

# The Click-Await Pipeline

A literate walkthrough of wallabidi's LiveView-aware click

Three race conditions, one converging code path.

Commits [9caa5c6](#) · [6e007c3](#) · [ad8d0a2](#).

# The problem

When an integration test clicks a link, what does it mean for the click to be “done”? For a static HTML page the answer is obvious: the browser has navigated and the new DOM has parsed. For a Phoenix LiveView application, the answer is subtler. A single `<a>` tag may, depending on its attributes and the state of the page at the moment of the click, resolve to:

1. a full page load (`<a href="/x">`),
2. a LiveView `push_navigate` — same socket, new route, client-side,
3. a LiveView `push_patch` — no route change, just a diff,
4. no navigation at all — the event is handled server-side with a state update and nothing else,
5. or, under CI load, a delayed form of any of the above that starts arriving only after the test harness has already given up waiting.

Each branch requires a different “the click is done” signal, and picking the wrong signal causes *silent* test failures: the click returns, the next assertion runs, and the page is mid-transition. The failing assertion looks unrelated.

This document walks through the pipeline that wallabidi uses to resolve these cases, focusing on three race conditions that required changes to the pipeline between `v0.2.9` and `v0.2.12`.

## What we are explaining

The click path spans two files. The in-browser side lives in `lib/wallabidi/cdp/pipeline.ex` and is compiled to JavaScript that Chrome evaluates in one shot. The host-side dispatch lives in `lib/wallabidi/browser.ex` and `lib/wallabidi/live_view_aware.ex`. The JS decides *what happened* the Elixir decides *what to wait for next*.

We will read the code in the order it executes, starting from the user’s `click(session, query)`.

## Classification: what did the user just click?

Before any click is dispatched, the pipeline asks a simple question: if this element is clicked, what kind of transition will it cause? The answer — `"none"` · `"patch"` · `"navigate"` · `"full_page"` — is the key the host uses to select a post-click await strategy.

Classification runs in JS alongside the click so that both the decision and the action happen atomically, with no round-trip in between. Here is the classifier, lifted directly from `Pipeline.classify_fn/0`:

```

"""
function(el, type) {
  if (!el) return "none";
  if (type === 'click') {
    var link = el.closest('[data-phx-link]');
    if (link) return link.getAttribute('data-phx-link') === 'redirect' ?
'navigate' : 'patch';
    var phxClick = el.getAttribute('phx-click');
    if (phxClick) {
      if (phxClick.startsWith('[')) {
        // JS command list – pick the strongest navigation signal present.
        // JSON-quoted command names are robust enough to string-match.
        if (phxClick.indexOf('"navigate"') !== -1) return 'navigate';
        if (phxClick.indexOf('"patch"') !== -1) return 'patch';
        if (phxClick.indexOf('"push"') !== -1) return 'patch';
        return 'none';
      }
      return 'patch';
    }
    if (el.type === 'submit' || el.type === 'image' || el.tagName ===
'BUTTON') {
      var form = el.closest('form');
      // phx-trigger-action fires a native form submit after the LV event,
      // so a full page load is the load-bearing transition – await that,
      // not the preceding LV patch.
      if (form && form.hasAttribute('phx-trigger-action')) return 'full_page';
      if (form && form.getAttribute('phx-submit')) return 'patch';
      if (form) return 'full_page';
    }
    var anchor = el.closest('a[href]');
    if (anchor && anchor.getAttribute('href') && !
anchor.getAttribute('href').startsWith('#')) return 'full_page';
    return 'none';
  }
  if (type === 'change') {
    var phxChange = el.getAttribute('phx-change') || (el.form &&
el.form.getAttribute('phx-change'));
    return phxChange ? 'patch' : 'none';
  }
  if (type === 'submit') {
    var f = el.closest('form');
    if (f && f.hasAttribute('phx-trigger-action')) return 'full_page';
    if (f && f.getAttribute('phx-submit')) return 'patch';
    if (f) return 'full_page';
    return 'none';
  }
  return 'none';
}

```

```
}  
""
```

A few things are worth pointing out.

*The `data-phx-link` attribute wins.* Phoenix stamps this on any `<a>` it intends to handle client-side. If the attribute is present and reads `redirect`, the click becomes a `push_navigate` — classified as `"navigate"`. If it reads anything else (notably `patch`), we classify as `"patch"`. We check this before inspecting `phx-click`, because a `data-phx-link` on an anchor's ancestor dominates any click handler on the link itself.

*JS command lists.* LiveView lets you write `phx-click` as a list of commands: `[["push", {...}], ["navigate", {...}]]`. The classifier doesn't parse the list — it string-matches on the JSON-quoted command names, taking the strongest navigation signal present. Parsing would be more robust; string matching is what we need given that the classifier runs inside a JS expression and has to stay short.

*`phx-trigger-action` is a trap.* A form with this attribute does a LiveView event *and then* a native form submission. The load-bearing transition is the full page load, not the preceding LV patch — so we classify the form as `"full_page"`, even though the server will definitely send a diff first. If we classified it as `patch` and awaited the diff, the test would continue while the real navigation was still in flight.

This last case was the fix in commit `007238b` (`v0.2.9`). The integration suite had a form-submit flake that was traced back to awaiting the LV patch instead of the page load. The fix is three lines in the classifier, but finding it required tracing one specific assertion failure back through the wrong await branch.

# Dispatch: preparing, snapshotting, and clicking

The classifier's result is only useful if the host knows what the LiveView was *about* to do before the click. Two snapshots need to happen *before* the click fires, or the post-click await has no reference to compare against.

## The pre-click setup

`click_full` is the op that combines classification, promise setup, the ref snapshot, and the click itself into one JS function. Reading it top-to-bottom gives the full story:

```

"""
var _prepared = false;
var _classification = "none";
var _count = els.length;
var _preRef = null;
(function() {
  // 1. prepare_patch - install onPatchEnd hook + promise
  var ls = window.liveSocket;
  if (ls && ls.main) {
    if (!window.__wallabidi_patch_hooked) {
      var orig = ls.domCallbacks.onPatchEnd;
      ls.domCallbacks.onPatchEnd = function(container) {
        orig(container);
        if (window.__wallabidi_patch_resolve) {
          var r = window.__wallabidi_patch_resolve;
          window.__wallabidi_patch_resolve = null;
          r(true);
        }
      };
      window.__wallabidi_patch_hooked = true;
    }
    window.__wallabidi_patch_promise = new Promise(function(resolve) {
      window.__wallabidi_patch_resolve = resolve;
      window.addEventListener('beforeunload', function() {
        if (window.__wallabidi_patch_resolve) {
          window.__wallabidi_patch_resolve = null;
          resolve('navigated');
        }
      }, {once: true});
    });
    _prepared = true;
  }

  // 2. classify first element
  if (els.length > 0) {
    _classification = ({classify_fn: () => els[0], toJson: () =>
      (to_string(interaction))});
  }

  // 2a. Snapshot the LV view's ref counter. Every phx-* event push
  // grabs `view.ref++` for its server-side reply. After the click,
  // waiting for `view.lastAckRef >= _preRef` tells us the server has
  // finished processing whatever event our click triggered - even

```

```

    // if that processing takes longer than patch/load deadlines. This
    // closes the gap where a slow handle_event + push_navigate
    // completes AFTER wallabidi's post-click awaits have given up.
    if (ls && ls.main && typeof ls.main.ref === 'number') {
      _preRef = ls.main.ref;
    }
  })();

  // 3. Capture return value BEFORE clicking – the click may navigate
  // the page and destroy the execution context. _ret becomes a
  // Promise that resolves to the return map once the click has
  // been dispatched (step 5). evaluate is called with
  // awaitPromise: true so the RPC response waits for it.
  var _retValue = {count: _count, classification: _classification, prepared:
    _prepared, preRef: _preRef};

  // 4. Wait for the source LiveView to finish joining before
  // dispatching the click. If the LV is mid-join, Phoenix's
  // [data-phx-link] handler may not be bound yet and the synthetic
  // click falls through to the default anchor behaviour – the
  // teamology u2i/teamology#586 nav-click flake. Bounded by a 5s
  // deadline so a genuinely-broken LV doesn't hang indefinitely.
  var _ret = new Promise(function(resolve) {
    var ls = window.liveSocket;
    if (!ls || !ls.main || !ls.main.joinPending) return resolve();

    var deadline = Date.now() + 5000;
    function check() {
      var ls2 = window.liveSocket;
      if (!ls2 || !ls2.main || !ls2.main.joinPending) return resolve();
      if (Date.now() > deadline) return resolve();
      setTimeout(check, 20);
    }
    check();
  }).then(function() {
    // 5. Click (fire-and-forget – if it navigates, we won't get a
    // response, but Chrome still processes the click).
    if (els.length > 0) {
      var _el = els[0];
      #{click_fn()}
    }
    return _retValue;
  });
  ""

```

Four things happen, in order, before the click is dispatched:

1. A *one-shot patch promise* is installed. `window.__wallabidi_patch_promise` resolves on the next `onPatchEnd` callback. The `beforeunload` listener resolves it with the string `"navigated"` instead if a full navigation intervenes — this lets the host distinguish “patch arrived” from “page went away before it could”.
2. The element is classified. Result written to `_classification`. See previous section.
3. The view’s ref counter is snapshotted into `_preRef`. `liveSocket.main.ref` is the counter LiveView increments for each outgoing event. The server echoes the ref back, updating `lastAckRef`. So after the click, `lastAckRef >= _preRef` is a sufficient condition for *the server has finished processing whatever event our click triggered*, whether the reply was

a diff, a redirect, or silent state update. This is the mechanism the commit `ad8d0a2` fix hinges on. More on this in Section 4.1.

4. *The return value is captured before the click fires.* This matters because the click itself may tear down the JS execution context — if the element triggers a full page load, Chrome destroys the context and any attempt to read `_classification` afterward would fail with an RPC error. By packaging `_retValue` up first and then returning a promise that clicks *and then* resolves to it, we get the classification back even when the click navigates.

## Waiting for `joinPending`

Between step 4 (capture) and step 5 (click) there is a small gate:

```
var _ret = new Promise(function(resolve) {
  var ls = window.liveSocket;
  if (!ls || !ls.main || !ls.main.joinPending) return resolve();

  var deadline = Date.now() + 5000;
  function check() {
    var ls2 = window.liveSocket;
    if (!ls2 || !ls2.main || !ls2.main.joinPending) return resolve();
    if (Date.now() > deadline) return resolve();
    setTimeout(check, 20);
  }
  check();
});
```

This is commit `6e007c3`. The situation: under CI load, a test can request a click on the very first interactable element of a LiveView page, before the client-side join has completed. If `joinPending` is still `true`, Phoenix's delegated `[data-phx-link]` click handler has not yet been bound to the document. Our synthetic click, dispatched at that moment, falls through to the anchor's default behaviour — a full page reload — instead of being intercepted for a `push_navigate`.

Downstream, the classifier had already returned `"navigate"`, so the host awaits a LiveView-style transition that never happens, and the assertion races the real navigation.

The fix waits up to 5 seconds for `joinPending` to clear. Longer than that and we assume the LV is broken and click anyway, so a genuinely stuck join doesn't block forever.

*A subtlety:* the read-the-latest-value pattern `var ls2 = window.liveSocket` inside `check()` is deliberate. The execution context can be swapped out from under us (e.g. by a concurrent navigation), in which case `ls` from the outer scope points at a dead object. Re-reading `window.liveSocket` on each tick avoids that.

## Post-click: four strategies, one dispatcher

Once the click JS returns `{count, classification, prepared, preRef}`, the host dispatches to the strategy named by `classification`:

```
defp do_post_click(_session, "none", _prepared, _pre_page_id, _pre_ref), do: :ok

defp do_post_click(session, "patch", true, pre_page_id, pre_ref) do
  case Wallabidi.LiveViewAware.await_patch(session, 1_000) do
    :ok ->
      :ok

    :page_navigated ->
      Wallabidi.SessionProcess.await_page_ready_after(session, pre_page_id)

    :timeout ->
      # Patch didn't fire in 1s. Three possibilities:
      # (a) click triggered a nav – bootstrap will fire page_ready
      # (b) server handle_event is slow (DB, state transitions) and
      #     will push_navigate / push_patch / diff shortly
      # (c) no LV event at all
      # If we captured a pre_ref, wait for the LV server to ack that
      # event – however long handle_event takes. Closes the teamology
      # slow-handle_event race where wallabidi returned before server
      # finished processing, leaving downstream assertions racing the
      # redirect. Then let page_ready pick up any resulting navigation.
      if is_integer(pre_ref) do
        _ = Wallabidi.LiveViewAware.await_ack(session, pre_ref, max_wait_time())
      end

      Wallabidi.SessionProcess.await_page_ready_after(session, pre_page_id,
1_000)
    end
  end

defp do_post_click(_session, "patch", false, _pre_page_id, _pre_ref), do: :ok

defp do_post_click(session, "navigate", _prepared, pre_page_id, _pre_ref) do
  case Wallabidi.SessionProcess.await_page_ready_after(session, pre_page_id) do
    :ok ->
      :ok

    :timeout ->
      post =
        case Wallabidi.Protocol.current_url(session) do
          {:ok, url} -> url
          _ -> nil
        end

      raise Wallabidi.NavigationTimeoutError,
        %{from: nil, to: post, timeout_ms: 5_000}
  end
end

defp do_post_click(session, "full_page", _prepared, pre_page_id, _pre_ref) do
  case Wallabidi.SessionProcess.await_page_ready_after(session, pre_page_id) do
    :ok ->
```



```

      :ok

      :timeout ->
      post =
        case Wallabidi.Protocol.current_url(session) do
          {:ok, url} -> url
          _ -> nil
        end

        raise Wallabidi.NavigationTimeoutError,
          %{from: nil, to: post, timeout_ms: 5_000}
      end
    end

    defp do_post_click(_, _, _, _, _), do: :ok
  end
end

```

"none" — nothing to wait for, return immediately. The click was a plain DOM event with no LV handler, no anchor, no form. The caller is responsible for any further synchronization.

"full\_page" / "navigate" — a full page load is coming. We defer to the session process's push-based page-ready signal (`await_page_ready_after`), which is fed by the bootstrap script injected into every page. On timeout, we raise `NavigationTimeoutError` rather than returning silently. That's commit `9caa5c6`, a.k.a. "stop letting navigation timeouts masquerade as passing tests".

"patch" with `prepared: false` — the patch promise was never installed (no LiveSocket). There's nothing to await; fall through.

"patch" with `prepared: true` — three sub-cases, and this is where the hard work is.

## The patch sub-dispatcher

```

defp do_post_click(session, "patch", true, pre_page_id, pre_ref) do
  case Wallabidi.LiveViewAware.await_patch(session, 1_000) do
    :ok ->
      :ok

    :page_navigated ->
      Wallabidi.SessionProcess.await_page_ready_after(session, pre_page_id)

    :timeout ->
      # Patch didn't fire in 1s. Three possibilities:
      # (a) click triggered a nav – bootstrap will fire page_ready
      # (b) server handle_event is slow (DB, state transitions) and
      #     will push_navigate / push_patch / diff shortly
      # (c) no LV event at all
      # If we captured a pre_ref, wait for the LV server to ack that
      # event – however long handle_event takes. Closes the teamology
      # slow-handle_event race where wallabidi returned before server
      # finished processing, leaving downstream assertions racing the
      # redirect. Then let page_ready pick up any resulting navigation.
      if is_integer(pre_ref) do
        _ = Wallabidi.LiveViewAware.await_ack(session, pre_ref, max_wait_time())
      end

      Wallabidi.SessionProcess.await_page_ready_after(session, pre_page_id,
1_000)
  end
end

```

```

    end
  end

```

The first branch, `:ok`, is the happy path — the promise resolved because `onPatchEnd` fired, the diff landed, we're done.

The second, `:page_navigated`, is the beforeunload escape hatch from Section 4.1.1: the patch promise resolved with `"navigated"` because a full navigation took over mid-wait. We hand off to `await_page_ready_after` and let the bootstrap signal completion.

The third, `:timeout`, is the one that took three fixes to get right.

#### 4.1.1 The 1-second patch budget and why it used to leak

`await_patch` is called with a 1-second budget. That was a deliberate compromise — most diffs arrive in tens of milliseconds, and a longer default would slow every test. The assumption was: if 1s passes and no patch arrived, either the click didn't trigger a LiveView event at all (`handle_event` doesn't exist for this phx-click), or the page is on its way to a full navigation that the bootstrap's page-ready signal will handle.

That assumption broke under two conditions: the server's `handle_event` is slow (more than 1s), *and* the handler eventually issues `push_navigate` rather than a diff. On a fast dev machine, patches arrive in milliseconds and the branch is unreachable. On CI, a `handle_event` doing real work (database queries, cross-service calls) can easily exceed 1s before issuing its `push_navigate`, and during that window, wallabidi had already returned. Downstream assertions ran against a page that was *about to* navigate away.

The `9caa5c6` commit added a proper error for the tail case — raising `NavigationTimeoutError` when even the fallback `page_ready` budget expired — which at least made the failure visible. But it didn't close the race; it just stopped hiding it.

`ad8d0a2` closes it. When `await_patch` times out but we have a `pre_ref`, we call `await_ack` with the full `max_wait_time()` budget (typically 5s) *before* falling through to `page_ready`. `await_ack` is the heart of the fix:

```

js = """
new Promise(resolve => {
  var deadline = Date.now() + #{timeout};
  var target = #{pre_ref_next};

  function check() {
    var ls = window.liveSocket;
    if (!ls || !ls.main) return resolve('no-liveview');

    // `ref` here is the NEXT ref the view will issue. Before the
    // click we snapshotted it as `target`. The server acks with
    // the ref it received; lastAckRef is the highest so far. Our
    // click triggered an event with ref === target, so we wait for
    // lastAckRef >= target.
    if (ls.main.lastAckRef !== null && ls.main.lastAckRef >= target) {
      return resolve('acked');
    }

    if (Date.now() > deadline) return resolve(false);
    setTimeout(check, 20);
  }
}

```

```

window.addEventListener('beforeunload', function() {
  resolve('navigated');
}, {once: true});

check();
})
"""

case Protocol.eval_async(session, js, timeout + 1_000) do
  {:ok, "acked"} -> {:ok, :acked}
  {:ok, "no-liveview"} -> {:ok, :no_liveview}
  {:ok, "navigated"} -> {:ok, :page_navigated}
  {:ok, false} -> {:error, :timeout}
  _ -> {:error, :timeout}
end

```

The key insight: `lastAckRef` is updated by the LiveView client for every server reply, regardless of reply shape. A diff updates it. A `push_navigate` updates it. A `push_patch` updates it. Even a silent handle-event-with-no-reply updates it (the server still acks the event). So waiting for `lastAckRef >= target` is the most general “server finished processing our event” barrier available.

The `beforeunload` escape hatch is still needed: if the server’s reply was a redirect, the new page will be loading by the time our JS runs, and the execution context will die before `lastAckRef` can be observed. The `beforeunload` listener resolves with `"navigated"` so the host falls through to the page-ready path.

Only after `await_ack` returns do we ask `page_ready_after` to catch any resulting navigation. By this point the server is genuinely done; we’re just waiting for the new page’s DOM to come in.

## Sequence diagram

In chronological order, showing the full happy-path for a slow `handle_event` + `push_navigate`:

```

t0    Host    click_full JS compiled and evaluated
t0+   Browser prepare_patch installs __wallabidi_patch_promise
t0+   Browser classify → "patch"
t0+   Browser snapshot liveSocket.main.ref → preRef
t0+   Browser wait for !joinPending (already clear)
t0+   Browser dispatch click → phx-click event sent, ref=preRef
t0+   Browser return {classification:"patch", preRef, prepared:true}
t0+   Host    await_patch(1000ms) enters

      ... server's handle_event runs for 2.3s ...

t0+2.3s Server  reply: push_navigate
t0+1s   Host    await_patch → :timeout (1s budget expired at t0+1s)
t0+1s   Host    await_ack(preRef, 5000ms) enters
t0+2.3s Browser lastAckRef updated to preRef → resolve("acked")
t0+2.3s Host    page_ready_after(1000ms) – catches the push_navigate's
t0+2.5s Host    new-page notification from the bootstrap
t0+2.5s Host    click() returns. Downstream assertions are safe.

```

Note that `await_patch` and `await_ack` run in series, not in parallel. We only reach `await_ack` after the patch budget has expired. This keeps the fast path fast — a 10ms patch still returns in 10ms — and pays the ack-wait cost only when we actually need it.

## What the literate form adds

Reading this code as a literate document makes one thing obvious that the source, annotated or not, doesn't: *the three race conditions aren't independent bugs, they're one bug at different resolutions*. Each one is a place where wallabidi returned to the caller while the browser or server was still processing the click. The fixes all have the same shape — insert a more specific barrier, between steps that were previously assumed to be instant.

The sequence diagram is the real payoff. The code tells you what each function does; the diagram tells you why the functions are composed in this order and not a different one. That ordering is not recoverable from the source: `await_patch` then `await_ack` then `page_ready_after` looks arbitrary until you see it as a cascade of increasingly-general barriers.

Prose also lets us name things the code can't. `"lastAckRef"` is a fine identifier; “the most general server-finished-processing signal available” is a better description, but it wouldn't fit in a variable name, and a comment carrying it would have to be attached to one of the several places `lastAckRef` is read or written. The literate form puts the description once, near the argument that motivates it.

Generated from `lib/wallabidi/cdp/pipeline.ex`,  
`lib/wallabidi/browser.ex`, and  
`lib/wallabidi/live_view_aware.ex`.