The work behind the workflow

What automation actually does, and how the code makes it happen.

A task becomes an automation when software can recognize when to start, find the right information, apply a decision, and carry out the next step.

The back end is the part you do not see: programs running on a server or an automation platform. They receive events, read and save records, and ask other systems to perform actions. A visual workflow builder arranges much of this logic for you.

01 / Receive an event

A form submission can send a webhook: an automatic message to a receiving address. Alternatively, a scheduled job can check an app for new records. The incoming data might include a lead ID, email address, and service requested.

02 / Check and organize the information

The program verifies the request, checks required fields, and matches the record to the correct customer. A database stores durable information such as lead status, scheduled follow-up time, and which actions have already happened.

03 / Apply the rules and connect the tools

An API is a defined way for one program to request work from another. Using a secure connection, the workflow can ask your customer database to update a record or your email service to send an approved message. Conditions decide which path runs.

04 / Run at the right time, then record the result

A scheduler or job queue holds work until it is due. A worker picks it up, checks the latest status, and performs the action. The system records success or failure so the owner can see what happened and investigate exceptions.

A practical next step

Have one recurring task in mind? Donahue Works can help you map the steps and scope an automation around it. Bring the task, the tools you use, and the result you want.

From a reminder to a rule

One example: checking whether a lead needs a follow-up.

STAGE 1 / A PERSON CARRIES THE PROCESS

Open the lead list. Find unanswered inquiries. Check when the last message went out. Check for a reply or booking. Write the next message, send it, and update the record. Repeat tomorrow.

The friction: the process depends on someone remembering, checking several places, and recording the result consistently.

STAGE 2 / WRITE THE DECISION AS A SMALL RULE

Before connecting any apps, define eligibility. In this simplified Python example, a lead is due only when its scheduled time has arrived and it has not replied, booked, or opted out.

```
def needs_follow_up(lead, now):
    return (
        lead["follow_up_at"] <= now
        and not lead["replied"]
        and not lead["booked"]
        and not lead["opted_out"]
    )
```

Read it as: "The follow-up is due, and none of the stop conditions is true." The rule only returns yes or no; it does not send a message.

STAGE 3 / CONNECT THE RULE TO AN ACTION

A scheduled worker reads lead records, applies the rule, and asks the messaging service to send approved copy. It then records the outcome and schedules the next step, if the agreed sequence allows one.

A small rule is the beginning

The surrounding connections, current data, scheduling, and error handling turn that rule into something the business can rely on.

From code to a working system

The simple decision stays small. The supporting system makes it dependable.

STAGE 4 / PUT THE PIECES TOGETHER

CAPTURE
Form saves lead

>

STORE
Database keeps state

>

CHECK
Worker applies rule

>

ACT
Service sends message

Illustrative pseudocode: named helpers stand in for real integrations.

```
for lead in database.due_leads(now):
    if needs_follow_up(lead, now):
        key = make_key(lead.id, lead.sequence_step)
        result = messaging.send_once(
            key=key, to=lead.email,
            text=approved_message(lead)
        )
        database.record_result(lead.id, result)
```

This is a teaching example, not deployable code. The helpers must be implemented, and a send-once guarantee depends on the provider and storage design.

WHAT A PRODUCTION WORKFLOW ALSO NEEDS

Fresh data + stop rules

Check replies, bookings, and opt-outs before sending. Route uncertain cases to a person.

Duplicate protection

Use persistent job IDs and atomic claims. Where supported, an idempotency key lets retries reuse the same operation rather than create another one.

Retries + visibility

Retry temporary failures with limits and increasing delays. If delivery is uncertain, reconcile the provider status before resending; alert an owner when needed.

Secure access + testing

Keep credentials outside code, verify incoming requests, limit access, and test normal, duplicate, failure, and stop paths before launch.

Further reading: Stripe webhook documentation [1] and idempotent requests [2]. These illustrate integration patterns; your tools may behave differently.

[1] Webhooks **[2] Idempotent requests**