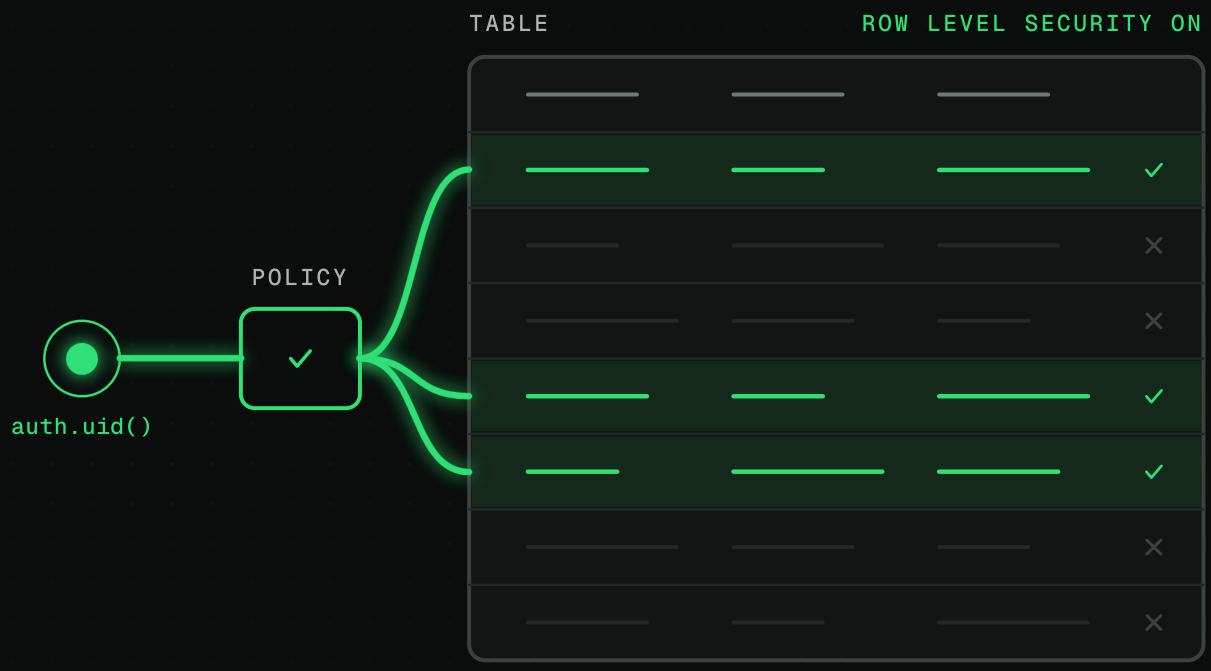


PROMPT PACK · FREE

Supabase RLS Review Prompt Pack

Eight prompts and a test matrix for proving that row level security protects a Supabase app before launch.

HOW TO USE THIS PROMPT PACK

Replace the assumption that RLS is on with executed evidence of what each role can read and change.

Eight review prompts, a policy test matrix, a defects reference, and a pre-launch checklist that make a coding agent prove, with files, policy names, and query results, that row level security protects a Supabase app's data.

WORKING METHOD

01 Give the agent read access to the repository and migrations. For the prompts that run queries, give it a local or staging database with seeded test users, never production.

02 Run the inventory prompt first. Its table tells you which tables, views, and buckets the later prompts need to cover.

03 Run one prompt per pass and keep each output. Static prompts read code and migrations. The exposure, isolation, and performance prompts need the non-production database.

04 Copy the policy test matrix once per table, write the expected result in every cell before testing, then fill in what the agent observed.

05 Match each finding to the defects reference, fix it in a migration, rerun the prompt that found it, and finish with the pre-launch checklist.

INSIDE

- ✓ 8 review prompts
- ✓ Policy test matrix
- ✓ Common policy defects reference
- ✓ Pre-launch RLS checklist

BEFORE YOU START

No prompt in this pack may be pointed at a production database. Tests that insert, update, and delete need seeded data and a transaction that is rolled back.

A clean result covers the tables, roles, and requests that were tested. It is not a penetration test and does not cover Edge Functions, Auth settings, or third-party services.

Migrations can differ from the live database. Changes made in the dashboard exist only in the database until someone writes them into a migration.

Do not paste key values or real customer rows into a prompt. Variable names and seeded test rows are enough.

CONTENTS

01	Table and RLS inventory	PROMPT
02	Policy-by-policy review	PROMPT
03	Anon and authenticated exposure	PROMPT
04	Elevated key usage in code	PROMPT
05	Storage bucket policies	PROMPT
06	Views and security definer functions	PROMPT
07	Tenant isolation test	PROMPT
08	Policy performance review	PROMPT
09	Policy test matrix	TABLE
10	Common policy defects reference	TABLE
11	Pre-launch RLS checklist	16 CHECKS

Table and RLS inventory

List every exposed table and view with its row level security state before judging any single policy.

USE WHEN	EXPECTED RESULT	REPLACE
At the start of every review, and again after any migration that adds a table or view.	One table of objects with RLS state, policy count, commands covered, roles, and the migration line that proves each entry.	`[EXPOSED SCHEMAS]` `[MIGRATIONS PATH]` `[NON-PROD DATABASE]`

PROMPT · COPY AND PASTE

```
Build an inventory of every table and view in [EXPOSED SCHEMAS] for this Supabase project. Read [MIGRATIONS PATH] in file order first. Then, only if a connection to [NON-PROD DATABASE] is provided, confirm against pg_class (relrowsecurity, relforcerowsecurity) and pg_policies. This task is read only: do not run DDL, do not write rows, and do not connect to production. If the only database available is production, stop and say so. For each object record the schema and name, whether row level security is enabled, the number of policies, which commands they cover (SELECT, INSERT, UPDATE, DELETE, ALL), the roles each policy names, and the migration file and line that proves it. Flag three states separately: RLS disabled, RLS enabled with no policies, and policies present while RLS is disabled. Mark any row you could not confirm in the database as migrations only. Output one table with RLS disabled first, then a one-line count per state. Do not propose fixes. Stop after the table.
```

HOLD THE LINE — Read only · Never production · Cite file and line

> TIP Exposed schemas are listed in the project's Data API settings. Most projects expose public, and some add others.

Policy-by-policy review

Read each policy as a rule about a request, and name the request it wrongly allows.

USE WHEN	EXPECTED RESULT	REPLACE
After the inventory, for every table that holds user, tenant, or payment data.	A finding per defective policy with file, line, policy name, and the request that succeeds but should not.	[TABLES IN SCOPE] [OWNERSHIP MODEL] [MIGRATIONS PATH]

PROMPT · COPY AND PASTE

```
Review every row level security policy on [TABLES IN SCOPE]. Intended access model: [OWNERSHIP MODEL]. Read each CREATE POLICY, ALTER POLICY and DROP POLICY statement in [MIGRATIONS PATH] in file order and use the final definition. For each policy state the command, the roles in its TO clause, whether it is permissive or restrictive, the USING expression, and the WITH CHECK expression. Then check for: a missing TO clause, an always-true expression, an INSERT or UPDATE check that does not tie the ownership column to auth.uid(), an UPDATE policy with no SELECT policy beside it, conditions that read user-editable metadata from the JWT, and several permissive policies on one command whose combination allows more than the model. Do not edit files, do not run SQL against production, and do not rewrite policies. For each finding give file, line, policy name, the request that succeeds but should not, and whether you verified or inferred it. Policies with no finding get one line. Stop after the last table.
```

HOLD THE LINE — Final definition only · Name the bad request · No policy rewrites

> TIP Write the ownership model as one sentence per table, for example: a user reads and edits only rows where user_id is their own id.

Anon and authenticated exposure

Measure what a caller with only the public key can do, signed out and signed in.

USE WHEN	EXPECTED RESULT	REPLACE
Before launch and after any change to grants, policies, or the list of exposed schemas.	A role by operation table for each table, built from executed statements, plus a list of mismatches with the expected access.	[NON-PROD DATABASE] [EXPOSED SCHEMAS] [TABLES IN SCOPE] [TEST USER A] [EXPECTED ACCESS]

PROMPT · COPY AND PASTE

```
Determine what a caller holding only the publishable key (or the legacy anon key) can do, signed out and signed in. Work only against [NON-PROD DATABASE]. If it points at production, stop. Step 1: list the table, view and function privileges granted to the anon and authenticated roles in [EXPOSED SCHEMAS]. Step 2: inside a transaction that you roll back, switch to the anon role and attempt SELECT, INSERT, UPDATE and DELETE on each table in [TABLES IN SCOPE]. Step 3: repeat as the authenticated role with the JWT claims of [TEST USER A]. Use seeded test rows only. For each attempt record the statement, the row count or error returned, and whether it matches [EXPECTED ACCESS]. A filtered read, update or delete returns zero rows with no error, so compare counts with the seeded data. Do not commit, do not change grants or policies, and do not print row contents beyond ids. Output a role by operation table per table, then the mismatches. Stop there.
```

HOLD THE LINE

- Non-production only
- Roll back every test
- Counts, not row contents

> TIP The publishable key ships in your app, so treat anyone on the internet as able to send these requests. That is the caller this prompt imitates.

Elevated key usage in code

Find every use of a key that skips row level security and decide whether a browser can reach it.

USE WHEN	EXPECTED RESULT	REPLACE
The app has any server code, Edge Functions, scripts, or AI-generated files that create a Supabase client.	Each client creation and elevated query classified as browser-reachable, server only with a check, or server only without a check.	`[CLIENT CODE PATHS] [SERVER CODE PATHS] [ENV AND CI FILES]`

PROMPT · COPY AND PASTE

```
Find every place this repository uses a Supabase key that bypasses row level security. Search [CLIENT CODE PATHS], [SERVER CODE PATHS] and [ENV AND CI FILES] for secret keys (sb_secret_ prefix), legacy service_role JWT keys, variables whose names suggest either, and every call that creates a Supabase client. For each client creation record the file and line, which key variable it reads, and whether that code can be bundled for or executed in a browser or mobile app. Check public environment variable prefixes, client-side components and import chains to decide. Then list every query made with an elevated key and state which authorization check runs before it, because policies will not apply. Never print a key value: show the variable name only. Do not rotate, edit or delete anything, and do not call any live project. Classify each use as browser-reachable, server only with a check, or server only without a check, citing evidence. If a key value is committed to the repository, report that first. Stop after the classification.
```

HOLD THE LINE

- Never print key values
- No live project calls
- Evidence for each class

> TIP A query made with a secret key is only as safe as the authorization check in the code before it. Ask for that check by file and line.

Storage bucket policies

State, per bucket, who can upload, overwrite, read, list, and delete, and which policy allows it.

USE WHEN	EXPECTED RESULT	REPLACE
The app stores uploads, documents, exports, or avatars in Supabase Storage.	A per-bucket access summary with the policy behind each permission and findings that cite file, line, and policy name.	[BUCKETS] [FILE ACCESS MODEL] [MIGRATIONS PATH] [NON-PROD DATABASE]

PROMPT · COPY AND PASTE

```
Review Supabase Storage access for [BUCKETS]. Intended model: [FILE ACCESS
MODEL]. First list each bucket and whether it is public or private, from
[MIGRATIONS PATH], seed files or, read only, [NON-PROD DATABASE]. Then read
every policy on storage.objects and map it to a bucket through its bucket_id
condition. Report a policy with no bucket_id condition as applying to every
bucket. For each bucket state who can upload, overwrite, read, list and
delete, and which policy grants it. Check for: a public bucket that holds
private files, since public buckets serve files without access checks, SELECT
policies broad enough to list a whole bucket, write policies that restrict
neither the path nor the owner, and upsert code paths that lack the SELECT and
UPDATE policies they need. Confirm the client code that uploads files or
creates signed URLs matches. Do not upload, delete or move files, do not
change bucket settings, and never touch production. Cite file, line and policy
name for each finding. Stop after the per-bucket summary.
```

HOLD THE LINE · No file operations · No bucket setting changes · Map policy to bucket

> TIP Decide public or private by the most sensitive file a bucket will ever hold, not by the files in it today.

Views and security definer functions

List every object that can read past table policies and show the call that proves it.

USE WHEN	EXPECTED RESULT	REPLACE
The schema has views, materialized views, RPC functions, triggers, or helper functions used inside policies.	Each view and security definer function marked safe or unsafe, with a concrete query or call for every unsafe one.	[EXPOSED SCHEMAS] [MIGRATIONS PATH] [NON-PROD DATABASE]

PROMPT · COPY AND PASTE

```
Find every database object that can read past row level security. From [MIGRATIONS PATH], and read only from [NON-PROD DATABASE] if provided, list: (1) views in [EXPOSED SCHEMAS] and whether each sets security_invoker = true, (2) functions declared SECURITY DEFINER, with schema, owner, search_path setting and the roles that hold EXECUTE, (3) materialized views and any view that selects from auth.users. For each view without security_invoker, name the underlying tables and the rows a caller could read that the table policies would hide. For each security definer function, state what it reads or writes, whether it checks the caller before acting, whether it sits in an exposed schema, and whether anon or authenticated can execute it. Do not alter, drop or call these objects, and do not connect to production. For each unsafe object report file, line, object name, and a concrete query or call that returns data it should not, marked verified or inferred. Safe objects get one line. Stop after the list.
```

HOLD THE LINE Do not call or alter · Never production · Verified or inferred

> TIP Row level security does not apply to functions. For a function, the controls are who holds EXECUTE and what the body checks.

Tenant isolation test

Act as one tenant's user and try every route into another tenant's rows.

USE WHEN	EXPECTED RESULT	REPLACE
Rows belong to an organization, team, workspace, or account, and more than one customer shares the same tables.	A pass or fail line per table and attempt, with the statement, rows affected, and the policy responsible for each failure.	[NON-PROD DATABASE] [TENANT COLUMN] [TEST USER A] [TEST USER B] [TABLES IN SCOPE]

PROMPT · COPY AND PASTE

```
Prove that one tenant cannot read or change another tenant's data. Use only
[NON-PROD DATABASE] with seeded data. If seeds are missing, stop and list what
is needed. Tenancy is carried by [TENANT COLUMN]. [TEST USER A] belongs to
tenant A and [TEST USER B] belongs to tenant B. Run every attempt inside a
transaction that is rolled back. For each table in [TABLES IN SCOPE], act as
authenticated with user A's JWT claims and try to: select tenant B rows by id,
insert a row carrying tenant B's [TENANT COLUMN], move a tenant A row to
tenant B, update and delete a tenant B row by id, and read tenant B data
through each exposed view and function. Repeat as user B against tenant A.
Record each statement with rows affected or the error. Expected every time:
zero rows or an error. Do not commit, disable a policy, or use an elevated
key. Output pass or fail per table and attempt, then each failure with its
policy. Stop there.
```

HOLD THE LINE Seeded data only · Roll back every test · No elevated key

> TIP Seed at least two rows per tenant in every table. A table with no tenant B rows passes every read test without proving anything.

Policy performance review

Find policy expressions that are evaluated per row or filter on unindexed columns, without changing which rows they return.

USE WHEN	EXPECTED RESULT	REPLACE
Policies are correct and tables are expected to grow, or list queries slow down once RLS is enabled.	A list of cost patterns by policy with a proposed expression or index, each marked measured or unmeasured.	`[TABLES IN SCOPE]` `[HOT QUERIES]` `[NON-PROD DATABASE]`

PROMPT · COPY AND PASTE

```
Review the row level security policies on [TABLES IN SCOPE] for avoidable query cost. Frequent queries: [HOT QUERIES]. From the policy definitions list: calls to auth.uid(), auth.jwt() or helper functions that do not depend on row values and are not wrapped in a select, columns compared in policies that have no index in the schema, subqueries that look up a membership table for each row instead of filtering the row's column against the caller's set, and policies with no TO clause. If [NON-PROD DATABASE] is provided, run EXPLAIN ANALYZE on [HOT QUERIES] as the authenticated role with test claims, before and after each proposed change, inside a transaction that is rolled back, and report the plans. Do not invent timings, do not run anything on production, and do not apply changes. Every proposed rewrite must return the same rows: state how you checked. Output the policy name, file and line, the cost pattern, the proposed expression or index, and whether it is measured or unmeasured. Stop after the list.
```

HOLD THE LINE Same rows returned · No invented timings · Propose, do not apply

> TIP Run this prompt last. Tuning a policy that is still wrong only makes the wrong answer faster.

Policy test matrix

One copy per table. Write the expected result in each cell before testing, then record what the test returned.

Read results by count. A row hidden by a USING expression is filtered without an error, so a blocked read, update, or delete usually shows as zero rows. A row rejected by a WITH CHECK expression raises an error. The service_role row is there as a reminder that policies do not protect anything reached with a secret key: the server code must check the caller itself.

CALLER	SELECT	INSERT	UPDATE	DELETE
anon, signed out	Expected: [ ] Observed: [ ]	Expected: [ ] Observed: [ ]	Expected: [ ] Observed: [ ]	Expected: [ ] Observed: [ ]
authenticated, row owner	Expected: own rows only. Observed: [ ]	Expected: allowed for own id. Observed: [ ]	Expected: own rows only. Observed: [ ]	Expected: [ ] Observed: [ ]
authenticated, another user in the same tenant	Expected: [ ] Observed: [ ]	Expected: [ ] Observed: [ ]	Expected: [ ] Observed: [ ]	Expected: [ ] Observed: [ ]
authenticated, user in another tenant	Expected: 0 rows. Observed: [ ]	Expected: error. Observed: [ ]	Expected: 0 rows affected. Observed: [ ]	Expected: 0 rows affected. Observed: [ ]
authenticated, payload carries another owner or tenant id	Not applicable	Expected: error. Observed: [ ]	Expected: error. Observed: [ ]	Not applicable
service_role, server code only	All rows, policies bypassed	Allowed, policies bypassed	Allowed, policies bypassed	Allowed, policies bypassed

Common policy defects reference

Match a finding to its row to see how to confirm it and which direction the fix takes.

Supabase's Security Advisor and Performance Advisor report several of these states automatically. Treat an advisor result as a lead to confirm against your access model: some findings are intentional, and a clean advisor run does not show that a policy's logic matches what you intended.

DEFECT	HOW IT SHOWS UP	HOW TO CONFIRM	FIX DIRECTION
RLS disabled on an exposed table	Any caller with the publishable key reads or writes rows the grants allow	relrowsecurity is false in pg_class, or Security Advisor flags the table	Enable RLS in a migration, then add policies for each needed command
Policies written, RLS never enabled	Policies look correct in review and have no effect	pg_policies lists the table while relrowsecurity is false	Enable RLS on the table and retest every role

DEFECT	HOW IT SHOWS UP	HOW TO CONFIRM	FIX DIRECTION
RLS enabled, no policies	API requests return no rows, so the app is switched to a secret key	Table has zero rows in pg_policies and RLS is on	Write policies for the public key roles. Keep the secret key on the server
Always-true expression	Every caller the policy names can read or change every row	USING or WITH CHECK is true or equivalent in pg_policies	Replace with a condition on the caller, or limit the policy to public data
TO clause omitted	The policy applies to every role, including anon	The roles column in pg_policies shows public	Name the role, usually authenticated, in the TO clause
Write check does not bind the owner	A user inserts or reassigns rows under another user or tenant id	As a test user, write a row carrying another id and see it succeed	Compare the owner or tenant column to the caller in WITH CHECK
UPDATE policy without a SELECT policy	Updates affect zero rows for users who should be allowed	The table has an UPDATE policy and no SELECT or ALL policy	Add a SELECT policy covering the rows the user may update
Policy trusts user-editable metadata	A user edits their own profile metadata and gains access	The policy expression reads user metadata from the JWT	Keep authorization data in app metadata or a table users cannot edit
View without security_invoker	The view returns rows that the base table policies hide	Query the view as a test user and compare with the base table	Set security_invoker = true on Postgres 15 or later, or move the view out of exposed schemas
Security definer function open to public roles	An API call returns or changes data the caller's policies forbid	The function is SECURITY DEFINER and anon or authenticated holds EXECUTE	Move it to an unexposed schema, revoke EXECUTE, set search_path, check the caller
Secret key in client code	Every policy is bypassed for whoever extracts the key	A secret or service_role key is read by code that ships to browsers	Remove it from the client, rotate the key, route the call through server code
Auth function evaluated per row	List queries slow down as tables grow	auth.uid() is unwrapped in the policy, or the compared column lacks an index	Wrap the call in a select and index the column, then confirm with EXPLAIN ANALYZE

11 / 11 CHECKLIST

Pre-launch RLS checklist

Run after the prompts and fixes. Each check names the evidence that closes it.

Tables and policies

4 CHECKS

- [ ] 01 Confirm every table in an exposed schema has RLS enabled by rerunning the inventory prompt; expect zero rows in the RLS disabled state.
- [ ] 02 Open each table that has RLS enabled and no policies, and record in writing whether it is server-only by design or a missing policy.

- [ ] 03 Check each policy's TO clause in pg_policies or the migrations; expect a named role on every policy, with anon used only for public data.
- [ ] 04 Read every INSERT and UPDATE policy and confirm its WITH CHECK compares the owner or tenant column to the caller; expect no always-true checks.

Roles and keys

4 CHECKS

- [ ] 05 Search the built client bundle for secret key prefixes and legacy service_role keys; expect no match in any file served to a browser.
- [ ] 06 List every server query made with a secret key and point to the authorization check before it; expect a file and line for each one.
- [ ] 07 Run the signed-out test for each table in a rolled-back transaction on staging; expect the observed result to equal the matrix's anon row.
- [ ] 08 Review the privileges granted to anon and authenticated on exposed tables and functions; revoke any the application never uses and retest the affected calls.

Views, functions, and storage

4 CHECKS

- [ ] 09 Query each exposed view as a test user and compare with the base table policies; expect no row the user could not read directly.
- [ ] 10 List SECURITY DEFINER functions with their EXECUTE grants; expect none callable by anon or authenticated unless the body verifies the caller and scope.
- [ ] 11 Confirm each SECURITY DEFINER function sets a fixed search_path in its definition; cite the migration line that sets it.
- [ ] 12 For each storage bucket, record public or private and the policies on storage.objects; expect no private file retrievable through a public bucket URL.

Evidence and regression

4 CHECKS

- [ ] 13 Fill one policy test matrix per sensitive table from executed statements; expect every observed cell to equal its expected cell before launch.
- [ ] 14 Run the tenant isolation test with two seeded tenants after the final migration; expect zero rows or an error for every cross-tenant attempt.
- [ ] 15 Run the Security Advisor on the staging project and resolve or document each RLS-related finding; attach the accepted exceptions with a reason.
- [ ] 16 Compare staging policies with the migrations by diffing pg_policies output against the repository; expect no policy that exists only in the database.

FIELD NOTES

Common mistakes, useful habits, hard limits.

01 **Treating enabled RLS as finished**

Enabling row level security with no policies denies every request made with the public key. Enabling it with a policy that is always true allows every request. Both states show RLS as on. Only a test as each role tells them apart.

02 **Fixing an empty result with the secret key**

When a query returns nothing after RLS is enabled, the missing piece is usually a policy or a signed-in session. Switching the client to a secret key makes the query work by bypassing every policy, for that query and for anyone who obtains the key.

03 **Testing only as the owner**

A policy that lets a user read their own rows passes every happy-path test, including when it also lets them read everyone else's. Isolation is shown by the requests that return zero rows or an error: another user, another tenant, and a signed-out caller.

PRACTICAL TIPS

Keep policies in migrations, not only in the dashboard, so a reviewer and an agent can read the same definitions the database runs.

Write one policy per command and role where you can. Small policies are easier to test cell by cell than one policy for ALL.

Save the isolation statements as database tests and run them in CI, so a later migration cannot quietly reopen a table.

When several permissive policies cover the same command, test their combination. Postgres joins permissive policies with OR, so the widest one wins.

Add explicit filters in application queries as well as policies. The policy is the security boundary, and the filter lets the planner do less work.

WHEN NOT TO USE THIS

When the only database you can reach is production. Create a local or staging copy with seeded users first, then run the prompts there.

For apps where clients never talk to Supabase directly and every query goes through your own server with its own authorization layer. Review that layer instead, and still enable RLS on exposed tables.

As a replacement for a security assessment your customers, insurer, or regulator requires. This pack reviews data access rules, not the whole application.

SOURCES · CHECKED 2026-11-22

Supabase Docs: Row Level Security · RLS should be enabled on every table in an exposed schema, and once enabled no data is accessible with a publishable key until policies exist. SELECT and DELETE use USING, INSERT uses WITH CHECK, UPDATE uses both and needs a corresponding SELECT policy. auth.uid() returns null for unauthenticated requests. Policies should name a role with TO. A secret key authorizes as service_role, which has bypassrls, and must never be used in a browser. Views bypass RLS by default and can set security_invoker = true on Postgres 15 and above. Security definer functions should set search_path and stay out of exposed schemas. User metadata is user-editable and app metadata is not. Wrapping auth functions in a select and indexing policy columns improves performance.
`https://supabase.com/docs/guides/database/postgres/row-level-security`

PostgreSQL Docs: Row Security Policies · With row security enabled and no policy, a default-deny policy applies. Superusers and roles with BYPASSRLS always bypass row security, and table owners bypass it unless FORCE ROW LEVEL SECURITY is set. Permissive policies combine with OR and restrictive policies with AND.
`https://www.postgresql.org/docs/current/ddl-rowsecurity.html`

PostgreSQL Docs: CREATE POLICY · A policy applies to PUBLIC, meaning all roles, when TO is omitted, and is permissive by default. Rows failing a USING expression are not visible and typically raise no error, while rows failing WITH CHECK raise an error. INSERT policies take only WITH CHECK, SELECT and DELETE only USING, and UPDATE both. UPDATE and DELETE that read columns also apply SELECT policies.
`https://www.postgresql.org/docs/current/sql-createpolicy.html`

PostgreSQL Docs: CREATE VIEW · By default access to a view's base relations, including their row-level security policies, is determined by the view owner. With security_invoker set to true, the policies and permissions of the invoking user are used instead.
`https://www.postgresql.org/docs/current/sql-createview.html`

PostgreSQL Docs: CREATE FUNCTION · A SECURITY DEFINER function executes with the privileges of its owner. search_path should be set to exclude schemas writable by untrusted users. EXECUTE is granted to PUBLIC by default for new functions and must be revoked to restrict use.
`https://www.postgresql.org/docs/current/sql-createfunction.html`

PostgreSQL Docs: pg_policies and pg_class · pg_policies lists each policy with schemaname, tablename, policyname, permissive, roles, cmd, qual and with_check. pg_class.relrowsecurity is true when a table has row-level security enabled, and relforcerowsecurity when it also applies to the table owner (https://www.postgresql.org/docs/current/catalog-pg-class.html).
`https://www.postgresql.org/docs/current/view-pg-policies.html`

Supabase Docs: Understanding API keys · Publishable keys (sb_publishable_ prefix) map to the anon role, or authenticated when a user is signed in, and are safe in browsers. Secret keys (sb_secret_ prefix) map to service_role, bypass RLS and are for backend use only. The anon and service_role JWT keys are the legacy equivalents and Supabase documents them as being deprecated.
`https://supabase.com/docs/guides/api/api-keys`

Supabase Docs: Securing your API · Grants control whether a role can access an object and RLS controls which rows it can access, and both should be used for every exposed object. RLS does not apply to functions, so EXECUTE should be granted only to roles that need it.
`https://supabase.com/docs/guides/api/securing-your-api`

Supabase Docs: Storage access control · Storage does not allow uploads to buckets without RLS policies. Policies are created on the storage.objects table. Uploading needs INSERT, and upsert also needs SELECT and UPDATE. Service keys bypass Storage policies.
`https://supabase.com/docs/guides/storage/security/access-control`

Supabase Docs: Storage buckets · Buckets are private by default. A public bucket bypasses access controls for retrieving and serving files, while access control is still enforced for uploading, deleting, moving and copying.

https://supabase.com/docs/guides/storage/buckets/fundamentals

Supabase Docs: Database advisors · The Security and Performance Advisors run checks that flag, among others, tables in public without RLS, policies on tables with RLS disabled, RLS enabled with no policy, always-true policies, policies that reference user metadata, security definer views, and auth functions re-evaluated per row. Some findings may be intentional and should be checked against the access model.

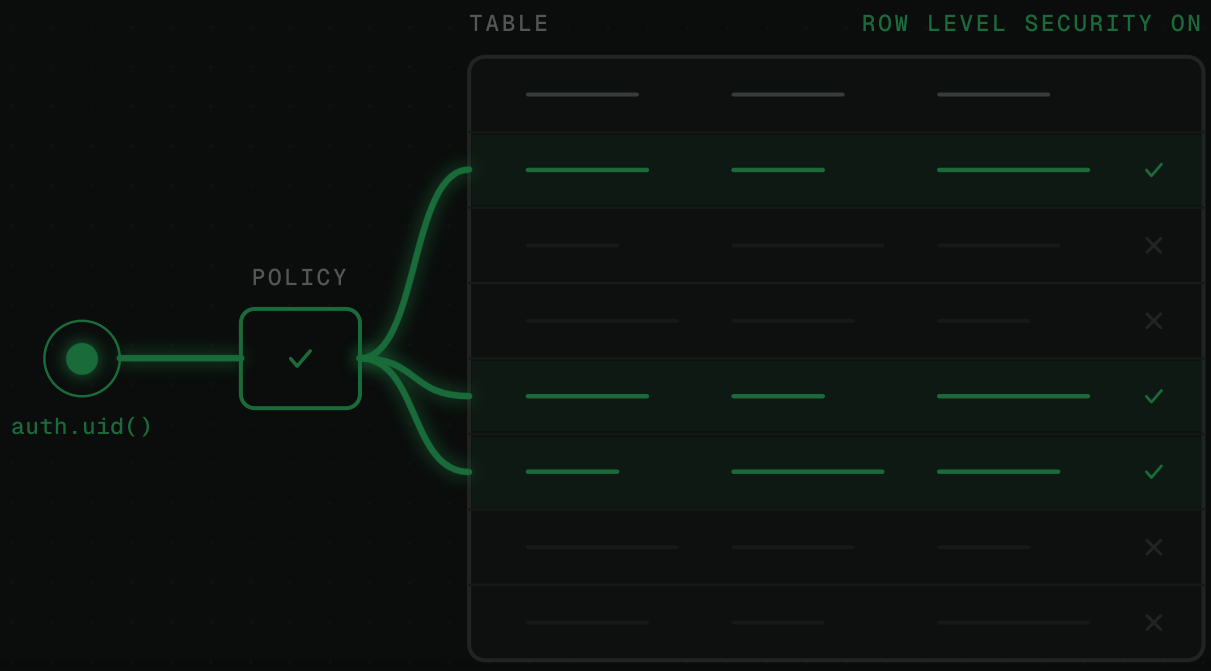
https://supabase.com/docs/guides/database/database-advisors

Supabase Docs: RLS performance and best practices · Recommendations: index columns used in policies, wrap functions in a select when the result does not depend on row data, add explicit filters in client queries, filter the row column against the caller's set instead of a per-row lookup, name roles with TO, and measure with EXPLAIN ANALYZE in a non-production environment.

https://supabase.com/docs/guides/troubleshooting/rls-performance-and-best-practices-Z5Jjwv

Supabase Docs: Testing overview · Database tests can switch role with set local role and set request JWT claims, should use begin and rollback for isolation, should cover the anonymous and authenticated roles, and should always include negative cases.

https://supabase.com/docs/guides/local-development/testing/overview



FAISAL KARKOH / RESOURCES

More working files for building with AI.

Prompt packs, checklists, templates and playbooks for shipping reliable AI products. Free to download and reuse in your own work.

LIBRARY
faisalkarkoh.com/resources

WRITING
faisalkarkoh.com/blog

WORK TOGETHER
faisalkarkoh.com/contact