n8n Pro-Tips (Public)

Table of Contents

21. Supabase Node → Postgres Node Migration

1. Nested Loops Are Not Supported: Use Sub-Workflows

> **Severity: Critical.** This causes silent data loss with no error or warning.

The Problem: n8n does not support loops within loops. If you place a loop inside another loop, the inner loop will only execute for the **first item** from the outer loop. All subsequent outer loop items are silently skipped: no error, no warning, just incomplete data.

What Happens

```
Outer Loop: items [A, B, C]
  → Inner Loop runs for A
  → Inner Loop silently fails for B (no error)
  → Inner Loop silently fails for C (no error)

Result: you think all 3 were processed. Only A was.
```

The workflow completes successfully. Executions show green. You only discover the problem when you check your data.

Common Symptoms

- A **loop within a loop** only processes the first item from the outer loop
- Database records are only partially populated after a run
- Workflow shows no errors but output count doesn't match expectations
- Inner loop results appear correct for the first item only

Important: The **Split Out** node itself works fine inside loops. The issue is specifically with **nested loops** (IF → loop back pattern inside another IF → loop back pattern).

The Fix: Sub-Workflows

Replace the inner loop with an **Execute Sub-workflow** node. The inner loop logic lives in a separate workflow that gets called once per outer item.

```
BROKEN: Nested loop:

Outer Loop (IF) → [Split Out → Inner Loop (IF) → Process] → Edit Fields → back to Outer L
oop

CORRECT: Sub-workflow pattern:

Outer Loop (IF) → Execute Sub-workflow → Edit Fields → back to Outer Loop
                          ↓
               [Sub-workflow: Split Out → IF → Process → done]
```

How to Implement

In the main workflow:

1. Replace the inner loop entirely with an **Execute Workflow** node
2. Pass the item data needed by the inner logic as input parameters
3. The outer loop continues as normal after the sub-workflow returns

In the sub-workflow:

1. Trigger: **Execute Workflow Trigger** node
2. Build the inner loop logic (Split Out → IF → Process) as normal
3. Return results back to the main workflow if needed

Why This Architecture Is Actually Better

Beyond fixing the bug, sub-workflows offer real advantages:

- Inner logic can be **tested and debugged independently**
- Sub-workflows can be **reused** across multiple parent workflows
- Cleaner, more maintainable workflow structure
- Easier to reason about data flow

---

2. Wait Node + Sub-Workflows: Parallel Processing for Approval Flows

The Problem: If you build an approval flow (e.g. draft email responses, then get Slack approval before sending) in a single sequential workflow with a Wait node, the workflow halts completely on the first item until the approval times out or is actioned. With 10 items, each with a 10-minute wait, you could be waiting 100+ minutes to process everything.

The Pattern

Use a **sub-workflow** instead, and configure the main workflow to **not wait** for the sub-workflow to complete before moving to the next item.

```
Main Workflow:
  [Trigger] → [For Each Item] → [Execute Sub-workflow (don't wait)] → [next item...]

Sub-workflow (runs independently per item):
  [Trigger] → [Draft response] → [Wait node] → [Send approval to Slack] → [Wait for respo
nse] → [Send / Discard]
```

With this setup:

- All 10 items fire off their sub-workflows simultaneously
- All 10 draft messages arrive in Slack at the same time
- You approve/reject them all in one batch, not one by one

Key Configuration

In the **Execute Workflow** node, set **“Wait for sub-workflow”** to **OFF**. This tells the main workflow to continue to the next item immediately rather than waiting for the sub-workflow to finish.

When to Use This

Any time you have a flow where:

- You process multiple items

- Each item requires a human action or an external wait (approval, webhook callback, scheduled send)
- You want all items to be “in flight” simultaneously rather than sequentially

---

3. Loops + Wait Node: Handling Rate Limits

The Problem: Running a long task against an API in a straight sequence will hit rate limits. For example, fetching full Slack threads for 500+ messages in one go will likely fail partway through.

The Pattern

Use a **Loop node** (Split in Batches) combined with a **Wait node** to pace your requests.

```
[Items] → [Split in Batches: 10] → [API Call] → [Wait: 3s] → ↩ loop
```

Example calculation for Slack (20 requests/minute limit):

- 60 seconds ÷ 20 requests = 3 seconds per request
- Set Wait node to **3 seconds** to stay safely within limits

Adjust batch size and wait time based on the API’s rate limit documentation.

Also Useful For

- Supabase bulk operations where you want to avoid overloading the database
- Any external API with a requests-per-minute cap
- Large-scale data processing where you want predictable, throttled execution

---

4. Recovering Lost Data Links

The Problem: After passing data through a Code node or certain transformations, you lose the reference to data from earlier in the workflow. Downstream expressions like `$('Node Name').json.some_field` no longer work as expected.

Solution 1: Reference a Specific Earlier Node

Use `$('Node Name').first().json` or `$node['Node Name'].json` to explicitly pull data from an earlier node by name, bypassing the "current item" context entirely.

```
// Pull from a specific earlier node
{
  {
    $("Set Message Fields").first().json.channel_id;
  }
}
```

Solution 2: JavaScript Filter to Match the Right Item

When you have multiple items and need to find the one that corresponds to the current item, use `.all().filter()`:

```
{
  {
    $("Set Message Fields")
      .all()
      .filter((ms) => ms.json.ts === $json.message_id)[0].jso
n.text;
  }
}
```

This is useful when a Code node breaks the 1:1 item pairing and you need to re-establish the relationship by matching on a shared key (like a message ID or timestamp).

Solution 3: `.map()` for Array Transformations

JavaScript's `.map()` is helpful when you need to transform an array of items, e.g.:

```
{
  {
```

```
    $("Get Channels")
      .all()
      .map((item) => item.json.channel_name);
  }
}
```

Solution 4 - pairedItem & itemMatching: Preserving Data Links in Code Nodes

The Problem: Code nodes break n8n's internal item-to-item linking. After a Code node, downstream nodes can no longer trace which output item corresponds to which input item. This is the root cause of the data linking issues described in Tip #4.

---

pairedItem: Preserve Links Through Code Nodes

When returning items from a Code node, use the `pairedItem` property to tell n8n which input item each output item corresponds to:

```
// "Run Once for All Items" mode
const items = $input.all();

return items.map((item, index) => ({
  json: {
    // your transformed data
    processed: item.json.someField.toUpperCase(),
  },
  pairedItem: index, // links this output to input item at sa
me index
}));
```

This preserves the chain so that nodes further downstream can still reference data from nodes **before** the Code node using standard expressions.

---

itemMatching: Access Linked Items from Earlier Nodes

When you need to access the linked item from an earlier node (after pairedItem has preserved the link), use `itemMatching`:

```
// Access the paired item from a specific earlier node
const earlierData = $("Earlier Node Name").itemMatching($inpu
t.item);
```

This is the n8n-native alternative to the manual `.all().filter()` workaround from Tip #4. It's cleaner, less error-prone, and doesn't require you to manually match on shared keys.

When to Use Each

- **pairedItem**: Use in every Code node where you're transforming items and want to maintain linkage for downstream nodes
- **itemMatching**: Use downstream of a Code node (that used pairedItem) when you need to pull data from a node that came before the Code node
- **Tip #4 workarounds**: Still useful as a fallback when pairedItem wasn't set, or for complex many-to-many relationships

---

5. Useful Internal Nodes

These built-in n8n nodes are often overlooked but save significant time:

Sort: Sort items by any field, ascending or descending. Avoids writing custom sort logic in Code nodes.

Limit: Cap the number of items flowing through. Useful for testing (process only first 5) or enforcing maximums.

Deduplication Node: Remove duplicate items based on a key field. Essential for idempotent workflows that might receive the same data twice.

Code Node: JavaScript or Python for custom logic. Use as a last resort: check if a built-in node (Sort, Deduplicate, etc.) already solves your problem first.

Edit Fields (Set): Normalize and reshape data. Prefer this over Code nodes for simple field mapping; it's visually clear and easier to modify later.

No-Op Node (Do Nothing): Acts as a flow control junction. When multiple branches converge back into a single loop, use a No-Op to collect all branches cleanly before the loop continues. Makes complex flows much easier to read.

```
Branch A ─┐
Branch B ──→ [No-Op] → [Loop back / Continue]
Branch C ─┘
```

---

6. When a Node Doesn’t Exist or Doesn’t Do What You Need

If n8n doesn’t have a native node for a service, or the existing node is too limited, here are your options in order of ease vs. power:

Option 1: HTTP Request Node (easiest)
Most APIs have REST endpoints. The HTTP Request node can call any API with full control over headers, authentication, and body. Check the service’s API docs and build the call manually.

Option 2: Apify (more powerful, harder to configure, may cost money)
For web scraping and browser automation tasks. More capable than HTTP Request for dynamic pages, but requires setup and has usage costs.

Option 3: Browser Simulation (most powerful, hardest to maintain)
Full browser automation for services with no API or that require human-like interaction:

- **Anchor Browser**
- **Browser-Use**
- **Browser Base**

These are the most capable but also the most brittle: any UI change in the target service can break your automation.

---

7. Binary Data Downstream

When you need to pass binary data (images, files, etc.) to downstream nodes after it's been fetched, standard `$json` references won't work: you need to access the binary data explicitly.

Method 1: Edit Fields Node

Use an Edit Fields node with this expression to reference binary data from an earlier node:

```
{{ $('REPLACE WITH NODE NAME').item.binary.data }}
```

Method 2: Code Node (Buffer)

For creating binary data from scratch: for example, generating a Markdown file programmatically: use a Code node with a Buffer:

```
const content = "# My Generated File\\n\\nContent here.";
const buffer = Buffer.from(content, "utf8");

return [
  {
    json: {},
    binary: {
      data: {
        data: buffer.toString("base64"),
        mimeType: "text/markdown",
        fileName: "output.md",
      },
    },
  },
];
```

Method 3: Extracting Binary Data to Base64/Text (Code Node)

The reverse of Method 2: when you have an incoming file (from a webhook upload, Read Binary File, HTTP Request, etc.) and need to extract its contents as base64 or readable text.

> **Important:** Don't access `binary.data.data` directly. n8n can store binary data in an external filesystem location (especially for larger files), so the base64 string may not be on the object. Always use n8n's built-in helper.

Use `this.helpers.getBinaryDataBuffer()`: this is the n8n-recommended approach. It handles both inline base64 and filesystem-stored binary data:

```
// Run Once for Each Item mode
const buffer = await this.helpers.getBinaryDataBuffer(0, "dat
a");
const meta = $input.first().binary?.data || {};

return {
  json: {
    base64: buffer.toString("base64"),
    textContent: buffer.toString("utf8"),
    fileName: meta.fileName || "unknown",
    mimeType: meta.mimeType || "application/octet-stream",
  },
};
```

Key distinction:

- `$input.first().binary.data` → gives you the **metadata object** (fileName, mimeType, etc.)
- `this.helpers.getBinaryDataBuffer()` → gives you the **actual file contents** as a Node.js Buffer

The binary key name (`data`) must match what the upstream node used. If a webhook or form upload created the binary, the key might be `file`, `attachment`, or the form field name. Check the upstream node's output to confirm, or detect it dynamically:

```
// Detect the binary key dynamically
const binaryKey = Object.keys($input.first().binary)[0];
const buffer = await this.helpers.getBinaryDataBuffer(0, bina
ryKey);
```

For non-text files (images, PDFs), skip `toString('utf8')` and work with the base64 string directly: that's what you'd pass to an API like OpenAI's vision endpoint or store in a database.

---

8. Always Output Data: Handle Empty Lookups Gracefully

The Problem: When a lookup node (e.g. Supabase "Get a row") finds no matching record, it outputs nothing: the item disappears from the flow entirely. This makes it impossible to handle the "not found" case in the same workflow branch.

This is particularly relevant for **Supabase**, which does not support native upsert operations, making the "check if exists → create or update" pattern very common.

The Setting

On any node that might return empty: go to **Settings** tab → enable **"Always Output Data"**.

When enabled, if the node returns no results, it outputs an **empty JSON object** `{}` instead of nothing. The item stays in the flow, and you can route it conditionally.

The Pattern: Create or Update (Manual Upsert)

```
[Get a row]  ← Always Output Data: ON
     ↓
  [IF node] : does row exist?
  TRUE ↓              FALSE ↓
[Update row]      [Create row + Slack notify]
```

IF condition to check for empty result:

```
={{ Object.keys($json).length > 0 }}
```

- Returns `true` → row exists, update it
- Returns `false` → row is empty `{}`, create it

Real-World Example: Supabase Channel Tracking

For each Slack channel:

1. **Get a row** (Supabase): look up channel by ID. Always Output Data: **ON**
2. **IF**: `Object.keys($json).length > 0`
   - **TRUE** → row exists → **Update row** with latest data
   - **FALSE** → row doesn't exist → **Insert row** + **Slack notification** (new channel detected)

Without Always Output Data, the FALSE branch would never trigger: items simply vanish when not found.

Nodes Where This Is Most Useful

- Supabase / Postgres "Get a row" or "Select" with 0 results
- HTTP Request returning empty body
- Any lookup that may legitimately return nothing

Critical Caveat: Always Output Data Is Node-Level, Not Per-Item

> **Severity: High.** The manual-upsert pattern above silently breaks when you feed the lookup more than one item at a time.

"Always Output Data" does **not** guarantee one output item per input item. It is a node-level fallback that only fires when the node produces **zero total output across the entire run.** It has no per-item awareness. This is general lookup-node behavior (Supabase `getAll`, Postgres, the Notion node, etc.), not specific to one provider.

Two distinct failure modes when multiple items flow into a single lookup:

Mixed run (some match, some don't):

```
Input: [Post A (exists), Post B (new)]
  → [Get rows]  ← Always Output Data: ON
  → Output: [Post A's row]   ← Post B silently dropped, NO {} placeholder
```

The node produced output for the matches, so the fallback never triggers. Post B does not become an empty `{}`. It just vanishes. The FALSE branch never sees it.

Fully empty run (nothing matches):

```
Input: [Post A (new), Post B (new)]
  → [Get rows]  ← Always Output Data: ON
  → Output: [ {} ]   ← ONE empty object, not one per input item
```

You get a single `{}` for the whole node, not one `{}` per input. Item count collapses from 2 to 1.

Why the Tip #8 Pattern Still Works (and When It Doesn't)

The manual-upsert pattern is only safe when the lookup processes **one item at a time** (inside a loop, or per-item execution). In that mode, "zero total output" and "this item didn't match" are the same condition, so the `{}` fallback correctly stands in for each missing item.

Fan multiple items into one lookup and that equivalence breaks. The per-item FALSE branch can no longer be trusted.

The Fix

Process one item at a time. Keep the lookup inside a loop (or per-item execution) so each "not found" is its own zero-output run, and the `{}` fallback correctly stands in for each missing item. Slower than a single batched query for large inputs, but it keeps the per-item branch logic valid.

---

9. The “Execute Once” Setting

Every n8n node has an **“Execute Once”** toggle in its Settings tab. When ON, the node runs only for the **first input item** regardless of how many items are passed in.

When Execute Once is actually useful:

- Triggering a single notification regardless of how many items came in
- Calling an API that doesn’t depend on item data (e.g. fetching a config once)
- Inserting a single summary record, not one per item

Rule of thumb: If the node references `$json` (i.e. it uses current item data), Execute Once should almost certainly be **OFF**. If it doesn’t reference item data at all, Execute Once may be appropriate.

---

10. Gemini Output Formatting Limitations

> Ref: Why Google Gemini’s Response Schema Isn’t Ready for Complex JSON

Gemini has significant limitations with complex JSON schema definitions in n8n’s Structured Output Parser. These can cause cryptic, hard-to-debug errors.

Known Unsupported Patterns

Nullable fields using JSON Schema union syntax are **not supported**:

```
{ "type": ["string", "null"] }
```

This produces errors like:

- `"Proto field is not repeating, cannot start list"`
- `"Unknown name 'type' at 'path'"`

Workarounds

Use Gemini’s nullable syntax instead:

```
{ "type": "string", "nullable": true }
```

For complex schemas: define the output structure in the prompt text itself rather than in the schema definition, then use a separate downstream step to parse and validate:

```
[Gemini Agent: output defined in prompt]
  → [OpenAI LLM Chain: parse/fix the output schema]
  → [Structured Output Parser]
```

General rule: For anything involving complex nested schemas or nullable fields, route to **OpenAI** instead of Gemini. OpenAI handles these more reliably in n8n.

---

11. Code Node Return Formats

> **Severity: High.** Incorrect return format causes `A 'json' property isn't an object [item 0]` errors.

The return format depends on the Code node's execution mode. Getting this wrong produces runtime errors.

"Run Once for Each Item" Mode

Return a **single object** with a `json` key. Do **not** wrap in an array.

```
// CORRECT: single object
return {
  json: {
    name: "example",
    value: 42,
  },
};
```

```
// WRONG: array wrapper causes errors in this mode
return [
  {
    json: {
```

```
      name: "example",
      value: 42,
    },
  },
];
```

Restrictions in this mode:

- `$input.all()` is **not available**: n8n only exposes the current item
- Use `$input.first()` or `$input.item` to access the current item

"Run Once for All Items" Mode

Return an **array of objects**, each with a `json` key.

```
// CORRECT: array of objects
return [{ json: { name: "item1" } }, { json: { name: "item2"
} }];
```

Both `$input.first()` and `$input.all()` are available in this mode.

Returning Binary Data

When returning binary alongside json, the `json` key must still be a valid object (even if empty):

```
// "Run Once for All Items" mode: array with binary
return [
  {
    json: {},
    binary: {
      data: {
        data: buffer.toString("base64"),
        mimeType: "text/markdown",
        fileName: "output.md",
      },
    },
```

```
  },
];

// "Run Once for Each Item" mode: single object with binary
return {
  json: {},
  binary: {
    data: {
      data: buffer.toString("base64"),
      mimeType: "text/markdown",
      fileName: "output.md",
    },
  },
};
```

Common Pitfall

The error `A 'json' property isn't an object [item 0]` usually means one of:

- The `json` value is `null`, `undefined`, a string, or a number instead of a plain object
- Destructuring failed silently (e.g., `$input.first().binary?.data` returned `undefined` and downstream code put a non-object into `json`)
- Wrong return format for the selected mode (array vs. single object)

Always add safety checks when building json from potentially missing data:

```
const meta = $input.first().binary?.data || {};

return {
  json: {
    fileName: meta.fileName || "unknown",
    mimeType: meta.mimeType || "application/octet-stream",
  },
};
```

12. HTTP Node: Always Return Binary Output

The HTTP Request node can be configured to **always return binary data**, even when the response might otherwise be interpreted as JSON or text. This is useful when downloading files (images, PDFs, etc.) and you need a guaranteed binary output shape for downstream nodes.

How to set it:
In the HTTP Request node, go to **Options** → set **Response Format** to **File** (binary). This forces the node to always output the response as binary data regardless of content type.

Why it matters:
Without this, the HTTP node may return JSON on some responses and binary on others, breaking downstream nodes that expect a consistent binary input. Forcing binary output ensures your workflow handles files predictably: a good complement to the "Always Output Data" setting (Tip #8).

---

13. OpenRouter for Vision: Multi-Model Fallback With One HTTP Call

> Companion to Tip #12 (HTTP binary output). Use this when you want provider-level redundancy on image analysis without wiring up multiple AI nodes.

The Pattern: OpenRouter accepts an array of model IDs in the `models` field and will automatically fall back to the next model in the list if the first one fails, errors, or returns nothing. This gives you provider-level redundancy for vision tasks (or any chat-completion task) with a single HTTP node: no Merge gates, no error branches between models.

Why Use This Instead of Native Gemini / OpenAI Nodes

- **Provider redundancy in one call**: list 3 models, OpenRouter tries them in order
- **Cheap-first, premium fallback**: start with a fast/cheap model, fall back to a stronger one only if needed
- **Mix providers**: combine Qwen, Gemini, Claude, GPT in the same fallback chain

- **Uniform response shape**: OpenAI-compatible API, so `$json.choices[0].message.content` works regardless of which model actually answered

---

Request Body Shape

```
{
  "models": [
    "qwen/qwen3.5-plus-02-15",
    "google/gemini-2.0-flash-lite-001",
    "google/gemini-2.5-flash"
  ],
  "messages": [
    {
      "role": "user",
      "content": [
        { "type": "text", "text": "Describe this image..." },
        { "type": "image_url", "image_url": { "url": "{{ $jso
n.dataUrl }}" } }
      ]
    }
  ],
  "stream": false
}
```

Key points:

- `models` **(array), not** `model` **(string)**: this is what enables the fallback chain
- `content` **is an array, not a string**: vision requests use multi-part content with `type: "text"` and `type: "image_url"` blocks
- `image_url.url` **accepts either an HTTPS URL or a base64 data URL**: data URLs are simpler when the image is already in the workflow as binary

---

Preparing the Image as a Data URL

OpenRouter (following the OpenAI vision spec) expects `image_url` to be a string. A base64 data URL is the cleanest way to pass a binary that's already in your workflow:

```
// Code node: Run Once for Each Item
const buffer = await this.helpers.getBinaryDataBuffer(0, 'dat
a');
const base64String = buffer.toString('base64');
const meta = $input.item.binary.data;

// Derive MIME type from filename: binary metadata's mimeType
can be missing/stale
const ext = (meta.fileName || '').split('.').pop().toLowerCas
e();
const mimeMap = {
  png: 'image/png',
  jpg: 'image/jpeg',
  jpeg: 'image/jpeg',
  gif: 'image/gif',
  webp: 'image/webp'
};
const mimeType = mimeMap[ext] || meta.mimeType || 'image/pn
g';

return {
  json: {
    dataUrl: `data:${mimeType};base64,${base64String}`,
    fileName: meta.fileName,
    mimeType
  }
};
```

Then reference `{{ $json.dataUrl }}` in the HTTP Request body. (See Tip #7 for the full binary-extraction discussion and why `this.helpers.getBinaryDataBuffer()` is the correct way to get bytes off binary metadata.)

HTTP Request Node Configuration

- **Method:** POST
- **URL:** `https://openrouter.ai/api/v1/chat/completions`
- **Authentication:** Predefined credential type → OpenRouter API
- **Headers:** `Content-Type: application/json`
- **Body:** JSON (specify body as JSON, paste the request shape above)
- **On Error:** `continueErrorOutput`: so a hard failure (all models in the list errored) routes to your fallback branch

---

Reading the Response

OpenRouter returns the OpenAI-compatible shape regardless of which model answered:

```
{{ $json.choices[0].message.content }}
```

If you want to know which model actually responded (for logging or cost tracking), it's at `$json.model`.

When OpenRouter's Built-In Fallback Isn't Enough

The `models` array handles provider/model failures gracefully, but it doesn't catch every case. You may still want a downstream Merge-gated fallback (Tip #14) if:

- OpenRouter itself is down (the whole HTTP call fails)
- You want a non-OpenRouter fallback (e.g., a direct Gemini call as a last resort)
- You need to gate on response quality, not just success/failure (e.g., the model returned a refusal or an empty description)

In those cases, pair the OpenRouter `models` array with a Combine-merge fallback to a native node downstream. See Tip #14 for a worked example combining both patterns.

Gotcha: Verify Model IDs Are Current

OpenRouter's model catalog churns. Model IDs are dated snapshots and can be deprecated. Check `https://openrouter.ai/models` periodically and update the array. If the first model in your list is dead, you're paying the latency cost of one failed attempt on every call before fallback kicks in.

---

14. Merge Node Modes: Append vs Combine

> **Severity: High.** Wrong merge mode causes either missing items (branch never fires) or duplicates (fallback runs when it shouldn't).

The Problem: When an IF or Switch node splits a flow into multiple branches and you want to reconverge them downstream, the Merge node's mode determines whether items actually pass through. The right choice depends on whether your branches are **mutually exclusive** (only one fires per item) or **both required** (both must produce data for the merge to make sense).

The Most Common Case: Either/Or Convergence (Append)

The typical pattern: an IF node splits the flow based on some condition, each branch does different processing, and a Merge node downstream collects whichever branch ran so the rest of the workflow can continue.

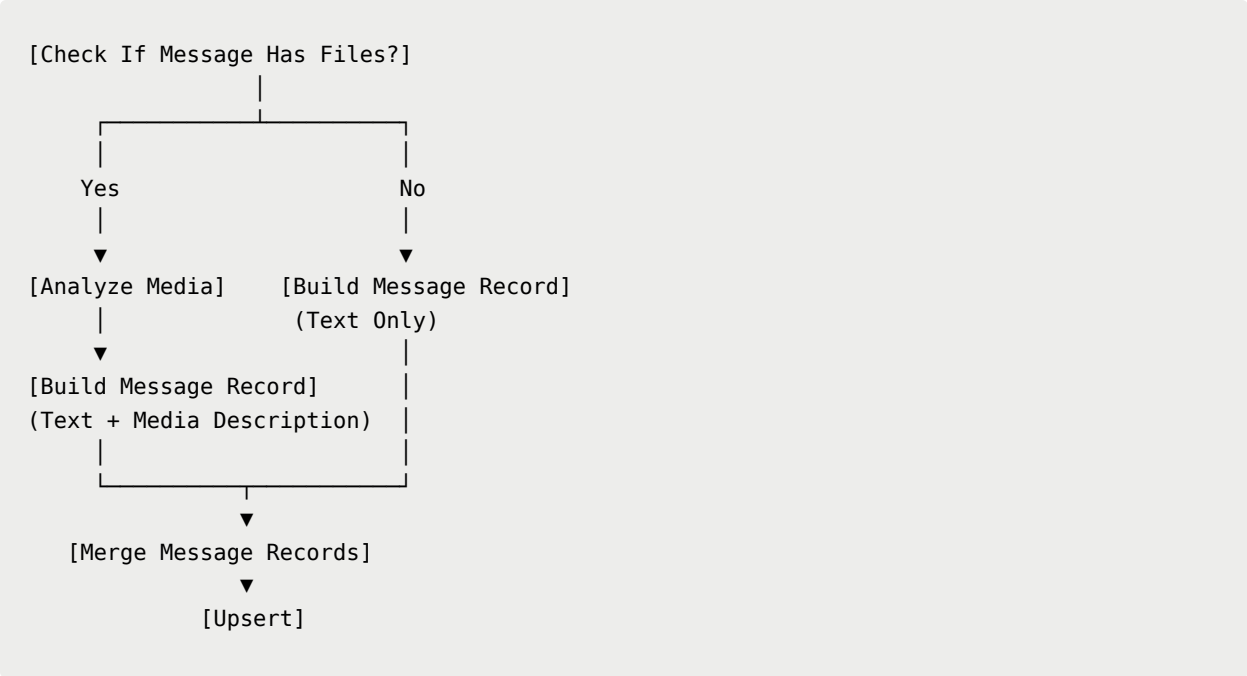
```
[IF: has files?]
  TRUE  → [Process with media analysis] ──┐
  FALSE → [Process text only] ────────────┴→ [Merge: Append] → [Continue...]
```

Because the IF guarantees only ONE branch fires per item, you want the Merge node to pass through whichever branch produced output. That's **Append mode**: it needs only one input to fire and forwards everything that arrived.

Use Append when:

- Branches are mutually exclusive (IF / Switch)
- Each branch produces the same shape of output (or compatible shapes)
- You want "whichever path ran, pass it along"

Real-world example: Slack message capture with conditional media analysis:



Both Set nodes produce the same column shape, so the downstream upsert doesn't care which branch ran. Append is correct.

The Exception: Both Inputs Required (Combine)

Combine mode (all possible combinations) needs **both inputs** to fire. It creates the cartesian product of its inputs. If one input has no data, the Merge produces nothing.

This is the right mode when:

- You're recombining data that was deliberately split (e.g. an error output and the original binary)
- The downstream node needs fields from BOTH branches to function
- You want the merge to act as a gate: only proceed if both sides produced data

Real-world example: OpenRouter vision call with Gemini fallback (combines patterns from Tip #13):

```
Slack Image
    ↓
Download Image
    ↓
```

```
Convert to Base64/Data URL
    ↓
 ┌──────────────────────┐
 │                      │
 ↓                      ↓
OpenRouter Vision   Gemini Vision
 ↓                      ↓
Description A      Description B
 │                      │
 └──────── Merge ───────┘
           ↓
Combined media description
           ↓
Attach to message
           ↓
Save to Supabase
```

Why Combine on the gate:

- On success, the error output has nothing → Combine doesn't fire → Gemini fallback skipped
- On failure, both inputs have data (binary + error) → Combine fires → Gemini runs with the binary it needs
- If you used Append here, the binary alone would fire the merge even on success, and Gemini would run unnecessarily, producing duplicate results.

The final Merge in this flow is Append, because the Success and Fallback paths are mutually exclusive: only one ever produces a description per item.

Want to See It in Action?

Paste this into a fresh n8n canvas to play with both patterns end-to-end. Add your own Slack and OpenRouter credentials to the relevant nodes after import.

▼ Click to expand: Reference workflow JSON

```
{
  "nodes": [
    {
      "parameters": {
        "method": "POST",
```

```
        "url": "https://openrouter.ai/api/v1/chat/completi
ons",
        "authentication": "predefinedCredentialType",
        "nodeCredentialType": "openRouterApi",
        "sendHeaders": true,
        "headerParameters": {
          "parameters": [
            {
              "name": "Content-Type",
              "value": "application/json"
            }
          ]
        },
        "sendBody": true,
        "specifyBody": "json",
        "jsonBody": "={\n\"models\": [\"qwen/qwen3.5-plus-
02-15\",\"google/gemini-2.0-flash-lite-001\",\"google/gemi
ni-2.5-flash\"],\n\"messages\": [\n    {\n\"role\":\"user
\",\n\"content\": [\n        {\n\"type\":\"text\",\n\"text
\":\"Analyze this image:\"\n        },\n        {\n\"type
\":\"image_url\",\n\"image_url\": {\n\"url\":\"{{ $json.da
taUrl }}\"\n          }\n        }\n      ]\n    }\n  ],\n
\"stream\": false\n}",
        "options": {}
      },
      "type": "n8n-nodes-base.httpRequest",
      "typeVersion": 4.4,
      "position": [
        2672,
        2160
      ],
      "id": "d9fceb18-8960-4a39-a4dd-ef4f650d0fca",
      "name": "Get Image Description - OpenRouter",
      "onError": "continueErrorOutput"
    },
    {
```

```
      "parameters": {
        "mode": "combine",
        "combineBy": "combineAll",
        "options": {}
      },
      "type": "n8n-nodes-base.merge",
      "typeVersion": 3.2,
      "position": [
        2960,
        2320
      ],
      "id": "a9c96f20-b911-44f4-b8d5-147c639ef540",
      "name": "Merge [COMBINE]"
    },
    {
      "parameters": {
        "resource": "image",
        "operation": "analyze",
        "modelId": {
          "__rl": true,
          "value": "models/gemini-2.5-flash",
          "mode": "list",
          "cachedResultName": "models/gemini-2.5-flash"
        },
        "text": "=Analyze this image:",
        "inputType": "binary",
        "options": {}
      },
      "type": "@n8n/n8n-nodes-langchain.googleGemini",
      "typeVersion": 1.1,
      "position": [
        3200,
        2320
      ],
      "id": "fc2c1e25-525d-4f75-9da9-76da4aad8270",
      "name": "Gemini Analysis",
```

```
      "onError": "continueRegularOutput"
    },
    {
      "parameters": {
        "assignments": {
          "assignments": [
            {
              "id": "626657af-65f1-4451-be42-8b046d1ba62
e",
              "name": "description",
              "value": "=MEDIA ANALYSIS: {{ $json?.choices
[0]?.message?.content }}",
              "type": "string"
            }
          ]
        },
        "options": {}
      },
      "type": "n8n-nodes-base.set",
      "typeVersion": 3.4,
      "position": [
        3200,
        2144
      ],
      "id": "aa8b314e-4ee7-4183-b877-a3b074c57aa1",
      "name": "Set Description",
      "executeOnce": false
    },
    {
      "parameters": {
        "assignments": {
          "assignments": [
            {
              "id": "626657af-65f1-4451-be42-8b046d1ba62
e",
              "name": "description",
```

```
              "value": "={{ $json?.content?.parts[0]?.text
? `MEDIA ANALYSIS:\\n\\n${$json.content.parts[0].text}`
:\"\" }}",
              "type": "string"
            }
          ]
        },
        "options": {}
      },
      "type": "n8n-nodes-base.set",
      "typeVersion": 3.4,
      "position": [
        3488,
        2320
      ],
      "id": "48748326-af9c-41b2-8284-3a776943bf61",
      "name": "Set Gemini Analysis Description",
      "executeOnce": false
    },
    {
      "parameters": {},
      "type": "n8n-nodes-base.merge",
      "typeVersion": 3.2,
      "position": [
        3760,
        2240
      ],
      "id": "d32747de-1ae7-4c3d-baaf-3f9eeac095ff",
      "name": "Merge [APPEND]"
    },
    {
      "parameters": {
        "url": "={{ $json.url_private_download }}",
        "authentication": "predefinedCredentialType",
        "nodeCredentialType": "slackApi",
        "options": {
```

```
          "response": {
            "response": {
              "responseFormat": "file"
            }
          }
        }
      },
      "type": "n8n-nodes-base.httpRequest",
      "typeVersion": 4.4,
      "position": [
        2112,
        2176
      ],
      "id": "206ae316-a450-4f0a-8173-969b7dbf3fc3",
      "name": "Download Image (Returns Binary)",
      "alwaysOutputData": false
    },
    {
      "parameters": {
        "mode": "runOnceForEachItem",
        "jsCode": "// n8n-recommended approach for getting
binary data\nconst buffer = await this.helpers.getBinaryDa
taBuffer(0, 'data');\nconst base64String = buffer.toString
('base64');\n\n// Grab metadata (still on the object)\ncon
st mimeMeta = $input.item.binary.data;\n\n// Derive MIME t
ype from filename extension\nconst ext = (mimeMeta.fileNam
e || '').split('.').pop().toLowerCase();\nconst mimeMap =
{ png: 'image/png', jpg: 'image/jpeg', jpeg: 'image/jpeg',
gif: 'image/gif', webp: 'image/webp' };\nconst mimeTypeFil
e = mimeMap[ext] || mimeMeta.mimeType || 'image/png';\n\nr
eturn {\n  json: {\n    base64: base64String,\n    fileNam
e: mimeMeta.fileName,\n    mimeType: mimeMeta.mimeType,\n
dataUrl: `data:${mimeTypeFile};base64,${base64String}`\n
}\n};"
      },
      "type": "n8n-nodes-base.code",
```

```
      "typeVersion": 2,
      "position": [
        2416,
        2176
      ],
      "id": "34370473-12e3-43cc-9d82-12ea65b93b30",
      "name": "Generate base64 String for File",
      "onError": "continueErrorOutput"
    }
  ],
  "connections": {
    "Get Image Description - OpenRouter": {
      "main": [
        [
          {
            "node": "Set Description",
            "type": "main",
            "index": 0
          }
        ],
        [
          {
            "node": "Merge [COMBINE]",
            "type": "main",
            "index": 0
          }
        ]
      ]
    },
    "Merge [COMBINE]": {
      "main": [
        [
          {
            "node": "Gemini Analysis",
            "type": "main",
            "index": 0
```

```
          }
        ]
      ]
    },
    "Gemini Analysis": {
      "main": [
        [
          {
            "node": "Set Gemini Analysis Description",
            "type": "main",
            "index": 0
          }
        ]
      ]
    },
    "Set Description": {
      "main": [
        [
          {
            "node": "Merge [APPEND]",
            "type": "main",
            "index": 0
          }
        ]
      ]
    },
    "Set Gemini Analysis Description": {
      "main": [
        [
          {
            "node": "Merge [APPEND]",
            "type": "main",
            "index": 1
          }
        ]
      ]
```

```
    },
    "Download Image (Returns Binary)": {
      "main": [
        [
          {
            "node": "Generate base64 String for File",
            "type": "main",
            "index": 0
          },
          {
            "node": "Merge [COMBINE]",
            "type": "main",
            "index": 1
          }
        ]
      ]
    },
    "Generate base64 String for File": {
      "main": [
        [
          {
            "node": "Get Image Description - OpenRouter",
            "type": "main",
            "index": 0
          }
        ],
        [
          {
            "node": "Merge [COMBINE]",
            "type": "main",
            "index": 0
          }
        ]
      ]
    }
  },
```

```
  "pinData": {}
}
```

Rule of Thumb

Branches are...	Use	Why
Mutually exclusive (IF)	**Append**	Only one fires; merge just forwards whichever ran
Independent, both needed	**Combine**	Both must produce data for downstream to make sense
Error gates (fallback)	**Combine**	Prevents fallback from firing when primary succeeded
Final output collection	**Append**	Collect results from multiple branches regardless

---

Quick Debugging Check

If a Merge node isn't behaving as expected:

- **Missing items?** You may be using Combine where Append is needed. One of your inputs is empty by design (e.g. an IF branch that didn't fire), so Combine produces nothing.
- **Duplicate items?** You may be using Append where Combine is needed. A branch you intended to skip is firing because Append only needs one input.

---

15. Recursive Sub-Workflows: API Pagination Without Loops

The Problem: Some APIs return paginated results with a cursor or `has_more` flag. You don't know how many pages exist upfront, so a fixed loop won't work. Building a loop-back pattern inside a single workflow risks the nested loop issues from Tip #1 if other loops are already present.

The Pattern

An **Execute Workflow** node calls **its own workflow ID**, creating clean recursion. Each execution handles one page and passes the next cursor to itself. An IF node checks the termination condition and either recurses or exits.

```
[Execute Workflow Trigger] → [Fetch Page (HTTP Request)] → [Process Results] → [IF: more pages?]
  TRUE  → [Execute Workflow: THIS workflow's ID, pass next cursor + depth] → [Return combined results]
  FALSE → [Return final results]
```

Each recursive call is its own isolated execution context: no data bleed, no nested loop issues. Results bubble back up through the return chain.

Key Configuration

- **Execute Workflow node**: set the workflow ID to the current workflow's own ID
- **Pass parameters**: send the pagination cursor/offset and a depth counter as input fields
- **Execute Workflow Trigger**: receives cursor and depth, uses them to fetch the next page
- **IF node**: checks both `has_more` (or equivalent) AND depth limit

---

Safety: Always Include a Depth Counter

APIs can return `has_more: true` indefinitely if something goes wrong. Always pass a `depth` parameter that increments each call, and include it in the termination condition:

```
{{ $json.has_more === true && $json.depth < 50 }}
```

Without this, a misbehaving API will trigger infinite recursion: burning through executions and potentially hitting n8n's execution limits.

When to Use This Over a Simple Loop

- **Unknown page count**: you can't set a fixed loop size

- **Workflow already contains a loop**: adding another would create nested loops (Tip #1)
- **Clean separation**: each page fetch is independently debuggable in execution history
- **Complex per-page processing**: each execution can do heavy transforms without accumulating memory

For simple, bounded pagination (known page count, no existing loops), a standard SplitInBatches + Wait loop (Tip #3) is simpler.

---

16. PostgREST Query Params in the Supabase Node's filterString

The Problem: The Supabase node (typeVersion 1) doesn't natively support `order`, `limit`, or other PostgREST query parameters: only filters. So getting the "latest row" or ordering results previously required dropping down to an HTTP Request node hitting the PostgREST API directly.

The Discovery

The Supabase node's `filterType: "string"` option passes the raw string directly to the PostgREST API. You can **append any PostgREST query parameter** to the filter string: not just filter conditions. The node doesn't validate or strip them.

Example: Get Unprocessed Weekly Digests, Ordered by Date

```
{
  "operation": "getAll",
  "tableId": "internal_weekly_slack_digest",
  "returnAll": true,
  "filterType": "string",
  "filterString": "=and=(channel_id.eq.{{$json.channel}},proj
ects_classified.is.null)&order=digest_date.asc"
}
```

This sends a single request that filters AND orders in one shot: entirely within the Supabase node.

What You Can Append

- `&order=column.asc` or `&order=column.desc`: sorting
- `&limit=1`: return only N rows
- `&order=created_at.desc&limit=1`: **"get latest row" pattern**
- `&select=col1,col2`: column selection (reduce payload size)

Important: Set `returnAll: true`

When using `&limit=` in the filterString, set `returnAll: true` on the node. Otherwise the node's own pagination logic may conflict with your manual limit and produce unexpected results.

When to Still Use HTTP Request → PostgREST Directly

- Complex `or()` logic across multiple columns
- Range queries (`Range` header for pagination)
- When you need response headers (e.g., `Content-Range` for total count)
- RPC calls (`/rpc/function_name`)
- Any case where you need full control over the HTTP request

---

17. Supabase Postgres Connections: Always Use the Transaction Pooler

> **Severity: High.** Wrong port = pool exhaustion under real workflow traffic. The smoking-gun error is `EMAXCONNSESSION max clients reached in session mode - max clients are limited to pool_size: 15`.

The Problem: Supabase exposes Postgres through three different connection endpoints. All three "work" in basic testing: the difference only surfaces once you have any concurrency. Pick the wrong one and n8n workflows will silently

degrade, then crash under load. The fix is a one-character change (the port), but it's invisible until production traffic hits.

The Three Endpoints

Connection Type	Host	Port	Use Case
Direct connection	`db.<project>.supabase.co`	5432	Long-running services, migrations. **Not for n8n.**
Session pooler	`aws-0-<region>.pooler.supabase.com`	5432	Persistent sessions, prepared statements. **Not for n8n.**
Transaction pooler	`aws-0-<region>.pooler.supabase.com`	**6543**	n8n, serverless, anything bursty. **Use this.**

The session and transaction poolers share the same hostname. The **port** is the only thing that distinguishes them.

Why n8n Specifically Needs Transaction Mode

n8n's execution model creates a connection per node run, holds it for the duration of that node, and releases. Under any real traffic (Slack triggers, scheduled workflows, webhooks firing in parallel), you'll have many short-lived connections.

- **Session mode**: connection assigned to a client for the entire session. The 15-connection pool exhausts almost immediately once executions overlap.
- **Transaction mode**: connection returned to the pool after each transaction. Same 15-connection pool can serve hundreds of concurrent executions.

The default Supabase pool size (15) is fine for transaction mode. Increasing the pool isn't usually the right fix: switching modes is.

Decoding the Error

```
(EMAXCONNSESSION) max clients reached in session mode - max clients are limited to pool_s
ize: 15
```

- `EMAXCONNSESSION`: Supabase's pooler signaling client connection limit reached
- `session mode`: confirms you're on port 5432 of the pooler
- `pool_size: 15`: the default per-project pool, not the bottleneck (transaction mode handles the same pool size fine)

When you see this error in n8n logs, the diagnosis is always the same: **wrong port**. Don't increase the pool size, don't add retries, don't add Wait nodes between Postgres calls: switch to port 6543.

Where to Find the Right Connection String

Supabase Dashboard → click the **"Connect"** button at the top of any project page. You'll get a modal with three tabs (Direct, Transaction, Session). Copy the transaction pooler string.

Alternatively: Project Settings → Database → "Connection string" section, same tabs there.

Note: the "Connection pooling" settings page (pool size, max clients) does **not** show the connection strings. That page is for configuring pool behavior, not retrieving credentials.

The n8n Credential Setup

For the Postgres credential in n8n:

- **Host**: `aws-0-<region>.pooler.supabase.com` (region matches your Supabase project)
- **Port**: `6543` ← this is the entire point
- **Database**: `postgres`
- **User**: `postgres.<project_ref>`
- **Password**: from Supabase dashboard
- **SSL**: Allow / Require (Supabase enforces TLS)

Trap: PostgREST/Supabase Node Hides This Class of Issue

The native n8n Supabase node calls PostgREST over HTTPS, so this whole class of issue is invisible there. You won't notice the pooling problem until you migrate to the Postgres node. **Plan for the credential setup as part of any Supabase → Postgres node migration (Tip #21).**

Sneakier Symptom: Intermittent Production Failures

Workflows succeed in manual testing but fail intermittently in production. If you see Postgres node errors clustering during high-traffic windows (Slack activity spikes, scheduled batch runs, webhook bursts), the pooler port is the first thing to check: even if the error message isn't exactly `EMAXCONNSESSION`. Pool exhaustion can also surface as generic timeouts or `connection terminated unexpectedly` errors when the pooler kills queued connections.

Edge Cases Where You'd Use Session Mode Instead

Almost never, in an n8n context. Session mode is for:

- Prepared statements (rare in n8n; the Postgres node uses parameterized queries, not prepared)
- `LISTEN/NOTIFY` (you'd use the Postgres Trigger node for this, separate connection lifecycle)
- Long-running transactions spanning multiple queries (uncommon in workflow logic)

If you're not doing any of these, transaction mode is correct.

---

18. Referencing Unexecuted Nodes: The `$if` Guard

The Problem: If an expression references a node that didn't execute in the current run (e.g., it's on a branch that was skipped), n8n throws a warning: *"An expression references this node, but the node is unexecuted."* This can cause unexpected empty values or errors downstream.

The Fix

Use `$if()` with `.isExecuted` to check whether the referenced node actually ran before pulling data from it:

```
{{ $if($("NodeName").isExecuted, $("NodeName").first().json.value, "") }}
```

- If `NodeName` executed → returns its data
- If `NodeName` did **not** execute → returns the fallback (empty string, default value, etc.)

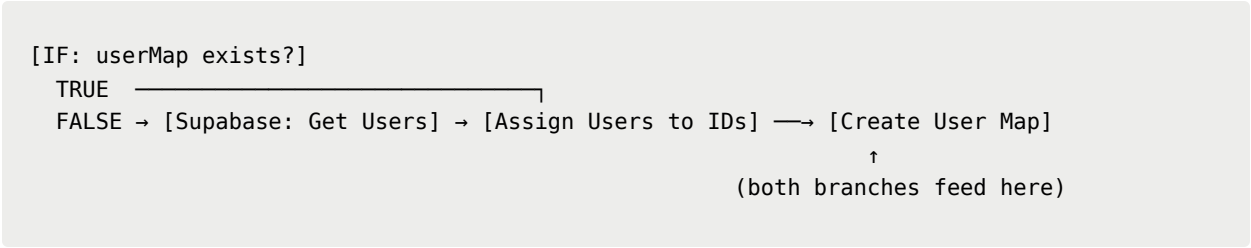
When to Use This

- **After branching logic (IF/Switch)**: when downstream nodes reference data from branches that may not have fired
- **Merge nodes collecting from optional branches**: where one or more inputs may be absent
- **Error fallback patterns**: where you reference both success and error path nodes but only one runs per execution

---

Real-World Example: Conditional Data Merge Without a Merge Node

A common pattern is an IF node that checks whether data already exists on the incoming item. If it does, skip the lookup; if not, fetch and build it. Both branches converge on the **same downstream node**, which uses `$if(.isExecuted)` to resolve which branch actually ran.



The “Create User Map” node uses a single expression to handle both cases:

```
={{
  $('Assign Users to IDs').isExecuted
    ? Object.assign({}, ...$('Assign Users to IDs').all().map(i => i.json))
    : $('Set Date and Channel Details').item.json.userMap
}}
```

- If "Assign Users to IDs" executed (FALSE branch ran) → build the map from the freshly fetched results
- If it didn't (TRUE branch ran) → pull the pre-existing `userMap` from the incoming item

This avoids the Merge node entirely: no worrying about Combine vs Append modes (Tip #14), and no extra nodes. Works well when the data shape is simple enough that a single expression can resolve the "which source?" question.

---

19. General Tips & Gotchas

Fixed-Count Loops (Circuit Breaker Pattern): n8n doesn't have a native "repeat N times" node. Use an IF node + counter: add a `run` field that increments each pass (`={{ ($('If').item.json.run || 0) + 1 }}`), then add an IF condition to exit when `run` reaches your max (e.g., `{{ ($json.run == null) || ($json.run != null && $json.run < 10) }}`). This acts as a circuit breaker to prevent infinite loops. Always include this on any loop: an unterminated loop will crash your instance with an out-of-memory error.

JavaScript expressions are powerful: use them. The expression editor supports full JS. Key examples:

- Time operations: `{{ $now.minus({ hours: 24 }).toISO() }}`
- Array `.map()` and `.filter()` for inline data transformation
- `JSON.stringify($json)` to serialize objects for passing to APIs or prompts

Check for a built-in node before writing a Code node. Sort, Deduplicate, Limit, Merge, Split Out: these cover many cases that feel like they need custom code.

File size limits will crash your instance. Files that are too large cause out-of-memory errors. Process large files outside n8n where possible, and pass references (URLs, paths) rather than file contents through the workflow.

Unterminated loops will crash your instance. Always include a counter circuit breaker in loops (see above). An infinite loop will consume all available memory and bring down the instance.

Error Handling: Continue on Fail Options

When a node fails in n8n, the default behavior is to stop the entire workflow. But you have two "Continue on Fail" options in every node's Settings tab that let the workflow keep running:

Option 1: "Using Error Output": The node gets a separate error output connector (a red port). When the node fails, execution routes through this dedicated error output instead of the main success output. This is ideal for building explicit fallback branches: for example, if an API call fails, route to an alternative provider or send an alert. The success output only fires when the node succeeds, giving you clean branching with no ambiguity.

Option 2: "Using Default Output": The node continues through its normal output, but the failed item's JSON now contains an `error` object with details about the failure. To detect whether the node succeeded or failed, you have two approaches:

- Check for the error directly: `{{ $json.error }}`: if it exists, the node failed
- Check for an expected output key: `{{ $json.expected_key }}`: if it doesn't exist, the node likely failed

Option 2 is useful when you want to handle success and failure in the same branch with a simple IF check, rather than splitting into separate error/success paths.

Preventing Silent Workflow Stops: Guard Against Empty Results

Even when a node doesn't technically "fail," it can return zero results: and this silently stops downstream execution. The most common case is a database lookup (e.g., Supabase "Get a Row") that finds no matching record. The node succeeds (no error), but outputs nothing, and every node after it simply never runs.

The fix: Add an IF node immediately after any lookup that might return empty, and check whether the result is an empty object:

```
{{ Object.keys($json).length === 0 }}
```

- **TRUE** (empty) → Handle the missing data case (create the record, send a notification, use a default value)
- **FALSE** (has data) → Continue with normal processing

This pairs with the "Always Output Data" setting (Tip #8). With Always Output Data enabled, an empty result outputs `{}` instead of nothing, which keeps the item in the flow so your IF node can actually evaluate it. Without that setting enabled, the item vanishes entirely and the IF node never fires.

Common places to add this guard:

- After any Supabase/Postgres lookup
- After HTTP requests that may return empty bodies
- After search operations that might find zero matches
- After any node where "no result" is a valid, expected outcome that you need to handle

---

20. Postgres Node: `queryReplacement` Comma-Splitting Bug

> **Severity: High.** Silent data corruption. Values shift positions and land in the wrong columns. Manifests as cryptic SQL type errors like `invalid input syntax for type double precision: "false"`.

The Problem: The Postgres node's `executeQuery` operation with `queryReplacement` does NOT do smart parsing of expressions. When you write:

```
={{ $json.field1 }},{{ $json.field2 }},{{ $json.field3 }}
```

n8n evaluates the whole expression to a single string, then splits the **resolved string** on every literal comma. If any of those values contains a comma in its data (URL lists, message text with punctuation, full names like "Smith, John", job titles),

each comma in the value becomes a fake parameter boundary. All subsequent parameter positions shift.

What Failure Looks Like

Suppose the attachments field resolves to a list of 4 URLs joined by commas. Your 13-parameter query suddenly has 16 values (4 URLs counted as 4 separate params). Result: `$6` becomes the first URL, `$7` becomes the second URL instead of `channel_id`, everything downstream shifts.

When the shifted parameter lands in a typed column, you get a cast error:

```
invalid input syntax for type double precision: "false"
```

The string `"false"` here was the `is_edited` boolean from `$10`, now landing in position `$12` where the query does `to_timestamp($12::double precision)`.

The error message is misleading: it suggests a type problem, but the root cause is a positional problem upstream.

The Fix: Return a JS Array Instead of a String

Wrap the values in an actual JS array expression:

```
={{ [$json.field1, $json.field2, $json.field3, $json.attachments_list, $json.message_tex
t] }}
```

When n8n's Postgres node sees `Array.isArray(value) === true`, it uses the array entries as positional parameters directly. **No string splitting.** Commas inside any value stay where they belong.

Side-by-Side

```
// BROKEN: string with commas, gets split on resolved commas
queryReplacement: "={{ $json.message_ts }},{{ $json.content
}},{{ $json.attachments }}"
```

```
// CORRECT: JS array expression, used directly
queryReplacement: "={{ [$json.message_ts, $json.content, $jso
n.attachments] }}"
```

When This Bites You Hardest

Any field that can contain a comma in its data:

- **User-generated text**: message content, descriptions, comments, full names, job titles
- **URL lists**: anything that uses `array.join()` defaulting to comma separator
- **CSV-like data**: already-delimited values being passed through

If your workflow ever stores Slack messages, user profiles, or any free-text input, **use the array form by default**. The string form only works safely when every single value is a known non-comma type (IDs, booleans, ISO timestamps, numbers).

When the String Form Is Fine

- All parameters are known to be comma-free (Slack channel IDs, user IDs, booleans, ISO timestamps)
- Quick prototyping with hardcoded values
- Test queries with literal values

Even then, the array form is safer and the syntax is barely different. Prefer the array form universally.

Diagnosing in the Wild

Symptoms that point at this bug:

- `invalid input syntax for type X` errors where the value clearly doesn't match the column type
- Values appearing in unexpected columns (URLs in `channel_id`, timestamps where text should be)

- A query that works in development but fails in production (because dev data didn't contain commas)
- Errors that disappear when you remove the offending field but reappear when you add it back

Quick check: inspect the node's resolved input. If you see your values strung together with commas as one big text blob, you're looking at the string form. Switch to array form.

---

21. Supabase Node → Postgres Node Migration

> **Origin story.** This entry exists because we hit recurring `Service unavailable - try again later` errors on the Supabase node in production. Direct Postgres queries against the same database worked fine. Diagnosis: PostgREST (the HTTP gateway the Supabase node calls) was flaking under load, not Postgres itself.

The Problem: The native n8n Supabase node communicates with the database via PostgREST over HTTPS. PostgREST is a separate service in Supabase's stack and is the flakiest layer. When it has a bad day, your workflows get `Service unavailable - try again later or consider setting this node to retry automatically (in the node settings)` errors. Direct Postgres connections bypass PostgREST entirely and have been rock-solid in our experience.

When to Migrate

Migrate now:

- Workflows currently throwing `Service unavailable` errors from the Supabase node
- New workflows being built (default to Postgres node going forward)
- Workflows needing complex operations PostgREST doesn't handle well: JOINs, CTEs, transactions, `RETURNING` clauses
- Workflows with manual upsert hacks (insert-then-update-on-error pattern from Tip #8): collapse into `INSERT ... ON CONFLICT` with one node
- Bulk operations where performance matters

Leave alone:

- Working Supabase node workflows with no errors: not worth the migration risk
- Workflows that lean heavily on Tip #16's `filterString` PostgREST tricks (translation cost is real, see below)

What You Gain

- **Reliability**: one fewer service in the path. PostgREST is the flakiest layer.
- **Native upserts**: `INSERT ... ON CONFLICT (col) DO UPDATE SET ...` kills the manual upsert hack
- **Full SQL**: JOINs, CTEs, transactions, `RETURNING`, window functions
- **Better bulk performance**: direct connection is noticeably faster for batch ops
- **Cleaner expressions**: column mapping via SQL is more readable than nested PostgREST query params

---

What You Lose

- **The `filterString` PostgREST trick (Tip #16)**: replaced by SQL `WHERE` / `ORDER BY` / `LIMIT`, which is arguably cleaner but requires rewriting any workflow that relied on it
- **Row Level Security**: direct connections authenticate as a database role, RLS policies don't apply. Fine for internal automation, but worth being explicit. Never expose direct Postgres credentials beyond n8n.
- **The visual operation picker**: every read/write is SQL. Less friction for SQL-comfortable users, more friction for others.
- **Migration cost**: every Supabase node needs rewriting

Configuration Trap: Use the Transaction Pooler

The single most important config decision. **See Tip #17 for the full breakdown.** Short version:

- Use `aws-0-<region>.pooler.supabase.com` on **port 6543** (transaction mode)

- NOT port 5432 (direct connection or session mode)
- Default pool size (15) is fine

Get this wrong and the migrated workflow will work in testing then crash under real traffic with `EMAXCONNSESSION`.

Per-Operation Translation Reference

Read all rows:

```
-- Old (Supabase node): operation=getAll, returnAll=true, tab
leId=my_table
-- New (Postgres node, executeQuery):
SELECT col1, col2, col3 FROM my_table;
```

Read with filter and order:

```
-- Old (Supabase node + filterString trick from Tip #16):
-- "and=(status.eq.active,deleted_at.is.null)&order=created_a
t.desc&limit=10"
-- New (Postgres node, executeQuery):
SELECT * FROM my_table
WHERE status = 'active' AND deleted_at IS NULL
ORDER BY created_at DESC
LIMIT 10;
```

Insert with returned ID:

```
-- Old (Supabase node): operation=create, then downstream use
s $json.id
-- New (Postgres node, executeQuery):
INSERT INTO my_table (col1, col2) VALUES ($1, $2) RETURNING i
d, col1, col2;
```

Upsert (the big win):

```
-- Old (Supabase node): manual hack: Insert → IF on error → U
pdate fallback. 3 nodes.
-- New (Postgres node, executeQuery): 1 node.
INSERT INTO my_table (id, col1, col2) VALUES ($1, $2, $3)
ON CONFLICT (id) DO UPDATE SET
  col1 = EXCLUDED.col1,
  col2 = EXCLUDED.col2;
```

Idempotent insert (won't error on duplicate):

```
INSERT INTO my_table (id, col1) VALUES ($1, $2)
ON CONFLICT (id) DO NOTHING;
```

Update by condition:

```
-- Old (Supabase node): operation=update, filters
-- New (Postgres node, executeQuery):
UPDATE my_table SET col1 = $1, updated_at = NOW() WHERE id =
$2;
```

Conditional insert (only if FK row exists):

```
-- Useful when a child table has an FK to a parent that may n
ot be tracked yet
-- (e.g., only log messages for channels that already exist i
n internal_slack_channels)
INSERT INTO child_table (col1, col2, parent_id)
SELECT $1, $2, $3
WHERE EXISTS (SELECT 1 FROM parent_table WHERE id = $3)
ON CONFLICT (col1) DO UPDATE SET col2 = EXCLUDED.col2;
```

Required Prerequisites Before Migrating a Workflow

1. **Postgres credential configured** with transaction pooler (Tip #17). Confirm port `6543`.

2. **Unique constraints exist** on any column you're using as an `ON CONFLICT` target. Check with:

```
SELECT conname, conrelid::regclass, pg_get_constraintdef(oid)
FROM pg_constraint
WHERE conrelid = 'my_table'::regclass AND contype IN ('u', 'p');
```

3. **Backup the workflow** by duplicating it in the n8n UI (right-click → Duplicate). Migrate the duplicate. Swap names when verified.
4. **Map all expressions** that reference `$json.id` or other returned fields from the Supabase node: ensure the new SQL has `RETURNING` to preserve those references.

Parameter Binding Gotcha

When passing values to `executeQuery`, use the **JS array form** for `queryReplacement` (see Tip #20) to avoid the comma-splitting bug:

```
// Safe for any value, including text with commas
queryReplacement: "={{ [$json.field1, $json.field2, $json.message_content] }}"
```

Settings to Preserve Manually

The MCP API can set most node parameters but not all. After creating the new Postgres nodes, manually verify in the n8n UI:

- **Retry On Fail**: re-enable on critical write nodes (was on the original Supabase nodes)
- **Wait Between Tries**: preserve any custom value (typically 5000ms)
- **On Error**: `Continue (using regular output)` for non-critical writes, `Stop workflow` for critical paths

These are under each node's **Settings** tab (not the parameters panel).

When NOT to Migrate

- The workflow is working fine: don’t fix what isn’t broken
- The workflow uses Supabase-specific features (RPC functions called via Supabase node, etc.): these are translatable but the lift might not be worth it
- The team isn’t SQL-comfortable: manageability matters too

The right migration strategy is incremental: new workflows default to Postgres node, broken workflows migrate immediately, working Supabase workflows stay until they break or until a refactor opportunity comes up.