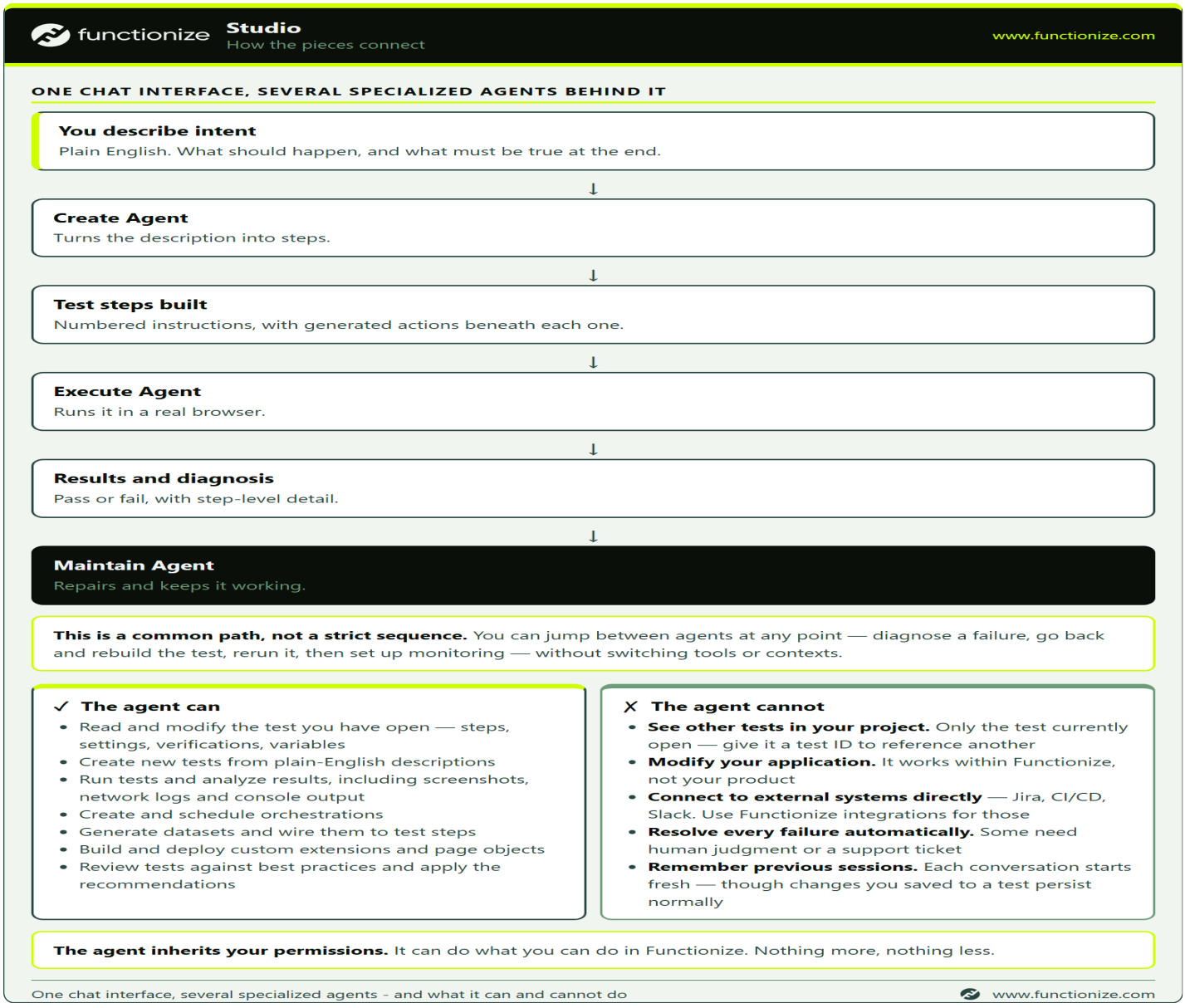
Agent Prompt Guide

Studio Basics · https://docs.functionize.com/hc/en-us/articles/42178282951063-Agent-Prompt-Guide

This guide explains how to prompt the Functionize AI Agent well: how the specialized agents connect to each other, what to say to get the result you want, and the patterns that produce reliable tests.

01 How the pieces connect



The Functionize AI Agent is a set of specialized agents that handle different parts of the test lifecycle. You interact with them through a single chat interface.

You describe intent → Create Agent → Test steps built → Execute Agent → Results and diagnosis → Maintain Agent

You describe what you want in plain English. The agent figures out which operation to perform and handles it without you switching tools or contexts. The flow above is a common path, not a strict sequence. You can jump between agents at any point: diagnose a failure, go back and rebuild the test,

rerun it, then set up monitoring.

What the agent can and cannot do

The agent can:

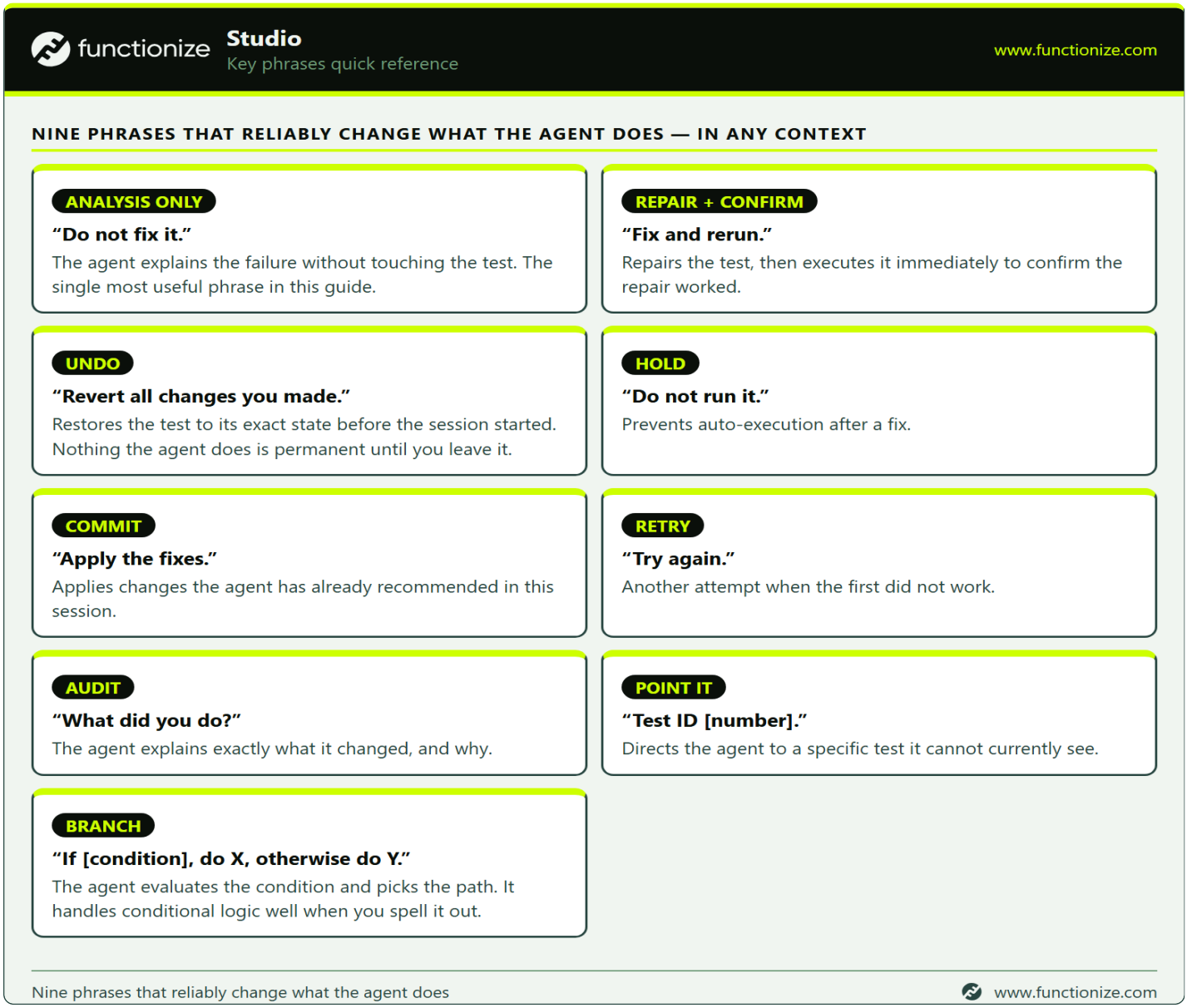
- Read and modify the test you have open: steps, settings, verifications, variables
- Create new tests from plain-English descriptions
- Run tests and analyze results including screenshots, network logs, and console output
- Create and schedule orchestrations (test suites)
- Generate datasets and wire them to test steps
- Build and deploy custom extensions and page objects
- Review tests against best practices and apply recommended changes

The agent cannot:

- See other tests in your project. It only sees the test currently open (provide a test ID to reference another)
- Modify your application. It works within Functionize, not your product
- Connect to external systems directly (Jira, CI/CD, Slack). Use Functionize integrations for that
- Resolve every failure automatically. Some issues require human judgment or a support ticket
- Remember previous sessions. Each conversation starts fresh, but any changes you save to a test persist normally

The agent inherits your permissions. It can do what you can do in Functionize, nothing more, nothing less.

02 Key phrases quick reference



These phrases reliably change agent behavior. They work in any context.

Phrase	Effect
Do not fix it.	Analysis only. The agent explains the failure without making changes
Fix and rerun.	Repair the test then immediately execute to confirm
Revert all changes you made.	Restores the test to its exact state before the session started
Do not run it.	Prevents auto-execution after a fix
Apply the fixes.	Applies changes the agent already recommended in this session
Try again.	Another attempt if the first did not work
What did you do?	The agent explains exactly what it changed and why

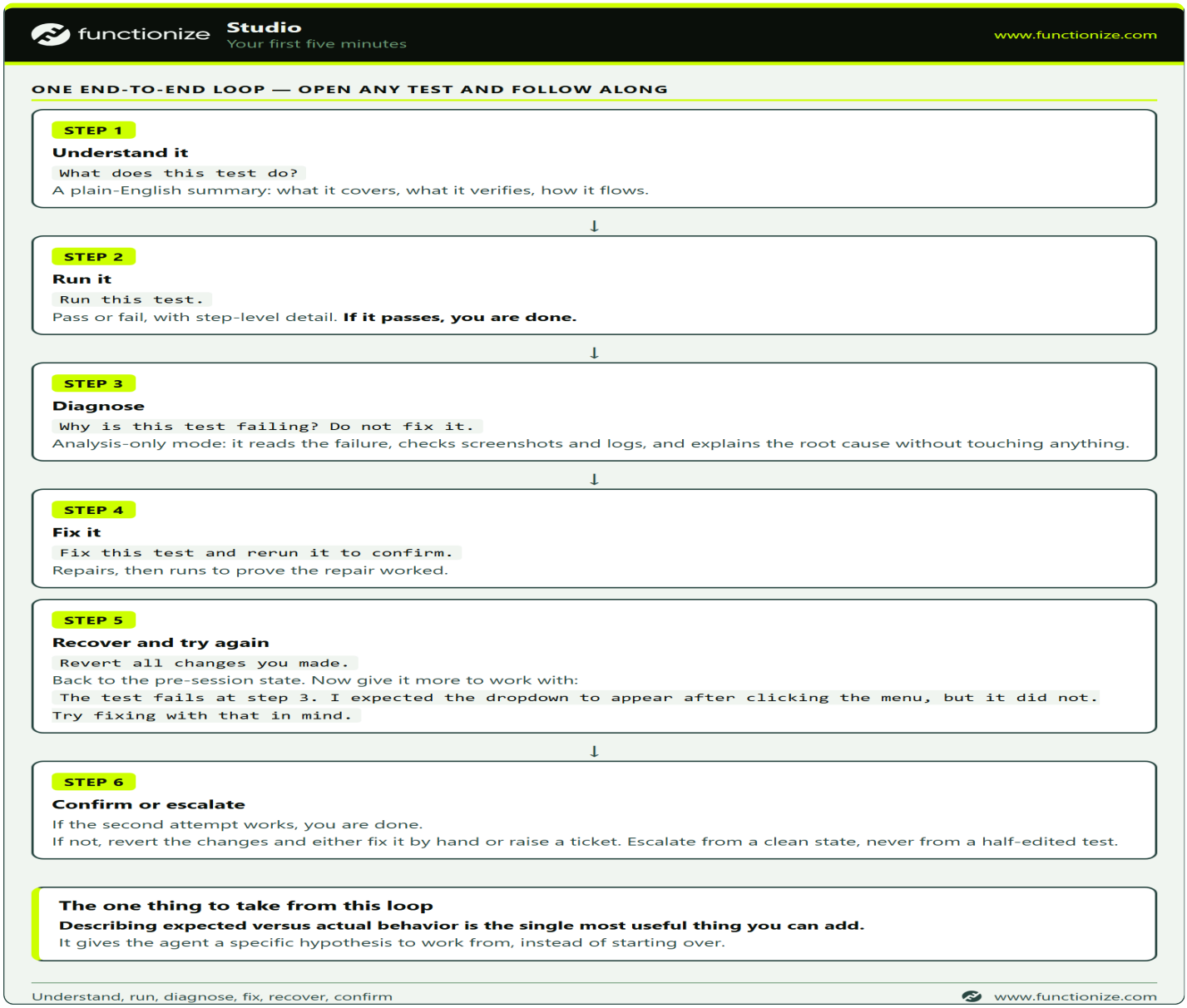
Test ID [number].	Directs the agent to a specific test it cannot currently see
If [condition], do X, otherwise do Y.	Conditional logic. The agent evaluates and picks the right path

03 Find what you need by goal

Find the row that matches your situation, then use the prompts in the sections that follow.

I want to...	Start with this	Then try
Understand why a test failed	Why is this test failing? Do not fix it.	What is the issue in step 3?
Fix a failing test	Fix this test and rerun it.	Fix and rerun.
Create a new test	Create a test that [describe the flow].	Does this test follow best practices?
Modify an existing test	Wait until the order-confirmation header is visible before continuing.	Make that action optional.
Adjust test settings	Show me all the current test settings.	Change the timing model to 5.
Run and check results	Run this test.	How many times has this test passed?
Understand what a test does	What does this test do?	Can you document this test?
Set up monitoring	Create an orchestration called [name].	Schedule it to run daily at 8am.
Work with test data	Create a dataset with 100 rows of customer data.	Map the data file to step 3.

04 Your first 5 minutes



One end-to-end flow that shows how the agent works. Open any test in Functionize and follow along.

Step 1: Understand what you are looking at

> What does this test do?

The agent reads every step and returns a plain-English summary: what the test covers, what it verifies, and how it flows.

Step 2: Run it

> Run this test.

The agent executes the test and returns pass/fail results with step-level detail. If it passes, you are done. If it fails, move to step 3.

Step 3: Diagnose the failure

> Why is this test failing? Do not fix it.

Adding "do not fix it" switches the agent to analysis-only mode: it reads the failure, checks screenshots and logs, and explains the root cause without touching the test.

Step 4: Fix it

> Fix this test and rerun it to confirm.

The agent applies a repair based on its analysis, then runs the test to confirm the fix worked. If it passes, you are done. If it fails again, continue to step 5.

Step 5: Recover 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.

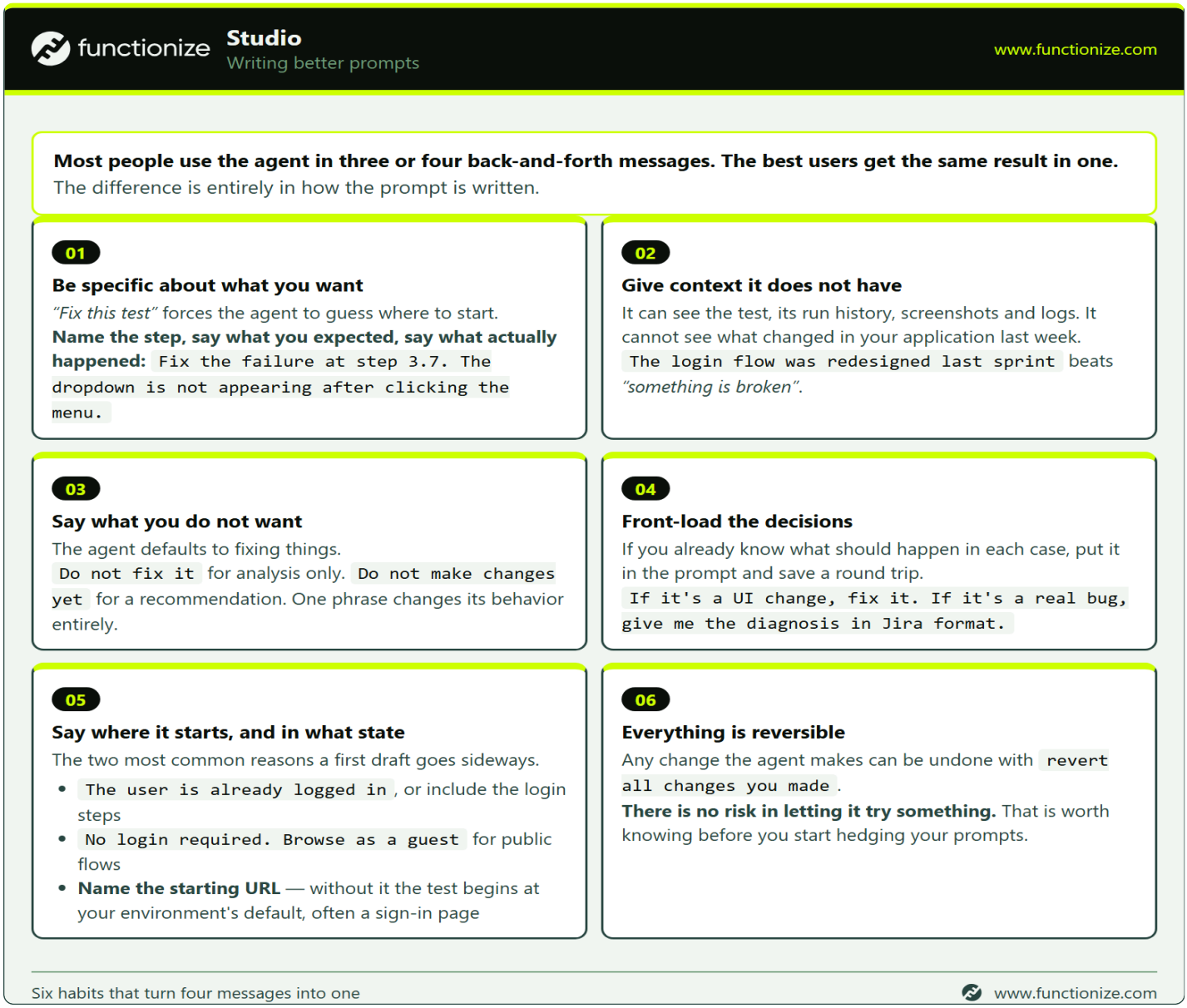
Tip: Describing expected vs. actual behavior is the single most useful thing you can add. It gives the agent a specific hypothesis to work from instead of starting over.

Step 6: Confirm or escalate

If the second attempt works, you are done. If not, revert changes and fix manually or submit a support ticket. Every ticket helps Functionize improve the agent.

---

05 Writing better prompts



Most people use the agent in three or four back-and-forth messages. The best users get the same result in one. The difference is how the prompt is written.

Be specific about what you want

"Fix this test" forces the agent to guess where to start. "Fix the failure at step 3.7. The dropdown is not appearing after clicking the menu" gives it a target. Name the step, describe what you expected, and say what actually happened.

Give context the agent does not have

The agent can see the test, its run history, screenshots, and logs. It cannot see what changed in your application last week. If you know something relevant, say it. "The login flow was redesigned last sprint" is more useful than "something is broken."

Say what you do not want

The agent defaults to fixing things. If you want analysis only, say "do not fix it." If you want a recommendation without execution, say "do not make changes yet." This single phrase changes the agent's behavior entirely.

Front-load decisions

If you already know what the agent should do in different scenarios, include that logic in your prompt. "If it's a UI change, fix it. If it's a real bug, give me the diagnosis in Jira format" saves a round trip. The agent handles conditional logic well when you spell it out.

Say where the test starts and what state it's in

Two of the most common reasons a first draft goes sideways are an unclear starting point and an unclear starting state. Spell both out:

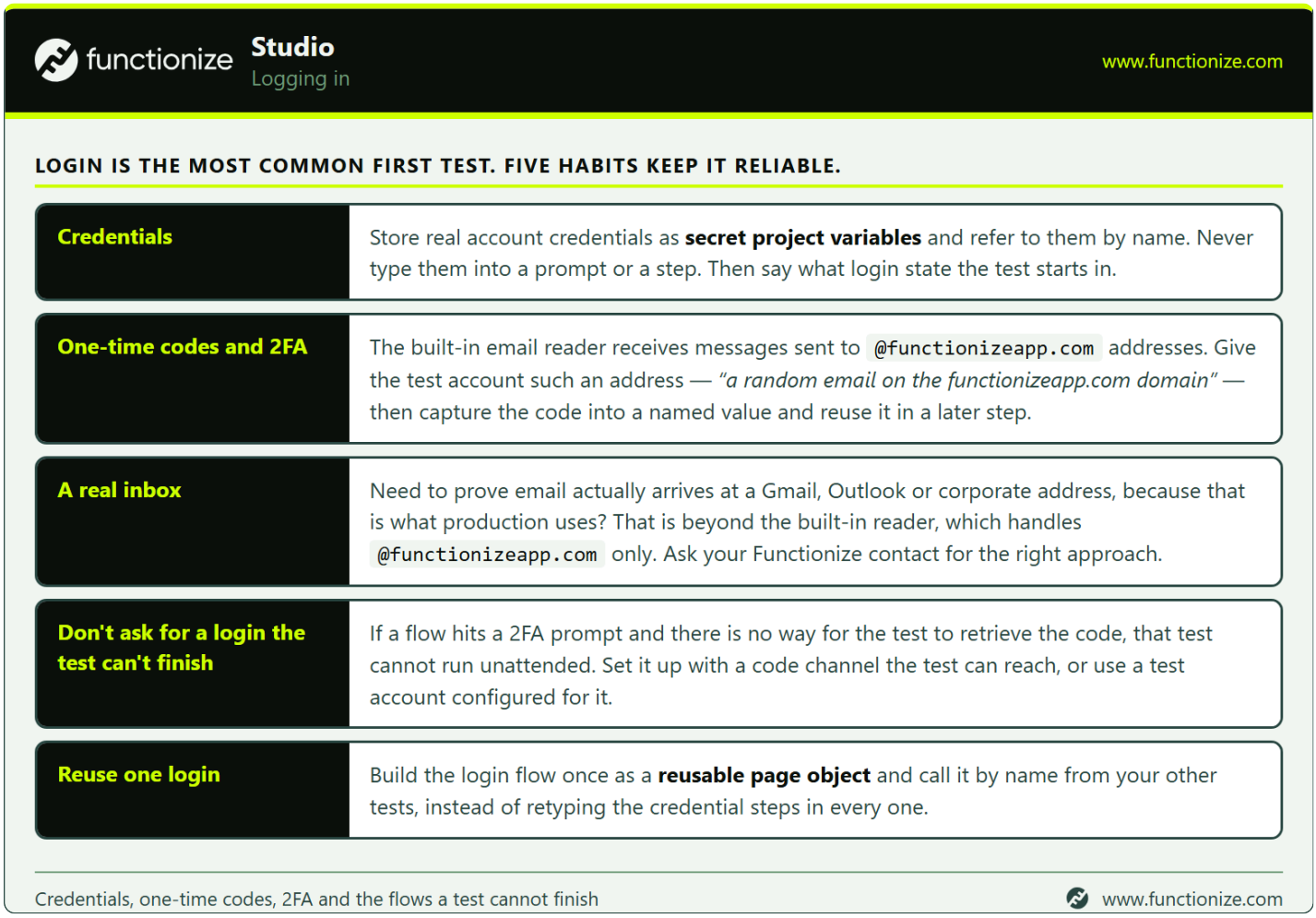
- "The user is already logged in", or include the login steps.
- "No login required. Browse as a guest" for public flows. Without this, the agent may add a login step you didn't ask for.
- Name the starting URL. If you don't, the test begins at your environment's default URL, which is often a sign-in or portal page rather than the screen you meant.

Everything is reversible

Any change the agent makes to a test can be undone. Say "revert all changes you made" and the test goes back to its pre-session state. There is no risk in letting the agent try something.

---

06 Logging in (passwords, one-time codes, 2FA, SSO)



Login is one of the most common first tests. A few habits keep it reliable.

Credentials

Store real account credentials as secret project variables and refer to them by name (see the credentials note under Settings and configuration). Say what login state the test starts in. See "Say where the test starts" under Writing Better Prompts.

One-time codes and 2FA by email

The built-in email reader receives messages sent to `@functionizeapp.com` addresses. If your flow receives a code by email, use a `@functionizeapp.com` address for the test account, for example "a random email on the functionizeapp.com domain", so the test can read the code back. Then capture the code into a named value and reuse it (see Reusing a Value the Test Produces).

Testing delivery to a real inbox

If you need to test that email actually arrives at a real Gmail, Outlook, or corporate address, because that is what your production flow uses, that is beyond the built-in reader, which reads `@functionizeapp.com` addresses only. Ask your Functionize contact about the right approach for reading a real mailbox.

Don't ask for a login the test can't finish

If a flow hits a 2FA prompt and there is no way for the test to retrieve the code, that test can't run unattended. Set it up with a code channel the test can reach, or use a test account configured for it.

Reuse one login, don't re-script it

Build the login flow once as a reusable page object and call it by name from your other tests, instead of retyping the credential steps in every test.

---

07 Chaining: one prompt, multiple actions

A single prompt can instruct the agent to diagnose, decide, act, and format output without a second message. Decide what you want as the end state, work backwards, and write it into one instruction.

Fix, confirm, and summarize

> Fix this test, rerun it to confirm the fix worked, then give me a one-paragraph summary of what you changed and why.

Three actions, one prompt. Useful when you need to document what happened or share a change summary with your team.

Failure triage with conditional output

> Look at this failure. If something changed in the application, update the test to match and rerun it. If it's a real bug, do not change the test. Give me the diagnosis in Jira ticket format: summary, steps to reproduce, expected result, actual result, and relevant logs.

The agent diagnoses the failure, decides which category it falls into, and either fixes and reruns or returns a formatted ticket ready to paste into your tracker.

Build, check quality, and schedule

> Check this test against best practices. Apply any recommended changes to timing, timeouts, and settings. Then create an orchestration called Checkout Regression, add this test to it, and schedule it to run daily at 6am.

The agent audits the test, applies its own recommendations, then sets up ongoing execution. Three distinct phases from a single instruction.

Data setup, wiring, and scheduling

> Create a dataset with 100 rows of realistic customer data: first name, last name, email, and phone number. Add it to the input on step 2.3. Then create an orchestration called Customer Data Regression, add this test to it, and schedule it to run every Monday at 8am.

Four separate tasks, one prompt: generate the dataset, map it to a step, create the orchestration, set the schedule.

Data-driven testing in one prompt

> Create a data source with columns: username, password, expected_result. Add 5 rows: 3 valid logins that should succeed and 2 invalid logins that should show an error message. Map username to step 2 and password to step 3. Then run the test to confirm the data injection works.

When a test uses data from a data source, write the step in plain terms, "enter the username and password for this row", then connect each data column to the right step field in the platform after the test is generated. That column-to-field connection is made in the platform, so a prompt line like "read column X into field Y" won't create it on its own. After connecting the columns, run the test once to confirm the data is being used.

Cross-test orchestration

> List all orchestrations in this project. If there is one called Nightly Regression, add this test to it. If there is not, create it, add this test, and schedule it to run every night at 11pm. Either way, confirm what you did.

Conditional logic lets the agent handle both cases without asking you which one applies. It checks, decides, and confirms.

Tip: Include conditional logic ("if X, do Y, otherwise do Z") so the agent can adapt without waiting for your next message.

08 Reusing a value the test produces

When a later step needs a value an earlier step produced, an order number, a confirmation code, a one-time code from an email, capture it into a named value and refer to that name. Don't phrase it as "read it and type it in."

Instead of this	Say this
Read the verification code from the email and enter it.	Capture the verification code from the email as a value called `otpCode`, then enter `otpCode` on the next screen.

Phrased the naive way, the test can lock in whatever value happened to be there when it was built, so it works once and then fails the next time the real value is different. Capturing it into a named value keeps every later step pointed at the live value.

09 Failure patterns



Grouped by the failure you are actually seeing. Find your pattern and use the prompts below.

Element not found

What it looks like: "Missing element" error, element timeout exceeded, step fails before the interaction happens.

Usual causes: the page has not fully loaded, the element moved or was renamed, or a timing issue.

> Fix this test and rerun it.

The agent identifies what changed, adjusts its element identification strategy, and confirms the fix. If it needs more time to find the element:

> Update the missing element timeout to 10 seconds and try again.

If the element genuinely no longer exists in the application, tell the agent what changed:

> The login button was moved to a dropdown menu in the latest release. Update the test to match.

Verification mismatch

What it looks like: a verification step fails because the expected text does not match what the page shows.

> Why is this test failing? Do not fix it.

If the data legitimately changed:

> Update the verification to match the current application behavior.

If the verification logic is the problem:

> Update step 2.15 to use contains logic instead of equals.

Timing or loading issue

What it looks like: the test passes sometimes and fails other times, or fails consistently on slower environments.

> Optimize the timing and performance settings on this test.

You can also be specific:

> Change the timing model to 5 and set the page load timeout to 20 seconds.

Application changed

What it looks like: the test was passing last week, and now multiple steps fail.

> Look at this failure. If something changed in the application, update the test to match and rerun it. If it looks like a real bug, do not change the test. Just give me the diagnosis.

Data-driven test failures

What it looks like: the test passes for some data rows but fails for others, or fails on a specific row.

> Which rows in the dataset failed? Show me the row number, the input values, and the failure reason for each.

If a specific row is the problem:

> Why did the test fail on row 47 but pass on row 1? Compare the two runs.

When the agent misdiagnoses

The most common reason is not enough context. Give it more specific information:

Prompt	Why it helps
It failed at step 3. I expected X but got Y.	Expected vs. actual is the most useful thing you can add

Can you look at the network logs for step 3 specifically?	Directs the agent to a data source it may not have checked
The job ID for the failing run is 60246851.	A run ID lets the agent look at exact execution data
Something changed in the app recently. The login flow was updated.	Gives the agent a hypothesis instead of guessing

10 Reporting and failure analysis

Prompt	What it does
Analyze the failures in this orchestration. Report with test case name, failure reason, and root cause. Do not fix anything.	Read-only failure report. The "do not fix" keeps it analytical.
Generate a report at the environment level: pass count, fail count, and pass percentage from May 1 to May 6.	Date-ranged stats by environment. Useful for weekly status reviews.
Give me execution count, passes, failures, and pass rate for April.	Monthly summary across the project.
Add the environment name as a column in the report you just generated.	Builds on a report from the same session. The agent carries context forward.

Tip: Name the columns you want. "Generate a report" gets the agent's best guess. "Report with test name, status, failure reason, and root cause" gets you exactly that.

11 Optimization and stability

To speed up a slow test, first find the bottleneck, then ask the agent to make those steps faster and confirm the total run time dropped.

Prompt	What it does
This test takes 18 minutes. Show me which steps are the bottleneck.	Execution time breakdown by step. Start here before applying fixes.
Make the slowest steps faster, then rerun and tell me the new total time.	Speeds up the slowest steps, reruns the test, and reports whether the total time dropped.
Are there steps with low element selection confidence? What can be improved for stability?	Surfaces flaky steps and recommends element identification changes.

12 Prompt reference

A flat lookup for prompts not covered in the sections above. For diagnosis and failure-related prompts, see Failure Patterns. For reporting, see Reporting and Failure Analysis.

Creating tests

Prompt	The agent will
Create a test that searches Google for Formula 1.	Generate a simple search test
Make a test on https://hawksshop.com/ that adds a hat to the cart.	Generate an e-commerce interaction test
Create a test that goes to google.com, searches for 'cat', and verifies the results.	Generate a test with a verification step

Generate an end-to-end regression test for https://www.fandango.com.	Generate broad coverage across the main flows. Split large journeys into separate tests and chain them
Create a test with a small viewport to simulate a mobile web test.	Generate a mobile-simulated test

Tip: When you verify, check a specific, stable outcome. "Verify the order total is $51.25" beats "verify the page looks right," and a value that stays on the page beats a banner that disappears.

Modifying tests

Prompt	The agent will
Delete step 1.21.	Remove a specific step
Wait until the order-confirmation header is visible before continuing.	Insert a wait anchored to a visible element or state
Update step 2.15 to use equals logic for the full expected text.	Change the match logic on a verification
Make that action optional.	Allow the step to continue even if it fails
Update step 3 to input the project variable `map_value`.	Map a variable as a step's input
Add a step to verify that the word 'door' does not appear in search results.	Add a negative verification

Anchor a wait to something visible, not to a number of seconds. "Wait until the results grid appears" adapts to slow and fast runs alike. "Wait 10 seconds" is either too short and flaky, or wasted time. Tell the agent the element or state the next step depends on, and let it wait for that.

Caution: An optional step that gets skipped can still let the test report a pass. Use it sparingly, and don't make the step that proves your outcome optional.

Settings and configuration

Prompt	The agent will
Show me all the current test settings.	Return a full configuration summary
Update the missing element timeout to 10 seconds.	Update that specific timeout
Set the page load timeout to 20 seconds.	Update the page load timeout
Change the timing model to 5.	Set the timing model to the recommended default
Enable continue on verification failure.	Allow the test to continue instead of stopping on verification errors
Set the region so this test runs on the west coast.	Set the geographic execution region
Set HTTP auth using the credentials stored as secret project variables.	Use credentials saved once as secret project variables. Keeps the real values out of the test and the chat.
Set HTTP auth. User: tomtom, password: 921mmdsiq91.	Add basic authentication credentials

Credentials: Two ways to give the agent credentials. Store the value once as a secret project variable and refer to it by name. Best for real accounts, since it keeps the value out of the test and the chat. Or pass it inline in the prompt, as in the last row above.

Running and checking results

Prompt	The agent will
Run this test.	Execute the current test
Execute my test using live execution.	Run with a live browser you can watch in real time

How many times has this test passed?	Return pass/fail statistics across all runs
What are all the executions on this test from today?	Return today's run history
Evaluate the error on run ID 60246851.	Analyze a specific historical run

Tip: Live execution opens a real-time browser view so you can watch the test run step by step. Useful for debugging complex interactions or confirming visual behavior.

Understanding and documenting tests

Prompt	The agent will
What does this test do?	Return a plain-English summary of the test's purpose
Can you document this test?	Generate inline documentation
Explain step 3.	Explain a specific step's action and intent
How many steps are in this test?	Return the total step count
Who created this test?	Return author and creation metadata
Is this test in a passing state?	Check the most recent run result
Does this test follow best practices?	Review quality, reliability, and recommended settings

---

13 Multi-turn workflows

When you want a back-and-forth conversation instead of a single chained prompt.

Workflow 1: Diagnose, fix, and confirm

#	Who	Message
1	You	Why is this test failing? Do not fix it.
2	Agent	Failing at step 3.2. The login element is not found within the timeout. The page structure changed. The element moved into a nested container.
3	You	Fix it and rerun to confirm.
4	Agent	Fixed. Updated the element identification for step 3.2 and reran the test. Test passed.

Workflow 2: Fix fails, revert, try again

#	Who	Message
1	You	Fix this test and rerun it.
2	Agent	Applied changes to step 4 and reran. Test failed again at the same step.
3	You	Revert all changes you made.
4	Agent	Reverted. Test is back to its original state.
5	You	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 and rerun.

6	Agent	Added a wait before the dropdown interaction and updated the element strategy. Reran the test. Test passed.

Workflow 3: Create, optimize, and schedule

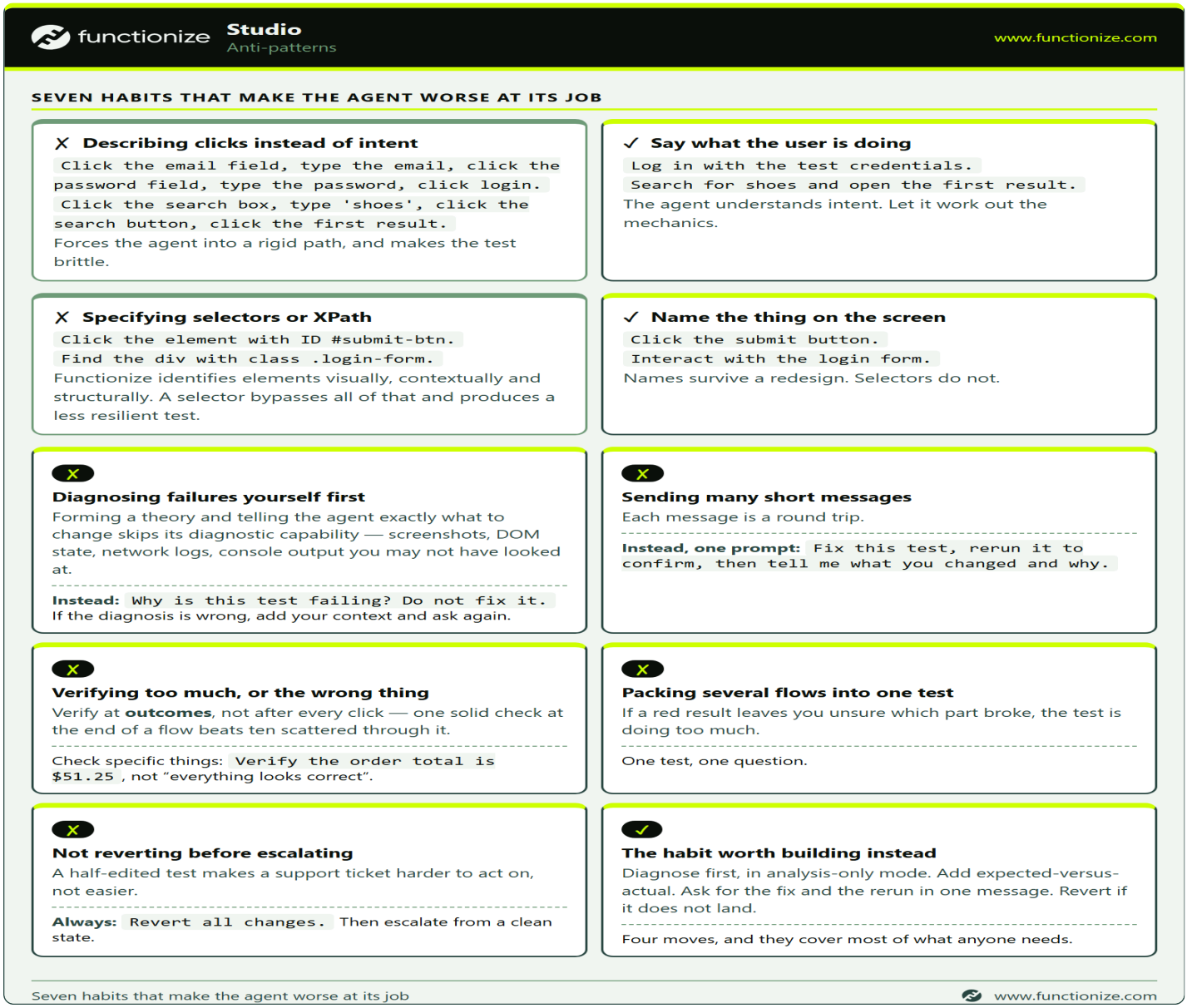
#	Who	Message
1	You	Create a test that searches Google for Formula 1 and verifies the F1 logo appears.
2	Agent	Created a test with 4 steps: navigate to Google, enter search term, submit search, verify F1 logo is visible.
3	You	Does this test follow best practices?
4	Agent	Mostly. I would recommend increasing the missing element timeout to 10 seconds and adjusting the timing model to 5.
5	You	Make those changes, then create an orchestration called F1 Daily that runs this test every day at 8am.
6	Agent	Applied the settings changes. Created the F1 Daily orchestration and scheduled it to run daily at 8am.

Workflow 4: Diagnose and escalate

#	Who	Message
1	You	Fix this test and rerun it.
2	Agent	Applied fixes to step 5 and adjusted timing. Reran the test. Still failing. The element at step 5 is not rendering in the DOM during execution.
3	You	Revert all changes.
4	Agent	Reverted. Test is back to its original state.

At this point, fix the issue manually or submit a support ticket at support@functionize.com. Always revert agent changes before escalating so the test is in a clean state.

14 Anti-patterns: common mistakes



Describing clicks instead of intent

Instead of this	Say this
Click the email field, type the email, click the password field, type the password, click login.	Log in with the test credentials.
Click the search box, type 'shoes', click the search button, click the first result.	Search for shoes and open the first result.

The agent understands intent. Describing individual clicks forces it into a rigid path and makes tests more brittle.

Specifying selectors or XPath

Instead of this	Say this
Click the element with ID #submit-btn.	Click the submit button.
Find the div with class .login-form.	Interact with the login form.

Functionize uses AI-based element identification, not selectors. The platform identifies elements visually, contextually, and structurally. Specifying selectors bypasses this and produces less resilient tests.

Diagnosing failures manually

Users often form a theory and tell the agent exactly what to change. This skips the agent's diagnostic capability, which has access to screenshots, DOM state, network logs, and console output that you may not have checked.

> Why is this test failing? Do not fix it.

Let the agent diagnose first. If its diagnosis is wrong, add your context and ask it to try again.

Sending too many short messages

Each message is a round trip. Instead of three separate messages:

> Fix this test, rerun it to confirm, then tell me what you changed and why.

Verifying too much, or verifying the wrong thing

Good verification is about placement and specificity:

- Verify at outcomes, not after every click. One solid check at the end of a flow beats ten redundant ones scattered through it.
- Check specific things, not "everything looks correct." "Verify the order total is $51.25", not "verify the page looks right."
- Verify things that stay on the page, not things that disappear. Check that the record saved or the field changed, not a pop-up or banner that auto-dismisses.
- For content that changes often, feeds, listings, live prices, check the shape ("at least one result appears") rather than the exact value that happens to be there today.

Instead of this	Say this
Verify the page looks right.	Verify the order total is $51.25.
Verify everything on the results page.	Verify that at least one result appears.

Packing multiple flows into one test

Keep each test to one flow with one reason to fail. Don't log in as an admin, create a product, switch to a guest, buy it, then delete it, all in one test. Split those into separate tests and chain them together with an orchestration. Long single tests are harder to diagnose and more likely to break partway through.

Not reverting before escalating

If the agent made changes that did not work, always revert before submitting a support ticket or fixing manually.

> Revert all changes you made.

---

15 Advanced capabilities

Orchestrations and test suites

Prompt	The agent will
List all my orchestrations.	Return all orchestrations in the project
Create an orchestration called F1 Daily that runs this test every day at 4am.	Create a scheduled suite
What time does the RegressionPack orchestration run?	Return the schedule for a named orchestration
Give me a report of this orchestration with test name, status, failure reason, fault, and summary.	Generate a structured run report

Data and variables

Prompt	The agent will
Show me all the project variables.	List project-level variables
Create a new project variable called `env_url` with value `staging.app.com`.	Add a variable
Create a data source with headers first, last, and email, and populate 5 rows.	Generate a test data CSV
Map the data file to step 3's input action.	Link test data to a specific step

Page objects and extensions

Extensions are custom code blocks that run at specific points in a test step's lifecycle. Page objects are reusable element collections. Use extensions when you need logic the agent cannot handle natively: OCR, custom validation, API calls during execution.

Prompt	The agent will
Import a page object in this test case.	Add a page object to the current test
Create an extension called OCRProcessor that reads the post-step screenshot and returns the text as `ocrData`.	Build a custom code extension
Attach the OCRProcessor extension to step 2 as POST_STEP.	Link the extension to a step and lifecycle hook
Redeploy the existing extension.	Push the latest version

Team and project info

Prompt	The agent will
List the users in this project.	Return all users with project access
Show me all the users in my team.	Return the full team member list
Show environment configurations for this project.	Return environment-level settings
List the orchestrations in this account.	Return all orchestrations across the account

---

Need help?

If the agent cannot resolve an issue after two attempts, revert your changes and reach out to support@functionize.com. Include the test ID, a description of what you tried, and any relevant run IDs. The support team can also help with advanced configurations, extension development, and CI/CD integration.

Related

- Writing Prompts That Produce Durable Tests — the shorter, opinionated version
- Reviewing AI-Generated Tests: What a Human Still Has to Check — what to check before you trust the output
- Exercise 4 - Change a Test by Asking — practice the phrasing