N8n Automation Nodes Cheat Sheet

This cheat sheet provides **JSON configuration snippets** for common n8n nodes, categorized by function. Each snippet shows how to use the node programmatically (as it would appear in a workflow's JSON). Use these examples as templates for building workflows. *(Note: Replace placeholder values like `YOUR_API_KEY` or `YOUR_ID` with real values. Credentials are referenced by name/id after you set them up in n8n.)*

AI & LLM Integrations

OpenAI Chat Model (GPT-3.5/GPT-4): Uses OpenAI's Chat API to generate responses. You specify the model, prompts, and parameters like temperature. This node can take a system prompt and user prompt to generate a completion. For example, to use the GPT-3.5 Turbo model with a system role and user message:

github.com
【46†L17-L24**
json
Copy

```
{
  "name": "ChatGPT",
  "type": "n8n-nodes-langchain.lmChatOpenAi",
  "typeVersion": 1,
  "parameters": {
    "model": "gpt-3.5-turbo",
    "temperature": 0.7,
    "maxTokens": 500,
    "systemPrompt": "You are a helpful assistant.",
    "userPrompt": "Hello! How can I automate tasks with n8n?"
  },
  "credentials": {
    "openAiApi": {
      "id": "YOUR_CRED_ID",
      "name": "OpenAI API"
    }
  }
}
```

Explanation: This config sets the OpenAI Chat node to use the GPT-3.5 model with a system message providing context and a user message. The response will be available as the node's output. Ensure you have an OpenAI API credential named "OpenAI API" configured, referenced in `credentials`. You can adjust `temperature` for randomness and `maxTokens` for length.

OpenAI Text Completion: For GPT-3 text completion models (like `text-davinci-003`), you can use the OpenAI node in completion mode. Provide a prompt and parameters. For example:

docs.n8n.io
【48†L15-L22**
json
Copy

```
{
  "name": "OpenAI Completion",
  "type": "n8n-nodes-langchain.openai",
  "typeVersion": 1,
  "parameters": {
    "model": "text-davinci-003",
    "prompt": "Summarize the following text: {{$json[\"content\"]}}",
    "temperature": 0.5,
    "maxTokens": 200
  },
  "credentials": {
    "openAiApi": {
      "name": "OpenAI API"
    }
  }
}
```

Explanation: This will send a prompt to OpenAI's completion endpoint to summarize some content from the input data. The `{{$json["content"]}}` syntax inserts data from the previous node. The node uses the OpenAI API credentials. The output appears in the node's JSON (`result` field containing the completion text).

AI Agents (with Tools): n8n supports advanced **Agent** nodes that let an AI model use tools (like web search, calculators, or other node functions) to fulfill tasks

docs.n8n.io
. For example, the *OpenAI Functions Agent* can call custom functions, and the *ReAct Agent* uses the ReAct strategy to decide which tool to use at each step. Below is a simplified example

of an AI Agent node configuration using an OpenAI model and a tool (e.g., a Google Search tool):
json
Copy

```
{
  "name": "AI Agent (ReAct)",
  "type": "n8n-nodes-langchain.agent",
  "typeVersion": 1,
  "parameters": {
    "agentType": "react",           // Agent type: "react",
"functions", "planAndExecute", etc.
    "model": "gpt-4",              // Underlying LLM model
    "memory": false,               // Whether to use conversational
memory
    "tools": [
      {
        "name": "googleSearch",    // A connected tool node's name
        "input": "What is n8n?"    // Sample query the agent will use
this tool for
      }
    ]
  }
}
```

Explanation: This agent is configured to use the ReAct strategy with GPT-4. The `tools` array would correspond to actual Tool nodes connected to the Agent node (for example, a Google Search node in the workflow). In practice, you add tools via the editor UI (they become sub-nodes). The agent will decide when to use the tool. For instance, it might use the Google Search tool to retrieve information needed to answer a question. **Note:** The agent node cluster handles the logic; ensure you have the appropriate tool nodes and credentials (like Google API keys) set up. Agents can also use other modes like **OpenAI Functions Agent**, **Plan-and-Execute Agent**, or **SQL Agent**, each allowing the AI to perform specific complex tasks (e.g., call defined functions, break a task into sub-tasks, or run SQL queries via provided DB credentials).

Embeddings and Vector Stores: n8n provides nodes to generate text embeddings with models (OpenAI, Cohere, Google PaLM, etc.) and to store/retrieve them in vector databases (Pinecone, Weaviate, etc.). For example, the **OpenAI Embeddings** node can take text and return a vector. Usage is straightforward: you specify the text field to embed and the model. The

output is usually an array of numbers representing the vector. These can be used with a **Vector Store** node (like Pinecone) to `add` or `query` vectors. Example for an OpenAI Embedding node:

json
Copy

```
{
  "name": "Text to Vector",
  "type": "n8n-nodes-langchain.embeddingsOpenAi",
  "typeVersion": 1,
  "parameters": {
    "model": "text-embedding-ada-002",
    "text": "={{ $json[\"content\"] }}"
  },
  "credentials": {
    "openAiApi": {
      "name": "OpenAI API"
    }
  }
}
```

Explanation: This takes the `content` field from input JSON and generates a 1536-dimensional embedding using OpenAI's ada model. You would typically send this vector to a storage or use it in a similarity search. For storing, a **Pinecone** (or other vector DB) node can be used, with operations like `Insert Vector` or `Query Vector` (you provide the index name, vector data, and any metadata or query vectors as needed). Because vector store usage can be complex, refer to the specific node docs for exact parameter names. The key concept is that **Embeddings nodes** convert text to numerical vectors, and **Vector Store nodes** allow saving and searching those vectors, enabling workflows like semantic search or retrieval-augmented generation.

API & Webhook Integrations

HTTP Request Node (REST APIs): This is the universal node to call any RESTful API. You can configure method, URL, headers, query params, and body (including JSON body). It supports authentication via credentials (OAuth2, API token, etc.) or custom headers. Below is an example GET request with query parameters:

community.faros.ai

community.faros.ai
json

Copy

```
{
  "name": "HTTP GET Example",
  "type": "n8n-nodes-base.httpRequest",
  "typeVersion": 1,
  "parameters": {
    "url": "https://api.example.com/data",
    "method": "GET",
    "responseFormat": "json",
    "queryParametersUi": {
      "parameter": [
        { "name": "q", "value": "search term" },
        { "name": "limit", "value": "10" }
      ]
    }
  }
}
```

To perform a POST with a JSON body, use `jsonParameters: true` and provide the `bodyParametersJson` object (or form data, etc.). For example, a POST request:

json
Copy

```
{
  "name": "HTTP POST Example",
  "type": "n8n-nodes-base.httpRequest",
  "typeVersion": 1,
  "parameters": {
    "url": "https://api.example.com/posts",
    "method": "POST",
    "responseFormat": "json",
    "jsonParameters": true,
    "options": {
      "bodyContentType": "json"
    },
    "bodyParametersJson": {
      "title": "Hello World",
      "content": "This is an example post via n8n."
    }
```

```
  },
  "credentials": {
    "httpBasicAuth": {
      "name": "Example API Auth"
    }
  }
}
```

Explanation: This POST will send a JSON payload with `title` and `content`. We set `bodyContentType` to JSON to ensure the `Content-Type: application/json` header. The credentials section (optional) can reference an HTTP Basic Auth, OAuth2, or API Key credential if needed for the API. The HTTP node is very flexible – you can also use it for GraphQL by sending a GraphQL query in the body or using the dedicated GraphQL node.

GraphQL Node: The GraphQL node is a convenience wrapper for GraphQL APIs. You specify the endpoint URL and the GraphQL query or mutation. It's essentially similar to an HTTP POST but simplifies sending the query. For example:

community.faros.ai

community.faros.ai
json
Copy

```
{
  "name": "GraphQL Query",
  "type": "n8n-nodes-base.graphql",
  "typeVersion": 1,
  "parameters": {
    "endpoint": "https://api.spacex.land/graphql/",
    "query": "query Launches($limit:Int!) {\n  launchesPast(limit:
$limit) {\n    mission_name\n    launch_date_utc\n  }\n}",
    "variables": "{ \"limit\": 3 }",
    "headerParametersUi": {
      "parameter": [
        { "name": "Authorization", "value": "Bearer YOUR_API_TOKEN" }
      ]
    }
  }
}
```

Explanation: This will query the SpaceX GraphQL API for the last 3 launches. We included a bearer token in headers (if the API required auth). The result will be JSON in the node's output. The GraphQL node automatically sends the request as JSON to the endpoint. If needed, you can also pass dynamic data by using n8n expressions inside the `query` or `variables`.

Webhook Trigger: The Webhook node starts a workflow via an incoming HTTP request. You provide a `path` (which becomes part of the URL n8n listens on) and optional HTTP method, authentication, and response settings. Example of a basic webhook that triggers on a POST request:

community.n8n.io

community.n8n.io
json
Copy

```
{
  "name": "Webhook Trigger",
  "type": "n8n-nodes-base.webhook",
  "typeVersion": 1,
  "parameters": {
    "path": "incoming-data",
    "methods": ["POST"],
    "responseMode": "onReceived",
    "responseData": {
      "statusCode": 200,
      "body": "Webhook received successfully"
    }
  }
}
```

Explanation: This creates a webhook at URL `/webhook/incoming-data` (the full URL is shown in n8n UI, including host and unique ID if not using a custom URL). It listens for POST requests. `responseMode: onReceived` means n8n will immediately send a response as soon as the request is received (without waiting for the whole workflow to finish), and we provide a static 200 OK response body. If you set `responseMode` to **"lastNode"** (default), the webhook will wait and respond with whatever the last node in the workflow returns. Use that mode if you want to return processed data. You can access incoming request data in the subsequent nodes: query parameters and body will be available in the JSON (e.g., `{{$json["body"]["fieldName"]}}`). This is ideal for creating webhooks for services like Stripe, GitHub, etc., or for custom integrations.

Respond to Webhook: When using the default **lastNode** response mode for webhooks, the final node's output becomes the response. But if you need to craft a custom response or send a reply mid-workflow, n8n offers a **Respond to Webhook** node. This node lets you explicitly return a status, headers, and body. For example, to return a JSON payload:

json
Copy

```
{
  "name": "Webhook Response",
  "type": "n8n-nodes-base.respondToWebhook",
  "typeVersion": 1,
  "parameters": {
    "responseMode": "responseNode",
    "options": {
      "responseData": {
        "body": "{ \"success\": true, \"message\": \"Processed\" }",
        "headers": {
          "Content-Type": "application/json"
        },
        "statusCode": 200
      }
    }
  }
}
```

Explanation: Place this node in your workflow where you want to terminate and respond to the waiting webhook caller. After this node executes, n8n sends the specified response and will not wait for further nodes (downstream nodes won't run). If you want to keep the workflow running *after* responding, use the option **"respond and continue"**. In most simple cases, you don't need this node and can just let the workflow end with the desired output, but it's useful for advanced control (e.g., send an immediate acknowledgment and then continue processing data in the background).

Data Processing & Logic Nodes

Code (Function) Node: The Code node (formerly the Function node) allows you to write custom JavaScript (and in recent versions, optionally Python) to manipulate data. It can operate on incoming items or generate new data. The JSON snippet for a Code node includes your code as a string in `functionCode`. For example, a code node that adds a new field to each item:

community.faros.ai
json
Copy

```
{
  "name": "Add Field",
  "type": "n8n-nodes-base.function",
  "typeVersion": 1,
  "parameters": {
    "functionCode": "for (const item of items) {\n  item.json.newField
= item.json.someField + 100;\n}\nreturn items;"
  }
}
```

Explanation: This JavaScript code loops through each input item (`items` is an array of objects `{ json: {...}, binary: {...} }`) and appends a new field. Here we set `newField` to `someField + 100` as an example calculation. The node must `return items;` at the end. After this runs, downstream nodes see the modified items. You can log to the console or perform more complex transformations. If you set **No Inputs**, the code node can act as a data generator (e.g., create an array of objects from scratch). The Code node is extremely powerful for any logic not covered by other nodes. *(Note: n8n also supports a separate **Code (Python)** mode if enabled, where you can write Python code. In JSON, this would include `"language": "python"` and the code in a similar field.)*

IF Node (Conditional): The IF node routes data based on conditions (true/false). It has two outputs: *true* (first output) and *false* (second output). In the JSON, you define `conditions`. You can compare numbers, strings, booleans, etc., and combine conditions with AND/OR (`combinator`). Here's an example IF node that checks if the field `status` equals "success":

community.n8n.io

community.n8n.io
json
Copy

```
{
  "name": "Check Status",
  "type": "n8n-nodes-base.if",
  "typeVersion": 2,
  "parameters": {
    "conditions": {
      "conditions": [
```

```
        {
          "leftValue": "={{ $json[\"status\"] }}",
          "rightValue": "success",
          "operator": {
            "type": "string",
            "operation": "equals"
          }
        }
      ],
      "combinator": "and"
    }
  }
}
```

Explanation: This IF node will output any incoming item to the **true** branch if the item's JSON `status` field is the string "success". All other items go to the **false** branch. The JSON structure under `parameters.conditions` can include multiple rules; here we have one rule comparing a string. `leftValue` uses an expression to pull the item's `status`. We set the operator type to "string" and operation "equals". The IF node supports other operations (e.g., *contains*, *greater*, *less*, *regex matches*, etc.) and data types (number, boolean, date, etc.). The `combinator` is used if you have multiple conditions – for example, you could set two conditions and use `"combinator": "and"` to require both true. **Important:** The IF node does not filter out items entirely; it *routes* them. So items always go to either the true output or false output. If you want to stop items, you might follow with other logic (or use the IF in combination with the Merge node to filter, as described below).

Switch Node (Multiple Routing): The Switch node is like a multi-IF or a switch/case statement. It lets you define multiple rules and up to … outputs. Each incoming item goes to the first matching output (or all matching, if configured), or to a fallback if no rule matches. For example, a Switch that routes based on a `category` field:

github.com

github.com
json
Copy
```
{
  "name": "Route by Category",
  "type": "n8n-nodes-base.switch",
  "typeVersion": 1,
```

```
  "parameters": {
    "dataType": "string",
    "value1": "={{ $json[\"category\"] }}",
    "rules": {
      "rules": [
        { "operation": "equal", "value2": "support" },   // Output 0
if category == "support"
        { "operation": "equal", "value2": "sales" }      // Output 1
if category == "sales"
      ]
    },
    "fallbackOutput": 2
  }
}
```

Explanation: This Switch node looks at the `category` value. If it equals "support", the item goes out via output 0 (first output); if "sales", via output 1; anything else falls through to output 2 (the fallback). We set `dataType: "string"` since we are comparing strings. The `rules` array defines two cases. `fallbackOutput: 2` means we have configured a third output for "none of the above". In the editor, you would set **Number of Outputs** to 3 in this case. You can add more rules for additional outputs. The operations can be equality, contains, greater than, etc., similar to IF. You can also switch on numbers, booleans, dates, or even use a JavaScript expression mode to direct items to an output index. The Switch node is useful for branching logic in workflows (e.g., handling different event types or categories differently).

Set Node (Edit Fields): The Set node allows you to add, remove, or rename fields in the JSON data without coding. It's often used to prepare or clean data. In JSON, you configure an `assignments` list for new values and you can toggle options to keep or delete other fields. Here's an example that sets two new fields and keeps existing data:

community.n8n.io

community.n8n.io
json
Copy
```
{
  "name": "Set Fields",
  "type": "n8n-nodes-base.set",
  "typeVersion": 3,
  "parameters": {
```

```
    "keepOnlySet": false,
    "values": {
      "number": [
        {
          "name": "year",
          "value": 2025
        }
      ],
      "string": [
        {
          "name": "statusMessage",
          "value": "Processed by n8n"
        }
      ]
    }
  }
}
```

Explanation: This Set node will add a numeric field `year` with value `2025` and a string field `statusMessage` with a static text. Because `keepOnlySet` is false, it will **retain all existing fields** from input items and just append these two. If `keepOnlySet` were true, the output would only have the fields we explicitly set (useful if you want to discard other data). The Set node can handle multiple data types (notice we used a number and a string; you could also set boolean, date, etc. using the respective sections). If you want to remove certain fields, you can use the **Remove Fields** option (in JSON, there's a `options.removeFields` you could list). The Set node is handy for mapping data before sending to an API or after receiving data to simplify output.

Merge Node: The Merge node takes two input streams and combines them. It can work in different modes: **Append** (just concatenates items), **Wait** (waits for both inputs then output together), **Merge By Index** (pair items one-to-one by their index), or **Merge By Key** (match items from two inputs on a key value). The JSON structure will have `mode` and possibly a `propertyName` or `joinBy` depending on mode. Example of using Merge in "By Key" mode to join data from two sources on an `id` field:

json
Copy

```
{
  "name": "Merge on ID",
  "type": "n8n-nodes-base.merge",
```

```
  "typeVersion": 1,
  "parameters": {
    "mode": "mergeByKey",
    "propertyName": "id",
    "outputDataFrom": "both"
  }
}
```

Explanation: This Merge is set to **mergeByKey** on the field `id`. It expects that each input has items with an `id` property. It will output a single stream of merged items: each output item combines the JSON from input1 and input2 where the `id` matched. The `outputDataFrom: "both"` means the output item will include fields from both inputs (you could also choose to output only data from one side, with the other just used to match). For using Merge, connect one node to Input 1 and another to Input 2 of this node. If an `id` from one side doesn't find a match on the other, that item can be dropped or passed through depending on additional options (`options.outputMissing...`). Other modes: **Append** simply concatenates Input1 and Input2 items (one after the other), **Wait** will pause until both inputs have executed (then output both sets; useful to synchronize parallel branches), **Merge By Index** takes item 0 from Input1 with item 0 from Input2, item 1 with item 1, etc., combining their JSON (you'd typically only use that if both inputs are same length and order). The Merge node is crucial for more complex flows where data splits and needs to come back together.

Item Lists (Aggregate) Node: The Item Lists node (previously called **Aggregate** in some contexts) helps manipulate arrays of items – for example, splitting an array into individual items or aggregating multiple items into one array. One common use is to **split** a single item that contains an array into separate items (each with one element of the array). Conversely, it can **aggregate** many items into a single array. A typical configuration for splitting might look like:

json
Copy

```
{
  "name": "Split Array",
  "type": "n8n-nodes-base.itemLists",
  "typeVersion": 1,
  "parameters": {
    "operation": "splitIntoItems",
    "property": "results"
  }
}
```

Explanation: If an incoming item has a field `results` which is an array (e.g., from an HTTP response or a previous computation), this node will output each element of `results` as a separate n8n item. The `operation: "splitIntoItems"` and specifying the property to split on does this. The opposite can be done with `operation: "aggregateItems"` which can collect all input items' data into a single array on one item (you specify how to aggregate, like collect all values of a field into an array). There are also other list operations (like removing duplicates, sorting items by a field, etc.). The Item Lists node is very useful for managing array data without writing code – for instance, splitting an API response that returned a list of records into individual items for further processing.

Looping (Split/Batches): n8n doesn't use traditional loops; instead, it processes multiple items in parallel through nodes. To explicitly loop a certain way, you can use the **Split In Batches** node (called "Loop Over Items" in UI). This node allows you to process items in batches and iterate. JSON example to process 10 items at a time:

json
Copy

```
{
  "name": "Batch Loop",
  "type": "n8n-nodes-base.splitInBatches",
  "typeVersion": 1,
  "parameters": {
    "batchSize": 10
  }
}
```

Explanation: Connect the **SplitInBatches** node in your flow where you want to throttle or loop through items. In the first run, it will pass the first 10 items forward and hold the rest. At the end of the loop (you must connect the last node in the loop back into the SplitInBatches node's **input2**), the SplitInBatches node will send the next batch when triggered from that second input. This essentially creates a loop: after the last node, connect it back to the SplitInBatches (select "Execute Next Batch" input). This is advanced usage for scenarios where you need to avoid processing all items at once (e.g., rate limiting API calls or handling large lists chunk by chunk). The loop ends when no more items remain; you can detect that using a Run IF connected to the "No Items" output of SplitInBatches (or simply let the workflow end after no more batches).

Error Handling (Try/Catch): For automation, you might want to handle errors gracefully. n8n has an **Error Trigger** node that can catch workflow errors globally, and a **Continue On Fail** option per node. While not a typical "node" to call programmatically, note that in the workflow JSON you can set `"continueOnFail": true` on any node's parameters to prevent a node failure from stopping the workflow. Additionally, you can use the **Stop And Error** node to

deliberately throw an error with a custom message (to, say, halt execution based on a condition and mark workflow as failed). A Stop And Error node JSON would look like:

json
Copy
```
{
  "name": "Stop on Condition",
  "type": "n8n-nodes-base.stopAndError",
  "typeVersion": 1,
  "parameters": {
    "message": "Terminating workflow due to business rule X"
  }
}
```

If this node executes, it will stop the workflow and produce an error with the given message. Use it after an IF or other check if you want to gracefully stop when something isn't right (instead of continuing). For catching errors, the **Error Trigger** is placed in a separate workflow; when any workflow errors, it can capture details and, for example, send an alert via email or Slack.

Workflow Automation & Triggers

Cron Trigger (Schedule): The Cron node (also called Schedule Trigger) starts workflows on time-based schedules. You can configure it to run at fixed times (every day at X, every week on Y, etc.) or at intervals. JSON configuration has a `triggerTimes` array. For example, to run every day at 9:00 AM and 5:00 PM:

community.n8n.io

github.com
json
Copy
```
{
  "name": "Daily Schedule",
  "type": "n8n-nodes-base.cron",
  "typeVersion": 1,
  "parameters": {
    "triggerTimes": {
      "item": [
        { "hour": 9, "minute": 0 },
        { "hour": 17, "minute": 0 }
```

```
      ]
    }
  }
}
```

Explanation: This sets two trigger times (Cron will handle both). The first object is 9:00, second is 17:00 (5 PM). By default, if you only specify hour/minute, it runs every day at that time. You can add `"weekday": ["Monday","Tuesday",...]` or `"dayOfMonth": [...]` to refine the schedule. Alternatively, Cron can be set in simple modes: `"mode": "everyMinute"` or `"mode": "everyX"` with a value and unit. For example, to trigger every 15 minutes: `{"item": [ { "mode": "everyX", "value": 15, "unit": "minutes" } ] }`. Cron triggers are great for daily reports, routine data syncs, etc. This node has no input; it simply fires on schedule and passes an empty item to start the workflow.

Email Trigger (IMAP): The Email Trigger node watches an IMAP inbox for new emails. Configuration includes server, email credentials, search criteria, etc. In JSON, it appears with those details (which are mostly sensitive like host, port, etc., so we won't list a full example here). Key fields: `"mailbox": "INBOX"` (or label/folder), `"criteria": "UNSEEN"` (to get unread emails), and your IMAP credentials reference. When triggered, the node outputs email data (subject, body, attachments as binary). Example snippet (with placeholders):

json
Copy

```
{
  "name": "Email Trigger",
  "type": "n8n-nodes-base.emailReadImap",
  "typeVersion": 1,
  "parameters": {
    "mailbox": "INBOX",
    "postProcessAction": "read",
    "options": {
      "criteria": "UNSEEN"
    }
  },
  "credentials": {
    "imap": {
      "name": "My Email Account"
    }
  }
}
```

Explanation: This checks the INBOX of the configured email account for unseen messages and marks them as read (`postProcessAction: "read"`). For each new email, it triggers the workflow with the email content. Use this for automations like parsing incoming support emails or lead notifications. You can combine it with an IF node to filter emails by subject, etc., and then route them (e.g., create tickets, send alerts, etc.).

Manual Trigger: The Manual Trigger is simply a node to start the workflow by clicking “Execute Workflow” in the editor. It has no parameters in JSON beyond defaults. Example:

json
Copy

```
{
  "name": "Manual Trigger",
  "type": "n8n-nodes-base.manualTrigger",
  "typeVersion": 1,
  "parameters": {}
}
```

Explanation: You typically wouldn’t include this in an exported workflow JSON if you plan to run it automatically, but it’s useful during development. It produces one empty item to kick off the flow.

Workflow Trigger (Execute Workflow): n8n allows one workflow to call another. There are two nodes for this: **Execute Workflow** (in the caller workflow) and **Workflow Trigger** (in the callee workflow). In the sub-workflow that is being called, you use a **Workflow Trigger** node to receive the call (essentially acts like Webhook but internal). In the parent workflow, you use **Execute Workflow** to invoke. Example of an Execute Workflow node that calls a sub-workflow by ID and waits for result:

community.n8n.io
【50†L1432-L1440**
json
Copy

```
{
  "name": "Run Sub-workflow",
  "type": "n8n-nodes-base.executeWorkflow",
  "typeVersion": 1,
  "parameters": {
    "workflowId": "123",       // The ID or name of the workflow to
execute
```

```
    "waitForCompletion": true,
    "inputs": {
      "inputData": "={{ $json[\"data\"] }}"
    }
  }
}
```

Explanation: This will execute the workflow with ID `123` (you can find a workflow's ID in its URL or list). It passes an input field `inputData` to the sub-workflow (the sub-workflow's **Workflow Trigger** node should be configured to accept that field). If `waitForCompletion` is true, the parent workflow pauses until the sub-workflow finishes, then resumes with whatever output the sub-workflow returned. If false, it triggers the other workflow and immediately continues (fire-and-forget mode). Use this to reuse common routines across workflows or to split complex processes. For instance, you might have a sub-workflow that takes data and writes to Google Sheets, which you can call from multiple other workflows instead of duplicating those nodes. **Note:** Ensure the sub-workflow has a **Workflow Trigger** node (which in JSON would simply be `"type": "n8n-nodes-base.workflowTrigger"` with any defined input schema). The outputs of the sub-workflow become the output of the Execute Workflow node.

External Hooks (n8n Trigger): There's also an **n8n Trigger** node that can fire when certain events in n8n happen (like when a workflow is activated, or a user triggers it via API). For most users, this is less commonly used, but it's good to know it exists for advanced orchestration.

Notable Integrations (Apps & Services)

(Below are examples of popular third-party service nodes with JSON usage. Each of these nodes requires appropriate credentials (set up in n8n's Credentials and referenced by name or ID in the JSON). The exact fields depend on the service's API.)

Slack Node (Send Message): The Slack node lets you post messages, get info about channels, manage files, etc., through Slack's API. A common operation is sending a channel message. For instance, to post a message to a channel:

community.faros.ai
json
Copy
```
{
  "name": "Send Slack Message",
  "type": "n8n-nodes-base.slack",
```

```
  "typeVersion": 1,
  "parameters": {
    "resource": "message",
    "operation": "send",
    "channel": "C01234567",  // channel ID or name
    "text": "Hello from n8n :tada:"
  },
  "credentials": {
    "slackApi": {
      "name": "Slack OAuth2"
    }
  }
}
```

Explanation: This will send the text "Hello from n8n 🎉" to the specified Slack channel. We chose `resource: "message"` and `operation: "send"`. Slack nodes often have multiple resources like message, channel, etc. The channel can be the Slack channel ID or name (if using name, ensure the credential has proper scopes to find it). The `credentials` points to a Slack OAuth2 credential (with scopes like `chat:write` etc. as needed

community.faros.ai
). If you want to use blocks or attachments, the Slack node allows a JSON mode for those fields (`jsonParameters: true` and provide a JSON object for attachments or blocks). The output of this node will typically include the Slack message ID and timestamp if successful. You can also use **Slack Trigger** node (not shown here) to listen for Slack events if you set up a Slack app webhook – useful for reactive workflows (e.g., respond when a message is posted).

Google Sheets Node: This node integrates with Google Sheets to read or write spreadsheet data. Operations include: *Read rows*, *Append row*, *Update row*, *Delete row*, *Lookup* etc. You must have Google Sheets credentials (OAuth2) set up. Here's an example to **append a new row** of data to a sheet:

github.com

github.com
json
Copy
```
{
  "name": "Append to Sheet",
  "type": "n8n-nodes-base.googleSheets",
  "typeVersion": 4,
```

```
  "parameters": {
    "operation": "append",
    "spreadsheetId": "1A2b3C4D5E6FgHiJkLMnoPQrstu",  // Google Sheet
document ID
    "sheetName": "Sheet1",
    "dataMode": "autoMap",
    "options": {
      "valueInputMode": "USER_ENTERED"
    }
    // When dataMode is autoMap, no need to specify values here;
    // it will take incoming item fields matching column names.
  },
  "credentials": {
    "googleSheetsOAuth2Api": {
      "name": "Google Sheets OAuth2"
    }
  }
}
```

Explanation: This is configured to append a new row to the sheet named "Sheet1" in the Google Sheets document with the given ID. We used `dataMode: "autoMap"`, which means the node will automatically map incoming fields to columns with the same header name. For example, if the incoming items have JSON like `{ "Name": "Alice", "Email": "alice@example.com" }`, and the Google Sheet has columns "Name" and "Email", those values will be placed accordingly. `valueInputMode: "USER_ENTERED"` tells Google Sheets to treat the input as if a user typed it (so formulas in cells will recalc, etc.). If you wanted to explicitly map fields, you could use `dataMode: "define"` and then provide a list of fields to send. For reading data, the **Get Many** (or `operation: "getAll"`) would require a range (e.g., "Sheet1!A:D") and returns an array of rows. The Google Sheets node is powerful for both exporting data from n8n to spreadsheets and importing data from spreadsheets into n8n for further processing.

Notion Node: The Notion node connects to Notion's API, allowing you to create or update pages in a database, retrieve database items, or append content to pages. For example, to **query a Notion database** for pages that match a filter, you might use the *Database Page: Get All* operation with a filter condition:

community.n8n.io

community.n8n.io

json
Copy

```
{
  "name": "Query Notion DB",
  "type": "n8n-nodes-base.notion",
  "typeVersion": 2,
  "parameters": {
    "resource": "databasePage",
    "operation": "getAll",
    "databaseId": "YOUR_NOTION_DATABASE_ID",
    "options": {
      "filter": {
        "singleCondition": {
          "key": "Email|email",
          "condition": "equals",
          "emailValue": "={{ $json[\"email\"] }}"
        }
      }
    }
  },
  "credentials": {
    "notionApi": {
      "name": "Notion API"
    }
  }
}
```

Explanation: This will fetch all pages from the specified Notion database where the Email property equals the `$json["email"]` value from the previous node. In Notion's API, filters can be complex; here we used a simple single-condition filter on an Email property. The `key` is formatted as `PropertyName|propertyType` in the node (the Notion node needs the property type to format the filter correctly, hence "Email|email"). The Notion node supports creating pages (you'd specify properties to set), updating pages, searching, etc. For a **Create** example, you'd use `operation: "create"` with a `pageId` (if adding to a page) or `databaseId` (if adding to a database) and provide the properties object. e.g., for a database, `properties: { "Name": {"title": [{"text": {"content": "New Item"}}]}, "Status": {"select": {"name": "Done"}} }`. The structure follows Notion API JSON. The node simplifies some of this, but often you use the **No Code** approach to set fields via UI. When working programmatically, referencing Notion's API docs for exact JSON of properties is helpful. The

output of a Notion node will be the JSON representation of the Notion page or database entries retrieved. Use this integration to automate adding meeting notes, updating task statuses, or generating dashboards in Notion.

GitHub Node: This node allows interactions with GitHub, such as creating issues, retrieving commits, managing repositories, etc. As an example, to **create a new issue** in a GitHub repository:

json
Copy

```
{
  "name": "Create GitHub Issue",
  "type": "n8n-nodes-base.github",
  "typeVersion": 1,
  "parameters": {
    "resource": "issue",
    "operation": "create",
    "owner": "octocat",             // GitHub username or org
    "repository": "Hello-World",    // Repo name
    "title": "Automated Issue from n8n",
    "body": "This issue was created by an n8n workflow."
  },
  "credentials": {
    "githubApi": {
      "name": "GitHub personal access token"
    }
  }
}
```

Explanation: This uses the GitHub node to create an issue in the `octocat/Hello-World` repository. The `githubApi` credential should be a Personal Access Token with repo permissions (or OAuth app token). The node could also update or read issues (different operations), list commits (`resource: "repository", operation: "getCommits"` for example), manage pull requests, etc. The output for create operations usually contains the data of the created object (issue details including its number, URL, etc.). This is useful for automation like logging errors or TODOs as GitHub issues, or posting deployment notes to a repo.

Database Nodes (MySQL, Postgres, etc.): n8n includes nodes for popular databases like MySQL, PostgreSQL, MSSQL, etc. These nodes let you run queries or operations (select/insert/update). A common usage is to run a custom SQL query. For instance, using the MySQL node to execute a query:

community.n8n.io
json
Copy

```
{
  "name": "MySQL Query",
  "type": "n8n-nodes-base.mySql",
  "typeVersion": 2,
  "parameters": {
    "operation": "executeQuery",
    "query": "SELECT * FROM users WHERE id = {{ $json[\"user_id\"]
}}",
    "additionalFields": {}
  },
  "credentials": {
    "mySql": {
      "name": "My MySQL DB"
    }
  }
}
```

Explanation: This will run the given SQL query on the connected MySQL database (using the credentials named "My MySQL DB"). We used an expression to inject an incoming `user_id` into the query. The result will be returned as items (each row as an item with columns as fields). You could also use `operation: insert` and specify table and column data in a structured way, but often raw SQL (with `executeQuery` or `execute`) is simplest for complex operations. Make sure your queries are safe (if using expressions, ensure they are sanitized or not directly from user input to avoid SQL injection). For PostgreSQL, the node is similar (just type is `postgres` and credentials type is `postgreSql`). These nodes allow you to integrate your workflow with existing databases for reading or writing data, essentially making n8n a part of your data pipeline.

Other Integrations: n8n has **500+ nodes** for various services. Some other notable ones:

- **AWS S3** (and other AWS services via dedicated nodes): e.g., S3 node can upload or download files.
- **Google Drive**: for file operations (upload, download, list files).
- **Email Send (SMTP)**: to send emails via SMTP or services like SendGrid.
- **Twilio**: send SMS or WhatsApp messages via Twilio.
- **Stripe**: create customers, process payments or react to Stripe events (Stripe Trigger).

- **Webhook (service-specific)**: Many services have trigger nodes (e.g., Stripe Trigger, GitHub Trigger, Slack Trigger) that listen to incoming webhooks from those services without you manually setting up the Webhook node.
- **Jira, Trello, Asana**: project management nodes to create/update tasks.
- **HubSpot, Salesforce**: CRM nodes to manage contacts, deals, etc.
- **CSV & XML**: nodes to parse or write CSV/XML, useful when dealing with file data.
- **HTTP Webhook (outgoing)**: If you need to call an external webhook, just use the HTTP node (as shown) or the specific integration if available.

For any specific service node, the pattern is: **resource** (what entity you're dealing with), **operation** (action to perform), and then fields for that operation (often mapping closely to the service's API fields). The best way to build these is often to configure the node in n8n's editor and then copy the JSON (via workflow export) as a reference.

Usage Tips:

- **Expressions:** In the JSON above, you see a lot of `={{ $json["..."] }}`. These are n8n expressions that pull data from previous nodes. In code, ensure they are wrapped in double curly braces inside the JSON string. At runtime, n8n evaluates them. You can also use `$node["NodeName"].json["field"]` to reference a specific node's output, or `$items()` to reference multiple items.

- **Credentials:** The `"credentials"` section in each node's JSON links to stored creds. In exports, they may appear as `{ "id": "some-id", "name": "Credential Name" }` or just the name. When programmatically creating workflows via the API, you might only need to set the credential name (if it's unique) or the ID. Always ensure the credential exists in n8n beforehand.

- **Node IDs and Position:** You might notice `id` and `position` in examples from exports community.faros.ai
  . These are not necessary when writing a cheat sheet, but in actual workflow JSON they place the node in the editor. They can be omitted if focusing only on the nodes' functional config.

Connecting Nodes: In the workflow JSON, there's a `"connections"` object that links node outputs to inputs. In this cheat sheet, we show individual nodes. When building a workflow programmatically, you'll need to construct that connections object. For example, to connect *Cron -> GraphQL -> Function -> Slack* as in our Slack reminder example, the JSON had:

json

Copy

```
"connections": {
  "Cron": { "main": [ [ { "node": "GraphQL", "type": "main", "index": 0 } ] ] },
  "GraphQL": { "main": [ [ { "node": "Summarize", "type": "main", "index": 0 } ] ] },
  "Summarize": { "main": [ [ { "node": "Slack", "type": "main", "index": 0 } ] ] }
}
```

- This indicates Cron's output connects to GraphQL's input, etc. community.faros.ai

  community.faros.ai
  . If creating workflows via the API, you will formulate a similar structure.

- **Testing and Iteration:** Start with simple nodes (Manual Trigger -> one node -> output) to ensure your JSON is correct, then expand. You can import JSON in n8n via *Workflow -> Import from JSON* to verify it visually.

This cheat sheet covered major categories and popular nodes (AI, webhooks, integrations, data processing). With these examples, you can mix and match to automate a vast array of workflows in n8n. Happy automating!

Ultimate N8n Automation Cheat Sheet: Comprehensive Guide to Node Configuration and JSON Usage

The following guide offers a detailed exploration of N8n automation capabilities, with special focus on JSON snippets for various node configurations. This cheat sheet serves as a comprehensive resource for both beginners and advanced users looking to leverage the full power of N8n's workflow automation platform.

Understanding N8n's Data Structure

N8n passes data between nodes as an array of objects, following a specific structure that's crucial to understand for effective workflow creation. All data in N8n follows this fundamental pattern[10]:

```
json
[
  {
    "json": {
      "property1": "value1",
      "property2": "value2"
    },
    "binary": {}
  },
  {
    "json": {
      "property1": "anotherValue1",
      "property2": "anotherValue2"
    },
    "binary": {}
  }
]
```

Each entry in this array is called an "item" and nodes process each item individually. This structure enables parallel processing of multiple data pieces through your workflow[10]. From version 0.166.0 onward, when using Function or Code nodes, N8n automatically adds the `json` key if it's missing and wraps items in an array if needed[10].

Working with N8n's JSON Data

When manipulating data in N8n, you'll frequently need to access properties from previous nodes. You can use expressions to do this:

```
text
{{ $json.property1 }}
```

For deeper nested properties:

```
text
{{ $json.parent.child.property }}
```

To access data from a specific previous node:

```
text
{{ $node["NodeName"].json.property }}
```

Edit Fields (Set) Node Configuration

The Edit Fields node is essential for manipulating workflow data. It allows you to set new data and overwrite existing data[1].

Manual Mapping Mode

```
json
{
  "newField": "{{ $json.existingField }}",
  "combinedField": "{{ $json.firstName }} {{ $json.lastName }}",
  "staticField": "This is a static value"
}
```

JSON Output Mode

```
json
{
  "mode": "jsonOutput",
  "jsonOutput": {
    "newField": "{{ $json.existingField }}",
    "processedData": {
      "id": "{{ $json.id }}",
      "timestamp": "{{ $now }}"
    }
  }
}
```

Dot Notation Support

By default, N8n supports dot notation in field names. Setting a name as `number.one` with value `20` produces:

```
json
{
  "number": {
    "one": 20
  }
}
```

To disable this behavior, select Add Option > Support Dot Notation and set it to off[1].

Code Node Implementation

The Code node provides powerful ways to transform data programmatically. It offers two operational modes[16]:

Run Once for All Items

This processes all incoming data at once, useful for operations on multiple items:

```
javascript
// Example: Get Array from Object
const newItems = [];
const inputData = $input.all().json;

for (const item of inputData) {
  newItems.push({
    json: {
      modifiedData: item.originalData,
      timestamp: new Date().toISOString()
    }
  });
}

return newItems;
```

Run Once for Each Item

This processes each item individually:

```javascript
// Example: Transform single item
const item = $json;
item.processed = true;
item.modifiedAt = new Date().toISOString();
return { json: item };
```

Splitting JSON into Separate Items

A common task is splitting nested JSON into individual items:

```javascript
// Split incoming webhook data into separate items
let results = [];
for (const item of $('Webhook').all()) {
  const students = item.json.body.students;
  for (studentKey of Object.keys(students)) {
    results.push({
      json: students[studentKey]
    });
  }
}
return results;
```

Using Expressions in N8n

Expressions allow dynamic parameter setting based on data from previous nodes, the workflow, or your environment[7].

Basic Expression Syntax

All expressions have the format `{{ your expression here }}`[7].

Accessing Data from Previous Nodes

To get data from a webhook body[7]:

```
text
{{ $json.city }}
```

This accesses the incoming JSON data using N8n's `$json` variable and finds the value of the `city` property.

AI Agent Node Configuration

The AI Agent node brings powerful AI capabilities to N8n workflows. It provides six LangChain agent options[3]:

Tools Agent (Default)

This agent uses external tools and APIs to perform actions and retrieve information[3]. JSON configuration example:

```
json
{
  "agent": "tools",
  "model": {
    "provider": "openai",
    "model": "gpt-4",
    "temperature": 0.7
  },
  "systemMessage": "You are a helpful assistant that uses tools to find information and accomplish tasks.",
  "memory": true,
  "verbose": true
}
```

Conversational Agent

This agent maintains context, understands user intent, and provides relevant answers. Ideal for chatbots and virtual assistants[3]:

```
json
{
  "agent": "conversational",
  "model": {
```

```
    "provider": "openai",
    "model": "gpt-3.5-turbo",
    "temperature": 0.8
  },
  "systemMessage": "You are a helpful customer service agent for
our company.",
  "memory": true
}
```

Building an AI Agent with Memory

To create an AI agent with long-term memory[1][2]:

```
json
{
  "agent": "tools",
  "model": {
    "provider": "openai",
    "model": "gpt-4",
    "temperature": 0.7
  },
  "systemMessage": "You are an assistant with memory
capabilities. Use your tools to remember important information
from our conversations.",
  "memory": true,
  "tools": ["memory-add", "memory-retrieve"]
}
```

Exporting and Importing Workflows

N8n saves workflows in JSON format, allowing for easy sharing and reuse[9].

Copy-Paste Method

You can copy part or all of a workflow using standard keyboard shortcuts (Ctrl+C/Cmd+C and Ctrl+V/Cmd+V)[9].

Importing JSON Workflows

To import a JSON workflow from an external source[2]:

1. Create a new workflow in N8n
2. Copy the JSON script
3. Paste it directly into the workflow editor using Ctrl+V/Cmd+V
4. Connect any required accounts
5. Customize as needed
6. Save the workflow

Looping Through JSON Data

Working with arrays of JSON objects is common in N8n[17]:

Using Item Lists Node

To split an array from incoming data:

```
text
{{ $json.body.block }}
```

This splits each block into separate items that can be processed individually[17].

Tools and Converters

N8n provides several conversion tools[5]:

XML to JSON Converter

```
json
{
  "operation": "convert",
  "source": "xml",
  "target": "json",
  "data": "{{ $json.xmlData }}"
}
```

CSV to JSON Converter

```
json
{
  "operation": "convert",
  "source": "csv",
  "target": "json",
  "data": "{{ $json.csvData }}"
}
```

Integrating with Third-Party Services

N8n excels at connecting various platforms and services[15].

Google Sheets Integration

```
json
{
  "operation": "appendOrUpdate",
  "sheetId": "YOUR_SHEET_ID",
  "range": "A:Z",
  "data": "{{ $json.rowData }}"
}
```

Automating JSON Generation with OpenAI

You can use the OpenAI node to generate structured JSON automatically[11]:

```
json
{
  "model": "gpt-4",
  "prompt": "Generate a JSON object with the following
structure: {{ $json.structure }}",
  "temperature": 0.2,
  "output": "json"
}
```

Advanced Features and Tips

Accessing Data from a Webhook

When receiving data through a webhook, access specific elements using expressions[14]:

```
text
{{ $json.body.ticker }}
{{ $json.body.tf }}
```

For string representations of JSON, you'll need to parse the string first[14]:

```
javascript
// In a Code node
const parsedBody = JSON.parse($json.body);
return { json: parsedBody };
```

Adding Data from Previous Node to Every Item

When splitting data but needing to retain information from the original source[18]:

```
javascript
// In a Code node
const results = [];
const eventName =
$('Webhook').first().json.body.event.event_name;

for (const student of $json.body.students) {
  results.push({
    json: {
      ...student,
      eventName: eventName
    }
  });
}
return results;
```

Comprehensive Guide to JSON Snippets for N8n Automation

When building automations in N8n, proper handling of JSON configurations is essential, especially for advanced nodes like the AI Agent module. This guide compiles key JSON snippets and patterns for effectively configuring various N8n components, with a special focus on the AI Agent functionality.

N8n's Fundamental JSON Data Structure

In N8n, data flows between nodes in a standardized JSON structure. Understanding this structure is crucial for effective automation:

```
json
[
  {
    "json": {
      "property1": "value1",
      "property2": "value2"
    },
    "binary": {}
  },
  {
    "json": {
      "property1": "anotherValue1",
      "property2": "anotherValue2"
    },
    "binary": {}
  }
]
```

Each item in this array represents a piece of data being processed through your workflow. From version 0.166.0 onward, N8n automatically adds the `json` key if it's missing and wraps items in an array if needed when using Function or Code nodes[9].

AI Agent Module JSON Configuration

The AI Agent module in N8n provides several agent types, each requiring specific JSON configurations. Here are snippets for the most common types:

Tools Agent (Default) Configuration

```
json
{
  "agent": "tools",
  "model": {
    "provider": "openai",
    "model": "gpt-4",
    "temperature": 0.7
  },
  "systemMessage": "You are a helpful assistant that uses tools to find information and accomplish tasks.",
  "memory": true,
  "verbose": true
}
```

This configuration sets up the default Tools Agent that can use external tools and APIs to perform actions and retrieve information[1].

Conversational Agent Configuration

```
json
{
  "agent": "conversational",
  "model": {
    "provider": "openai",
    "model": "gpt-3.5-turbo",
    "temperature": 0.8
  },
  "systemMessage": "You are a helpful customer service agent for our company.",
  "memory": true
}
```

The Conversational Agent is ideal for chatbots and virtual assistants, as it can maintain context and understand user intent[5].

Memory-Enabled Agent Configuration

```
json
{
  "agent": "tools",
  "model": {
    "provider": "openai",
    "model": "gpt-4",
    "temperature": 0.7
  },
  "systemMessage": "You are an assistant with memory
capabilities. Use your tools to remember important information
from our conversations.",
  "memory": true,
  "tools": ["memory-add", "memory-retrieve"]
}
```

This configuration creates an AI agent with long-term memory capabilities for maintaining context across interactions[1].

Handling AI Agent Output Issues

One common challenge with the AI Agent is that it often outputs results as a string inside a JSON object, which can cause issues when further processing is needed:

```
json
{
  "output": "{\n \"countries\": [\n {\n \"name\": \"China\",\n
\"population\": 1411778724\n },\n {\n \"name\": \"India\",\n
\"population\": 1387297452\n },\n {\n \"name\": \"United
States\",\n \"population\": 331893745\n },\n {\n \"name\":
\"Indonesia\",\n \"population\": 276362965\n },\n {\n \"name\":
\"Pakistan\",\n \"population\": 225199937\n }\n ]\n}"
}
```

To process this output, you'll typically need to use a Code node to parse the string into a proper JSON object[1]:

```
javascript
// Parse the string output from AI Agent into usable JSON
const outputStr = $json.output;
let parsedData;
try {
  parsedData = JSON.parse(outputStr);
  return { json: parsedData };
} catch (error) {
  // Handle case where output is not valid JSON
  return { json: { error: "Could not parse output as JSON",
original: outputStr } };
}
```

Providing JSON Examples to AI Agents

When instructing AI agents to output structured JSON, you may encounter the "Single '{' in template" error. This happens because the curly braces in your JSON example conflict with N8n's expression syntax[4].

To avoid this, you can use one of these approaches:

1. Use a Code Node to Generate the Example

```
javascript
// Generate JSON example before the AI Agent node
const example = {
  "data": [
    {
      "Item1": "This is a test",
      "Somedata": {
        "Frog": 6,
        "Cat": 7,
        "Dog": 9
      }
    }
```

```
  ]
};
return { json: { example: JSON.stringify(example) } };
```

Then in your AI Agent node, you can reference this example with `{{ $json.example }}`[4].

2. Use the fromAI Function (Newer Method)

For newer versions of N8n, the `$fromAI` function provides a cleaner approach to handle AI agent outputs[14]:

```
text
{{ $fromAI.json.specificProperty }}
```

This allows you to directly reference properties from the AI's output in subsequent nodes without manual parsing.

Working with HTTP Tools in AI Agents

When configuring HTTP tools for AI agents that require JSON parameters:

```
json
{
  "url": "https://api.example.com/endpoint",
  "method": "POST",
  "headers": {
    "Content-Type": "application/json"
  },
  "bodyParametersUi": {
    "parameter": [
      {
        "name": "ids",
        "value": "{{ [\"ID\"] }}"
      }
    ]
  }
}
```

The proper format for array parameters must be explicitly described in the tool's description to ensure the AI correctly formats them[6].

Preserving Input JSON Fields in AI Agent Output

A common challenge is maintaining input fields (like IDs) in the AI agent's output. Since the AI agent might not reliably include these fields in its response, a better approach is to use a Code node after the AI Agent to merge the original input with the agent's output[10]:

```
javascript
// Preserve the ID from input while using AI agent output
const originalId = $node["PreviousNode"].first().json.id;
const aiOutput = $json.output;

return {
  json: {
    id: originalId,
    aiResult: aiOutput
    // Add any other fields you want to preserve
  }
};
```

Processing Multiple Items with AI Agents

By default, the AI Agent processes each item individually, which can be time-consuming for batch processing. To process multiple items in a single call[8]:

```
javascript
// Collect all items into a single array for the AI agent
const allItems = $input.all().map(item => item.json);
return {
  json: {
    combinedData: allItems,
    prompt: "Process all these items at once: " +
JSON.stringify(allItems)
  }
};
```

Structured Output Formatting for AI Agents

To get consistent structured output from AI agents, provide clear output format instructions in your system prompt. For JSON outputs[11]:

```
json
{
  "options": {
    "systemMessage": "You are a helpful assistant. Always respond in valid JSON format following this structure:\n\n{\n \"category\": \"[Category of the request]\",\n  \"response\": \"[Your detailed response]\",\n  \"nextSteps\": [\"step1\", \"step2\", \"etc\"]\n}\n\nEnsure your entire response is valid JSON."
  }
}
```

Using Expressions to Access JSON Data

N8n provides a powerful expression syntax for accessing and manipulating JSON data:

```
text
{{ $json.property1 }}
```

For deeper nested properties:

```
text
{{ $json.parent.child.property }}
```

To access data from a specific previous node:

```
text
{{ $node["NodeName"].json.property }}
```