Getting Started

Studio Basics · https://docs.functionize.com/hc/en-us/articles/42177580685591-Getting-Started

Functionize Studio lets you create, run and maintain tests by describing what you want in plain English. This guide covers what it is, how to phrase your first request, and what the agent does with it.

functionize **Studio**
The Studio workspace

www.functionize.com

THE LEFT-HAND NAVIGATION, AND WHAT EACH AREA IS FOR

New Session	The starting point for everything. Describe what you want in plain language — a new test, a change to an existing one, an explanation of a failure. The **+** on the prompt box attaches a screenshot when a picture is clearer than a paragraph. · `studio.functionize.com/app`
Projects	The container for tests, and the level at which settings live — access and auth, cookies, timeouts, browser behavior, execution presets. One project per application or environment. · `/app/projects`
Sessions	Every conversation you have had with the agent, kept. Reopen one to continue where you left off rather than re-explaining. · `/app/sessions`
Tests	Every test, with status, tags, last updated, last run and browsers. Filter by tag. Open one to see its steps, its history and its settings. · `/app/tests`
Orchestrations	Groups of tests run as one job, on demand or on a schedule. This is where your smoke, critical and regression tiers live, and where your time-to-feedback number comes from. · `/app/orchestrations`
Workflows	Scheduled jobs written in plain language that produce a *result* rather than a verdict — a price watch, a daily summary. Not a test; it does not pass or fail. · `/app/workflows`
Reports	Execution breakdown, trends per day and pass rate by project, over a date range, with export. Weekly and monthly shape — not a live view. · `/app/reports`
Assets → Extensions	Your own code — Node, Python, Go or Java — running inside a test. For the things no description can do: generating a one-time passcode, reading a mailbox, calling an internal API. · `/app/extensions`
Integrations → MCP	Connection details for driving Studio from Claude Code, Claude Desktop, Cursor, VS Code, Gemini CLI, Codex CLI, ChatGPT and others. · `/app/mcp`

Where settings are: not in the sidebar. Project-level settings are inside the project; a single test's overrides are behind **Test Settings** in the test editor toolbar. The rule of thumb is to change things at project level so every test inherits it, and override on a single test only when that test genuinely differs.

Every area in the left-hand navigation, and what it is for

www.functionize.com

What is Functionize Studio?

Functionize Studio is a chat interface built into the platform. You describe what you want in plain English, and the agent handles it, creating tests, running them, diagnosing failures, and fixing tests when your application changes.

How it works

Behind the chat are specialized agents for creating, executing, diagnosing, and maintaining tests. They identify elements on the page visually, contextually, and structurally, not through CSS selectors or XPath. When your application changes, the model adapts how it finds an element instead of breaking because a selector moved. Tests self-heal. This is also why you describe what a user does, not how the UI is built.

One chat interface, multiple agents

Everything happens in a single chat. You type a message and the system routes it to the right agents, in the right order. "Fix this test and rerun it" runs diagnosis, then the repair, then a fresh run, automatically. If a repair does not hold up, the system can re-check before running. And if the first attempt does not resolve the issue, it can try again using the context from what it just tried, which is why a second attempt often works when the first did not.

Infrastructure

Functionize is cloud-hosted. We manage the infrastructure. Tests execute on isolated virtual machines, and the platform scales to run thousands of tests in parallel on enterprise plans. There is nothing to install or maintain on your side.

Where to find it

Open any test in Functionize. The test editor has three panes: the agent chat on the left, the test's steps in the middle, and the browser and its data on the right. The chat is the single interface for everything described above: creation, execution, diagnosis, and maintenance. Type a message and press Enter.

If you do not have a test yet, that is fine. The first thing we will do is create one.

Your first few minutes

Before you start: pick a project and make sure your target URL is set in the environment settings. If you need a new project or environment, ask the agent. It can set those up for you.

1. Create a test

Tell the agent what you want to test. Use your own application URL, or try a public site:

> "Create a test that goes to https://en.wikipedia.org, searches for 'software testing', and verifies the search results list shows entries for 'software testing'."

The agent builds the test steps from your description: navigate, type, click, verify. Generation takes about 30 to 90 seconds depending on complexity. You will see each step appear one by one in the editor. Wait for the agent to finish before running or editing the test.

Tip: Describe the flow in terms of what a user does, not how the UI is structured. "Log in with test credentials" is better than "click the email field, type the email, click the password field." Store real usernames and passwords as secret project variables (Project → Variables) and refer to them by name, e.g. `{{fze.project.test_user}}`. Never type an actual password into the chat. It would be saved in the test and captured in screenshots.

2. Run it

> "Run this test."

The agent executes the test and returns pass/fail results with step-level detail, including screenshots at each step.

3. If it passed

You have a working test. A few things to try next:

> "Does this test follow best practices?"

The agent reviews timing, settings, and structure and recommends improvements.

Or document what you built:

> "Can you document this test?"

Or set it to run on a schedule:

> "Create an orchestration called Morning Check that runs this test every day at 8am."

An orchestration is a saved schedule that runs one or more tests automatically.

If your test passed, you are done. The steps below cover what to do when a test fails.

4. If it failed, diagnose

> "Why is this test failing?"

This gives you a read-only diagnosis. The agent analyzes screenshots, network logs, and DOM state without touching the test. If you are combining steps in one prompt ("diagnose this, then fix it"), add "do not fix it" to pause between analysis and repair. On its own, a diagnostic prompt is already read-only, so the phrase is optional.

5. Fix and confirm

> "Fix this test and rerun it to confirm."

The agent repairs the test and runs it to verify the fix worked. If it fails again, move to the next step.

6. Revert and try again

> "Revert all changes you made."

This restores the test to its exact state before the session started. Now give the agent more context:

> "The test fails at step 3. I expected the dropdown to appear after clicking the menu, but it did not. Try fixing with that in mind."

Tip: Describing expected vs. actual behavior is the most useful thing you can add. It gives the agent a specific hypothesis instead of starting from scratch.

If the second attempt does not work, revert and reach out to support@functionize.com with the test ID and what you tried.

What to try next

Find your goal in the left column. Use the prompts in the two columns to the right.

I want to...	Say this	Then try
Fix a failing test	Fix this test and rerun it.	What did you change?
Understand why it failed	Why is this test failing?	What is the issue in step 3?
Create another test	Create a test that [describe the flow].	Does this test follow best practices?
Modify a test	Make step 2.3 wait for the results table to appear before continuing.	Make that action optional.
Run and check results	Run this test.	Show me the results from the last run.
Set up scheduled runs	Create an orchestration called [name].	Schedule it to run daily at 8am.

Key phrases

These phrases change how the agent behaves. They work in any context.

Phrase	What it does
Do not fix it.	Analysis only. Pauses between diagnosis and repair in a compound prompt.
Fix and rerun.	Repairs the test then runs it to confirm.
Revert all changes you made.	Restores the test to its state before the session started.
Do not run it.	Applies a fix without running the test afterward.
Try fixing step 3.2 again, the dropdown did not open.	Retries a specific repair with context on what went wrong.

What did you do?	Explains exactly what the agent changed and why.

Good to know

Changes are reversible. Any change the agent makes in a session can be undone with "revert all changes you made," which restores the test to its state before the session started.

Generated is not the same as done. A test is not finished when the steps appear. It is finished when it runs green and its final step checks the thing you actually care about. Generated steps can be incomplete or drift from your intent, so end a test on a specific outcome, "verify the results list shows entries for 'software testing'" rather than "verify the page loads."

The agent sees more than you do. On every run, the agent has access to step-level screenshots, network logs, console output, and DOM state. When you ask it to diagnose a failure, it checks all of these, not just the error message.

Sessions start fresh. Each conversation is independent. The agent does not remember previous sessions. But any changes you save to a test persist normally between sessions.

One prompt can chain related steps. "Fix this test, rerun it to confirm, then tell me what you changed" is a single efficient message. But keep unrelated work in separate messages, and keep separate user flows in separate tests. One test should prove one thing.

Where to go next

- Your First Test in Functionize Studio: A Visual Walkthrough — build one, with screenshots
- Starting a Session: The Studio Home Screen — where every piece of work begins
- Your First 30 Minutes with Functionize Studio — a guided first half hour
- Projects: Boundaries, Defaults and What Inherits What — the container everything else sits in