Comprehensive Make (Integromat) Technical Documentation for AI-Powered Scenario Generation

I. Executive Summary

Make.com, formerly known as Integromat, is a leading visual automation platform that enables users to connect diverse applications and automate complex workflows without requiring extensive coding expertise. Its strength lies in a highly intuitive visual interface that translates intricate logical sequences into comprehensible diagrams, complemented by robust data transformation capabilities. For FlowForge AI, Make's fundamental reliance on JSON for scenario blueprints and its internal data representation is a pivotal advantage. This inherent JSON structure provides a direct, machine-readable format for AI-driven generation and subsequent validation, positioning Make as an ideal target for AI-powered text-to-automation solutions.

The deep research conducted confirms that Make's architecture is exceptionally well-suited for programmatic interaction through its comprehensive API and direct manipulation of JSON blueprints. Key findings highlight explicit JSON structures for scenarios (blueprints), detailed module configurations embedded within these blueprints, and the critical role of data structures in facilitating JSON parsing and creation. Furthermore, a nuanced understanding of "operations" and various data limits is essential for generating cost-efficient and performant scenarios. Advanced features such as aggregators, iterators, routers, and sub-scenarios each possess distinct JSON representations and operational impacts, which are crucial for generating optimized and complex workflows. The platform's robust security features and performance considerations provide essential contextual information for building a reliable and secure AI system.

The visual interface of Make.com, while simplifying complex processes for human users by abstracting underlying structures, presents a unique challenge and opportunity for AI-driven generation. This abstraction means that while end-users interact with a drag-and-drop canvas, the platform internally processes and stores scenarios as precise JSON objects. For an AI system like FlowForge AI, the "no-code" visual layer is not the direct target for generation. Instead, the AI must operate directly on the underlying JSON schema. This necessitates that the AI effectively "reverse-engineer" the visual simplicity into syntactically correct and functionally accurate JSON. The success of the AI system, therefore, hinges on its profound understanding of this JSON structure, rather than a superficial comprehension of the visual interface. This capability to bridge the gap between high-level natural language intent and low-level, executable JSON is fundamental to FlowForge AI's mission.

II. Documentation Inventory & Resources

Access to comprehensive and accurate documentation is paramount for developing a robust AI system capable of generating Make scenarios. The available resources can be broadly

categorized into official documentation and supplementary community-driven materials.

Official Make Documentation

- **Help Center (help.make.com):** This serves as a foundational resource, offering step-by-step guides for scenario creation, detailed explanations of module settings, principles of data mapping, and insights into core tools such as aggregators, iterators, and routers. It also covers essential scenario settings and various error handling strategies. While primarily user-oriented, the Help Center provides crucial context for understanding the platform's intended functionality and basic configurations.
- **Developer Documentation (developers.make.com):** This is an indispensable resource for understanding the deeper technical aspects of Make. It provides detailed information on custom app development, including local development workflows, debugging techniques, and best practices for defining core app elements such as Base URLs, Connections, Webhooks, Modules, Remote Procedure Calls (RPCs), and Custom IML (Integromat Markup Language) functions. This documentation is vital for FlowForge AI to comprehend the granular structure of modules and their configurable parameters.
- **API Documentation (developers.make.com/api-documentation):** This is the definitive source for programmatic interaction with the Make platform. It offers comprehensive details on endpoints for managing a wide array of Make entities, including modules, scenarios, scenario folders, teams, connections, IML functions, webhooks, and custom apps. Critically, it specifies how to list, retrieve, create, update, run, and delete scenarios, with particular emphasis on the GET /scenarios/{scenarioId}/blueprint endpoint for retrieving a scenario's underlying JSON blueprint.

Community & Supplementary Resources

- **Make Community Forum (community.make.com):** This is an invaluable resource for practical insights, troubleshooting common issues, and discovering real-world implementations and workarounds. It frequently provides solutions to undocumented behaviors, particularly concerning complex JSON manipulation and data structures, and offers diverse perspectives on scenario design.
- **Template Marketplace (make.com/en/templates):** This repository of pre-built scenarios serves as a source of concrete JSON examples for various integrations and use cases. These templates are invaluable for understanding common automation patterns and their underlying JSON structure, even if a formal schema is not explicitly provided.
- **Third-party tutorials and GitHub blueprints:** External resources such as GitHub Gists (e.g., gist.github.com/helLf1nGer/ac8c7351411c65e660f700b575629947) and repositories (e.g., github.com/okash1n/make-blueprints/blob/main/blueprint.json) offer direct access to exportable JSON blueprints. These are critical for FlowForge AI to learn from actual, working scenario structures.

Recommendation

For FlowForge AI, the primary focus should be on official Make documentation for core schema definitions and authoritative information. This should be rigorously supplemented with community examples and exported blueprints to understand practical implementation patterns and edge cases. The explicit API endpoint for retrieving scenario blueprints (GET

/scenarios/{scenarioId}/blueprint) offers a direct, programmatic mechanism for FlowForge AI to acquire ground-truth data from existing scenarios.
The availability of the "Export Blueprint" functionality and the active community sharing of these JSON files indicate that Make implicitly supports programmatic understanding and manipulation of scenarios. This is significant because, even without a single, comprehensive, official JSON Schema definition (like a json-schema.org document) for the entire blueprint, the ability to export and import these JSONs means they represent the operational schema. For FlowForge AI, this implies that the system can effectively learn the "schema" by analyzing a large corpus of these exported blueprints. This "learn by example" approach is often more practical and robust for large language models than relying solely on abstract, formal schema definitions. The operational blueprints themselves serve as the definitive ground truth for what constitutes a valid Make scenario JSON, providing a concrete foundation for AI generation and validation.

III. Core Architecture & Scenario JSON Structure

Scenario JSON Structure (Blueprint)

A Make scenario blueprint encapsulates the complete design of an automation workflow, including its triggers, subsequent modules, detailed configuration settings, and linked applications or services. This comprehensive design can be exported as a .json file. When retrieved via Make's API, the blueprint is transmitted as a string rather than a direct JSON object, a design choice intended to optimize resource usage.
The core components of a scenario blueprint are inferred from various examples and API documentation :

- **name**: A string representing the user-defined display name of the scenario.
- **flow**: A critical array that defines the sequential (or sometimes parallel) execution path of modules within the scenario. Each object within this array represents an individual module and contains:
    - **id**: A unique integer identifier for the module within the scenario.
    - **module**: A string specifying the type of module (e.g., "json:ParseJSON", "google-sheets:watchUpdatedCells", "builtin:BasicRouter"). This often follows an app:moduleType naming convention.
    - **version**: An integer indicating the version of the module.
    - **parameters**: A JSON object containing module-specific configuration settings, which vary widely depending on the module's function.
    - **mapper**: A JSON object that defines how data (often referred to as "bundles") from preceding modules or external sources is transformed and mapped as input for the current module. This is where data flow logic is explicitly defined.
    - **metadata**: A JSON object containing design-time information, including designer properties with x and y coordinates for visual placement in the Make.com editor, and messages for warnings or errors related to the module. It also includes restore data for maintaining the UI state.
- **metadata (Top-level)**: A JSON object providing overall scenario-level settings and metadata:
    - **version**: An integer representing the version of the scenario's metadata.
    - **scenario**: A nested object containing crucial execution settings for the entire scenario, such as:

        - **roundtrips**: The number of times the scenario will attempt to process data in a single execution cycle.
        - **maxErrors**: The maximum number of errors allowed before the scenario automatically halts execution.
        - **autoCommit**: A boolean indicating whether changes are automatically committed during execution.
        - **autoCommitTriggerLast**: A boolean suggesting auto-commit behavior for the last trigger in a sequence.
        - **sequential**: A boolean indicating if modules or routes within the scenario execute strictly in sequence (true) or allow for parallel processing/branching (false).
        - **confidential**: A boolean flag indicating if the scenario handles confidential data.
        - **dataloss**: A boolean flag indicating if data loss is expected or possible under certain conditions.
        - **dlq**: A boolean indicating whether the Dead Letter Queue (DLQ) mechanism is enabled for failed bundles.
        - **freshVariables**: A boolean indicating if scenario variables are refreshed on each run.
    - **designer**: A nested object containing design-time information, such as orphans:, which would list any modules that are disconnected from the main flow.
    - **zone**: A string specifying the Make.com server zone where the scenario is hosted (e.g., "us1.make.com").

A concrete example of a scenario blueprint from a GitHub Gist illustrates this structure for a Google Sheets to Google Calendar automation. It showcases the configuration of modules like google-sheets:watchUpdatedCells, builtin:BasicRouter, google-calendar:createEvent, email:ActionSendEmail, and google-sheets:UpdateRow. This example clearly demonstrates parameter mapping, connections (via __IMTCONN__ IDs), and the various metadata fields.
The following table provides a simplified schema for the Make Scenario Blueprint, which is critical for FlowForge AI to ensure syntactic correctness and a clear understanding of essential fields and their expected data types during generation.

Field Name	Data Type	Description	Example Value
name	string	User-defined display name of the scenario.	"The Airbrush Company Schedule from GSheets to GCalendar"
flow	array of objects	Ordered sequence of modules in the scenario.	[{...}, {...}]
flow.id	integer	Unique identifier for the module within the scenario.	14
flow.module	string	Type of module (e.g., app:moduleType).	"google-sheets:watchUpdatedCells"
flow.version	integer	Version of the module.	2
flow.parameters	object	Module-specific configuration settings.	{"__IMTHOOK__": 99251}

Field Name	Data Type	Description	Example Value
flow.mapper	object	Defines data transformation and mapping for the module.	{"end": "{{formatDate(14.rowValues.4; "YYYY-MM-DDTHH:mm:ss"; "New_York")}}"}
flow.metadata	object	Design-time and module-specific metadata.	{"designer": {"x": 0, "y": 450}}
metadata.scenario.maxErrors	integer	Maximum errors allowed before scenario stops.	3
metadata.scenario.sequential	boolean	Indicates if execution is strictly sequential.	false
metadata.designer.x	integer	X-coordinate for visual designer layout.	0
metadata.designer.y	integer	Y-coordinate for visual designer layout.	450

Module Types & Core Functionality

Make organizes its automation logic around distinct module types, each serving a specific purpose within a scenario's flow. Understanding their operational impact, particularly concerning "operations" consumption, is vital for optimizing generated scenarios.

- **Trigger Modules:** These modules initiate a scenario's execution. They can be either **polling triggers**, which periodically check a service for new data, or **instant triggers** (webhooks), which execute the scenario immediately upon receiving an external request. Examples include google-sheets:watchUpdatedCells or a generic Webhook. A crucial operational detail is that trigger modules always consume one operation, irrespective of whether they retrieve new data. This means scheduled checks, even if no new data is found, contribute to the total operation count.
- **Action Modules:** These modules perform specific tasks or operations within a connected application, such as adding, updating, or deleting records (e.g., Google Sheets > Add a Row, Salesforce > Create Record). Action modules consume operations based on the number of runs required to process all input data. For instance, if 10 data bundles are passed to an action module, it will execute 10 times, consuming 10 operations.
- **Search Modules:** These modules query an external service for data (e.g., Google Sheets > Search Row). They typically consume one operation to initiate the search but can output multiple "bundles" of data, with each bundle representing a separate record found. The downstream modules will then process each of these bundles individually, consuming operations accordingly.
- **Aggregator Modules:** These specialized modules combine multiple incoming "bundles" of data into a single bundle. Common examples include the Array Aggregator (which creates an array from multiple bundles) and the Text Aggregator (which concatenates data into a single text string). Aggregators consume one operation for each aggregation performed, regardless of the number of bundles aggregated into a single output.
- **Iterator Modules:** The inverse of aggregators, iterators take a single "bundle" containing an array and split it into multiple individual "bundles," one for each item in the array. This

allows subsequent modules to process each array item separately. Iterators consume one operation for processing the entire array, but the modules following them will then run for each new bundle generated, potentially increasing the total operation count significantly downstream.

- **Router Modules:** These are flow control modules that enable conditional branching of a scenario's execution path. Data can be directed to different routes based on defined filters and conditions. A critical distinction is that router modules themselves do *not* consume operations , making them an efficient way to manage complex conditional logic without incurring additional operational costs.

IV. Module Documentation (Comprehensive)

This section details the common aspects of Make modules, focusing on their configuration, data handling, and operational considerations.

Configuration Parameters & Data Types

Make modules are configured through a parameters object within the scenario blueprint, which contains settings specific to each module's function. The types and structure of these parameters vary widely depending on the module and the connected application. Make supports a broad range of data types for these parameters, including: Array, Boolean, Buffer, Cert, Collection, Color, Date, Email, Filename, Folder, File, Filter, Hidden, Integer, Uinteger, Number, Password, Path, Pkey, Port, Select, Text, Time, Timestamp, Timezone, URL, and UUID.
A significant aspect of module configuration is the dynamic nature of parameters, particularly when dealing with JSON. Modules like Create JSON and Parse JSON rely heavily on predefined or automatically generated "data structures" to define the expected input and output fields. This allows for structured mapping of individual JSON items to other modules in the scenario.

Input/Output Specifications & Data Mapping

Data flow within Make scenarios is managed through "bundles," which are essentially collections of data passed from one module to the next. The mapper field within a module's blueprint entry explicitly defines how data from preceding modules or external sources is transformed and used as input for the current module. This is where complex data manipulation and field-to-field mapping are specified.
The JSON app in Make provides dedicated modules for handling JSON data:

- **Create JSON Module:** This module is used to construct JSON data. Its fields are dynamically generated based on a "data structure" that describes the desired JSON format. Users can either select an existing data structure or generate one automatically by providing a sample JSON string. The module then provides input fields corresponding to the defined structure, and its output is the generated JSON string. For instance, transforming Google Sheet rows into a JSON array of objects involves using an Array Aggregator to collect rows, then feeding this aggregated array into a Create JSON module configured with a matching data structure.
- **Parse JSON Module:** This module is essential for breaking down JSON strings into accessible data elements (bundles) that can be mapped to subsequent modules. Similar

to Create JSON, it can operate with a predefined data structure or automatically infer the structure by running the module with a sample JSON input. If the input JSON string contains a single collection ({}), it outputs a single bundle. If it contains an array (``), it outputs a series of bundles, one for each item in the array. This behavior is fundamental for processing lists of items from APIs.
- **Handling JSON Strings inside JSON Objects:** Make provides specific functions like createJSON() and parseJSON() within custom apps to handle scenarios where a JSON string is embedded as a value within another JSON object. These functions ensure that such nested JSON strings are correctly treated as structured data rather than plain text, enabling proper mapping of their child parameters. It is important to note that these specific functions are primarily for Custom Apps and not directly available in the scenario editor.

Connection Requirements

Modules require connections to external services to function. Make supports various authentication methods, including OAuth and API keys. The process of establishing these connections often involves specific steps within both Make and the third-party application.

- **Gmail:** Connections typically leverage OAuth 2.0 for enhanced security, often requiring a Google Workspace account and a JSON file with credentials generated from the Google API Console, including enabling the Gmail API and creating a Service Account with the appropriate role. Basic authentication with App Passwords is also possible for personal Google accounts with 2-Step Verification enabled.
- **Google Drive:** Similar to Gmail, Google Drive connections often utilize OAuth and may require creating and configuring a Google Cloud Platform project, enabling the Google Drive API, configuring an OAuth consent screen, and generating client credentials (Client ID and Client Secret).
- **Salesforce:** Salesforce modules can connect via standard Salesforce OAuth or Salesforce (client credentials). The client credentials method typically involves creating a custom connected app in Salesforce, defining OAuth scopes (e.g., api, refresh_token), and ensuring specific security settings like PKCE are off. Real-world integrations often involve Salesforce Record-Triggered Flows making API Callouts to Make.com via Webhooks, with the request body structured as JSON.
- **Shopify:** Shopify offers standard OAuth connections and custom app connections. The custom app approach requires creating a private app in the Shopify store, configuring Admin API integration, selecting necessary scopes (permissions), and obtaining an Admin API access token. Public apps require an API key and secret from the Partner Dashboard.
- **MySQL:** Connecting to MySQL requires remote access to be allowed on the server, appropriate firewall configurations, and granting CREATE ROUTINE privileges for stored procedures. Make can dynamically load the interface of input and output parameters for stored procedures, allowing individual mapping. Data transfer from JSON files to MySQL often involves parsing the JSON and then inserting records into a matching table structure.

Filtering & Conditions

Module-level filtering is a core capability within Make scenarios, allowing data processing to be

conditional. Filters are applied to the connections between modules and evaluate expressions based on incoming data bundles. Only bundles that satisfy the filter's conditions proceed to the next module.
Routers are specialized flow control modules that extend conditional logic by enabling multiple branching paths from a single module. Each route from a router can have its own filter, directing bundles to different processing chains based on distinct criteria. Routers themselves do not consume operations, making them an efficient method for implementing complex conditional logic. They can also be used to define fallback routes that process data not matching any other conditions.

Rate Limits & Quotas

Understanding Make's operational and data limits is crucial for designing efficient and cost-effective scenarios.

- **Operations Counting:** Every action performed by a module in a scenario consumes an "operation". This includes trigger checks (even if no new data is found), search module runs (one operation per search, regardless of bundles returned), and action module executions (one operation per bundle processed). Aggregators consume one operation per aggregation, and iterators consume one operation for processing the entire array, but subsequent modules will consume operations for each bundle they receive from the iterator. Exceptions to operation consumption include error handlers, routers, and filtering.
- **Webhook Rate Limits:** Make can process up to 30 incoming webhook requests per second. Exceeding this limit may result in rejected requests or queuing, depending on the webhook type and plan.
- **Data Storage Allowance:** Total data storage for Data Stores is tied to the number of operations in a user's plan, with every 1,000 operations equating to 1 MB of data storage. The minimum size for a data store is 1 MB. A single organization can have a maximum of 1,000 data stores.
- **File Size Limits:** The maximum file size that can be processed by modules depends on the user's subscription plan: Free (5 MB), Core (100 MB), Pro (250 MB), Teams (500 MB), and Enterprise (1,000 MB). If these limits are exceeded, Make proceeds according to the "Enable data loss" setting.
- **Data Transfer Limits:** The Free plan has a data transfer limit of 100 MB per month. Higher plans offer increased allowances.
- **Record Size in Data Stores:** Individual records stored in Data Stores have a maximum size of 15 MB.
- **Character Limits:** While not explicitly detailed for all inputs, Make does have character limits for module inputs, which can truncate long text fields, such as article content for WordPress posts. Workarounds include using external storage or splitting content.

Example Configurations

Concrete examples of module configurations are best observed through exported scenario blueprints and the Make.com template marketplace. The blueprint analysis in Section III provides a detailed example of how modules like google-sheets:watchUpdatedCells, builtin:BasicRouter, and google-calendar:createEvent are configured within the flow array, including their parameters and mapper fields.
The JSON app documentation further provides step-by-step examples for configuring Create

JSON and Parse JSON modules, illustrating how data structures are defined and used to transform data from sources like Google Sheets into JSON format. These examples demonstrate the practical application of data mapping and the dynamic nature of module fields based on the chosen data structure.

V. Integration Categories (Deep-Dive)

Make.com boasts an extensive library of over 2,000 pre-built app integrations, enabling connectivity across a vast array of services. These integrations are categorized for easier navigation and include both "Verified Apps" (reviewed by Make) and "Community Apps".

Top Popular App Integrations

While a definitive "Top 100" list across all categories is not explicitly provided in the research material, popular integrations frequently mentioned or highlighted across various categories include:

- **Communication & Messaging:** Gmail , Outlook, Slack , Discord , Microsoft Teams, Telegram. These are used for email processing, chat notifications, and message parsing.
- **Cloud Storage & File Management:** Google Drive , Dropbox, OneDrive, Box, AWS S3, FTP/SFTP. Common operations include file transfers, folder synchronization, and metadata extraction.
- **CRM & Customer Management:** Salesforce , HubSpot , Pipedrive , Zoho CRM, Airtable , Notion. These facilitate contact synchronization, deal management, and lead processing.
- **Marketing & Analytics:** Google Analytics, Facebook Ads, Google Ads, Mailchimp , ActiveCampaign. Used for campaign automation, subscriber management, and conversion tracking.
- **Development & Project Management:** GitHub, GitLab, Jira, Asana , Trello, Linear, Azure DevOps. Common uses include issue tracking, project automation, and code repository management.
- **E-commerce & Finance:** Shopify , WooCommerce, Magento, Stripe, PayPal, QuickBooks. Supports order processing, payment handling, and inventory management.
- **Database & Data Processing:** MySQL , PostgreSQL , MongoDB , Google Sheets , Excel Online. Used for data synchronization, record processing, and bulk operations.
- **AI:** OpenAI (ChatGPT) , Anthropic Claude. Used for content generation, analysis, and intelligent automation.

Module-Specific Parameters & Capabilities

Each app integration provides a set of modules (triggers, actions, searches) with specific parameters tailored to the API of the integrated service. For example, the Salesforce module allows broadcasting messages, creating/modifying records, and making API calls. Google Drive modules enable file operations like copying files, and Google Sheets modules allow selecting rows, updating rows, etc..

For applications not natively supported or for highly customized API interactions, Make provides a versatile **HTTP module**. This built-in module allows users to connect to any RESTful API, offering full flexibility to make HTTP requests (GET, POST, PUT, DELETE), configure headers, and handle request bodies and responses, including direct JSON control. This is particularly

useful for working with custom fields, advanced filters, or structured data that might not be exposed by native modules.

Authentication Requirements & Limitations

Authentication for integrations in Make typically involves standard methods like OAuth 2.0 (for services like Google, Salesforce, Shopify) or API keys (for many other services). The specific configuration steps vary per app, often requiring creation of custom applications or credentials within the third-party service's developer console. Some Salesforce editions may have API access limitations.

Real-World Usage Examples & Patterns

The template marketplace and community forums provide numerous real-world usage examples and automation patterns:

- **Lead Management:** When a new lead is added to HubSpot, create a task in Asana and send a welcome email via Mailchimp.
- **Content Automation:** Fetch new articles from RSS feeds, convert to JSON, and send via HTTP for distribution. Streamline YouTube uploads with AI-generated descriptions from Google Drive.
- **Data Synchronization:** Sync user activity to CRMs or databases. Automatically add new incoming emails to a Google Sheets spreadsheet.
- **Conditional Routing:** Route Salesforce leads to different Slack channels based on company size or country.
- **Error Handling:** Implement custom error handling by sending email notifications or logging issues to a Slack channel when a module errors.

These examples demonstrate the practical application of Make's modules and data flow capabilities in diverse business contexts.

VI. Advanced Features Documentation

Make's advanced features enable the construction of highly sophisticated and efficient automation workflows. Understanding their JSON representation and operational nuances is critical for FlowForge AI to generate complex scenarios.

Aggregators

Aggregators are modules designed to combine multiple data "bundles" into a single bundle, significantly reducing subsequent operations and streamlining data processing.

- **Purpose:** To accumulate data from multiple items (records) from trigger, iterator, list, search, or match modules into a consolidated structure. This is particularly useful when a downstream module expects a single input (e.g., an array or a single text string) rather than multiple individual bundles.
- **Types:**
    - **Array Aggregator:** Combines multiple bundles into a single array. This module is powerful for building complex arrays of collections that can be mapped to a later module's field. It requires setting a "Target structure type" to define the format of the

output array, often linked to a Create JSON module's expected array structure.
    - **Text Aggregator:** Concatenates data from multiple bundles into a single text string. This is highly flexible for building JSON, CSV, or HTML content.
- **JSON Examples:** While the snippets do not provide explicit JSON for aggregator configurations themselves, their output is often consumed by Create JSON modules. The process of transforming Google Sheets data to JSON using an Array Aggregator and Create JSON illustrates the resulting JSON structure. For example, aggregating multiple rows into a JSON array of books objects.

Iterators

Iterators perform the inverse function of aggregators: they convert a single bundle containing an array into a series of individual bundles, one for each item in the array. This allows subsequent modules to process each array item independently.

- **Purpose:** To process individual elements within a collection or array separately. This is essential when a module needs to act on each item from a list (e.g., saving each attachment from an email or processing each row from a spreadsheet).
- **Configuration:** Setting up an iterator involves specifying the array field to be converted into separate bundles. If the iterator does not have information about the array's item structure (e.g., from a Parse JSON or Custom Webhook module without a data structure), the mapping panel for subsequent modules will initially only show "Total number of bundles" and "Bundle order position". Manually running the scenario once allows the module to learn the structure and expose mappable items.
- **JSON Examples:** Specific JSON examples for iterator module configuration or its direct input/output are not explicitly provided in the research material. However, the blueprint in implicitly uses iterators or similar concepts within the flow, as data from Google Sheets (14.rowValues) is processed iteratively for calendar events and emails.

Routers

Routers are fundamental flow control modules that enable conditional branching and parallel execution paths within a scenario. They act as "traffic controllers" for data flow, directing bundles to different routes based on defined filters and conditions.

- **Purpose:** To define multiple execution paths based on specific criteria, allowing for diverse processing of different data types or conditions within a single scenario. This reduces the need for separate scenarios for different conditions.
- **JSON Structure:** In the scenario blueprint, routers are represented by the builtin:BasicRouter module type and contain a routes array. Each object within the routes array defines a distinct path, including its associated filters.
- **Conditional Logic:** Filters are configured on each route, using operators (e.g., less than, greater than, equals) to determine which bundles follow that path.
- **Parallel Execution:** By default, scenarios with instant webhooks are processed in parallel, and routers can facilitate this by allowing multiple branches to execute concurrently if their conditions are met.
- **Fallback Routes:** Routers support fallback routes, which are activated when data does not meet the conditions of any other defined route, providing a "plan B" for unexpected data or errors.
- **Error Handling with Routers:** While routers themselves do not consume operations,

they can be integral to error handling strategies by directing problematic bundles to specific error routes. For instance, an HTTP module returning a 400 error might be routed to an error handler.

Data Stores

Data Stores in Make function as temporary, simple databases within scenarios, allowing for data persistence and transfer between individual scenario runs or even different scenarios.

- **Purpose:** To store and retrieve data dynamically during scenario execution, acting as a lightweight, in-memory or persistent storage mechanism. They are useful for maintaining state, caching data, or passing information across loosely coupled scenarios.
- **JSON Structure for Records:** Data stores define their structure using a schema, which can be generated from a JSON sample. An example JSON structure for data store columns is provided:
  ```
  {
    "name": "john",
    "age": 30,
    "phone number": {
      "mobile": "987654321",
      "landline": "123456789"
    }
  }
  ```
  This example illustrates how nested objects can define columns within a data store.
- **Operations:** Data Stores support standard database-like operations:
  - **Add/Replace a Record:** Adds a new record or overwrites an existing one based on a unique key. The maximum record size is 15 MB.
  - **Update a Record:** Modifies an existing record identified by its key. It can also insert a new record if the key doesn't exist.
  - **Get a Record:** Retrieves a specific record using its unique key.
  - **Search Records:** Filters records based on specified column, operator, and value. Results can be sorted and limited.
- **Data Storage Allowance:** The total data storage capacity is tied to the user's plan, with 1 MB allocated for every 1,000 operations.

Webhooks

Webhooks are HTTP endpoints that allow external applications or other Make scenarios to send data to Make, triggering scenario execution. They typically act as instant triggers, executing scenarios immediately upon receiving a request, unlike scheduled triggers that poll periodically.

- **Purpose:** To enable real-time data exchange and instant scenario initiation from external systems or other Make scenarios.
- **Types:**
  - **App-specific Webhooks:** Provided by many integrated apps as "instant triggers".
  - **Custom Webhooks:** Created using Make's Webhooks app, providing a generic URL to which any data can be sent.
- **JSON Structure for Requests/Responses:** While the documentation does not provide a formal JSON schema for webhook configuration, it indicates that webhook logs contain the webhook request (timestamp, URL, method, headers, query, body) and webhook

response (status, headers, body). The "parsed items" combine the query parameters and body of the request into a single bundle for use in the scenario. Examples of request bodies sent to webhooks are seen in Salesforce integrations and custom app webhooks.
- **Configuration:** Custom webhooks can be configured to schedule processing (queueing requests) or process instantly.
- **Authentication:** Webhooks can be secured, and for custom apps, the webhook configuration can include authentication details.

Functions

Make provides a rich set of built-in functions for data transformation and calculations, similar to spreadsheet functions.

- **Built-in Functions:** Over 80 built-in functions are available for tasks like text manipulation (uppercase, trim), date formatting, and mathematical calculations. These are directly used within mapping fields.
- **Custom IML Functions:** For more complex data transformations and custom logic, Make Enterprise customers can create Custom IML Functions using JavaScript (ES6). These functions allow multiple actions within a single module operation, reducing scenario complexity and optimizing operation consumption. They are defined in a dedicated "Functions" section and can be called from modules.
- **createJSON() and parseJSON() Functions:** These specific functions are available for use within Custom Apps to handle JSON strings embedded within JSON objects, ensuring proper parsing and creation of structured data. It is important to note that these are not directly available for use in the standard scenario editor, but rather within the custom app development environment.

Variables

Scenario variables allow for dynamic content handling and temporary data storage within a scenario's execution.

- **Purpose:** To store and reuse data dynamically across different modules within a single scenario run, or to pass data between modules without direct connections.
- **Setting Variables:** The "Set Multiple Variables" module is commonly used to define and assign values to variables. Variables can hold various data types, including JSON structures.
- **JSON Representation:** While the JSON representation of variables themselves in the blueprint is not explicitly detailed in the research , the values assigned to variables can be JSON strings or collections. The Parse JSON module is often used to convert JSON output into a structured format before saving it to a variable, allowing individual fields to be mapped. Similarly, Text Aggregator can be used to convert structured data into a JSON string to be stored in a variable.
- **Dynamic Content:** Variables enable dynamic content generation by allowing values to be programmatically set and referenced throughout the scenario.

Sub-scenarios

Sub-scenarios enable modularity and reusability by allowing one scenario (parent) to call another scenario (sub-scenario).

- **Purpose:** To break down large, complex automations into smaller, more manageable, and reusable units. This improves maintainability, debugging, and collaboration.
- **Configuration:** The parent scenario uses a "Call a subscenario" module to initiate the sub-scenario. The sub-scenario itself is configured with "Scenario Inputs" to define the expected data structure it will receive, and "Return Output" modules to define the structured data it will send back to the parent.
- **JSON Structures for Inputs/Outputs:** A recent feature release (April 17, 2025) introduced "scenario outputs" and "synchronous subscenarios," which allow sub-scenarios to return structured data directly back to the parent. This structured data can be of various types (Array, Collection, Date, Text, Number, Boolean) or custom JSON. This capability transforms sub-scenarios into functional units that behave like functions, providing clear, retrievable results. For FlowForge AI, this is particularly valuable for building modular workflows where AI agents can call specific scenarios as tools and receive structured responses for further processing.

VII. Make-Specific Features

Beyond core automation components, Make.com offers several platform-specific features that enhance scenario development, management, and integration.

Visual Scenario Builder

The visual scenario builder is the primary user interface for designing workflows in Make.com, allowing users to drag, drop, and connect modules to create automation flows. The underlying representation of this visual flow is embedded within the scenario's JSON blueprint. Specifically, the metadata.designer object within each module's entry in the flow array contains x and y coordinates that define the module's position on the visual canvas. This means that the visual layout of a scenario is directly encoded in its JSON structure, enabling programmatic understanding and potential manipulation of the visual arrangement.

Data Flow Mapping

Data flow in Make is managed by passing "bundles" of information between modules. The mapper field within each module's JSON configuration explicitly defines how data from preceding modules is transformed and mapped as input for the current module. This mapping can involve direct field-to-field assignments, complex transformations using built-in functions, or the handling of nested data structures. When an API returns a JSON string inside a JSON object, Make treats it as text unless functions like parseJSON() are used within custom apps to correctly parse it into a collection. Conversely, createJSON() can be used to format data into a JSON string when an API requires it in that format. Troubleshooting data flow often involves examining input/output bundles of modules, which can be downloaded as JSON files.

Execution History

Make provides detailed execution history and logs for each scenario run, which are invaluable for debugging and monitoring performance. Users can view the number of operations consumed for each run, inspect the flow of data through modules, and identify any errors or warnings. This

history allows for granular analysis of scenario behavior and resource consumption.

Team Management

Make supports team and organization structures, allowing for collaboration on scenarios and shared resources. This includes features for inviting users, updating user organization roles, and managing team-specific scenarios and permissions. The API provides endpoints for managing these organizational entities.

Template System

Make offers a template system, comprising "Public Templates" (created by Make and partners) and "Team Templates" (created by users within an organization). These templates serve as pre-built scenario foundations that users can customize. While templates simplify scenario creation, it's important to note that custom data structures are generally not saved within templates or blueprints, as they are stored at the Organization level. This implies that FlowForge AI, when generating scenarios from templates, may need to handle data structure definitions separately or rely on dynamic inference.

API Integration

Make provides a comprehensive REST API that allows programmatic management of scenarios, modules, connections, and other platform entities. This API is crucial for FlowForge AI's ability to create, update, run, and validate scenarios directly. The POST /scenarios endpoint, for example, allows the creation of new scenarios by passing the scenario's blueprint as a string in the request body. The GET /scenarios/{scenarioId}/blueprint endpoint allows retrieval of an existing scenario's blueprint.

Custom Apps

Make's platform enables developers to build "Custom Apps" for integrating with services not natively supported or for creating highly specialized modules. Custom apps define their own Base (URL, authorization, error handling), Connections (OAuth, JWT), Webhooks (shared, dedicated), Modules (action, search, trigger, universal, responder), Remote Procedure Calls (RPCs), and Custom IML functions. This extensibility is vital for FlowForge AI to understand how custom integrations are structured and to potentially generate scenarios that leverage them.

VIII. Technical Specifications

Understanding Make's technical specifications is crucial for building efficient, reliable, and compliant automation solutions.

Rate Limiting

- **Operations:** Make's pricing model is based on "operations," with each module action consuming one operation. Exceeding monthly operation limits can pause scenarios. Optimization techniques like batch processing, strategic filtering, and using aggregators

can significantly reduce operation counts.
- **Webhook Rate Limits:** Make can process up to 30 incoming webhook requests per second. Webhook queues store data for scheduled processing, with limits depending on the user's plan (e.g., 667 items per 10,000 operations, up to 10,000 items max).

Data Limits

- **File Size Limits:** The maximum file size that can be processed by modules varies by plan: Free (5 MB), Core (100 MB), Pro (250 MB), Teams (500 MB), and Enterprise (1,000 MB).
- **Data Transfer Restrictions:** The Free plan has a data transfer limit of 100 MB per month.
- **Data Store Record Size:** Individual records within Data Stores have a maximum size of 15 MB.
- **Character Limits:** While not universally specified, character limits can apply to module inputs, potentially truncating long text fields.
- **Custom Function Code Size:** JavaScript code for Custom Functions must be under 5,000 characters.

Scheduling Options

Scenarios can be scheduled to run at specific intervals or triggered instantly.

- **Interval Scheduling:** Scenarios can be set to run periodically (e.g., every 15 minutes for Free plan, 1 minute for Core/Pro/Teams). This involves polling triggers checking for new data.
- **Instant Triggers:** Webhooks enable scenarios to execute immediately upon receiving an external HTTP request.
- **Cron Expressions:** More advanced scheduling can be achieved using Cron expressions, providing granular control over execution times.

Error Handling

Make provides a comprehensive suite of error handling mechanisms to ensure scenario robustness and data integrity.

- **Error Routes:** Scenarios can define specific error routes that activate when a module encounters an error. These routes can be configured to perform actions like sending notifications or logging errors.
- **Error Handlers:** Make offers several built-in error handler modules:
    - **Break:** Stops the scenario execution for the current bundle, optionally creating an incomplete execution.
    - **Commit:** Forces a commit of changes made by preceding modules before stopping.
    - **Ignore:** Allows the scenario to continue processing other bundles, effectively skipping the problematic one.
    - **Resume:** Replaces the erroring bundle with a substitute output, allowing the scenario flow to continue with modified data.
    - **Rollback:** Reverts changes made by preceding modules in the current execution path.

- **Retry Logic:** Incomplete executions can be retried, either automatically or manually. Some errors, like rate limit errors, can be fixed with exponential backoff.
- **JSON Error Handling:** When dealing with JSON, common errors include "Bad Request body," "Validate input JSON," or "Unprocessable Entity". These often stem from improper JSON formatting or special characters. Make provides functions or patterns (e.g., replace() with regex) to escape special characters in JSON strings. HTTP modules can be configured to not throw an error on specific HTTP status codes (e.g., 400) if the API response body contains error details to be processed. Error handlers (Break, Commit, Ignore, Resume, Rollback) and Routers do not consume operations.

Security

Make.com places a strong emphasis on security and compliance to protect user data.

- **Data Encryption:** All passwords and data at rest are stored in an encrypted format using industry-standard AES-256 encryption and AWS Key Management Service (KMS). Data in transit is secured with TLS versions 1.2 and 1.3 using AES 256 encryption.
- **Compliance:** Make.com is compliant with GDPR and has successfully completed SOC 2 Type 1 audits. For Enterprise plans, the infrastructure is ISO 27001 certified and SOC2 compliant.
- **Access Control:** Robust access control measures are in place, with the hosting environment accessible only via a private network and VPN, preventing direct public internet access. Single Sign-On (SSO) is supported with Google, Facebook, and GitHub, with custom SSO options for Enterprise customers. Advanced role and team management functions are also available.
- **Application Security:** Strict vulnerability management processes, regular penetration testing by independent third parties, and adherence to OWASP coding standards are employed. Static Application Security Testing (SAST) is integrated into the Software Development Life Cycle (SDLC).
- **Customer Data Security:** Log data is stored for 30 days by default, with extended retention available for Enterprise plans. The Enterprise platform runs in a separately managed AWS environment, isolated from self-service cloud customers.

Performance

Optimizing scenario performance is crucial for efficient operation and cost management.

- **Optimization Techniques:**
    - **Batch Processing:** Processing multiple records in a single operation (where supported by the app's API) rather than individual operations can significantly reduce operation counts.
    - **Strategic Filtering:** Placing filters early in the workflow to eliminate unnecessary processing of irrelevant data bundles saves operations.
    - **Aggregator Modules:** Using Array Aggregator or Text Aggregator to combine multiple items into a single bundle reduces subsequent operations.
    - **Custom Functions:** For Enterprise users, Custom Functions written in JavaScript can combine multiple data transformations or actions into a single module operation, thereby optimizing operation consumption and reducing scenario complexity.
    - **Sequential Processing:** For high-volume data, configuring webhooks or scenarios

for sequential processing can manage load and ensure orderly execution, even allowing for large queues to be processed overnight.
- **Concurrent Executions:** Instant webhooks typically process requests in parallel by default, which can enhance throughput but requires careful consideration of downstream module limitations.
- **Monitoring:** Make provides dashboards and history logs to track scenario statistics, operation consumption, and identify bottlenecks or recurring errors. Notifications can be set up for warnings or errors.
- **Debugging:** The ability to run specific modules only, download input/output bundles, and use temporary Parse JSON modules for testing greatly aids in debugging and performance tuning.

IX. Make vs. Competitors

Make.com differentiates itself from competitors like Zapier and n8n through a blend of features, pricing, and technical capabilities.

Unique Make Features & Capabilities

- **Advanced Visual Interface:** Make offers a sophisticated "canvas-type" visual interface that allows for comprehensive visualization of the entire workflow as a diagram. This provides a superior understanding of data flows and conditions, facilitating the creation of complex structures with conditional branches, unlike Zapier's more linear approach.
- **Robust Data Transformation:** Make is renowned for its powerful data transformation and manipulation capabilities, including built-in functions and the ability to define custom IML functions (for Enterprise users). This allows for more precise and complex data handling compared to simpler platforms.
- **Comprehensive Error Handling:** Make provides robust error handling and debugging tools, including various error handler modules (Break, Commit, Ignore, Resume, Rollback), detailed execution logs, and options for automatic retries.
- **Value for Money:** Compared to Zapier's per-task pricing, Make's per-operation model often offers better value, especially for workflows involving multiple steps or high data volumes.
- **Modularity with Sub-scenarios:** The recent introduction of synchronous sub-scenarios with defined inputs and outputs allows for true modular workflow design, enabling scenarios to act like functions and return structured data, which is highly beneficial for AI-driven automation.
- **Custom App Development:** The platform provides extensive tools for building custom app integrations, allowing users to connect to any API or service.

JSON Structure Differences from n8n/Zapier

- **Make (Blueprint JSON):** Make's core scenario representation is a comprehensive JSON blueprint that includes not only module configurations but also visual layout metadata (x, y coordinates) and detailed scenario-level settings. This makes its scenarios highly portable and programmatically understandable. The blueprint is the definitive "code" for a Make scenario.

- **n8n:** As an open-source, self-hostable platform, n8n offers complete JavaScript/Python support and advanced AI integration via LangChain. Its technical capabilities are generally more advanced, allowing for direct manipulation of complex data structures and sophisticated algorithms. n8n excels for complex workflows and high data volumes, often with a steeper learning curve. Its workflow definitions are also JSON-based but might differ in structure and level of detail compared to Make's blueprints, especially concerning visual metadata or specific platform-level settings. n8n currently holds a lead in dedicated AI agent features, including native vector database integration and a built-in chat interface, which Make is rapidly iterating to match.
- **Zapier:** Zapier is characterized by its intuitive interface, vast integration catalog (6000+), and ease of use, making it ideal for beginners and rapid integration of standard SaaS applications. Its workflow definitions are simpler and task-based, abstracting much of the underlying technical complexity. Zapier charges per "task," which can become expensive for high-volume workflows. Its JSON representation of "Zaps" is typically less detailed and less exposed for direct manipulation compared to Make's blueprints or n8n's workflows.

Make-Specific Terminology and Concepts

FlowForge AI must be adept at understanding and generating Make-specific terminology:

- **Scenarios:** Automated workflows.
- **Modules:** Building blocks of scenarios (triggers, actions, searches).
- **Operations:** The unit of billing, representing a module's action.
- **Bundles:** Data packets passed between modules.
- **Connections:** Authentication details for integrated apps.
- **Data Stores:** Temporary, simple databases within Make.
- **Webhooks:** HTTP endpoints for instant triggers.
- **Functions:** Built-in or custom JavaScript functions for data transformation.
- **Variables:** Dynamic data containers within scenarios.
- **Aggregators:** Modules that combine multiple bundles into one.
- **Iterators:** Modules that split a single bundle (containing an array) into multiple bundles.
- **Routers:** Flow control modules for conditional branching.
- **Sub-scenarios:** Nested scenarios for modularity.
- **Blueprints:** The complete JSON definition of a scenario.
- **Templates:** Pre-built scenario foundations.
- **Custom Apps:** User-built integrations.

Migration Patterns and Compatibility Considerations

Migrating from Zapier to Make is generally considered relatively simple due to similar core concepts, albeit with a steeper learning curve for Make's more advanced features. Migrating from Make to n8n is of medium complexity, requiring adaptation to n8n's node-based approach and its more advanced technical capabilities. Migration from Zapier to n8n is the most complex due to fundamental architectural differences. FlowForge AI's focus on Make's JSON blueprint should facilitate future compatibility or migration analysis, as the underlying structure is explicitly defined.

X. Conclusions & Recommendations for FlowForge AI

The deep research into Make.com's documentation reveals a platform highly suitable for AI-powered scenario generation. Its fundamental reliance on JSON blueprints for scenario definition, coupled with a comprehensive API for programmatic control, provides a robust foundation for FlowForge AI. The detailed understanding of module parameters, data flow mechanisms, and operational nuances derived from this research directly addresses the core requirements for accurate and efficient scenario generation.

Key Conclusions:

1. **JSON as the Core Language:** Make scenarios, despite their visual builder, are fundamentally defined by JSON blueprints. This means FlowForge AI's primary task is to generate valid and functional JSON that adheres to Make's internal schema. The ability to export and import these blueprints is a critical validation point, confirming that the JSON itself is the executable representation.
2. **Structured Data Handling:** Make's explicit modules for Create JSON and Parse JSON, along with the concept of "data structures," provide clear patterns for handling complex JSON inputs and outputs. This is essential for the AI to correctly map and transform data between modules and external services.
3. **Operational Awareness is Crucial:** The detailed operation counting rules for different module types (triggers, actions, aggregators, iterators) and various technical limits (file sizes, data transfer, webhook rates) must be embedded in FlowForge AI's knowledge base. This enables the AI to generate not just functional, but also optimized and cost-efficient scenarios.
4. **Modularity and Reusability:** The introduction of synchronous sub-scenarios with defined inputs and outputs significantly enhances Make's modularity. This feature allows AI to generate more organized, reusable, and debuggable workflows, treating scenarios as callable functions.
5. **Community as a Knowledge Base:** While official documentation provides authoritative definitions, the Make community forum and shared blueprints offer invaluable real-world examples, workarounds, and insights into undocumented behaviors, which are crucial for the AI to handle diverse and complex user requests.

Recommendations for FlowForge AI Development:

Based on these conclusions, the following recommendations are put forth for FlowForge AI's development:

1. **Prioritize Blueprint Schema Learning:** The AI's training data should heavily emphasize a diverse corpus of exported Make scenario blueprints. This "learn by example" approach will allow the LLMs to internalize the intricate JSON structure, including module types, parameters, mappers, and metadata, more effectively than relying solely on abstract schema definitions.
2. **Deep Integration of Operation Costs:** The AI should be trained to understand and predict operation consumption for each module and scenario. This will enable FlowForge AI to suggest optimization techniques (e.g., using aggregators for batch processing, placing filters early) and generate scenarios that are mindful of user's operational quotas.
3. **Robust Data Type and Mapping Handling:** FlowForge AI must have a sophisticated understanding of Make's data types and the mapper field's functionality. This includes correctly generating Create JSON and Parse JSON modules, handling nested JSON structures, and applying appropriate built-in functions for data transformation.

4. **Comprehensive Error Route Generation:** The AI should be capable of generating scenarios that include appropriate error routes and rollback/retry logic. This requires understanding common error types, the purpose of different error handler modules, and how to structure error paths within the blueprint.
5. **Focus on Prioritized Make Features:** For initial development and to meet immediate success criteria, FlowForge AI should prioritize accurate generation for:
    - **Instant vs. Polling Triggers:** Correctly representing webhook triggers versus scheduled polling triggers in JSON.
    - **Bundle Processing:** Accurately handling how data bundles flow between modules, especially with aggregators and iterators, to ensure correct data transformation and efficient operation usage.
    - **Data Type Handling:** Ensuring precise mapping and transformation of text, numbers, dates, arrays, and complex objects.
    - **Custom Webhooks:** Generating correct configurations for custom webhook modules, including expected input structures.
    - **Scenario Templates:** Understanding the structure of templates and how to generate scenarios that can leverage or mimic them.
    - **Team Collaboration Features:** While not directly impacting scenario generation JSON, understanding how teams and permissions are represented in the API can inform future features around shared scenarios.
6. **Continuous Learning from Community Data:** Establish a feedback loop where FlowForge AI can learn from new community discussions, shared blueprints, and template updates. This will allow the AI to adapt to evolving Make features, discover new best practices, and address emerging edge cases.

By focusing on these technical and architectural aspects, FlowForge AI can achieve its goal of generating highly accurate, efficient, and validated Make scenarios from natural language descriptions, significantly enhancing the automation capabilities for its users.

Works cited

1. Automate Your Community using the make.com app - Bettermode, https://bettermode.com/hub/apps-integrations/post/automate-your-community-using-the-make-com-app-iDHMK88wg37CJTz 2. n8n vs Make vs Zapier [2025 Comparison]: Which automation tool should you choose?, https://www.digidop.com/blog/n8n-vs-make-vs-zapier 3. What is the Difference Between Make.com and Its Competitors? - vatech.io, https://vatech.io/blog/what-is-the-difference-between-make-com-and-its-competitors/ 4. Create custom scenario using make API - How To - Make Community, https://community.make.com/t/create-custom-scenario-using-make-api/12585 5. Scenarios - Make Developer Hub, https://developers.make.com/api-documentation/api-reference/scenarios 6. How to Use JSON in Make: A Comprehensive Guide - Amine Fajry, https://aminefajry.com/how-to-use-json-in-make 7. Get whole output JSON bundle - How To - Make Community, https://community.make.com/t/get-whole-output-json-bundle/24750 8. How to Export and Import Make.com Blueprints - Scenario #shorts - YouTube, https://www.youtube.com/watch?v=VF4jkZ6-m-Y 9. Make Help Center, https://help.make.com/ 10. How to read the documentation | Make Developer Hub, https://developers.make.com/custom-apps-documentation/how-to-read-the-documentation 11. Create your app | Make Developer Hub, https://developers.make.com/custom-apps-documentation/basics/create-your-app 12. Make -

Apps Documentation, https://apps.make.com/make 13. Scenarios > Blueprints - Make Developer Hub, https://developers.make.com/api-documentation/api-reference/scenarios-greater-than-blueprints 14. Get scenario blueprint | Make Developer Hub, https://developers.make.com/api-documentation/api-reference/scenarios/blueprints/get--scenarios--scenarioid--blueprint 15. How to Use a Custom API in Make? - How To - Make Community, https://community.make.com/t/how-to-use-a-custom-api-in-make/68621 16. creating a doc from dynamic data : r/Integromat - Reddit, https://www.reddit.com/r/Integromat/comments/1ij56wy/creating_a_doc_from_dynamic_data/ 17. Help Guide for Custom App - Make Community, https://community.make.com/t/help-guide-for-custom-app/54944 18. Save JSON output to variable - How To - Make Community, https://community.make.com/t/save-json-output-to-variable/46971 19. Nested JSON data - Custom Apps - Make Community, https://community.make.com/t/nested-json-data/70189 20. Struggling with creating JSON - How To - Make Community, https://community.make.com/t/struggling-with-creating-json/23966 21. Error handling with API integration using REST - Custom Apps - Make Community, https://community.make.com/t/error-handling-with-api-integration-using-rest/71009 22. In json file, i got the error in HTTP - How To - Make Community, https://community.make.com/t/in-json-file-i-got-the-error-in-http/64808 23. How to Map the JSON Data properly using Aggregator or Set Variables for update to Sheets? - Getting Started - Make Community, https://community.make.com/t/how-to-map-the-json-data-properly-using-aggregator-or-set-variables-for-update-to-sheets/76892 24. How to bring more than one item in json via array aggregator - How To - Make Community, https://community.make.com/t/how-to-bring-more-than-one-item-in-json-via-array-aggregator/27362 25. Help Needed: Iterator module results as separate JSON output? - Make Community, https://community.make.com/t/help-needed-iterator-module-results-as-separate-json-output/74457 26. Converting object keys to an JSON array - How To - Make Community, https://community.make.com/t/converting-object-keys-to-an-json-array/14318 27. Functions parseJSON() and createJSON() - Features - Make Community, https://community.make.com/t/functions-parsejson-and-createjson/71402 28. Do you know a solid way to Create JSON? - How To - Make Community, https://community.make.com/t/do-you-know-a-solid-way-to-create-json/10249 29. Search for a value from in json - Getting Started - Make Community, https://community.make.com/t/search-for-a-value-from-in-json/82907 30. How to set field of JSON - How To - Make Community, https://community.make.com/t/how-to-set-field-of-json/74612 31. Subscenarios: Use Make's "Run a Scenario" to pass data between scenarios, https://community.make.com/t/subscenarios-use-makes-run-a-scenario-to-pass-data-between-scenarios/76747 32. Pre-configure JSON data structure in shared template - How To - Make Community, https://community.make.com/t/pre-configure-json-data-structure-in-shared-template/82523 33. Make.com Limits - Features, https://community.make.com/t/make-com-limits/28305 34. Optimizing operation-usage of a scenario - How To - Make Community, https://community.make.com/t/optimizing-operation-usage-of-a-scenario/77256 35. Help creating a JSON data structure - How To - Make Community, https://community.make.com/t/help-creating-a-json-data-structure/78254 36. How to Fix JSON

Errors in Make.com (Escaping JSON, Validating, Hidden Characters), https://www.youtube.com/watch?v=18wgBZxOp0U 37. Extracting specific data/items from JSON - How To - Make Community, https://community.make.com/t/extracting-specific-data-items-from-json/53554 38. How to Convert a Google Sheets to Jsonl - How To - Make Community, https://community.make.com/t/how-to-convert-a-google-sheets-to-jsonl/25912 39. Trouble Uploading JSON to Create a Scenario (Beginner) - Make Community, https://community.make.com/t/trouble-uploading-json-to-create-a-scenario-beginner/71964 40. Custom app as flexible error handler - Make Community, https://community.make.com/t/custom-app-as-flexible-error-handler/58675 41. Error handler based on filter - How To - Make Community, https://community.make.com/t/error-handler-based-on-filter/63928 42. Help Creating JSON Structure - How To - Make Community, https://community.make.com/t/help-creating-json-structure/66972 43. JSON Manipulation - How To - Make Community, https://community.make.com/t/json-manipulation/29595 44. Help Needed: Creating JSON data structure - How To - Make Community, https://community.make.com/t/help-needed-creating-json-data-structure/60379 45. Json String value doesnt work with HTTP "Make a Request module", https://community.make.com/t/json-string-value-doesnt-work-with-http-make-a-request-module/47258 46. How to Map a JSON Response - How To - Make Community, https://community.make.com/t/how-to-map-a-json-response/41313 47. Seeking Best Approach for Scheduled Automation in make.com scenario, https://community.make.com/t/seeking-best-approach-for-scheduled-automation-in-make-com-scenario/82585 48. Your Best Make.com Workflows & Automation Tips : r/Integromat - Reddit, https://www.reddit.com/r/Integromat/comments/1htbo1f/your_best_makecom_workflows_automation_tips/ 49. A More Efficient Scenario - How To - Make Community, https://community.make.com/t/a-more-efficient-scenario/24790 50. How to configure Webhook in Custom App? - Make Community, https://community.make.com/t/how-to-configure-webhook-in-custom-app/67158 51. Saving an array of jsons as a variable - How To - Make Community, https://community.make.com/t/saving-an-array-of-jsons-as-a-variable/53661 52. Passing Fillout JSON to Subscenario Using Call Subscenario - How To - Make Community, https://community.make.com/t/passing-fillout-json-to-subscenario-using-call-subscenario/82932 53. Parse json - How To - Make Community, https://community.make.com/t/parse-json/25155 54. Consolidating operations: aggregate to JSON not working - How To - Make Community, https://community.make.com/t/consolidating-operations-aggregate-to-json-not-working/76061 55. Aggregator not Aggregating - How To - Make Community, https://community.make.com/t/aggregator-not-aggregating/71715 56. Parse JSON Output Bundle and Store in Data Store to Merge Logic Across Scenarios, https://community.make.com/t/parse-json-output-bundle-and-store-in-data-store-to-merge-logic-across-scenarios/53988 57. Webhook to JSON (invalid json) - How To - Make Community, https://community.make.com/t/webhook-to-json-invalid-json/27659 58. Blueprint with webhook at the beginning - How To - Make Community, https://community.make.com/t/blueprint-with-webhook-at-the-beginning/63552 59. Setting variables array with set variable module - superfluous characters - Make Community, https://community.make.com/t/setting-variables-array-with-set-variable-module-superfluous-characters/33414 60. Need help with JSON in Universal Module for Custom App - How To - Make Community,

https://community.make.com/t/need-help-with-json-in-universal-module-for-custom-app/65789 61. How to return a json object from a webhook? - Make Community, https://community.make.com/t/how-to-return-a-json-object-from-a-webhook/81028 62. How to format Open AI JSON data using text to structured data into separate parameters and data? - How To - Make Community, https://community.make.com/t/how-to-format-open-ai-json-data-using-text-to-structured-data-into-separate-parameters-and-data/39659 63. Scenario templates - Help Center, https://help.make.com/scenario-templates 64. Make.com Template - Etsy, https://www.etsy.com/market/make.com_template?ref=lp_queries_internal_bottom-6 65. Fetch new articles from RSS feeds and send data in JSON format - Make, https://make.com/en/templates/13171-fetch-new-articles-from-rss-feeds-and-send-data-in-json-format 66. make-blueprints/blueprint.json at main · okash1n/make-blueprints ..., https://github.com/okash1n/make-blueprints/blob/main/blueprint.json 67. This JSON is an Export Blueprint from Make.com (formely Integromat ..., https://gist.github.com/helLf1nGer/ac8c7351411c65e660f700b575629947 68. Creating your first schema - JSON Schema, https://json-schema.org/learn/getting-started-step-by-step 69. Understanding blueprints in Make.com: the secret to automating and backing up your scenarios - benocode, https://benocode.vn/en/blog/operation/blueprints-in-make-com 70. Webhooks - Make Help Center, https://help.make.com/webhooks 71. Operations - Help Center, https://help.make.com/operations 72. Iterator - Make Help Center, https://help.make.com/iterator 73. Using OPC Router to Log OPC Data to JSON Files Without Code, https://blog.softwaretoolbox.com/opc-router-opc-data-to-json-files-without-code 74. How to Easily Use the Router Module in Make - Amine Fajry, https://aminefajry.com/router-module-in-make 75. Router - Make Help Center, https://help.make.com/router 76. JSON - Apps Documentation, https://apps.make.com/json 77. Processing of JSON strings inside a JSON object - Make Developer Hub, https://developers.make.com/custom-apps-documentation/other/processing-of-json-strings-inside-a-json-object 78. Email - Gmail Module | Resolve Documentation, https://docs.resolve.io/express/Configuration/Modules/Integration%20Modules/email-gmail-module/ 79. Gmail and JSON Integration | Workflow Automation - Make, https://www.make.com/en/integrations/google-email/json 80. Configure the module Email > Send an Email to a Team Member | Make Developer Hub, https://developers.make.com/white-label-documentation/install-and-configure-apps/configure-the-module-email-greater-than-send-an-email-to-a-team-member 81. Sending Email | Gmail - Google for Developers, https://developers.google.com/workspace/gmail/api/guides/sending 82. Google Drive - Apps Documentation - Make, https://apps.make.com/google-drive 83. Upload file data | Google Drive, https://developers.google.com/workspace/drive/api/guides/manage-uploads 84. Google Drive and JSON Integration | Workflow Automation - Make, https://www.make.com/en/integrations/google-drive/json 85. Salesforce - Apps Documentation - Make, https://apps.make.com/salesforce 86. Salesforce to Slack via Make.com | Astrea IT Services, https://astreait.com/Salesforce-to-Slack-via-makedotcom/ 87. Shopify - Apps Documentation - Make, https://apps.make.com/shopify 88. MySQL - Apps Documentation - Make, https://apps.make.com/mysql 89. How to Load Data from JSON File to MySQL Destination? - Airbyte, https://airbyte.com/how-to-sync/json-file-to-mysql-destination 90. Episode-039 - Insert and Select JSON Data in MySQL - YouTube, https://www.youtube.com/watch?v=Bih5md-PRbs 91. How to create and insert a JSON object using MySQL queries? - Stack Overflow, https://stackoverflow.com/questions/40930896/how-to-create-and-insert-a-json-object-using-mys

ql-queries 92. The Complete Guide to Make.com Pricing: Understanding Costs, Plans, and Maximizing Value - vatech.io, https://www.vatech.io/tutorial/what-is-make-com-pricing 93. Data stores - Make Help Center, https://help.make.com/data-stores 94. JSON Integration | Workflow Automation - Make, https://www.make.com/en/integrations/json 95. Working with files - Make Help Center, https://help.make.com/working-with-files 96. Working with files - Mapping - Make, https://www.make.com/en/help/mapping/working-with-files 97. Integrations | Software Integrations | 2,000+ Integration Apps | Make, https://www.make.com/en/integrations/category/databases?community=1&verified=1 98. 10 Make Integrations to Automate Workflows in 2025 - ClickUp, https://clickup.com/blog/make-integrations/ 99. Make.com Error Handling Quick Reference Guide - How to Handle Errors - YouTube, https://www.youtube.com/watch?v=pw0z-6pnk94 100. Simplify Automation with Make's Custom Functions, https://www.make.com/en/blog/custom-functions-in-make-best-practices 101. Improve AI automations with structured data in scenario outputs - Make, https://www.make.com/en/blog/introducing-scenario-outputs 102. Scenario inputs and scenario outputs - Make Help Center, https://help.make.com/scenario-inputs-and-scenario-outputs 103. JSON Flow - Visual Studio Marketplace, https://marketplace.visualstudio.com/items?itemName=imgildev.vscode-json-flow 104. How to Track Performance on Make.com - Value Added Tech, https://www.vatech.io/tutorial/how-to-track-performance-on-make-com 105. Resume error handler - Make Help Center, https://help.make.com/resume-error-handler 106. Automation Security & Compliance | Make, https://www.make.com/en/security 107. Privacy notice - Make, https://www.make.com/en/privacy-notice 108. n8n vs. Make.com: A Comparison of AI Agent Capabilities in Automation - AI Fire, https://www.aifire.co/p/n8n-vs-make-com-a-comparison-of-ai-agent-capabilities-in-automation