REASON             request req-015: no
reason from the approver
  grant 16  petra.smith@example.com      OPS_READER       NO_REASON             request req-016: no
reason from the requester
  grant 19  sven.jones@example.com       FINANCE_READER   GRANT_BEFORE_APPROVAL granted 2026-09-09
15:50, last approval 2026-09-11 15:50
  grant 22  ben.smith@example.com        OPS_READER       NO_END_DATE           assignment type
noExpiration holds no end date
  grant 25  elena.jones@example.com      FINANCE_READER   PAST_END_DATE         ended 2026-10-15,
grant still present
  grant 28  hugo.smith@example.com       LEGACY_ADMIN     UNMAPPED_ROLE         role LEGACY_ADMIN
```

```
is not in role_packages

Counts by finding
  NO_REQUEST            1
  NO_APPROVAL           1
  SELF_APPROVAL         1
  NO_REASON             2
  GRANT_BEFORE_APPROVAL 1
  NO_END_DATE           2
  REMOVED_IN_ENTRA      1
  PAST_END_DATE         1
  UNMAPPED_ROLE         1

30 grants read: 20 matched to a request, an approver, a reason and an end date; 10 with at least
one finding.
Evidence for every grant written to loop-out/data/wa/grant_evidence-planted-error.csv (30 rows).

Three matched grants, as an auditor would read them
  grant 1: ada.jones@example.com holds FINANCE_READER; asked 2026-08-05 10:11:00 (Quarter-end close
reconciliation); approved 2026-08-06 14:11:00 by Tess Owner (Approved: role matches the stated
task, review in 180 days); access ends 2027-02-02
  grant 2: ben.smith@example.com holds SALES_READER; asked 2026-08-07 09:45:00 (Pipeline review for
the regional plan); approved 2026-08-08 10:11:00 by Uma Owner (Approved: owner confirmed the need
with the team lead); access ends 2027-02-04
  grant 4: dev.smith@example.com holds OPS_READER; asked 2026-08-11 10:21:00 (Throughput study for
the new site); approved 2026-08-12 11:46:00 by Wes Owner (Approved: role matches the stated task,
review in 180 days); access ends 2027-02-08
```

Ten of 30 grants are flagged, with eleven findings between them. The other 20 each get a full evidence row in grant_evidence.csv; the last block prints three of them as an auditor would read them: who asked and why, who approved and why, and the date the access ends.

Run 2: the clean records

The same script on the same seed without --plant-error. Every grant matches, and no finding is printed. On these complete records, no check fires. Six of the 30 requests were made by an admin (adminAdd), so the requester is not the person who got the access; they match like the rest, because the script matches on the person the request assigns.

```
python3 guides/prove-who-approved-each-data-access/who_approved_each_grant.py --demo --out loop-
out/data/wa
```

```
Demo: checking the generated records' column names against Microsoft Graph's metadata
  Graph metadata at the pinned commit: 8200681 bytes, SHA-256 ff4f707d562dd1aa...
  Checked 22 columns against Graph's own property names; not found: 0
Generated records written to loop-out/data/wa/generated/ (seed 124; not a company's data).

Run at 2026-11-04 10:48:03 UTC. Exports taken 2026-11-04 (UTC).

Input checks
  [ok] AS_OF is a real date written YYYY-MM-DD
  [ok] every file is there with every column the script reads
      read 30 rows from grants.csv
```

```
      read 6 rows from role_packages.csv
      read 30 rows from requests.csv
      read 30 rows from approvals.csv
      read 30 rows from assignments.csv
  [ok] every date and time is written as YYYY-MM-DD[T]HH:MM:SS (UTC)
  [ok] every grant names a grantee and a role
  [ok] every request id is there once
  [ok] every state, type and result is one Microsoft documents
  [ok] every role in role_packages holds at least one grant
  [ok] every role maps to one access package

Findings (one line per gap; a grant can have more than one)
  none

Counts by finding
  NO_REQUEST            0
  NO_APPROVAL           0
  SELF_APPROVAL         0
  NO_REASON             0
  GRANT_BEFORE_APPROVAL 0
  NO_END_DATE           0
  REMOVED_IN_ENTRA      0
  PAST_END_DATE         0
  UNMAPPED_ROLE         0

30 grants read: 30 matched to a request, an approver, a reason and an end date; 0 with at least one
finding.
Evidence for every grant written to loop-out/data/wa/grant_evidence.csv (30 rows).

Three matched grants, as an auditor would read them
  grant 1: ada.jones@example.com holds FINANCE_READER; asked 2026-08-05 10:11:00 (Quarter-end close
reconciliation); approved 2026-08-06 14:11:00 by Tess Owner (Approved: role matches the stated
task, review in 180 days); access ends 2027-02-02
  grant 2: ben.smith@example.com holds SALES_READER; asked 2026-08-07 09:45:00 (Pipeline review for
the regional plan); approved 2026-08-08 10:11:00 by Uma Owner (Approved: owner confirmed the need
with the team lead); access ends 2027-02-04
  grant 3: chloe.jones@example.com holds HR_ANALYST; asked 2026-08-09 10:11:00 (Headcount analysis
for the annual plan); approved 2026-08-10 12:13:00 by Vic Owner (Approved: the project named in the
request needs this role); access ends 2027-02-06
```

Run 3: the input checks stop a bad export

--plant-input plants eight faults in the generated files, aimed at seven of the eight input checks: the header of approvals.csv renamed (justification to reason), a grant date written 03/04/26, a grant with no grantee, one request id twice, a request state Graph does not define (approved), an expiration type Graph does not define (forever), a role in the map that no grant holds, and one role listed twice. One check (every state, type and result is one Microsoft documents) catches two of the faults, so seven of the eight input checks fail. The one that passes is the date in the settings.

```
python3 guides/prove-who-approved-each-data-access/who_approved_each_grant.py --demo --plant-input
--out loop-out/data/wa
```

```
Demo: checking the generated records' column names against Microsoft Graph's metadata
  Graph metadata at the pinned commit: 8200681 bytes, SHA-256 ff4f707d562dd1aa...
  Checked 22 columns against Graph's own property names; not found: 0
Generated records written to loop-out/data/wa/generated-planted-input/ (seed 124; not a company's
data).

Run at 2026-11-04 10:48:04 UTC. Exports taken 2026-11-04 (UTC).

Input checks
  [ok] AS_OF is a real date written YYYY-MM-DD
  [FAIL] every file is there with every column the script reads: approvals.csv has no column
justification
      read 30 rows from grants.csv
      read 8 rows from role_packages.csv
      read 31 rows from requests.csv
      read 0 rows from approvals.csv
      read 30 rows from assignments.csv
  [FAIL] every date and time is written as YYYY-MM-DD[T]HH:MM:SS (UTC): not readable:
grants.csv.created_on: 1
  [FAIL] every grant names a grantee and a role: 1 grant row(s) with no grantee or no role
  [FAIL] every request id is there once: 1 request row(s) repeat an id or have none
  [FAIL] every state, type and result is one Microsoft documents: unknown values:
requests.csv.state: 1, assignments.csv.schedule_expiration_type: 1
  [FAIL] every role in role_packages holds at least one grant: 1 role(s) match no grant: a typo or
a stale row
  [FAIL] every role maps to one access package: 1 role(s) listed more than once

INCOMPLETE: 7 input check(s) failed, so no findings are computed. Fix the input.
```

How to read the results

- A finding is a question, not a verdict. It says a record is missing or out of order. The access may still be right; the record that shows it is not there.
- A matched grant is a complete chain, not a good decision. The script does not know whether the approver was the right owner or the reason was a good one. A person reads the reasons in grant_evidence.csv.
- Check the counts add up. The summary line gives grants read, grants matched and grants with a finding; they must add up to the rows of grants.csv, and so must the rows of grant_evidence.csv.
- Keep the file with the date of the export. The evidence is as of AS_OF and of the exports you took.

Run it on your own data

To run it on your own data, change four settings at the top of the script and nothing else. Set DATA_DIR to the folder that holds your five CSV files. Set FILE_NAMES to the names of those files.

In COLUMN_MAP, change the name on the right of each line to the column name in your file; leave the names on the left as they are, and names are matched without regard to case. Set AS_OF to the day you took the exports. Export only current grants (for Snowflake, the rows of GRANTS_TO_USERS where DELETED_ON is empty), list in the role map only roles that hold a grant, write every date and time as YYYY-MM-DD HH:MM:SS in UTC (a T between date and time, fractions of a second and a Z, +00:00 or +0000 ending are accepted), and save every file as CSV with a header row. Export each request with its assignment expanded, so assignment_id and assignment_target_principalName are filled, and each assignment with its state. Then run the script with no options.

We tested those sentences on a reader's data built for the purpose: other file names, other column names in mixed case, a grantee typed in capitals, a reviewer id in other letter case, a request an admin made, a request row with no requester, an approval with no named reviewer, a reason that starts with an equals sign, and at least one gap for each of the nine findings (a small invented set; that run's output is not reproduced here). Done literally, the script printed all nine and nothing failed.

Checklist

Done	Item	Note
□	Five CSV files exported, with a header row	grants.csv, role_packages.csv, requests.csv, approvals.csv, assignments.csv
□	Only current grants in grants.csv	For Snowflake, rows of GRANTS_TO_USERS where DELETED_ON is empty
□	DATA_DIR, FILE_NAMES, COLUMN_MAP and AS_OF set	Top of the script; nothing else changed
□	Every input check shows ok	If one fails, fix the file; no findings are computed until it passes
□	Every finding has an owner and a next step	See the table of findings
□	grant_evidence.csv read by a person	Reasons and approvers read, not only counted
□	The export date is kept with the file	AS_OF and the date you took the exports

Method

- Records: generated with Python's random.Random(124). Thirty grants, six roles, twenty people who hold access, one admin who made six of the requests, five approvers, all invented, all example.com addresses. Each assignment ends 180 days after its request was delivered. Nothing in them is a measurement of any company or platform.

- Column names: taken from the Graph v1.0 metadata file in microsoftgraph/msgraph-metadata at commit c4b98b7b (8,200,681 bytes, SHA-256 ff4f707d562dd1aa1dfaa591f4e17f4e6f50d517dc65b0278d216fb4fc4e547b). The script stops if the downloaded file differs.
- Matching rule: a request counts if its state is delivered, its requestType is userAdd, adminAdd or onBehalfAdd (Microsoft's v1.0 page marks onBehalfAdd not supported; it is counted if it appears), and the person it assigns (assignment.target.principalName) and its access package equal the grant's grantee and the package the role maps to. If several match, the one completed closest before the grant is used; if none was completed before it, the closest after. Names are compared without regard to case.
- Approval rule: a stage counts if its reviewResult is Approved and its reviewer is named. A reviewer id of all zeros, which Microsoft documents for a stage nobody reviewed, counts as not named. Ids are compared without regard to case.
- Removal rule: a grant is reported REMOVED_IN_ENTRA if the person's most recent assignment for the package has a state other than delivered, or a delivered request of type userRemove, adminRemove, systemRemove or approverRemove names that assignment's id. Microsoft documents that for a removal the request's assignment holds the id of the assignment it removes (accessPackageAssignmentRequest (read 2026-11-04)).
- Not a check: if the grants the script numbered differ from the grants it read, it stops with an internal error. No input of ours trips it, and it is not a check on your data. The date setting check is tripped by an AS_OF that is not a date; no run plants that.
- Mapping tables written out: the known request states, request types, results, expiration types and assignment states, and the request types the script counts as creating or ending an assignment, are lists at the top of the script (values copied from Microsoft's documentation); edit them if you disagree.
- Spreadsheet safety: a text cell in grant_evidence.csv that starts with =, +, -, @, a tab or a carriage return gets a ' in front, so a spreadsheet shows it as text and does not run it as a formula. Reasons are typed by requesters, so this matters.
- What trips each check: NO_REQUEST (grant 3), NO_APPROVAL (7), SELF_APPROVAL (11), NO_REASON (15 and 16), GRANT_BEFORE_APPROVAL (19), NO_END_DATE (22, and 3), REMOVED_IN_ENTRA (9), PAST_END_DATE (25), UNMAPPED_ROLE (28). Input checks: file and column present (renamed header), date readable (03/04/26), grantee and role named (blank grantee), request ids once (repeated id), values Microsoft documents (approved, forever), role map holds a grant (a stale role), one package per role (a role twice).

Limits

- This is not a client result and not a measurement. The records are generated to show the check. They say nothing about how often real grants lack a request.
- It checks that a record exists, not that the decision was right. A named approver may not own the data, and a reason may be one word. A person has to read them.
- It only knows the approvals you export. Access given outside Entra, or approved by email or in a ticket, shows up as NO_REQUEST even if someone said yes. To count an approval given by email, add it to your requests and approvals files and re-run, or accept the finding.
- It matches by sign-in name and role. If your platform names people differently from Entra's principalName, map them in the export first. A grant held through a group or a role nested in another role needs the export to list the person and the role they end up holding.
- Request history may not reach back far enough. A grant older than the history you export has no request in the files and is reported as NO_REQUEST.
- An assignment that carries a duration and no end date is reported as NO_END_DATE, because the script reads only schedule_expiration_endDateTime. Export the end date, or work it out from the start and the duration, before you run it.
- It does not change anything. It does not remove a grant, ask for a reason or end access.
- Evidence, not a conclusion. What it prints is one input to a review. It does not say any review or framework is satisfied.

Sources

Every document linked above, read 2026-11-04:

- accessPackageAssignmentRequest resource type (read 2026-11-04)
- approvalStage resource type (read 2026-11-04)
- accessPackageAssignment resource type (read 2026-11-04)
- expirationPattern resource type (read 2026-11-04)
- List approval stages (read 2026-11-04)
- Get accessPackageAssignmentRequest (read 2026-11-04)
- Snowflake ACCOUNT_USAGE GRANTS_TO_USERS view (read 2026-11-04)
- microsoftgraph/msgraph-metadata at commit c4b98b7b7808ee5f4e7d752378339d70ef472154 (read 2026-11-04)

- Data Access Concierge (Data Meaning) (read 2026-11-04)

Have the record exist before anyone asks for it

The script finds gaps after the fact. Data Access Concierge makes the record at the start: every permission it gives carries a reason, the owner's yes and an end date. People ask for data on the catalog page, and the owner approves in one screen.

Data Meaning sets it up and runs it in your own environment. Access given by hand outside it still needs a separate check.

See how Data Access Concierge works