STRETCHLEARN · WORKBOOK

# Make.com Automation — Workbook

This workbook turns the course into a working Make.com practice: an inventory of the manual tasks worth automating, a first two-module scenario, a mapping and filtering plan, a branching and looping design, and an error-handling and operations-budget review. Work one section per module, building each piece in your real Make account as you go and filling each worksheet. By the end you will have a small library of live, documented scenarios with the error handling to run them unattended, instead of a pile of fragile half-built automations.

## The Make Canvas and Your First Scenario

Inventory your repetitive tasks, pick the cheapest high-value one, build it as a two-module scenario, and set a schedule that fits the job and your operations budget.

### Exercise: Inventory the Tasks Worth Automating

For one week, list every repetitive app-to-app task you do by hand, then score each for automation fit. This is the most important input to the whole project, because it tells you which glue work actually fills your day. Do this before building anything.

• List 10 to 15 tasks where you copy or retype data from one app into another, and note how often each happens.

• For each task, write down the two or three apps involved and whether Make connects to them natively.

• Estimate the minutes each task costs per week and rank the list by total time spent.

• Tag each task as Simple two-app move, Needs branching or a list, or Too rare to bother, and circle your first candidate.

### Worksheet: First Scenario Spec Sheet

Pin down the smallest useful version of your first automation before opening the canvas. Keep it to a trigger plus one action so it is cheap to run and easy to debug. Fill this out and build exactly what it describes.

Trigger app and event (e.g. Google Sheets Watch New Rows)

Action app and event (e.g. Slack Create a Message)

Fields to map from trigger into the action (list the exact field names)

Schedule interval and active hours/days

Test plan: what test record will I add, and what output proves it worked?

### Worksheet: Operations Budget Planner

Estimate the monthly operation cost of this scenario before you turn scheduling on, so cost is a design decision rather than a surprise. Compare the result against your plan allowance and adjust the schedule if needed.

Modules in the scenario (count = operations per run)

Runs per day at chosen schedule (e.g. every 15 min = 96)

Estimated operations per month (modules x runs per day x ~30)

My plan's monthly operation allowance (e.g. Core = 10,000)

Adjustment if over budget (slower schedule / earlier filter / webhook instead of polling)

### Checklist: First-Scenario Launch Checklist

[ ] Trigger and action connections created and authorized successfully
[ ] Run once completed with a real test record and the expected output appeared
[ ] Each module's data bubble inspected to confirm the right values flowed through
[ ] Watch trigger's starting point and max-results-per-run set so it does not import a backlog
[ ] Scenario renamed clearly and scheduled at the slowest interval the task tolerates

## Mapping, Transforming, and Filtering Data

Map the right fields with the right types, transform messy values with built-in functions, and add early filters so only qualifying records cost operations.

### Exercise: Map a Run and Catch the Type Mismatches

Run your scenario once with real sample data, then walk the mapping panel field by field to verify each mapped value lands in a field of the matching type. Most beginner failures are type mismatches, so find them here before they break a live run.

• Run once, then open the trigger's output bubble and note the data type of each field you use (text, number, date, array).

• For each mapped destination field, confirm the type it expects matches the type you mapped in.

• Flag any field that is sometimes empty in your data, and decide on a fallback value for it.

• Identify any array fields (bracketed icon) that will need an iterator rather than a direct map.

### Worksheet: Function Transformation Worksheet

For each value that arrives in the wrong shape, plan the function that fixes it in transit. Test each one against the live preview in the mapping panel before saving. Reuse this whenever a destination app rejects your data.

Source field and the problem (extra spaces / wrong casing / raw timestamp / currency symbol / blank)

Function to apply (trim / lower / upper / formatDate / formatNumber / replace / ifempty)

Exact format string or fallback value used

Live preview result (paste what the panel showed)

Nested functions needed? (e.g. lower(trim(email)) for a clean match key)

### Exercise: Add an Early Filter and Prove It

Place a filter as close to the trigger as possible so unqualified bundles stop before any expensive work. Test with records that should and should not pass. A bundle stopped at the first line costs nothing; one stopped at the last wasted everything before it.

• Decide the single condition that defines a record worth processing (e.g. Amount greater than 5000 AND Stage equals Closed Won).

• Confirm each filter operator matches the field's data type (Greater than for numbers/dates, Contains/Equals for text).

• Run once with a record that should pass and one that should not, and confirm only the right one continues.

• Test a record missing the filtered field to see whether it passes or blocks, and add an Exists check if needed.

### Checklist: Data-Quality Checklist

[ ] Scenario run once so the mapping panel holds real sample data, not empty placeholders
[ ] Every mapped field's type matches the destination field's expected type
[ ] Optional or sometimes-blank fields given a fallback with ifempty
[ ] Messy values (dates, casing, spacing, symbols) normalized with functions and confirmed in the live preview
[ ] A labeled filter placed early so unqualified bundles stop before expensive modules

## Branching and Looping: Routers, Iterators, and Aggregators

Design routers that send records down the right path, iterators that process every item in a list, and aggregators that fold many items back into one clean output.

### Worksheet: Router Branch Design Sheet

Plan each branch of your router as one clear question about the record, with its own filter and purpose, before building. Order specific branches first and a catch-all last. Keep this beside you while you wire the router.

Branch label (its one-line purpose, e.g. Urgent tickets to on-call)

Branch filter condition(s) and operator

Modules on this branch (what it does when matched)

Branch order position (specific cases first)

Catch-all branch defined last with no filter? (yes/no)

### Exercise: Build an Iterate-Then-Act Loop

Take a real list field, such as order line items or email attachments, and unpack it with an iterator so the modules after it run once per item. Watch the operations multiply and decide whether you need every item. This pattern handles most list-shaped business data.

• Identify the array field you need to process and map it into a new Iterator module.

• After the iterator, add an action and confirm the panel now offers one item's fields (name, quantity, price, etc.).

• Run once and verify the action ran the correct number of times (one per list item).

• Note the operations the run used, and decide whether a filter after the iterator should cut it down.

### Exercise: Collapse Many Bundles Into One Digest

Use an aggregator to turn many bundles into a single combined output, such as one summary email instead of fifty. Getting the source module right is the part beginners miss, so test that the combined output contains every record.

• Choose the aggregator type for your goal (Text for a digest, Array for a bulk action, Numeric for a sum or count).

• Set the aggregator's source module to the point where the group of bundles begins.

• Map the per-bundle content with a line break so each record becomes one formatted line.

• Run once and confirm the single output contains every record, not just the last one.

### Checklist: Branching and Looping Checklist

[ ] Each router branch has one focused filter and a clear label, with a catch-all branch placed last
[ ] Array fields are unpacked with an iterator before any per-item module
[ ] Operations cost of iterators reviewed, with a filter cutting the list down where possible
[ ] Aggregator source module set correctly so the combined output captures the full group
[ ] Iterate-then-aggregate used where a list must be processed item by item then recombined

## Reliable, Production-Ready Automations

Add error handling that retries or recovers, switch to webhooks and the data store where they fit, watch the operations budget, and document each scenario for handoff.

### Worksheet: Error-Handling Plan Sheet

Decide in advance how each fragile module should behave when it fails, so a single bad record or brief outage never silently stops the automation. Match the directive to the situation, and always add an alert for failures a human must see.

Module most likely to fail and why (external API / missing field / rate limit)

Error directive chosen (Resume / Ignore / Rollback / Break / Commit) and why

Retry settings if Break (number of attempts and interval, e.g. 3 attempts every 15 min)

Incomplete-executions storage enabled? (yes/no)

Failure alert path (Slack channel or email that gets notified on a persistent error)

## Exercise: Decide Webhook vs Polling and Add Memory

For each scenario, choose between scheduled polling and an instant webhook, and decide whether it needs a data store to avoid double-processing. Webhooks give speed; the data store gives memory. Test the deduplication so the same record is never handled twice.

• For this scenario, does speed or volume justify a webhook, or is polling every 15 minutes fine? Decide and note why.

• If using a webhook, set up the Custom Webhook module, capture one real event, and confirm the fields appear in the panel.

• Decide what unique key (email, order ID) identifies a record, and whether a data store should remember processed keys.

• Test the dedupe path: send the same event twice and confirm the scenario only acts on it once.

## Worksheet: Scenario Inventory and Documentation Sheet

Document every scenario so the system survives handoff and the 3 a.m. failure. Fill one row per scenario and keep this inventory current; it is the first thing a teammate or future-you will need.

Scenario name (descriptive, e.g. New Order to Sheet plus Slack)

Trigger type and schedule (polling interval or webhook)

Apps and connections it touches

Webhooks or data stores it depends on

Owner and one-line purpose/assumptions note

## Checklist: Production-Readiness Checklist

[ ] Scenario tested with Run once on safe/test destinations before pointing at production
[ ] Error handler added to fragile modules with the right directive (Break+retry for outages, Ignore for batch rows)
[ ] Incomplete-executions storage on and a failure alert wired so errors reach a human
[ ] Operations budget reviewed: schedule slowed, filters early, webhooks used where speed matters
[ ] Scenario named clearly, noted, and added to the inventory with its owner and dependencies

# Your Action Plan

1. Spend one week logging repetitive app-to-app tasks and rank them by time spent.
2. Pick the simplest high-value task and spec it as a two-module trigger-plus-action scenario.
3. Build it, Run once with a test record, inspect every data bubble, then schedule it at the slowest tolerable interval.
4. Run once with real data and verify every mapped field's type matches its destination.
5. Normalize messy values with functions (trim, lower, formatDate, ifempty) and confirm each in the live preview.
6. Add an early labeled filter so only qualifying records consume operations, and test pass and fail cases.
7. Where records need different handling, add a router with focused branches and a catch-all placed last.

8. Unpack list fields with an iterator, then aggregate where you need one combined output instead of many.
9. Add error handlers (Break with retries, plus a failure alert) and turn on incomplete-executions storage.
10. Choose webhooks over polling where speed matters, dedupe with a data store, then document every scenario in the inventory.