Roc GUI: A manual for building native applications

The Roc GUI contributors
main

Contents

Choose a path 6
What you build 7
- Vocabulary 7
- What the platform provides 7

Getting started 9
- Use a published platform 9
- Work from a checkout instead 9
- Build and run the example 9
- Start an application 10
- Run the specifications 11
- Where to go next 11

Testing your application 12
- Two runners 12
  - Host flags 12
- Granting capabilities 13
- A first window specification 13
- What the window runner drives 15
- Where screenshots come from 15
- Iterating with an agent 16
- Reports are not captures 16

Building an application 17
- State, render, and actions 17
  - Pure application assembly 18
- Asynchronous actions 19
  - Superseding and cancelling tasks 19
  - Timers and subscriptions 20
- Elements and layout 20
  - Appearance 25
  - Rows produced on demand 27
  - Keyboard shortcuts and focus 28
  - Modifiers 29
- Translations and local state 30
  - Preserve a component across parent renders 30
  - Keys identify instances within a scope 31
  - Persistent application collections 31
  - Local updates and delegation 32

- Memoization is opt-in 32
- Tasks follow the accepting boundary 33
- Directory capabilities and tasks 33
- Single-file capabilities 34
- Dropped files 35
- Recent files and folders 35
- Watching granted files 36
- Encoded image data 37
- Shipped application assets 38
  - Validating an asset set at startup 39
  - What a store refuses 39
- Process and pseudo-terminal capabilities 40
- HID device capabilities 40
- System observation capabilities 40
- Durable application data 41
- Audio output capabilities 41
- Clipboard capabilities 42
- HTTP requests 42
- Read-only SQLite capabilities 43
- TCP endpoint capabilities 44
- Retained canvas 44
  - A canvas drawn for its size 45

Resource access 47
- What you start with 47
- How you get more 47
- A grant is a value 48
- Narrowing, and handing things on 48
- Taking it back 49
- Remembering what you were given 49
- Why it is built this way 50
- What this costs you 51
- What is deliberately absent 51

Specification language reference 52
- A behaviour specification 52
- Grants 53
- Locators 54
- Operations 56
- Assertions 59
- What the semantic runner proves 62
- Window specifications 63
  - One vocabulary, split by capability 63
  - Window-only steps 64
  - Screenshots 65
  - What input is real 66
  - What the probe cannot see 67
- Selecting and repeating cases 67

Measuring scaling 69
- Benchmark policy 69
- Read a scale ladder 70
- Verify depth with a process-stack budget 70
- Bound recorder detail overhead 70
- Workload families 71
- Compare two captures 73

Understanding captures 74
- Start with the summary 74
- Choose the question you are asking 74
- What semantic runs measure 77
- What a drawn frame measures 77
- Place work on one clock 78
- Count native rendering work 79
- Record an interactive application 81
- Read a capture while it is recorded 82
- Cycle targets 83
- Comparison identity and privacy 83

Troubleshooting 85
- roc build cannot find libhost.a 85
- The Windows build rejects the SDK's shader tools 85
- An application stops building after a Roc update 85
- The executable rejects an argument 85
- A click locator is missing or ambiguous 85
- A specification passes but performance columns are NULL 85
- A comparison is rejected 86
- The documentation build cannot render diagrams 86

How Roc GUI works 87
- The model in one page 87
- From state to pixels 89
  - The application renders a value 89
  - The platform lowers the description 89
  - The host validates and commits 91
  - The host materialises GPUI views 91
- One event turn 95
- Boundaries determine update scope 96
  - Memoization is an explicit behavioral contract 96
- Asynchronous action turns 97
  - Task lifetime, supersession, and cancellation 97
- There is no node-level diff 98
- Origin of the Action-State model 99
- How this compares 100
- Semantic tests use the production graph 101
- Threading and observability 101
- Correctness boundaries 102
- Responsibilities when extending the platform 102

Contributing 104
- Development environment 104
  - Two compilers 104
- Validate a change 105
- Complete a platform feature 106
- Fixture-backed steps 106
- Generated bindings 107
- Documentation 107
- Native linker inputs and releases 108
  - Continuous integration 108
  - Host source companions 109
  - The Windows import library 109
  - Linux producers with Nix 111
  - Windows inputs from Nix and native Windows 112
  - Generated system-library interfaces 112
  - Generated macOS interfaces 113

Release notes 114
- roc-gui 0.1.0-rc1 114
  - Use this release 114
  - Which Roc to use 114
  - If you built against a checkout 115
  - Targets 115
  - Known issues 115
  - Examples 115
  - Verify what you downloaded 115

Roc GUI: A manual for building native applications

Roc GUI is a small native GUI platform built around one predictable loop: state is rendered into elements, user events produce actions, and actions produce the next state. This manual takes an application from its first window through specifications that verify it, and on to repeatable scaling measurements.

Choose a path

- **New to Roc GUI**: Read What you build, then follow Getting started to build the counter and a minimal application.
- **Application author**: Continue through Building an application for the state/action/render model, elements, layout, and local state boundaries. Read Resource access before an application touches anything belonging to the person using it — a document, a folder, a network service, a device.
- **Specification author**: Read Testing your application. The executable you build runs both specification runners itself, so checking behaviour, layout, and appearance needs no test framework and no second build. Specification language reference has the complete step vocabulary.
- **Advanced application author**: Read How Roc GUI works for the rendering and event lifecycles, boundary replacement model, and comparison with other GUI architectures.
- **Performance investigator**: Start with Measuring scaling, then use Understanding captures to choose and interpret SQLite evidence views.
- **Platform contributor**: Read How Roc GUI works before Resource access and Contributing. These chapters cover host ownership, event routing, evidence ownership, validation, and feature completeness.

What you build

Roc GUI is an action/state platform for Roc, hosted on GPUI. You describe application data and how that data looks. The platform owns the window, turns the description into native views, and routes button presses back to Roc.

A Roc GUI application has three parts:

- **State**: The data that completely describes the application at a point in time.
- **Render**: A pure function from state to an element tree.
- **Action**: The result of handling an event: an update, no change, a proposal to the parent, or asynchronous work with a pending state and completion.



This model keeps application decisions in ordinary Roc values and functions. There is no mutable widget object to synchronise with a second copy of the application state.

Vocabulary

- **Application**: Your Roc code: its state, render functions, and event handlers.
- **Platform**: The Roc package selected by the application's `app` header. It defines the GUI API, lowers elements, and routes events.
- **Host**: The Rust executable support linked into the application. It owns the window, mounted graph, and GPUI views.
- **Element**: An immutable Roc value describing one piece of UI, such as text, a button, a row, or a column.
- **Element tree**: The nested elements returned by `render`; for example, a column whose children are text and a row of buttons.
- **Mounted graph**: The host-owned representation of the element tree the application displays.
- **Translation**: A way to embed a component that works with a smaller part of the application state, such as one counter on a page of counters.
- **Render boundary**: A part of the UI that can update without rebuilding its siblings. A translation creates one.
- **Specification**: A `.scm` file that drives user actions and assertions against an application. The steps it contains decide whether the semantic runner or the window runner takes it.
- **Capability**: Authority to reach one resource outside the application, such as a directory, an HTTP origin, or the clipboard.
- **Grant**: The opaque handle that carries a capability into application code. A grant cannot be forged, widened, or turned back into a path or a URL.
- **Observatory**: The opt-in recorder embedded in the host. It writes semantic and performance evidence to a SQLite `.rgstats` capture.

What the platform provides

- text;
- semantically named buttons and checkboxes;
- single-line and multiline text editors;

- images decoded from in-memory bytes;
- horizontal rows, vertical columns, panels, and modal dialogs;
- scrolling regions, virtual lists, and a retained canvas;
- components that can update independently;
- `.scm` specifications stored beside each application;
- opt-in SQLite captures for correctness and performance evidence.

The supported targets are Linux x86_64 with Wayland and Apple Silicon macOS.

Start with state, a render function, and actions. As the application grows, the application guide introduces local components, preserving their identity, and reusing unchanged UI. Those tools are available when you need them; they are not prerequisites for building your first screen.

Getting started

Roc GUI runs on Linux x86-64, macOS on Apple silicon, and Windows x86-64.

Use a published platform

An application names a released platform by its URL and needs nothing else. Build it with a current Roc nightly. The release page carries the URL and names the nightly the release was built and tested with:

```
app [State, main] { pf: platform "https://github.com/lukewilliamboswell/roc-gui/
releases/download/<version>/<bundle>.tar.zst" }
```

```
roc build --opt=dev --output=my-app main.roc
./my-app
```

Every release also publishes an archive of the examples with their headers already pointing at that platform, so they build without a checkout.

Work from a checkout instead

Contributors build the platform locally. Install a current Roc nightly as `roc` and the `blueprint` binary from the roc-blueprint 0.8.0 release. On Linux x86-64 or Apple silicon macOS, install Nix with flakes enabled for the native toolchain, then enter the development shell from the repository root:

```
blueprint shell
```

`Blueprint.lock` pins the Nix package set and the tooling compiler. Rustup selects the toolchain in `rust-toolchain.toml`. On Windows, install the Rust toolchain directly. No application names a compiler: each builds with the `roc` on the machine.

The host archive is part of every Roc GUI executable. Build it first:

```
python3 build.py
```

The result is `platform/targets/<target>/libhost.a`, where `<target>` is `x64glibc`, `arm64mac`, or `x64mingw`. Re-run `build.py` after a host change.

Build and run the example

```
roc build --opt=dev --output=counter examples/counter/main.roc
./counter
```

```
```

The window contains two independent counters. Their buttons show the normal application cycle: a press runs a Roc handler, the handler returns a new counter state, and only that counter's render boundary is rendered again.

Start an application

Create a directory with a `main.roc` file and point its platform path at this repository's `platform/main.roc`:

```
app [State, main] { pf: platform "../../platform/main.roc" }

import pf.Gui

State : { greeting : Str }

render : State -> Gui.Elem(State)
render = |curr_state| Gui.text(curr_state.greeting)

main : Gui.Program(State)
main = Gui.run({ init: |_access| { greeting: "Hello from Roc" }, render })
```

Choose an executable name explicitly when building your application:

```
roc build --opt=dev --output=my-app path/to/main.roc
./my-app
```

Pass `--host-help` to the executable to list native host options:

```
./my-app --host-help
```

Capabilities are absent by default. Grant each one explicitly. For example, the terminal workspace needs a local interactive shell profile:

```
roc build --opt=dev --output=terminal-workspace examples/terminal-workspace/
main.roc
./terminal-workspace --host-cap-process local-shell
```

The application receives only that fixed profile; it cannot select an executable, arguments, or inherited environment.

`--host-cap-*` options provision development and automation authority. They use the production registries and operations but are not trusted user consent. Interactive resource access uses the trusted broker workflows described in the resource access chapter.

Run the specifications

Every application keeps its specifications in one `specs/` directory beside `main.roc`. Run the counter's cases with:

```
python3 scripts/run_specs.py 'examples/counter/specs/*.scm'
```

The driver builds the host beside the locked linker inputs. It then builds the application, classifies each selected `.scm` file by the steps it contains, runs it in a fresh process, and prints the capture path for every passing case. Semantic specifications use the production event route and mounted graph without opening a GPUI window; window specifications open the real window. Pass `--only semantic` or `--only window` to run one kind.

Where to go next

Read Building an application to add interaction and split state into local components. Then read Testing your application and add a `.scm` file, so behaviour remains executable as the application changes. Specification language reference has the full step vocabulary.

Testing your application

Every Roc GUI executable can test itself. Both specification runners live in the host archive, so `roc build` links them into your application. There is no test binary, no test framework, no extra dependency, and nothing to add to your `app` header.

You select a runner with a command-line flag at startup, so an application launched without one behaves exactly as it does for anyone using it. Geometry recording is off unless a window specification asked for it, gated at runtime rather than at compile time, so the binary you ship is the binary the specification drives.

Build the application, then run both kinds of specifications against that same executable.

Two runners

A specification is an S-expression file describing what a user does and what should be true afterwards. The steps a file contains decide which runner takes it; nothing about its location or its name carries that meaning.

The **semantic runner** drives the mounted graph with no window. It is fast, deterministic, and answers questions about state and structure.

The **window runner** opens your real window, drives it through the same render path and event route an interactive session uses, and answers questions the semantic runner cannot: whether a control was actually laid out, whether it is on screen rather than merely mounted, how large it is, and what it looks like.

```
roc build --opt=dev my-app.roc

./my-app --host-run-spec specs/greeting.scm
./my-app --host-run-window-spec specs/greeting-window.scm \
    --host-window-report=.test-out/report.json \
    --host-window-shot-dir=.test-out
```

`./my-app --host-describe-specs specs/*.scm` prints one JSON line per specification giving the runner it needs and the capabilities it declares.

Host flags

Flag	Meaning
`--host-run-spec PATH`	Run one specification on the semantic runner.
`--host-run-window-spec PATH`	Run one specification against the real window.
`--host-window-report=PATH`	Where the window run writes its JSON report.
`--host-window-shot-dir=PATH`	Where screenshots are written.
`--host-window-timeout-ms=N`	Per-step deadline for the window runner, 1000 to 600000, default 15000. This bounds a step that

Flag	Meaning
	never settles; `(settle)` carries its own smaller default.
`--host-window-allow-missing-shots`	Report a screenshot that could not be taken as `unavailable` instead of failing the run. Use it where capture is impossible, such as a continuous integration runner that cannot grant screen recording.
`--host-describe-specs PATH...`	Print each specification's runner and declared grants as JSON.

`--host-help` lists these flags along with the capability and measurement flags.

Granting capabilities

An application that reads files, reaches the network, or drives a terminal does nothing useful without the authority to do it, and a specification run is no exception. The same `--host-cap-*` flags that grant a capability to an interactive session grant it to a specification run. Running without a grant is the supported way to exercise a denial path, because the application takes the same refusal it would take from a real broker.

```
./my-app --host-run-spec specs/saving.scm --host-cap-dir ./documents
```

`--host-help` lists every capability flag. The flags confer the authority, which is why what a run may touch is visible in the command that started it.

A specification also states what it needs, in a `(grants ...)` clause beside its steps. A runner reads that clause and passes the matching flags, so a case carries its own requirements instead of depending on how it was launched. The specification language reference lists the grant vocabulary.

```
(test "saving writes the note"
  (grants
    (directory "fixture"))
  (steps ...))
```

A first window specification

Start from the greeting application in the getting-started chapter and give it one button:

```
greet = |_, _| Gui.Action.update({ message: "Hello from Roc" })
```

```
Gui.col({}, [
    Gui.text(state.message),
    Gui.button({ caption: "Greet", label: "Greet", on_press: greet }),
])
```

Write `specs/greeting.scm` beside it:

```
(test "Greeting appears when the button is pressed"
  (steps
    (settle)
    (expect-on-screen (role button :name "Greet"))
    (screenshot "before")
    (click (role button :name "Greet"))
    (expect-visible (text "Hello from Roc"))
    (screenshot "after" :region (text "Hello from Roc") :pad 16)))
```

Run it:

```
roc build --opt=dev my-app.roc
./my-app --host-run-window-spec specs/greeting.scm \
    --host-window-report=.test-out/report.json \
    --host-window-shot-dir=.test-out
```

The report names every step and points at every image:

```
{
  "schema_version": 2,
  "spec": { "name": "Greeting appears when the button is pressed" },
  "run_mode": "window-scenario",
  "outcome": "pass",
  "window": { "width_points": 480, "height_points": 240,
              "requested_width_points": 480, "requested_height_points": 240,
              "scale_factor": 2.0 },
  "steps": [
    { "ordinal": 0, "line": 3, "kind": "settle", "status": "pass" },
    { "ordinal": 2, "line": 5, "kind": "screenshot", "status": "pass",
      "name": "before", "file": "02-before.png", "bytes": 22064 }
  ],
  "screenshots": [
    { "name": "before", "file": "02-before.png", "bytes": 22064 }
  ]
```

```
}
```

Images are named `NN-NAME.png`, numbered by step. Paths in the report are relative to the report itself, so it contains no home directory.

A failing step stops the run, and says what was expected in the words the specification used:

```
{ "ordinal": 3, "line": 6, "kind": "expect-visible", "status": "fail",
  "message": "line 6: (text \"Hello from Roc\") matched 0 nodes; expected exactly
one" }
```

The exit status is 0 when every step passed and 1 when one did not, so a specification is usable directly from a shell script or a build rule.

What the window runner drives

Keyboard input is real. `focus` moves focus through the same handle the host uses, `press-key`, `key`, and `type` reach the window's production keymap, and `expect-focused` asks the focus handle rather than a model of it.

Pointer input is simulated. `click` invokes the same handler the production click closure invokes, because GPUI exposes no usable pointer entry point. On a text field it takes focus instead, which is what a pointer press does there. What is **not** simulated is whether the click could have landed: the control must have been laid out, must survive clipping by the window and by every scrolling region around it, must be enabled, and must not be behind a modal dialog. A `click` that a real user could not have performed fails rather than passing quietly. There is no `hover` step, because hover styling is driven by GPUI's own pointer position and a step that could not honestly move it would not mean what its name says.

The specification reference has the full step vocabulary and describes what geometry recording cannot see.

Where screenshots come from

On Linux the host photographs its own frame: the renderer reads back the frame it presents, under any compositor, and nothing drawn over the window appears in the picture. Windows asks the window to render itself into a bitmap.

On macOS screenshots are taken by `screencapture`, which needs Screen Recording permission for whatever launched your application; without it the run reports `unavailable` and names the permission. A window that is not on the primary display also reports `unavailable` rather than guessing. That screenshot is a photograph of a region of the screen, not of the window's own buffer, so anything drawn over your window is in the picture: the dock, a notification, another application's window. Keep the window

unobstructed while a specification runs, and treat a shot containing something you did not expect as evidence about the screen rather than about your application.

An unavailable screenshot fails the run unless `--host-window-allow-missing-shots` is given. Every other assertion still runs either way, so a machine that cannot photograph anything can still verify behaviour.

Iterating with an agent

The report is meant to be read by a program as well as a person, which makes a window specification a usable feedback loop for an AI agent changing your interface. The agent runs the specification, reads the report and the images, changes the application, and runs it again.

Three details serve that loop. A failing step names the locator as it was written, so the message points at the specification rather than at an internal identifier. Screenshots taken before a failing step are kept, so the image of the state an assertion stopped at survives. The `screenshots` array is a flat index, so the agent finds images without walking the steps.

The loop verifies the assertions you wrote and hands back images. It does not judge whether the result looks good.

Reports are not captures

A window run writes a JSON report and images into a working directory. A measurement run writes a `.rgstats` SQLite capture. They are different kinds of evidence and neither replaces the other.

A capture is durable and archived, and records that a step ran and how it went. It never contains application text, paths, or pixels. A report is working evidence for whoever is iterating, and carries exactly what a capture must not: the failing locator as written, the message, and the screenshots. Keep reports and images out of version control; they describe one run on one machine.

Building an application

Start with one state type, a render function, and event handlers that return actions. Ordinary functions are enough to organise a small application. This chapter introduces local components when part of the UI needs to update independently, then keys and memoization when their additional guarantees help.

State, render, and actions

An application exports a state type and `main : Gui.Program(State)`. The whole platform arrives through one import:

```
import pf.Gui

State : { count : I64 }

render : State -> Gui.Elem(State)
render = |curr_state| Gui.col({}, [
    Gui.text("Count: ${curr_state.count.to_str()}"),
    Gui.button({
        caption: "Add one",
        label: "Increment count",
        on_press: |prev_state, _event|
            Gui.Action.update({ count: prev_state.count + 1.I64 }),
    }),
])

main : Gui.Program(State)
main = Gui.run({ init: |_access| { count: 0 }, render })
```

`init` takes the application's authority — a `Gui.Access` — and returns the first state. It is the only place authority enters an application, and there is no way to produce one, so an application holds exactly what it was handed and a library holds exactly what it was passed. An application that acquires nothing ignores it, as this one does. One that acquires later keeps it in its state, because the task that acquires runs after `init` has returned. The Resource access chapter explains why the platform is built this way.

Every acquisition is a question asked of that value, from inside a task: `access.pick_directory!()`, `access.pick_file!(types)`, `access.app_data!()`, `access.assets!(config)`, `access.audio!()`, `access.clipboard!()`, `access.device!()`, `access.http!()`, `access.process!()`, `access.system_monitor!()`, and `access.tcp_connect!()`. Each answers with a handle only for what the host has granted, and the handle carries its own operations. Code that was never handed the authority has nothing to ask.

`render` is pure: it reads state and returns a `Gui.Elem(State)`. A button handler receives the state associated with its render boundary and an `Gui.Event.Press`. It returns an action:

- `Gui.Action.update(next_state)`: Propose the next state and update the required render scope, subject to the component's memoization policy.

- `Gui.Action.none()`: Keep the existing state and element tree unchanged.

The names in the example distinguish two moments. `curr_state` describes the UI being rendered. The handler's `prev_state` is the state immediately before its action is applied.

Roc values are immutable. A handler that captures `curr_state` keeps that snapshot alive in its closure. Later updates to a shared list or string may therefore require cloning storage to preserve the captured value. Derive actions from `prev_state` to avoid unnecessarily capturing the renderer's snapshot. This does not eliminate cloning: other retained values, including memoization snapshots, can still require it. Renaming a captured value does not change its ownership or lifetime.

Using the handler argument also matters when handlers are retained across updates: it supplies the latest state, whereas a captured render snapshot still describes the earlier render.

Use `Gui.Action.none()` when an event is valid but has no effect, such as pressing a decrement button at a lower bound. It is both clearer and less work than returning an unchanged state with `Gui.Action.update`.

An action is a value describing a transition, not a transition performed. Constructing one changes nothing; returning it from a handler is what asks the platform to apply it. Every transition is applied on one thread, one at a time, so a handler's first argument is always the state that precedes it. This model is Richard Feldman's, from his design document *Design Idea: Action-State*; the architecture chapter records that origin and the reasoning behind it.

Pure application assembly

`Gui.run({ init, render, window })` describes an application with ordinary initial state and a pure renderer. `window` is optional. Resource acquisition and window-dependent interactions belong in actions.

`on_open: Some(opened!)` is what the application does as its window opens. The platform calls `opened!(state)` once, after the first state is shown and before any input, exactly as it calls an event handler: it may act, and the action it returns is applied. It is how a first screen shows something only an effect can learn, such as the files the application remembered, without waiting for a person to ask. Its turn is one cycle with the trigger `open`, which no element caused. An application without `on_open` has no such turn.

```
main = Gui.run({
    init: Observatory.init,
    render: View.render,
    on_open: Some(|state| Gui.Action.update({ ..state, recent: state.access.recent!
() })),
})
```

Reusable modules export ordinary render functions such as `Workspace.render : Workspace.State -> Gui.Elem(Workspace.State)`. A parent composes them through

`Gui.translate` or `Gui.translate_with`; no setup-time registration is needed. The host assigns each mounted boundary its lifetime. The module does not call `Gui.run` or choose the application's window.

Asynchronous actions

Use `Gui.Action.task({ pending, run, resolve })` when an event needs to wait for external work. `pending` describes the UI while that work runs, such as a loading indicator. After accepting that state, the platform runs `run` on its bounded worker pool and passes the result to `resolve` on the UI thread.

The resolver receives `prev_state`, the state immediately before completion, so it can preserve edits made while the worker was running. Capture only the request data needed by the worker; derive the completion action from the resolver's argument. Resource operations require the grants described in the resource access chapter.

Tasks belong to the component that accepts them. Removing that component cancels its tasks. The local-state section explains how a parent can own work that must survive a child closing.

Superseding and cancelling tasks

`Gui.Action.keyed_task({ key, pending, run, resolve })` is a task that supersedes the component's previous task with the same `key`. A person moving a selection, paging a list, or running a statement again wants the newest answer only; the older task is cancelled, and its `resolve` is never called. `Gui.Action.cancel(state, key)` installs `state` and cancels the component's task with `key`, for an explicit Cancel control or for closing what a task watches.

A cancelled task that has not started never runs. One that is running is interrupted where the capability allows: a `Gui.Timer` or `Gui.Files.Watch` wait returns `Canceled`, and a SQLite statement stops with `Interrupted`. The timer or watch itself keeps running, so a later task may wait on it again. Other work, such as an HTTP request, runs to its end, and its result is dropped.

```
run_query = |state, database, sql| Gui.Action.keyed_task({
    key: "query",
    pending: { ..state, status: Querying },
    run: || database.query!(sql),
    resolve: |latest, outcome| Gui.Action.update(show(latest, outcome)),
})

cancel_query = |state| Gui.Action.cancel({ ..state, status: Canceled }, "query")
```

Keys are scoped to the component that accepts the task, so two instances of one component never supersede each other. Cancelling a key with nothing in flight only installs the state.

Timers and subscriptions

`Gui.Timer.start!({ interval_ms })` creates an opaque `Gui.Timer.Handle` for intervals from 1 through 60,000 milliseconds. `handle.next!()` waits for one tick and belongs inside `Gui.Action.task`; the next wait is started only after the previous completion has been applied. This pull contract bounds backpressure at one in-flight wait and does not accumulate missed ticks.

`handle.cancel!()` is idempotent, wakes an outstanding wait, and makes it resolve as `Canceled`. Dropping the last typed `Box(U64)` handle also releases the host registry entry. Applications retain the handle in state and decide whether a `Fired` result schedules another wait.

Elements and layout

- `Gui.text(value)`: Display a string, inheriting colour and size from its container.
- `Gui.button({ ... })`: Display a controlled, styled action button. `caption` is visible text, `label` is its stable semantic name, as in every other control, and `enabled` defaults to `True`. Enabled buttons participate in keyboard tab order and activate with Enter or Space. Disabled buttons remain visible with a subdued appearance and accept neither pointer nor keyboard activation. Props include padding, sizing, growth, foreground and interaction colours, border, radius, font size, font weight, and overflow. A call that gives only `caption`, `label`, and `on_press` is the concise text button. A button accepts `on_hover_enter` and `on_hover_exit` as optional `Some(|prev_state, event| Gui.Action.update(...))` callbacks. Their defaults are `None`: unconfigured buttons allocate no hover routes. `Gui.Event.Hover` carries no fields. Hover callbacks receive current state and return ordinary actions, including cell-owned tasks. Native hover transitions follow GPUI pointer movement with no mouse button held; repeated movement inside a button emits no additional enter. Disabled buttons emit neither hover callback. Removal does not synthesize exit. Any other element takes the same two modifiers, `on_hover_enter` and `on_hover_exit`: the element is wrapped in a hover region that reports the pointer entering and leaving it and presents nothing. The region takes its element's share of the parent's space, so wrapping changes no layout.

Keep button labels and keyed projections stable. Rebuilding a hovered cell to change its color preserves its native identity and inside state; it does not produce another enter. `hover_bg` remains a native visual refinement independent of these callbacks. Set it to `Default` when application actions own the hover color, as in `benchmarks/hover-grid`.

```
Gui.button({
    caption: "Preview",
    label: "Preview item",
    on_press: |_, _| Gui.Action.none(),
    on_hover_enter: Some(|prev_state, _| Gui.Action.update({ ..prev_state,
preview: True })),
    on_hover_exit: Some(|prev_state, _| Gui.Action.update({ ..prev_state, preview:
False })),
```

```
})
```

- `Gui.checkbox({ ... })`: Display a semantic checkbox and handle `Gui.Event.Check`. `box_bg`, `box_checked_bg`, `box_border`, and `mark_color` colour the indicator itself, which `fg` does not reach; each defaults to the host's own value. Props include enabled state, gap, padding, sizing, growth, foreground and interaction colours, border, radius, font size, font weight, and overflow. Enabled checkboxes participate in keyboard tab order and toggle with Space; disabled checkboxes do not accept focus or activation.
- `Gui.textarea({ ... })`: Display a controlled multiline text editor and handle `Gui.Event.Input`, which carries the complete requested value. `label` is the stable semantic name; `placeholder`, `enabled`, and `read_only` control editing behaviour. A read-only textarea retains multiline layout and scrolling for response and log views. Textareas share the native style fields and default to a 160-logical-pixel editing viewport.
- `Gui.image({ label, bytes, format, ... })`: Decode and render capability-safe in-memory image bytes through GPUI. The explicit format is one of `Png`, `Jpeg`, `Webp`, `Gif`, `Svg`, `Bmp`, or `Tiff`; fit defaults to `Contain` and also supports `Cover`, `Fill`, `ScaleDown`, and `None`. `grayscale` applies a native rendering effect. The declared box is authoritative: `width`, `height`, and the minimum and maximum fields size the element, and `fit` is the rule that maps the decoded pixels into that box. Image sources are values, never ambient filesystem paths or URLs, and `label` is the stable semantic name.
- `Gui.text_input({ ... })`: Display a controlled, single-line native editor. `label` is its stable semantic name, `value` is the authoritative application value, and `placeholder` is shown only while empty. Committed edits deliver the complete value in `Gui.Event.TextChange`; Enter delivers `Gui.Event.TextSubmit`. The host retains focus, caret, selection, and an active IME composition across controlled rerenders. Disabled fields accept neither edits nor submission. Text values are bounded to one MiB of UTF-8. Text-input labels are non-empty and unique within their component instance scope because they also preserve native editor identity across controlled rerenders. Distinct keyed instances can reuse the same local input label.
- `Gui.styled_text({ value, fg, font_size, font_weight, font_face })`: Display a string in its own colour, size, weight, and face. Plain `Gui.text` inherits all four from its container, so every typographic step otherwise costs a one-child row or column that exists for no other reason; a wordmark, an eyebrow label, and a status line are each a string, not a layout.
- `Gui.rich_text({ spans, fg, font_size, font_weight, font_face })`: Display several styled runs as one text element, such as a line of highlighted source. Each run is built with `Gui.span({ text, fg, bg, font_weight, underline, monospace })`; only `text` is required, and a run that sets nothing keeps the element's own colour, weight, and face, which it inherits from its container unless the element sets them. The runs lay out, wrap, and clip as one string under their container's `text_overflow`, and a locator matches the whole text, the runs' texts in order. A rich text element is one native text node however many runs it has, and its native work is counted as styled text.

- `Gui.scroll({ label, content, axis, ... })`: Constrain content to available space and scroll it vertically, horizontally, or in both directions. The axis defaults to `Vertical`; `label` is the stable semantic identity. A scroll region shares the native `Gui.Style` fields, so it carries its own size, ground, inset, and corner rather than needing a column or panel wrapped around it to supply them.
- `Gui.virtual_list({ label, row_height, items })`: Present stable-keyed rows at one fixed logical height. Each item is an `{ key, content }` record; keys must be unique within the list. The complete item graph remains available to semantic inspection and event routing, while GPUI materialises only the current viewport range. A virtual list shares the native `Gui.Style` fields as well, and `row_gap` holds that many logical pixels clear at the bottom of each row inside `row_height`, so rows read as separate surfaces while the list's scrolling arithmetic stays exactly one `row_height` per row. Without them a list has no ground, inset, corner, or spacing of its own, and each row has to repaint the surface the list is sitting on.
- `Gui.virtual_rows({ label, row_height, count, render_row })`: Present `count` rows at one fixed logical height, building each with `render_row` only while it is near the viewport, so the cost of a render follows the rows on screen rather than `count`. It brings a row into view from state with `scroll_to` and reports the rows on screen through `on_range`. It shares the style fields and `row_gap` of `Gui.virtual_list`. See Rows produced on demand.
- `Gui.row({ ... }, children)`: Lay out children horizontally.
- `Gui.col({ ... }, children)`: Lay out children vertically.
- `Gui.keyed_col(render_item, { ... }, { key, get, set, on_delegate })`: Lay out a `Gui.KeyedSeq` vertically while preserving each item's component lifetime by its complete `Gui.Key`. Inserts, moves, removals, and replacements are applied as one mounted-graph transaction and one native keyed transaction. Construct a sequence with `Gui.KeyedSeq.empty()` and update it with `insert_before`, `remove`, `move_before`, `set`, or one atomic `apply_all`; `Before(anchor)` and `End` describe placement. Duplicate inserted keys, missing targets, and missing anchors reject the operation without publishing a partial revision. `on_delegate : parent, Gui.Key -> Gui.Action(parent)` receives the complete key of the delegating item. Item actions project through their key, so a move does not retarget an action and a removed item cannot be recreated by a late completion. If the platform has missed the sequence's one-generation journal, it reconciles the full ordered snapshot while retaining every surviving key. An exact journal renders only inserted and set items; a move preserves the existing item component and a removal retires it, its routes, and its pending task delivery. The full snapshot is materialised only for stale recovery.
- `Gui.panel({ ... }, children)`: Group children vertically in a labelled surface that is padded, bordered, and rounded by default. `heading` is the panel's own visible heading, painted above its children in `heading_size`, `heading_weight`, and `heading_color`; each zero or `Default` keeps the panel's own type. It is distinct from `label`, which remains the semantic locator and is never painted, so a panel named for what a specification calls it can still say something else to a reader. An empty heading, the default, paints none.
- `Gui.popover({ label, ... }, anchor, content)`: Annotate `anchor`, any element, with a non-modal surface presenting `content` beside it. The surface opens `delay_ms` (default 500, at most 60000) after the pointer comes to rest on the anchor, and at once when

keyboard focus enters the anchor. It closes when both the pointer and focus have left, and Escape that no dialog takes closes every presenting popover until the pointer or focus enters it again. `placement` is `Below` (the default), `Above`, `Start`, or `End`; a surface the window has no room for moves to the opposite side and stays inside the window. The surface floats above the rest of the window, outside any scroll region's clipping, and blocks nothing: its content is read, not operated. `label` names the surface as a `tooltip` for specifications and assistive technology; a closed surface is not presented, so nothing in it can be found. The style fields dress the surface, and a modifier applied to a popover applies to its anchor. The surface's text wraps within the surface, whatever truncation its anchor's container applies, unless the surface's own `text_overflow` says otherwise. `on_hover_enter` and `on_hover_exit` are optional handlers for the anchor's pointer edges. The host owns when a surface presents, so opening and closing one runs no application code.

- `Gui.tooltip(anchor, text)`: Annotate an element with a popover presenting one line of text, named by that text.

```
Gui.popover(
    { label: "title in result row 0", delay_ms: 400 },
    cell,
    [Gui.text("title · TEXT"), Gui.text(value)],
)
```

- `Gui.split({ label, size, on_resize, ... }, start, end)`: Lay two panes out side by side (`axis: Horizontal`, the default) or one above the other (`Vertical`), with a divider between them. `size` is the extent in logical pixels of the pane on `side`, `Start` (the default) or `End`; the other pane takes whatever space is left, however wide its content would like to be. The size is the application's: dragging the divider, or the keys below while the divider holds keyboard focus, ask for one through `on_resize` with a `Gui.Event.Resize`, `{ size, collapsed }`, and the split shows whatever `size` it is given next, kept within `min` (default 0) and `max` (default 16384). A drag is measured from where the pointer pressed, and asks only for a size the divider does not already show, so each size it asks for is one `drag` cycle. The divider takes part in keyboard tab order; Left and Right (Up and Down for a vertical split) move it by `step` (default 16) pixels, and Home and End move it to the start or end of its range. These are the divider's own shortcuts: they answer only while the divider holds focus, never while focus is in a pane, and each is a `key` cycle. A `collapsible` pane is folded away while `collapsed`: Enter on the divider asks to toggle it, and a drag that would leave the pane less than half of `min` asks to fold it, keeping `size` to return to. A folded pane stays mounted, with its component state and tasks, but it is neither drawn nor presented, so no locator finds anything in it. `label` names the divider as a `separator` for specifications and assistive technology, and its value is the size, or `collapsed`. `thickness` (default 6), `color`, `hover_color`, `active_color`, and `focus_color` dress the divider; `width`, `height`, and `grow` size the split, which fills its container by default.
- `Gui.tabs({ label, tabs, selected, on_select, on_close, ... })`: A strip of tabs, one of them selected. Each `Gui.TabItem` is `{ key, title, closable }`: the tab shows `title`,

which also names it as a `tab` for specifications, and a closable tab has a close button named `Close` and its title. Pressing a tab asks for it through `on_select`, and its close button through `on_close`, each with a `Gui.Event.Tab` carrying the tab's `key`; the application owns the list and the selection, and the strip shows the tab whose key is `selected` as chosen. While keyboard focus is in the strip, Left and Right choose the tab before and after the selected one, Home and End the first and last, and Delete asks to close the selected tab when it is closable. The strip's colours, `font_size`, `font_face`, and `max_tab_width` dress it.

- `Gui.drop_target({ label, types, on_drop, ... }, children)`: Lay children out as `Gui.col` does, in a column that accepts files a person drags from outside the application and drops on it. `types` are the kinds of file it accepts, written as `access.pick_file!` offers them; an empty list accepts every file. A drop is delivered to `on_drop` with a `Gui.Event.Drop`, `{ files, refused }`: `files` holds each dropped file of an accepted type, in the order dropped, each a `{ name, file }` granted to the application exactly as a chosen file is, and `refused` names every other item dropped with its reason, `NotFile` for a folder or anything that is not an ordinary file, `Unsupported` for a file of another type, `AccessDenied`, or `Unavailable`. While files it would accept are dragged over it the target shows `drop_bg` and `drop_border`; files it would refuse change nothing, so it never invites a drop it will not take. A drop behind a modal dialog is not delivered. `label` names it as a `drop-target` for specifications, and each drop is one `drop` cycle, whatever it held. The section on dropped files says what the grant is.

```
Gui.split(
    {
        label: "Inspector divider",
        side: End,
        size: state.inspector.size,
        min: 240,
        max: 1100,
        collapsible: True,
        collapsed: state.inspector.collapsed,
        on_resize: |current, event| Gui.Action.update({ ..current, inspector:
{ ..current.inspector, size: event.size, collapsed: event.collapsed } }),
    },
    main_view,
    inspector,
)
```

- `Gui.dialog({ label, on_dismiss, ... }, children)`: Present one modal surface above an input-blocking scrim. The first enabled control gains focus, Tab and Shift+Tab stay within the dialog, Escape calls `on_dismiss`, and closing the dialog restores its opener by stable semantic identity. Controls behind the scrim stop answering input but keep their own enabled or disabled appearance; the dialog does not restyle them as disabled. Render at most one dialog at a time.

Row, column, panel, dialog, popover, drop-target, scroll, virtual-list, and button props share the native `Gui.Style` fields: gap, padding, width, height, growth, colours, borders, radius, shadow, font size, font weight, per-axis overflow, and the two axis placements. `font_weight` is zero for the native default, or 100 through 900; 700 is bold.

A colour is written as a `0xRRGGBB` literal wherever a `Gui.Color` is expected, as in `bg: 0x24404b`, and a value above `0xffffff` is rejected when the application is compiled. A colour held in a name is annotated, `ink : Gui.Color` then `ink = 0x1f1c17`, and one computed at run time is `Rgb(value)`. `Default` keeps the control's native colour.

`Gui.adaptive(light, dark)` is a colour for each appearance, both `0xRRGGBB`. The host keeps the pair and chooses one each time it paints: `light` while the window's scheme is light and `dark` while it is dark. An application that writes its palette as adaptive pairs follows the system's scheme with no code of its own, and a change of scheme repaints the window without rendering the application again. The window's scheme is the system's unless the application has chosen one with `Gui.Appearance.prefer!`.

Appearance

`Gui.Appearance.current!()` answers the system's appearance as `{ scheme : [Light, Dark], motion : [Full, Reduced] }`: whether the desktop asks for a light or a dark scheme, and whether it asks for non-essential motion to be reduced. A desktop that states no preference is `Light` and `Full`. On Linux the host reads the XDG desktop portal's `org.freedesktop.appearance` settings, and GNOME's `enable-animations` for reduced motion where the portal does not publish its own key.

`Gui.Appearance.next_change!(known)` waits until the system's appearance differs from `known` and answers the new one. It belongs inside a keyed `Gui.Action.task`, so a newer wait supersedes it; cancelling the task interrupts the wait, which then answers `known`. An application that decides something by the appearance, such as whether to animate, starts the next wait from the completion of the last.

`Gui.Appearance.prefer!(preference)` chooses the scheme adaptive colours resolve to: `System`, `Light`, or `Dark`. It is a task's work, not a render's, and it lasts until the next `prefer!`; the system's appearance, as `current!` answers it, does not change. An application that keeps a person's choice across launches stores it and chooses it again at startup.

Reading the appearance takes no authority. A specification provisions the system's appearance with a `(theme ...)` grant and changes it with `system-theme`, so a case never depends on the desktop it runs on; see Specifications.

`padding` is the shorthand for all four sides. `padding_top`, `padding_right`, `padding_bottom`, and `padding_left` each override one side with `Px(n)` and default to `Same`, which takes the shorthand. A pill-shaped control wants generous horizontal and tight vertical padding, which is the shorthand plus two sides rather than a fixed height chosen to cancel the padding out.

`font_face` is `Default` for the host's proportional face or `Monospace` for the platform's fixed-pitch one, which is what columnar output and changing digits need. The host names the face each operating system ships rather than a generic word only one font stack resolves.

`min_width`, `min_height`, `max_width`, and `max_height` are floors and ceilings for the two sides, given as `Px(n)`. A fixed length is a basis a sibling's overflow can shrink, so a row of uniform thumbnails beside captions of different lengths needs a floor to keep one shape.

`text_overflow` decides what a string does when it is wider than the space it was given. `Wrap` reflows onto further lines and grows the container, which is the default. `NoWrap` keeps one line and leaves the rest to `overflow_x`. `Ellipsis` keeps one line and ends it with a marker, so a shortened value never looks like a complete one.

`disabled_bg`, `disabled_fg`, and `focus_color` replace the host's own control-state colours. The host defaults suit its dark ground: a blue-grey disabled fill at 55% opacity, and an amber focus ring. An application that names its own disabled colours gets those at full opacity, because a chosen colour is already the colour it wants to be.

`border_width` is likewise the shorthand, with `border_top`, `border_right`, `border_bottom`, and `border_left` overriding one side each. A dense panel divides its regions with a rule on the one edge that faces the next region, rather than giving every region a complete box whose outer edges are also drawn.

`shadow` is the blur radius of a soft drop shadow under the element, offset down by `shadow_y` and painted in `shadow_color` at `shadow_alpha` percent of that colour. `shadow: 0`, the default, paints none. A raised card on a near-white ground separates from its paper by shadow where a hairline border has too little contrast to read, and the colour is the application's own rather than a host grey, so the shadow can be cast in the page's ink.

`justify` distributes children along the container's own layout axis and `align` places them across it. `justify` is `Start`, `Center`, `End`, `Between`, or `Around`; `align` is `Start`, `Center`, `End`, `Baseline`, or `Stretch`. Both default to `Default`, which keeps the element's native placement. A grown child with `justify: End` puts a trailing status field at the far edge without a spacer element, and a grown empty-state column with `align: Center` and `justify: Center` sits in the middle of the area it was given. `Auto`, `Fill`, and `Px(n)` express dimensions. An optional non-empty `label` provides a stable semantic locator, such as `(role panel :name "Directory view")`; it does not add visible text.

Rows and columns accept a `List(Gui.Elem(State))`, which makes ordinary list transformations useful for repeated UI:

```
render_names : List(Str) -> Gui.Elem(State)
render_names = |names| Gui.col(
    { width: Fill, gap: 6 },
    names.map(|name| Gui.text(name)),
)
```

The empty `{}` retains neutral defaults. Semantic element names are not visible labels. Give each simultaneously visible button a unique, stable name such as `"Delete row ${row.id.to_str()}"`; specifications use that identity instead of screen position.

Keyboard focus has a visible outline. Tab and Shift+Tab traverse enabled buttons, checkboxes, and text inputs in rendered child order, so application layout order is also the

default keyboard order. Keep labels concise and semantic names stable across state updates.

`Gui.run` also accepts a `window` record with an initial logical `width`, `height`, and titlebar `title`. Defaults allow an `init`/`render`-only configuration; windows remain resizable and layout elements using `Fill` follow the available viewport.

`background` and `foreground` set the colour behind the root element and the ink text inherits when it names none. The background reaches the whole window, including its rounded corners and the edge above the root element, which a grown root element cannot cover. Both default to the host's own ground, so an application that says nothing keeps it.

Rows produced on demand

`Gui.virtual_list` receives every row as an item. GPUI draws only the rows in view, but every row is still built, lowered, and mounted, so its cost follows the list's length. `Gui.virtual_rows` receives a row count and a row function instead:

```
Gui.virtual_rows({
    label: "Query rows",
    row_height: 24,
    count: result.rows.len(),
    render_row: |index| result_row(result, index),
    scroll_to: state.rows_scroll,
    on_range: Some(|prev_state, visible| Gui.Action.update({ ..prev_state,
on_screen: Some(visible) })),
})
```

The host owns the viewport, so the host names the rows to build: those the viewport shows and one viewport's worth either side of them. Only those rows are rendered, lowered, and mounted. When scrolling brings the viewport within half a viewport of the edge of the mounted rows, the platform renders the list again for the new range. The list is its own render boundary: that render calls the `render_row` of the owner's most recent render and lowers the list alone, without rendering its owner. A row the viewport reaches before the list has been rendered for it holds its place, blank, for the frames in between.

The boundary is transparent. An action raised by a row, or by `on_range`, belongs to the list's owner exactly as if the rows had been built there, and a task it starts is owned by that owner too. A row may contain translations, buttons, editors, and nested lists like any other element.

- `row_key : U64 -> U64`: The stable identity of the row at an index; it defaults to the index. Keys are unique among the rows mounted together. A key drawn from the data keeps a row's native state — a pending press, focus, an editor — with its data when rows are inserted above it.
- `scroll_to : [None, Some(Gui.ScrollRequest)]`: Bring a row into view from state. A `Gui.ScrollRequest` is `{ row, align, serial }`, and `align` is a `Gui.RowAlign`: `Start`, `Center`,

`End`, or `Nearest`, which moves the list the least distance and not at all when the row is already in view. The list moves once for each distinct `serial`. Holding a request in state across later renders leaves a person's own scrolling alone; jumping again, even to the same row, or restoring a remembered position takes a new serial. The rows around the requested one are built in the render that asks for them, so a jump lands on built rows without building any row in between.

- `on_range : [None, Some((a, Gui.Event.VisibleRows => Gui.Action(a)))]`: Receive `{ start, end }`, the half-open range of rows the viewport intersects, whenever it changes. This is the hook for paging: an application that holds only part of its data reads the rest as the viewport approaches it. An action that changes nothing still lets a pending rebuild of the list happen.

Visible rows are a fact about a presented frame, so `on_range` is delivered by GPUI frames alone. Without a window, the semantic runner takes the viewport to be as tall as the window the application asked for: a list mounts the rows such a viewport shows and one viewport below them, and a scroll request moves that viewport by the same arithmetic the window uses.

Keyboard shortcuts and focus

A shortcut is a key chord and a handler, declared by the element it belongs to rather than registered with the window:

- `elem.shortcuts([{ keys, on_press }, ...])`: Answer each chord anywhere in the window while `elem` is mounted and presented.
- `elem.focus_shortcuts([{ keys, on_press }, ...])`: Answer each chord only while keyboard focus is inside `elem`.
- `elem.request_focus(serial)`: Move keyboard focus to the first enabled control inside `elem` once for each distinct `serial`. Holding a serial across renders leaves focus wherever a person has since moved it; a new serial moves it again, and 0 asks for nothing.

`keys` names modifiers and a key joined by `-`: `ctrl-k`, `shift-down`, `f2`, `[`. `secondary` is Ctrl on Linux and Windows and Cmd on macOS. The handler receives an `Event.Key` whose `keys` is the chord in the host's canonical spelling, modifiers in a fixed order, whatever spelling was declared.

- `Gui.chord(keys)`: Show a chord as text, declared exactly as a shortcut declares it, in the spelling of the platform it runs on: `Gui.chord("secondary-k")` reads the Command symbol (`U+2318`) followed by `K` on macOS and `ctrl-K` on Linux and Windows. It inherits colour and size from its container and restyles as text does. A hint built this way always names the keys the person presses; a hint written as literal text is right on one platform only. A chord the host itself takes, such as `escape`, may be shown, since naming it is not declaring it.

```
navigation = [
    { keys: "down", on_press: |state, _| Gui.Action.update(select_next(state)) },
    { keys: "up", on_press: |state, _| Gui.Action.update(select_previous(state)) },
]
```

```
Gui.col({ width: Fill }, rows).shortcuts(navigation)
```

A keystroke reaches at most one shortcut. The host's own keys come first: Tab and Shift+Tab move focus, Escape dismisses, a focused button takes Enter and Space, a focused checkbox Space, and a focused text field its editing keys and every character it types. Declaring Tab, Shift+Tab, Escape, or the App access chord is refused, because none of them could ever arrive. Otherwise the elements enclosing the focused control are asked first, innermost first, with both their window and focus shortcuts; then every other presented element's window shortcuts, where the element later in the window's order wins a chord two declare. While a dialog is open only the shortcuts declared inside it are live. A shortcut's handler runs like any other handler, as a `key` cycle, owned by the component that declared it.

Shortcuts and focus requests are carried by the element they are declared on: an element that is not a popover is wrapped in a region that takes its share of the parent's layout and presents nothing, and a popover or tooltip takes them itself. Always wrap the same element the same way, so that the region, and the controls inside it, keep their identity across renders.

Modifiers

A property record is how an element is written. The code that builds an element knows what it is, so it states the element's presentation in one place. A caller that receives an element something else built, such as a themed component or a translated child, adjusts it with modifier methods instead.

Every presentation field is also a method on an element: `width`, `grow`, `padding`, `gap`, `bg`, `fg`, `radius`, `font_size`, `align`, `justify`, the per-side `padding_top` and `border_bottom`, and so on for each shared style field, together with `label`. Control-specific fields such as `enabled`, `placeholder`, `read_only`, `axis`, `row_gap`, `fit`, `grayscale`, and `heading` follow the same rule, as do the handlers `on_press`, `on_hover_enter`, `on_hover_exit`, `on_check`, `on_change`, `on_submit`, `on_input`, `on_dismiss`, and `on_pointer`, and the keyboard's `shortcuts`, `focus_shortcuts`, and `request_focus`. A modifier that does not apply to an element leaves it unchanged. A typographic modifier on `Gui.text` styles that string, and a modifier on a translated component applies to the root that component renders.

In `examples/counter`, `Counter.render` owns how a card looks and the page decides how much of the row each translated card takes:

```
share : Gui.Elem(State) -> Gui.Elem(State)
share = |card| card.width(Fill).grow(True)

cards = Gui.row({ width: Fill, gap: 24 }, [
    share(Gui.translate(|counter| Counter.render("Left", counter), |parent|
```

```
parent.left, |parent, counter| { ..parent, left: counter })),
    share(Gui.translate(|counter| Counter.render("Right", counter), |parent|
parent.right, |parent, counter| { ..parent, right: counter })),
])
```

The component stays reusable at any width because it never chose one, and the page states its division of space without reaching into the component.

Translations and local state

Begin by splitting a large render function into ordinary helper functions. Introduce a translation when a part of the UI should work with its own smaller state and update independently. For example, each counter can own its counting logic while the page holds two counter states.

`Gui.translate(render, get, set)` introduces a local rendering, delegation, and task-ownership boundary. `get` reads the child's state from the parent; `set` puts an updated child back while preserving the rest of the parent:

```
render = |curr_state| Gui.col({}, [
    Gui.text(curr_state.title),
    Gui.translate(
        |counter| Counter.render("Left", counter),
        |parent| parent.left,
        |parent, counter| { ..parent, left: counter },
    ),
])

main = Gui.run({ init, render })
```

An unkeyed translation has a fresh lifetime whenever its parent reconstructs it. Its own local updates preserve its lifetime and parent-owned application data survives reconstruction unless an action changes it.

Preserve a component across parent renders

Use `Gui.translate_with(render, config)` when a component should keep its lifetime across parent rebuilds or reordering. A stable key lets an in-flight task still reach that same component after the parent renders:

```
Gui.translate_with(
    |counter| Counter.render("Left", counter),
    {
```

```
        key: "left",
        get: |parent| parent.left,
        set: |parent, counter| { ..parent, left: counter },
    },
)
```

The required configuration fields are `key`, `get`, and `set`. `on_delegate : parent -> Gui.Action(parent)` defaults to `Gui.Action.update`. `memo` defaults to `None`; `Some(|prev_state, curr_state| prev_state == curr_state)` opts into comparison of child inputs.

`Gui.try_translate(render, config)` has the same configuration but its getter returns `Try(child, [Removed])` and its setter returns `Try(parent, [Removed])`. Use it for removable collection entries. Resolve a stable application identity against the latest state; do not capture a render-time row index. Adapters do not decode keys. A successful setter preserves the logical identity and presence of its projection and unrelated parent data. A missing projection cannot recreate deleted state. If a setter returns `Err(Removed)`, the whole candidate is discarded before invoking delegation handlers or launching tasks. Emitting a boundary whose getter returns `Err(Removed)` is an application error; omit it from the element tree instead.

Keys identify instances within a scope

Use a string literal for a fixed component, `Gui.Key.from_str` for a computed name, or `Gui.Key.id(U64)` for a numeric application identifier. The string `"7"` and `Gui.Key.id(7)` are distinct keys. Compute dynamic keys when creating model entries and reuse them during rendering.

Keys must be unique within their owner and native-container scope. A container's kind and semantic label help define that scope. Reordering within that scope preserves the lifetime; changing key or scope, removing and remounting, or remounting the owner creates a new lifetime. Instance IDs are never recycled within a session. The architecture chapter describes the identity protocol.

A key promises the same logical component and stable state projection for its lifetime. Rebinding either requires a different key. The platform does not compare functions to infer identity. Keep container kinds and semantic names stable when native interaction state should survive. A translation adds no padding or layout surface.

Persistent application collections

Keep a small collection in a list unless measurements give you a reason to change it. For a large collection with frequent single-entry updates, retaining an old state can force the list to be copied. An index limits how much shared storage each update needs to copy; it does not eliminate copying.

`Index(value)` is a persistent map keyed by application `U64` identifiers. `Gui.Index.empty()`, `Gui.Index.get(index, id)`, `Gui.Index.set(index, id, value)`, and `Gui.Index.remove(index,`

`id)` provide lookup, insertion/replacement, and removal. Lookup returns `Try(value, [Missing])`. Updating an index preserves older snapshots and copies at most sixteen radix branches, rather than an entire collection. Numeric storage identifiers are separate from mounted-instance IDs and from the full-digest `Gui.Key` used by translations.

Store dynamic keys alongside their model entries. A removable collection can keep its display order separately, project entries through `Gui.Index.get`, and translate `Missing` into `Removed`. This lets a local action replace one entry without copying an unrelated flat list. The row-boundaries application uses the same persistent index as the platform's production route registry.

Local updates and delegation

Sometimes a child change also affects its parent: a counter may contribute to a page total, or a review decision may need approval from the queue. Choose delegation when the parent must participate in the change.

`Gui.Action.update(next_child)` commits through the adapters and renders only its originating boundary. Parent and sibling renderers and comparators are not visited. A parent summary responds through explicit delegation or composition inside the same render boundary; reading child state does not establish a tracked dependency.

`element.lift(get, set)` adapts a prebuilt element without adding a render boundary, lifetime, or memoization. It does not consume delegation. Its non-fallible adapters must remain valid for the existing owner's lifetime.

`Gui.Action.delegate(next_child)` constructs a candidate parent through the nearest translation, then calls `on_delegate(candidate)`. That handler may:

- return `Gui.Action.update(candidate)` to accept;
- return an update with revised parent data;
- return `Gui.Action.none()` to veto the entire candidate;
- return `Gui.Action.task` to start parent-owned work; or
- return `Gui.Action.delegate` to continue to the next parent policy.

Delegation reaching the root becomes a root update. Handlers receive current state. Veto commits none of the candidate, including its child changes.

Memoization is opt-in

Add memoization when a parent frequently rebuilds an expensive child whose input has not changed. A key alone preserves lifetime; memoization additionally lets the platform reuse the child's rendered UI. Start without it so there is no comparison policy to maintain or extra state snapshot to retain.

Ordinary translations require no equality and retain no comparison snapshot. A keyed translation enables memoization with `memo: Some(compare)`. Comparison runs only for boundaries reached by rendering and with a valid saved snapshot. A hit retains the subtree and captured handlers; it does not suppress accepted state transitions or task launches. Local updates never scan siblings.

The compared input must cover all variable rendering and captured-handler dependencies, including labels, increment amounts, and layout properties. Renderer captures and comparators must remain semantically stable for a keyed lifetime. Returning `True` promises that keeping the old subtree and handlers is equivalent to rebuilding them. Equality is suitable only when it satisfies that promise: custom equality can omit fields, and floating-point equality can equate signed zero even if rendering distinguishes it.

Descendant updates invalidate strict-ancestor snapshots without rendering those ancestors. This prevents stale `A -> B -> A` reuse. Enabling or disabling memo preserves instance identity and task ownership but rebuilds the boundary and resets its cache. Saved inputs can inhibit Roc's in-place updates; measure the production scaling cases before choosing a comparison policy.

Tasks follow the accepting boundary

Tasks launch only after their pending-state graph transaction is accepted. Resolvers receive the latest state. Matching keyed rebuilds and reordering preserve task ownership. Parent-owned tasks survive removal of their source child. If the owner or its projected state has disappeared, completion delivery is discarded. Reusing a removed key cannot revive old tasks.

Discarding a completion releases its result and captures through ordinary reference counting; it does not promise cancellation of the underlying operation. Use capability-specific cancellation for blocked work and application request IDs to reject superseded results within a surviving instance. Route revisions likewise prevent stale events from reaching replacement handlers.

See `examples/counter` for plain translations, `examples/review-queue` for explicit parent policy and removable state, and `examples/terminal-workspace` for keyed resource-bearing panes.

Directory capabilities and tasks

`access.pick_directory!()` requests a trusted host-owned Open Project selection. The selected project grants read authority, while cancellation is a normal `Canceled` result and denial is `PickDirectoryErr(AccessDenied)`. The returned `Gui.Files.Dir.Read` is an opaque, reference-counted handle: it may be stored in state or captured by tasks, but applications cannot forge it or recover an ambient path. Internally it uses the platform-wide typed `Resource.Handle` representation, leaving the same ABI shape available to future resource APIs.

Open Project is presented by the operating system, never by application code. On Linux Wayland the host asks the XDG Desktop Portal with directory, single-selection, read-only and modal options and accepts one portal-returned `file://` directory. On macOS the host asks the running window for the native directory panel, and the window owning that panel is the only thread that shows it. Either way the host opens the selected directory once at the trusted broker boundary and places only the resulting capability in the registry.

An absent chooser is `PickDirectoryErr(Unavailable)`: a missing portal or D-Bus on Linux, and on macOS a request made before the window exists or from the window's own

thread, which is the thread that would have to show the panel. One chooser may be active at a time, and cancellation or refusal starts a short host-owned cooldown that prevents application-driven prompt loops. An already provisioned development grant never prompts and is not affected by that cooldown.

Every root grant records whether its source was trusted chooser selection or development provisioning, and has session lifetime until the application remembers it. Derived child handles carry their immediate parent and inherited source. This metadata is host-private; no path or grant identity enters captures. `--host-cap-dir` remains development and automation provisioning and never represents consent.

`directory.list!()` lists at most 10,000 direct children and 4 MiB of UTF-8 names. Entries distinguish files, directories, symlinks, and other file types. `directory.open_dir!(name)` acquires a direct child directory without following the final symlink; names containing traversal or multiple path components are rejected. `directory.read!(name)` reads one direct ordinary file as a bounded `List(U8)` without following links. The 64 MiB bound returns `ResourceLimit` instead of constructing an unbounded application value. `directory.sha256!(name)` answers the SHA-256 digest of one direct ordinary file as 64 lowercase hexadecimal digits. The host streams the file in fixed chunks without following links, so none of its bytes becomes an application value; a file over 1 GiB answers `ResourceLimit`, and every other failure is the same `ReadFileErr` a read of that file would give. An application compares a file with a recorded identity this way, such as a specification with the `spec_hash` of its capture, without reading it. The host counts files hashed, hashes refused, and bytes hashed, which `expect-hash-counters` asserts; no name, path, or digest is evidence. Run filesystem operations inside `Gui.Action.task`, as `examples/folder-browser/main.roc` shows.

Single-file capabilities

`access.pick_file!(types)` asks the person for one file in the same trusted surface Open Project uses, and grants read authority for exactly that file. Each `Gui.Files.FileType` is a `label`, a list of `extensions` written without a leading dot, and a list of `mime_types`; an empty list offers every file. On Linux Wayland the XDG Desktop Portal is asked for one file with one filter per type. On macOS the window's native panel is asked for one file. Whatever presented the choice, the host then checks the chosen name against every offered extension and answers `PickFileErr(Unsupported)` for any other file, so the filter is a guarantee rather than advice. A type offered only by MIME type is left to the chooser, because the host does not sniff content. An extension that is a pattern rather than a name, or a type that offers nothing, is `PickFileErr(InvalidName)`.

Cancellation is `Ok(Canceled)`, and the chooser follows the same one-at-a-time rule and refusal cooldown as Open Project. A chosen file is `Chosen({ name, file })`, where `file` is an opaque, reference-counted `Gui.Files.File.Read`. It names the file and not its folder, so it reveals no sibling and takes no name from the application. `file.read!()` reads the whole file under the same 64 MiB bound and without following a link, `file.sha256!()` hashes it as `directory.sha256!(name)` hashes a child, and `Gui.Sqlite.open_file_read!(file)` opens it in place as a read-only database.

The grant is recorded as a `document` root with `read` and `derive`, with its origin and its lifetime, and a database opened from it is derived from it. Withdrawing the file withdraws the database, even after the application has released the file handle and kept only the connection. `--host-cap-file PATH` provisions the file the chooser answers with, and `--host-cap-file-canceled` answers it with a cancellation; both are development and automation provisioning and never represent consent.

Filesystem failures use the `Gui.Files.FileErr` tag union. Its `ListDirectoryErr`, `OpenReadDirectoryErr`, `PickDirectoryErr`, `PickFileErr`, `ReadFileErr`, and `WatchDirectoryErr` tags identify the failed operation directly; each carries a portable `Gui.Files.Reason` value.

`Revoked` is distinct from an invalid handle. Each directory operation resolves the opaque handle and its immutable root lineage while holding the broker registry mutex. That check is the revocation linearization point. Revocation that wins the mutex causes that root and every derived child to reject future list, child-open, and read operations as `Revoked`. An operation that already won the check owns an `Arc<Dir>` and completes under its ordinary bounds; revocation does not close that accepted resource. Task submission alone is not acceptance, so queued work that has not reached the check is rejected.

The trusted host access snapshot exposes only counts grouped by source, session lifetime, read rights, and active or revoked state. Root identifiers, names, paths, and URIs remain internal and never enter capture evidence.

Retain a root-to-current list of directory handles to implement Back and clickable breadcrumbs without exposing paths. Relist a retained handle before committing navigation, and attach a monotonically increasing request identity to asynchronous work so a superseded completion cannot replace a newer view.

Dropped files

A file a person drags onto the window and drops on a `Gui.drop_target` is granted exactly as a file chosen with `access.pick_file!` is: one `Gui.Files.File.Read` that reads that file and nothing beside it. The drop is the choice, made in a gesture the operating system's drag-and-drop delivers rather than the application, so no further question is asked. The host admits each dropped item against the target's `types` with the check a chosen file passes — an ordinary file, resolved so the handle never follows a link, whose name carries an accepted extension — and grants only what passes. Grants are made as the target's route asks for them, so a drop that reaches no live route grants nothing.

Each granted file is a `document` root whose origin is `drop`, recorded apart from a chooser's `trusted-selection` so evidence says which it was, and with the same consent-only enforcement. It is listed, withdrawn, and described exactly as a chosen file is. The document owner counts drops delivered, files they granted, and items they refused, which `expect-drop-counters` asserts; no name or path is evidence.

Recent files and folders

An application may remember a file or folder a person gave it, so a later run can reopen it without asking again. `access.remember_file!(file)` and

`access.remember_directory!(directory)` remember a root the person chose, dropped, or was provisioned; a handle derived from another, such as a subfolder, names no place a person chose and answers `Err(Unsupported)`, and a withdrawn one `Err(Revoked)`. Remembering takes the application's `Access`, not only the handle, so a library handed a file can read it but cannot keep it. The grant and everything derived from it are then recorded with a `remembered` lifetime.

`access.recent!()` lists what was remembered, most recent first, as `Gui.Files.Recent` entries `{ key, name, kind, status }`. The host keeps the list, outside the application's own storage and outside every capture; the application sees an entry's `key`, which stays the same across restarts, its `name`, whether it is a `File` or a `Directory`, and whether it can be reopened now. No path is ever part of an entry. The list holds at most 128 entries; remembering an entry again moves it first, and one more than the bound forgets the oldest.

The host checks every entry against what is at its place as it lists it, and again before it reopens one. `status` is `Available`, or `Unavailable(reason)` with a `Gui.Files.Unavailable`: `Missing` when nothing is at its place, `Replaced` when something else is — another file renamed over it, or a folder where a file was — `AccessDenied` when the application may no longer read it, and `Unreadable` for any other failure. A replaced file has the remembered name and is still refused, because it is not the file the person chose.

`access.reopen_file!(key)` and `access.reopen_directory!(key)` grant what an entry names again, as a new root with its original origin and a `remembered` lifetime, or answer why not; an entry that is not in the list is `Forgotten`, and one of the other kind is `Unsupported`. `access.forget_recent!(key)` removes an entry; the grants it produced this session are not withdrawn. Withdrawing a remembered grant in the trusted App access surface forgets its entry too, and the surface lists every remembered entry with a control that forgets it, so a person can forget without the application's help.

Listing checks each entry, so it is bounded work rather than free: an application reads it where blocking briefly is acceptable, such as its `on_open`, or in a task. The recent-list owner counts grants remembered, entries reopened, reopens and remembers refused, entries forgotten, and the entries it holds, which `expect-recent-counters` asserts.

An interactive run keeps the list in the user's state folder, beneath `roc-gui` and the application's name. `--host-recent PATH`, `--host-recent-each DIR EXT`, and `--host-recent-copied NAME` provision a list for the run instead, most recent first, recorded as provisioned; a provisioned list is never written back. A specification or smoke run without them starts with an empty list of its own.

Watching granted files

`directory.watch!()` subscribes to changes of a granted directory's direct children, and `database.watch!()` to changes of an open database: its file and its write-ahead log. Both return an opaque `Gui.Files.Watch` with the pull contract of a timer. `watch.next!()` waits for the next change and belongs inside `Gui.Action.task`; exactly one wait may be outstanding, and the next is started only after the previous completion has been applied.

A wait answers a `Gui.Files.Change`. `Changed({ names, replaced, overflowed })` names what changed since the previous wait returned, each name once: the children of a watched directory that were created, removed, renamed, or written, or a watched database's own name when a writer committed to it or it was replaced. `replaced` is set when a name may now refer to a different file, because a file was renamed over it or it was removed or created; an open database still reads the file it opened, so a replaced database is read by opening it again. Nothing beneath a child directory is reported.

Changes are coalesced rather than queued. However fast a file is written, a watch holds only the set of names that changed, at most 256; beyond that, or when the operating system's own queue overflowed, `overflowed` is set and `names` is incomplete, so reread everything the watch covers. A wait returns once writing has paused for 10 milliseconds, or 100 milliseconds after the first change it reports, so a reader woken by a commit reads that commit and a writer that never pauses is still reported. A database reader opening the log and index SQLite keeps beside a database is not a change to it.

A watch is derived from the grant it watches and recorded as a `watch` grant beneath it. Withdrawing that grant ends the watch with `Revoked`, and releasing the directory or database it watches ends it with `Canceled`, as does `watch.cancel!()`, which is idempotent, and dropping the last handle. `Gone` ends a watch whose directory was itself removed or moved away. Every later wait of an ended watch answers the same. A process holds at most 64 watches; starting another, or exceeding the operating system's limit, is `WatchDirectoryErr(ResourceLimit)` or `WatchDatabaseErr(ResourceLimit(_))`.

```
wait = |state, watch| Gui.Action.task({
    pending: state,
    run: || watch.next!(),
    resolve: |latest, change| match change {
        Changed(changes) => reread(latest, changes)
        _ => Gui.Action.update({ ..latest, watching: None })
    },
})
```

Encoded image data

`Gui.ImageData.inspect!(bytes, format)` validates bounded encoded bytes already obtained through application data or an explicit capability and reports pixel dimensions. It accepts neither paths nor URLs. Raster inspection uses the maintained `image` decoder and SVG inspection uses `usvg`; inputs above 64 MiB, dimensions above 32,768, and images above 200 million pixels are rejected.

Failures carry the operation tag `InspectImageErr` and one of `Corrupt`, `InvalidDimensions`, `ResourceLimit`, or `Unsupported`. Image presentation continues to use the byte-based `Gui.image` properties. `Gui.ImageData` owns metadata and integrity inspection only; it does not duplicate layout or rendering props.

The native image-data owner counts staged image nodes, staged encoded bytes, inspections, and inspection failures. Semantic specifications compare those numeric counters with `expect-image-owner-counters`; captures never store encoded content, image names, paths, dimensions, or application text.

Shipped application assets

An application's own artwork is not user data and not durable state. It is the application's own read-only content, so it is provisioned automatically and prompts nobody, and it belongs to a surface of its own.

Small artwork needs no surface at all. A compile-time file import places the bytes in the executable and `Gui.image` renders them:

```
import "icons/folder.png" as folder : List(U8)

icon = Gui.image({ label: "Folder", bytes: folder, format: Png })
```

`Assets` is for the larger asset — a background banner, a photograph — that should not be paid for in executable size. An application opens a store once and reads a file when it wants it; the bytes flow into that same `Gui.image`.

A store is anchored to a named root, not to an ambient path. Three roots exist because moving an executable, changing the working directory, and installing content apart from the program are three separate events, and none may silently change what another means:

- `Gui.Assets.beside_executable(path)`: A directory beneath the executable's own directory. The artwork travels with the program, so the store resolves the same way however the program was launched.
- `Gui.Assets.working_directory(path)`: A directory beneath the process working directory. The root follows the shell, which is what running from a project checkout wants.
- `Gui.Assets.content_directory`: The content directory the host provisioned for this application. The application names no path here, so an asset store is never a route to a directory the host did not hand it.

`access.assets!(config)` returns an opaque `Gui.Assets.Store`; `store.read!(path)` returns the bytes of one asset beneath it. Both touch the disk, so both belong inside `Gui.Action.task`, which runs them on the bounded worker pool and resolves on the UI thread:

```
Gui.Action.task({
    pending: { ..state, banner: Loading },
    run: || {
        store = state.access.assets!(Gui.Assets.beside_executable("assets"))?
        store.read!("banners/hero.png")
    },
```

```
    resolve: |latest, result| match result {
        Ok(bytes) => Gui.Action.update({ ..latest, banner: Loaded(bytes) })
        Err(_) => Gui.Action.update({ ..latest, banner: Failed })
    },
})
```

`Gui.Assets.stub` is the resource-free store value. An application that keeps a store in its state needs one to write that state down in a pure `expect`; reads through it fail, and it proves nothing about asset resolution.

Validating an asset set at startup

An asset set may declare a `roc-assets.manifest` file at its root: an `asset_set` name, a `schema` version, a `content_version`, and an optional `sha256` digest, one `key = value` per line with `#` comments.

```
# The gallery's shipped artwork
asset_set = gallery
schema = 1
content_version = 7
sha256 = 3b1f...c90a
```

`Gui.Assets.with_manifest(config, expectation)` makes the open check that file, so a half-updated or mismatched asset set is reported at startup rather than surfacing much later as one image that will not draw. The four comparisons are reported separately: `AssetSetMismatch`, `SchemaMismatch`, `ContentVersionMismatch`, and `ContentHashMismatch`. A manifest that is absent, unreadable, or not a manifest is `ManifestMissing`, `ManifestUnreadable`, or `ManifestMalformed`; an expectation the host cannot use, such as a digest that is not 64 hexadecimal characters, is `InvalidExpectation` and is answered before any file is touched.

The comparison is against the manifest's own declarations. Nothing walks or hashes the files beneath the root, so an open costs the same for one asset as for ten thousand.

What a store refuses

A store root is a directory handle, never an ambient filesystem capability. The host resolves every root and every asset path through the one resolver that serves granted directories. An absolute path, an empty path, a path holding a NUL, and any `.` or `..` component are refused rather than rewritten, and each step of a walk refuses to follow a symbolic link, so nothing inside a store can name anything outside it. A final entry that is a symbolic link or is not a regular file is `Unsupported`. One asset reads into one value up to 64 MiB; beyond that the read is `ResourceLimit`. The host never calls `chdir`.

The native asset owner counts opens, refused opens, manifest checks, reads, refused reads, and bytes read. Semantic specifications compare those six numbers with `expect-asset-counters`. No path, file name, or asset content ever becomes evidence.

Process and pseudo-terminal capabilities

`access.process!()` returns an opaque process grant only when the application is launched with `--host-cap-process=local-shell` or `--host-cap-process=test-program`. The grant fixes the only profile that may be started. `grant.spawn!` accepts only terminal dimensions; applications never provide a profile, executable path, arguments, environment variables, or command lines to the host.

`Grant.spawn!` creates a real pseudo-terminal with an explicitly bounded row and column size. `read!`, `write!`, `resize!`, and `cancel!` are methods on its typed `Gui.Process.Pty` handle. Reads and writes are capped at 65,536 bytes per call, one read may wait on a PTY, and cancellation wakes that wait. Blocking operations belong in `Gui.Action.task`, whose completion is reconciled with the latest state on the ordinary event route.

Failures carry their operation in the tag, such as `SpawnProcessErr(AccessDenied)` or `ReadProcessErr(Busy)`. Process grants and PTYs are opaque `Box(U64)` resources and cannot be confused with other handle kinds.

HID device capabilities

`access.device!()` returns an opaque grant only when the host has authorised one exact virtual device or physical USB vendor/product identity. Applications cannot request arbitrary hardware, inspect operating-system device paths, or read serial numbers. `grant.discover!()` lists only comparison-safe identity within that grant, and `grant.connect!()` connects exactly the granted device.

Connections are typed `Box(U64)` resources. `connection.transact!` exchanges one bounded byte request and response; device-specific framing and values belong in an application or protocol package rather than the platform. Physical HID and deterministic virtual transports enter this same operation. Blocking discovery, connection, and transactions run through `Gui.Action.task`.

Failures retain their operation in tags such as `DiscoverDeviceErr(NotFound)` and `TransactDeviceErr(Protocol)`. Connection and transaction lifecycle counts are recorded by the host that owns those events.

System observation capabilities

`access.system_monitor!()` returns an opaque typed sampler only when the host has granted read-only system observation. `sampler.sample!()` refreshes CPU, memory, aggregate process disk I/O, network I/O, and a bounded process list on a task worker. `sampler.close!()` is idempotent. Applications cannot request another machine or broaden the grant.

Every snapshot field is a tagged `Value` or `Unavailable(Unsupported)` value. Permission denial is an `AcquireSystemErr(AccessDenied)` operation failure, not a fabricated unavailable

sensor. Overall CPU is normalised across the machine and uses tenths of one percent in `0..1000`. Per-process CPU uses the same unit and can exceed `1000` for multi-core use. Memory and I/O counters are bytes as reported by `sysinfo`; the platform does not infer unavailable measurements.

Processes are bounded and sorted by CPU, memory, then PID. Only PID, display name, CPU, and memory cross the ABI; user, executable path, arguments, environment, hostname, and machine identity do not.

Durable application data

`access.app_data!()` acquires the application's private read-write directory granted by `--host-cap-app-data PATH`. The result is an opaque typed `Gui.Files.Dir.ReadWrite` handle; applications cannot recover the host path.

`directory.read_utf8!(name)` distinguishes `Missing` from `Value(Str)`. `directory.write_utf8_atomic!(name, value)` synchronises a private temporary file before replacing the direct child in the same directory. Names are flat ASCII identifiers of at most 128 bytes, values are UTF-8 of at most one MiB, and the ordinary operation-tagged `Gui.Files.FileErr` surface reports acquisition, read, and write failures.

Run these blocking operations inside `Gui.Action.task`, with a monotonic request identity when an edit can supersede pending work. The Settings Center uses this route and retains a newer draft when an older load completes.

Audio output capabilities

`access.audio!()` acquires an opaque output handle for ordinary mixer output. Playing sound through the shared system mixer is an automatically provisioned service with no prompt, and the handle lasts for the session; exclusive access or unusual routing would be separate scope. Acquisition fails when the machine has no usable output device. `output.load!(directory, name)` loads one direct file through an existing `Gui.Files.Dir.Read` grant; names with traversal or multiple components are rejected and encoded input is bounded to 64 MiB. The host delegates media decoding and playback to Rodio and Symphonia.

`Gui.Audio.Output` and `Gui.Audio.Track` are opaque host resources whose operations are methods on the handle, like every other resource. A track's `play!`, `pause!`, `seek!`, `status!`, and `stop!` are bounded operations intended for `Gui.Action.task`. Status reports the decoder-owned duration, playback position, and `Playing`, `Paused`, or `Stopped` state. `Gui.Audio.AudioErr` identifies the failed operation and carries a portable reason such as `DecodeFailed`, `OutputUnavailable`, or `InvalidCapability`.

Specification runs may receive a harness-owned null-output grant. It drains the same paced Rodio mixer, decoder, and player state used by a system output; it changes only the final device sink.

Clipboard capabilities

`access.clipboard!()` acquires text clipboard authority only when the host was launched with `--host-cap-clipboard`. Its opaque `Gui.Clipboard.Handle` uses the common typed `Box(U64)` representation and cannot be manufactured or confused with another resource family.

`handle.read_text!()` returns a bounded UTF-8 snapshot with a monotonic sequence. Applications compare sequences to suppress stale reads and deduplicate repeated content. `handle.write_text!(text)` schedules replacement on GPUI's owning UI context. Both directions reject values larger than 64 KiB. `Gui.Clipboard.ClipboardErr` identifies acquisition, reading, and writing with distinct operation tags carrying a portable reason.

Observe changes with one pull-based `Gui.Timer` subscription inside `Gui.Action.task`. Pause by cancelling that timer; sensitive values are discarded before they enter state. Clipboard content is application data and never enters host diagnostics or observatory captures.

HTTP requests

Applications declare `roc-lang/http` v1.0.0 and build its canonical `Request` values. `access.http!()` returns an opaque `Gui.Http.Client` only when the host was launched with `--host-cap-http-origin ORIGIN`. The grant names one exact scheme, host, and effective port; requests and redirects cannot leave it. Acquire failures are `AcquireHttpErr(reason)` and exchange failures are `SendHttpErr(reason)`, retaining the operation together with the portable reason. Provisioning grants authority for development and automation; they are not a substitute for an application's user-visible trusted-destination setup.

The host disables ambient proxy discovery, rejects credentials embedded in URLs, and pins an accepted non-literal destination resolution for the exchange. Literal loopback or private addresses remain usable only when named explicitly by the provisioned origin.

`client.send!(config, request)` performs one bounded exchange and returns the package's canonical `Response`. Call it from `Gui.Action.task` so connection, TLS, and response work runs on the native worker pool and its reducer returns through the ordinary UI-thread completion dispatcher.

Requests and config make policy explicit: method, URI, headers, body, timeout, `max_response_bytes`, and `max_redirects`. `Gui.Http.Config` carries a default for each limit — a 30 second timeout, a 1 MiB response, and three redirects — so `{ timeout_ms: 2_000 }` states only the limit it moves. URLs must use `http` or `https`. The platform accepts standard methods, at most 64 headers and 32 KiB of header data, a request body up to 1 MiB, a timeout from 1 through 60,000 milliseconds, a response limit from 1 byte through 4 MiB, and from zero through ten redirects. Response bodies remain `List(U8)` so downstream packages share the same nominal types. Portable failures distinguish invalid inputs, denied or invalid capabilities, unsupported schemes, connection and timeout failures, and redirect and body limits.

Give each submitted request a monotonically increasing identity. In the task resolver, compare that identity with the latest state and return `Gui.Action.none()` for an obsolete

result. This prevents a slow earlier response from replacing a newer response without creating a second completion route.

Read-only SQLite capabilities

`Gui.Sqlite.open_read!(directory, name)` opens one direct ordinary child of a granted directory as a read-only database capability. The operation rejects path traversal and symlinks and never exposes an ambient path. The database is read in place, so its size is not bounded by memory, and the file SQLite opens is checked to be the one the directory grant names. The returned `Gui.Sqlite.Db` is an opaque, reference-counted `Box(U64)` handle whose final release closes the native connection. The connection is derived from the directory grant, so withdrawing the folder withdraws the database with it. `Gui.Sqlite.open_file_read!(file)` opens a granted file the same way, derived from the file grant.

The connection is read-only and cannot attach another database. It is not opened as immutable: a database in write-ahead-log mode is read through its log, so every query sees the writes another process has committed, including a writer that is still running. SQLite keeps a write-ahead-logged database's shared index files beside it; the database file itself is never written. An open database never stops another program deleting it or moving another file over its name, on any operating system; a watch of the database reports the replacement.

`database.query!(sql)` prepares and executes one read-only statement. `database.query_with!(sql, params)` binds a `List(Gui.Sqlite.Value)` to the statement's `?` placeholders in order. Values are bound, never spliced into the statement text, and the number of values must match the placeholders. Results preserve SQLite's `Null`, `Integer`, `Real`, `String`, and `Bytes` value kinds. Queries are bounded to 64 KiB of SQL, 256 columns, 256 parameters, 10,000 rows, and 16 MiB of returned text and blob bytes; the same 16 MiB bounds the bytes of the parameters. Limit violations and invalid, corrupt, busy, inaccessible, or non-database resources are typed errors. Failures use `OpenDatabaseErr(reason)`, `QueryDatabaseErr(reason)`, and `WatchDatabaseErr(reason)`, so callers can distinguish the failed operation before matching the portable reason. Every reason retains SQLite's bounded diagnostic string; `Gui.Sqlite.detail` extracts it for user-facing status without flattening the public error type.

A result longer than the row limit is read in pages. `database.page!({ sql, params, rows })` returns a `Gui.Sqlite.Page` holding at most `rows` rows, from one to the row limit, and `more`, which is `True` when the result continues past the page. The next page is a further call whose parameters carry the position: a keyset such as `WHERE id > ? ORDER BY id` bound to the last key seen, or `LIMIT -1 OFFSET ?` bound to the rows already read. A keyset page costs the same at any depth; an offset page costs more the deeper it lies. Each page is one statement, so no read transaction is held between pages and a writer is never blocked by a reader turning pages.

```
next = database.page!({
    sql: "SELECT id, title FROM books WHERE id > ? ORDER BY id",
```

```
    params: [Integer(last_id)],
    rows: 1000,
})
```

Open and query calls are blocking capability operations. Run them inside `Gui.Action.task`, retain the last successful view while work is pending, and use a monotonic request identity in the resolver so an older completion cannot replace a newer result. `examples/database-browser` demonstrates this pattern.

`database.watch!()` reports each commit a writer makes to the database, and its replacement, as the watching section describes. `examples/observatory` watches a capture while its recorder is still writing it.

TCP endpoint capabilities

`access.tcp_connect!()` opens an ordered byte stream to the single numeric `IP:PORT` endpoint granted with `--host-cap-tcp`. Without that host grant it returns `ConnectTcpErr(AccessDenied)`. The application cannot select or derive a different destination from the opaque handle, and releasing its final reference closes the socket and removes the host registry entry.

`stream.read_up_to!(max_bytes)` reads between one byte and a one-MiB requested maximum; an empty list means orderly end of stream. `stream.write_all!(bytes)` writes the complete value or fails and accepts at most 16 MiB. `stream.close!()` is idempotent. Connections, reads, and writes have finite native timeouts, and failures use the operation-tagged `Gui.Tcp.TcpErr` union: `ConnectTcpErr`, `ReadTcpErr`, `WriteTcpErr`, or `CloseTcpErr` wraps a portable reason.

The numeric endpoint has no DNS resolution or ambient destination selection. The command-line grant provisions development and automation authority; a user-facing client obtains trusted connection authority through a named, revocable destination broker rather than treating that flag as user consent.

TCP operations block and belong inside `Gui.Action.task`. Protocol packages build on the byte transport rather than being reimplemented by the platform. The Redis Explorer adapts a stream with `roc-redis`'s `ByteIo.from_empty_eof`, so command encoding, bounded RESP2 decoding, typed commands, and exchange policy remain the responsibility of the established platform-independent package.

Retained canvas

`Gui.canvas` paints an ordered list of keyed primitives in native GPUI, built with `Gui.rectangle({ ... })`, `Gui.ellipse({ ... })`, `Gui.line({ ... })`, and `Gui.canvas_text({ ... })`. A canvas and every primitive have stable semantic labels. Primitive keys are non-zero and unique within the canvas; they preserve direct-manipulation identity independently of list position.

The controlled `on_pointer` handler receives `Begin`, `Move`, and `End` in integer canvas coordinates. The topmost hit-tested primitive is captured on `Begin`, and that same optional target key accompanies the complete gesture, including when a state update replaces the canvas subtree.

A text primitive sets one line of `value` in the canvas's inherited font, at `size` logical pixels and in `color`, within the box from `x` to `x + width` with its top at `y`. `align` places it at the box's `Start`, `Center`, or `End`. Its semantic rectangle is that box, one line tall: `size` and a quarter again. Text labels the shapes beneath it and is never a pointer target, so a caption drawn over a bar leaves the bar as the target.

`on_hover` receives pointer movement with no button pressed. `Move` carries the point and the topmost keyed primitive under it, and is delivered once for each new point; `Leave` is delivered once when the pointer leaves the canvas, with the last point delivered. A pressed pointer is always an `on_pointer` gesture, so hovering never interrupts a drag. `on_wheel` receives wheel and trackpad scrolling as `dx` and `dy` logical pixels, positive towards the content's end, with the pointer's point and target; a wheel that reports lines is converted at the window's line height. A canvas that handles the wheel consumes it, so an enclosing scroll region does not also move. Both handlers are optional. A canvas without one is not listened to for it, and moving the pointer across it records nothing.

Each delivered hover records one `hover` cycle and each wheel scroll one `wheel` cycle, through the same recorded dispatch as a `drag` phase, whether a person or a specification moves the pointer.

A canvas drawn for its size

A canvas is laid out like any other element: its `width` and `height` are lengths, and `Fill` with `grow` lets it take the space its container gives it. Its primitives are placed in integer logical pixels and are not scaled, so a canvas that fills a column learns the column's width through `on_size` and is drawn for it:

```
Gui.canvas({
    label: "CPU history plot",
    primitives: bars(state.history, state.plot_width),
    on_pointer: |_, _| Gui.Action.none(),
    on_size: Some(|current, laid_out| if laid_out.width == current.plot_width
Gui.Action.none() else Gui.Action.update({ ..current, plot_width:
laid_out.width })),
    width: Fill,
    grow: True,
    height: Px(112),
    min_width: Px(240),
})
```

- `on_size : [None, Some((a, Gui.Event.CanvasSize => Gui.Action(a)))]`: Receive `{ width, height }`, the size in logical pixels of the surface the primitives are drawn on,

inside any border. A size is a fact about a drawn frame, so it is delivered after the frame that lays the canvas out at a size it has not yet reported, once for each size. A later frame at the same size delivers nothing; nor does a render that rebuilds the canvas under the same identity, which keeps the size already reported. A canvas mounted again, under a new lifetime, hears its size again. Without a handler the host reports nothing.

Each delivered size records one `resize` cycle, whose target is the canvas. The handler draws for the size it hears; a handler that also changes the canvas's own dimensions must reach a size it keeps, or each frame would ask for another. A canvas that answers the width it was already drawn for with `Gui.Action.none()` makes that turn change nothing.

Only a canvas reports its size. Every other element lays its content out natively, so text wraps, rows share their width, and a scroll region clips without the application knowing the numbers; only a canvas's content is placed by the application in pixels of its own.

Without a window, the semantic runner lays a canvas out at the one extent it knows, the window the application asked for: a fixed dimension is its own size, a flexible one is the window's, each within the canvas's fixed minimum and maximum, less its border. It delivers each presented canvas's size after every step, by the same rule, so a specification sees the application draw for the size a window of that size would give a canvas that fills it.

Resource access

An application can only do what it has been given.

That is the whole rule. Everything in this chapter follows from it, and if some part of the platform ever seems to contradict it, the platform is wrong.

Most of what you build needs nothing from this chapter. A window, a layout, a timer, your own state: none of it touches anything belonging to anyone else, so none of it needs permission. This chapter is about the moment your application wants something that belongs to the person using it — their documents, their network, their devices, their clipboard.

What you start with

Every application begins with a useful private environment, provided automatically and without a prompt:

- a window, its input, and a clock;
- private storage of your own, isolated from every other application;
- the assets you shipped inside your own executable.

None of these holds anyone's data but yours, and none of them can reach anything else, which is exactly why they need no decision from anybody. An application that uses only these never asks the person for anything, and should not.

How you get more

Someone gives it to you. Usually that someone is the person using your application, and the way they give it to you is by **choosing the thing**.

```
access.pick_directory!()
```

This opens the operating system's own folder chooser. If the person picks a folder, you are handed a value that reads that folder. If they cancel, you are handed nothing, and cancelling is an ordinary outcome rather than a failure.

When one file is all you need, ask for one file:

```
access.pick_file!([{ label: "Captures", extensions: ["rgstats"], mime_types: [] }])
```

The person picks a file of a type you offered, and you are handed a `Gui.Files.File.Read` that reads that file and nothing beside it: not its siblings, not its folder. Ask for the smallest thing that does the job; a document viewer that asks for a folder has asked for every other document in it.

Dropping is choosing too. A person who drags a file from their file manager and lets go of it over your window has chosen it as surely as one who picked it in a chooser, in a gesture the operating system delivers rather than your application. Declare where a drop may land and what it may be:

```
Gui.drop_target({ label: "Captures", types: capture_types, on_drop:
open_dropped }, children)
```

Each dropped file of a type you declared arrives in `on_drop` as a `Gui.Files.File.Read`, exactly as a chosen file does, and everything else dropped is named and refused. The grant records that it arrived by a drop rather than a chooser, so the evidence can say which.

There is no second step. No "this application would like to access your files" dialog appears before or after, because the choosing **was** the permission. Someone who picked `~/Photos/2026` has said everything that needs saying, and asking them again only teaches them to click past questions without reading.

That is the practical point. The usual way a permission system fails is that people stop reading it. Every decision this platform asks for is attached to a thing the person just picked, shown in a window they recognise, at the moment they wanted something to happen.

A grant is a value

What you get back is an ordinary Roc value. You hold it in your state, pass it to functions, and store it in a record:

```
State : { project : [None, Some(Gui.Files.Dir.Read)] }
```

Three things are true of it, and between them they are the reason this works.

You cannot make one up. There is no constructor. A `Gui.Files.Dir.Read` exists because someone handed you one.

Its type says what it is for. A `Gui.Files.Dir.Read` reads a directory. There is no permission list to consult alongside it, because the type you are holding is the permission, and at every call site you can see exactly what you are handing over.

Nothing is ambient. No function anywhere in the platform conjures authority out of nothing. Every grant you hold traces back to something a person chose, or to the small private environment above.

Narrowing, and handing things on

You can always give away less than you hold, and never more.

```
covers = project.open_dir!("covers")
```

`covers` reads one subfolder. It cannot see its parent, its siblings, or anything above it. When some part of your application needs to work with the covers, give it `covers` and not `project`.

This is the part that protects **you**, rather than the person using your application.

A Roc package you depend on is code you did not write, compiled into your binary. In most ecosystems such a package holds everything your program holds: it can read any file your process can read and open any connection your process can open. The attacks that actually happen to working developers are overwhelmingly this shape.

Here a library can use only what you hand it. A charting package that takes your data and returns an element cannot read your documents, because you never gave it anything that reads documents and it has no way to ask for one. You do not have to audit it to know that — you can see what it received.

> **NOTE**
> This bounds the blast radius; it is not a guarantee of good behaviour. A library you hand a network client to may use that client for anything. Give the narrowest thing that does the job, and notice when a library asks for something broad.

Taking it back

A person can withdraw a grant. The trusted **App access** surface lists what this application is holding — what each grant is over, how it arrived, and what it allows — with a control to withdraw each one. It belongs to the platform rather than to the application: the application cannot draw it, suppress it, or tell that it is open, which is the only way a withdrawal control is worth anything.

When a grant is withdrawn, everything derived from it stops at the same moment: the subfolder handles, the database opened from that folder, the watch waiting for that database to change, the connection opened through that device. You do not have to hunt them down and close them. They refuse their next use with a typed error that says the authority was withdrawn, rather than one that says something went wrong, and a watch that is waiting wakes to say so.

Two limits, stated plainly because a model that overpromises here is worse than one that does not try:

- Work already finished stays finished. Bytes you have read are in your application's memory, and withdrawing a grant cannot reach in and remove them.
- A grant given for a session ends with the session. Nothing carries over to the next run unless it was remembered, which the next section describes.

Remembering what you were given

A person who opened a document yesterday expects to find it again today without choosing it again. Your application may remember a file or folder it was given:

```
access.remember_file!(chosen.file)
```

What is remembered is a record, not authority, and it is kept by the platform, not by your application: the place the person chose and what was there. Your application sees each entry of its recent list only as a key, a name, and whether it can be reopened now. Reopening asks the platform to grant the entry again, and the platform first checks that the thing at that place is still the thing that was chosen. A file that is missing, one that something else was renamed over, or one your application may no longer read is refused with that reason, never reopened: a name is not an identity, and the person chose a file, not whatever later takes its name.

Remembering takes your `Access`, not only the handle. A library you hand a file to can read it, but it cannot keep it for next time; only the application a person gave it to can.

A remembered grant is listed in the App access surface as remembered, beside a list of everything remembered. Withdrawing a remembered grant forgets it, and each remembered entry can be forgotten there directly, with no help from your application — which is the point of offering it there.

Why it is built this way

The idea is older than most of the software that ignores it.

In 1966 Dennis and Van Horn described a **capability**: an unforgeable reference that both names a resource and carries the right to use it. The essential property is the absence of ambient authority. You cannot act on something merely because you know its name.

In 1988 Norm Hardy wrote up the **confused deputy**, still the clearest argument for why that matters. A compiler on a shared system could write to a billing file its users could not. A user asked the compiler to write output to that path, and the compiler — holding authority the user lacked — destroyed the billing records. Nothing was broken into. Access control lists answer "who is asking", and the answer was "the compiler", which was true and useless. Capabilities answer a better question: what did the requester bring with them?

File paths are that problem's natural habitat. A path is a name anyone can write down, and whether it works depends on who happens to be running. Hand over a directory handle instead and the question cannot arise.

The **powerbox** — a trusted chooser where the act of choosing is the act of granting — came out of the E language and CapDesk in the early 2000s, and is now the mainstream answer rather than an exotic one. WebAssembly's WASI has no ambient `open`: a module receives directory handles or reaches nothing. Deno made permission explicit and per-resource. XDG Desktop Portals on Linux, and the file and photo pickers on macOS and iOS, are the same shape — the operating system shows the chooser and the application receives only what was chosen.

The counter-example is worth keeping in mind too. Android first showed a list of permissions at install time, and people learned to scroll past it, because a list of abstract

capabilities presented before anything has happened is not a question anyone can answer. Every platform that started there has since moved the decision to the moment of use and attached it to the thing being used.

What this costs you

It would be dishonest to present this as free.

- You have to thread grants through your application state. A function that reads a file needs the handle passed to it, so your state has to hold it and your call sites have to show it.
- You cannot open a path you were not given, including for good reasons. A convenience you might expect — reading a configuration file from a known location — is private storage here instead.
- Development and automation need grants with nobody present to give them. Those flags exist, and they are recorded as provisioning rather than as consent, so evidence never reports a test fixture as a decision a person made.
- It does not make untrusted **data** safe. A malformed image is dangerous to decode however narrowly you were granted the file it arrived in.

If a rule in this chapter costs you something and protects nobody, that is a defect in the platform, and worth saying so.

What is deliberately absent

- No permission list in a manifest, and no approval at install time.
- No "allow this application to access your files" prompt, before or after a choice.
- No interface that takes an absolute path.
- No switch that grants everything, for convenience or for compatibility.

The last is the load-bearing absence. A model with an escape hatch becomes a model everybody escapes, and then the sentence at the top of this chapter stops being true.

Specification language reference

The complete `.scm` vocabulary: locators, operations, assertions, window steps, and case selection. Read Testing your application first if you are verifying an application for the first time; this chapter is the one to reach for once you know what you want to say and need to know how to say it. Steps that depend on this repository's own fixtures are described in Contributing instead, so a vocabulary entry here means the same thing in any application.

Specifications describe the application as a user sees it: find text, identify a button by its semantic name, press it, and check the result. Put `.scm` files in a `specs/` directory beside the `main.roc` they exercise. Both runners read the same directory, because the steps a file contains decide which runner takes it:

```
my-app/
├── main.roc
└── specs/
    └── editing.scm
```

The files contain S-expression data. They are not Scheme programs and do not execute arbitrary code.

A behaviour specification

```
(test "Counter increments independently"
  (steps
    (expect-visible (text "Counter"))
    (expect-visible (text "-1"))
    (click (role button :name "Left increment"))
    (expect-not-visible (text "-1"))
    (expect-visible (text "0"))))
```

Run one application's specifications with a repository-relative glob:

```
python3 scripts/run_specs.py 'examples/counter/specs/*.scm'
```

Run every example and benchmark specification with no pattern:

```
python3 scripts/run_specs.py
```

Each specification gets a fresh process. This isolates application state and produces one `.rgstats` SQLite capture beneath a timestamped `.test-out/specs/` directory. A failing assertion makes the command fail and prints the specification line and reason.

Grants

A specification states the capabilities it needs beside its steps, so a case can be read on its own and nothing is inferred from a file name:

```
(test "a corrupt track is isolated"
  (grants
    (directory "fixture")
    (audio null))
  (steps
    (click (role button :name "Open library"))
    (expect-visible (text "1 track skipped"))))
```

A specification with no `(grants ...)` clause, and one with an empty `(grants)`, both receive no capability at all. Write the empty clause when the absence is the point: a denial case takes the same refusal from the same production acquisition call that a real broker would give it.

Each grant may appear once. A path is relative to the application directory, the one holding `main.roc`, and a path that leaves it is a parse error naming its line, as an unknown grant or a malformed one is.

- `(app-data "PATH")`: Private application-data storage, seeded from `PATH`. The run receives a fresh private copy, so it writes through the production capability without changing the checked-in directory.
- `(assets "PATH")`: The content directory an `Assets.content_directory` store resolves to. The application names no path of its own, so the case names the one the host provisions. Omit the grant and the open is `AccessDenied`, which is how a case exercises an application's missing-artwork path.
- `(audio null)`: A paced null output in place of the system audio device. It changes only the final sink; decoding, mixing, playback, seeking, and status remain the production path.
- `(clipboard system)` **and** `(clipboard fixture)`: Text clipboard authority. The system form reaches the real clipboard; the fixture form gives the case a clipboard the `clipboard-text` step drives, which is rejected without it.
- `(device virtual)` **and** `(device virtual N)`: One deterministic virtual HID device, optionally with `N` controls, reached through the same host capability as physical hardware.
- `(device "VID:PID")`: One physical HID device by hexadecimal vendor and product identifier.
- `(directory "PATH")`: Read access to one directory, reached through the same opaque capability operations as an interactive grant.
- `(directory canceled)`: A directory chooser the person dismisses without choosing. The chooser opens and the acquisition call runs, but no authority is produced and `access.pick_directory!()` answers `Ok(Canceled)`. Cancelling is not a failure, so this is its own provisioning rather than an absent grant, and it is how a case shows that an

application returns to rest instead of accusing the person of a denial. It cannot be combined with `(directory "PATH")`.

- `(directory copy "PATH")`: Read access to a private copy of one directory's files, made for the run and discarded after it, in place of `(directory "PATH")`. It is the grant a case needs to change what the application reads with `replace-file` and `remove-file`, without touching the directory it names.
- `(file "PATH")`: Read access to one file, which `access.pick_file!(types)` answers with. The host checks it against the offered types exactly as it checks a file chosen in the operating system's chooser.
- `(file canceled)`: A file chooser the person dismisses without choosing, so `access.pick_file!(types)` answers `Ok(Canceled)`. It cannot be combined with `(file "PATH")`; either may be combined with a directory grant.
- `(file recording)`: The capture this run is recording, as the file `access.pick_file!(types)` answers with. The production recorder is writing it while the application reads it, which is how a case watches a recording grow. It is semantic-only, because only the semantic runner records a capture.
- `(http-origin "ORIGIN")`: HTTP access to exactly one origin. A request to any other origin is denied by the host.
- `(process local-shell)` **and** `(process test-program)`: One PTY child profile. `test-program` is the fixed deterministic child; `local-shell` is the user's shell.
- `(recents ITEM...)`: The application's recent list for the run, in place of the person's own, the first item most recent. An item is a file or folder relative to the application directory, `(each "DIR")` for every ordinary file directly in a folder, in name order, `(each "DIR" "EXT")` for those whose extension is `EXT`, or `(copied "NAME")` for a direct child of the `(directory copy ...)` grant, which a `remove-file` or `replace-file` step can then take away or replace. Each entry is recorded as provisioned, and a provisioned list is never written anywhere the next run would read. A case without this grant starts with an empty list. Every lifecycle of a case starts from the same list, so a benchmark sample never opens on what the sample before it remembered.
- `(server "PATH" PORT)`: A loopback service the runner starts before the case and stops after it. `PATH` names a script beside the application, and `PORT` is the port it reports readiness on. This grants the harness, not the application: a case still needs `(http-origin ...)` or `(tcp ...)` to reach it. `(system-monitor standard)`, `(system-monitor unavailable)`, and
- `(system-monitor processes N)`: A deterministic system sampler delivering its snapshots through the same ABI as the physical source.
- `(tcp "IP:PORT")`: TCP access to exactly one endpoint.
- `(theme light)` **and** `(theme dark)`: The system appearance the case sees, in place of the desktop's. `(theme dark reduced-motion)` also asks for reduced motion. It grants no authority; a case without it sees a light scheme with full motion, whatever the machine running it prefers.

Locators

`within` scopes any locator to a named part of the mounted graph. Text and prefix locators follow, then role locators in alphabetical order by role.

- `(within ANCESTOR TARGET)`: Matches `TARGET` nodes that are strict descendants of a node matched by `ANCESTOR`. Either locator may itself use `within`. The ancestor never matches itself, and several matching ancestors do not duplicate a target. Multiple target matches remain ambiguous for interactions.
- `(text "value")`: Text whose entire value equals `value`.
- `(text-prefix "prefix")`: Text beginning with `prefix`.
- `(button-prefix "prefix")`: Button names beginning with `prefix`.
- `(canvas-item-prefix "prefix")`: Keyed canvas primitives whose labels begin with `prefix`. This locator supports deterministic primitive counts in scaling specifications.
- `(checkbox-prefix "prefix")`: Checkbox labels beginning with `prefix`.
- `(role button :name "name")`: A button with the semantic `name` passed to `Gui.button`. A tab of a tab strip is a tab, not a button; its close button is a button.
- `(role canvas :name "name")`: A retained drawing surface identified by its stable semantic name.
- `(role canvas-item :name "name")`: A keyed canvas primitive identified by its stable semantic label.
- `(role checkbox :name "label")`: A checkbox identified by its visible label.
- `(role column :name "name")`: A column identified by its semantic name.
- `(role dialog :name "label")`: The active modal dialog identified by its semantic label.
- `(role drop-target :name "label")`: A drop target identified by the label `Gui.drop_target` gave it.
- `(role image :name "name")`: An encoded image identified by its application-supplied semantic name. Neither encoded bytes nor source filenames are captured.
- `(role panel :name "label")`: A panel identified by its semantic label.
- `(role row :name "name")`: A row identified by its semantic name.
- `(role scroll :name "name")`: A scroll region identified by its semantic name.
- `(role separator :name "label")`: A split's divider, identified by the label its split gave it. Nothing in the pane a collapsed split folds away matches any locator.
- `(role tab :name "title")`: One tab of a tab strip, identified by its title.
- `(role tooltip :name "label")`: A popover's presenting surface, identified by its semantic label. A closed popover presents nothing, so neither it nor anything in its content matches.
- `(shortcut "CHORD")`: Every presented element that declares a shortcut for `CHORD`, which is compared in the host's canonical spelling, so `"shift-ctrl-k"` finds a shortcut declared as `"ctrl-shift-k"`.
- `(chord "CHORD")`: Text the host spelled from `Gui.chord`, found by the chord it shows in the host's canonical spelling rather than by its rendering, so one specification locates `(chord "secondary-k")` on every platform while a person reads the Command symbol (`U+2318`) followed by `K` on macOS and `ctrl-K` elsewhere. The rendering is also text, and a text locator finds it.
- `(role textarea :name "name")`: A multiline editor identified by its semantic name. Locator matching never records its value.
- `(role textbox :name "label")`: A controlled text input identified by its semantic label.
- `(role virtual-list :name "name")`: A fixed-row virtual list identified by its semantic name. A list of rows produced on demand mounts only the rows near its viewport, so only those rows can be located.

An interaction locator must resolve to exactly one live node. Ambiguous and missing targets fail instead of choosing one by traversal order. When a text locator matches nothing, the run names on standard error the texts nearest to it under the locator's scope, or says that the scope itself matched nothing. The capture records only the failure; application text never enters it. Use unique semantic names or scope repeated component-local names through a named container:

```
(replace-text
  (within (role panel :name "Left editor") (role textbox :name "Value"))
  "Changed")
```

Both runners resolve scoped locators through the same mounted-graph matcher. Scopes do not depend on native view materialisation, so they also work for logically mounted virtual-list rows. A canvas primitive is a strict semantic child of its owning canvas, so `(within (role canvas :name "Stage") (canvas-item-prefix "Dot"))` counts matching primitives on that canvas. A primitive is a leaf and cannot itself contain further matches.

Row and column locators describe the mounted application graph. They do not assert pixel geometry or operating-system accessibility.

Operations

Steps that act on the application, in alphabetical order.

- `(await-count LOCATOR N)`: Apply task completions until the locator matches `N` nodes, then continue. Output that arrives over several tasks, such as a terminal's, is chunked by the platform rather than by the application, so a specification that counted `await-task` steps would be asserting how an operating system flushed a pipe. This waits for what the application shows instead, and fails if the graph has not said it before the task budget expires. It is also how a specification waits out a popover's delay: the window runner presents frames on the window's own clock, and the semantic runner, which has no clock, lets every pending delay elapse before it waits for a task. Follow it with `expect-count` to assert the same number exactly. On the window runner the step ends on a presented frame carrying those nodes, so a following `expect-on-screen` asks about something the window has actually drawn. It waits for a state, not for a change. A count that already holds when the step begins satisfies it immediately, having awaited nothing, so a locator matching something left over from an earlier step proves nothing about this one. Wait for what the step under test produces, such as the output of the command it sent.
- `(await-task)`: Wait for exactly one accepted worker task and apply its completion through the normal latest-state route. Concurrent tasks may complete in a different order from the order they were issued. A task that resolves to another task requires another `await-task` step.
  On the window runner this claim is weaker, and knowingly so: frames are what apply completions, so the task has usually landed before the step runs, and an application holding a subscription open — a terminal reading, a timer playing, a watch waiting —

never reaches a moment with nothing in flight. The step therefore counts completions rather than waiting for the next one: it is satisfied by one that no earlier `await-task` waited for and that landed after the last step that waited for the window, `settle`, `await-count`, or `await-ticks`, even if it landed during the step that started it. It cannot attribute that completion to a task. Where the distinction matters, say what the application should show with `await-count` instead of counting tasks.

- `(await-ticks N)`: Apply `N` successive timer wait completions through that same worker and latest-state route. `N` must be positive.
- `(click LOCATOR)`: Press exactly one matching enabled control through the application's normal Roc event route. A disabled control consumes no event and does not call Roc, matching the interactive host.
  A canvas-item locator presses the one primitive it names: the pointer goes down and up at the primitive's centre, through its canvas's pointer route, as a `drag` that does not move. A mark is therefore pressed by what it means rather than by where the capture's data put it. The step fails when the canvas's own hit test would give that point to another primitive drawn over it.
- `(hover-enter LOCATOR)` **and** `(hover-exit LOCATOR)`: Rest the pointer on, or take it off, the located node. The transition goes through the canonical mounted graph to every hover target the pointer then rests on: each popover or hover region whose anchor contains the node, outermost first, and the node itself when it is a button with a hover handler. Installed handlers are delivered through the normal Roc route; a popover's surface presents after its delay. A node that is no hover target and lies in no anchor is refused. Duplicate transitions and disabled controls dispatch nothing. Both runners use the same transition owner. The window runner requires an entering target to be painted and visible, then dispatches a GPUI mouse-move event at its centre. Exit moves the simulated pointer outside the window; GPUI performs hit testing and delivers the resulting transitions.
- `(expect-background LOCATOR 0xRRGGBB)`: Compare the explicit application background in the mounted graph. Native hover refinements are separate. Missing backgrounds fail as unavailable; colors are not recorded as count evidence.
- `(mark-native-work)`: Start observing completed native frames in a window run. The underlying production counters remain active independently of recording.
- `(expect-native-work :button-renders-max N :boundary-renders-max N :boundary-elements-max N :cached-prepaint-subtrees-min N :cached-paint-subtrees-min N :replayed-scene-operations-min N :fresh-hitboxes-max N :fresh-mouse-listeners-max N :element-states-moved-min N)`: Bound actual button and boundary view renders and boundary view elements created in every completed frame since the mark. Each limit is optional, but at least one is required. GPUI-owned limits distinguish cached subtree replay from fresh hitbox and listener insertion and finish-time element-state movement. Minimums prove that reuse work actually occurred; maxima bound fresh reconstruction. Element creation measures the host handing a view to GPUI even when that view subsequently reuses its rendered subtree. A missing mark or no completed frame fails as unavailable. Choose limits from the changed patch frontier and the application's ordinary native controls; these assertions belong to controlled interactions without unrelated input or resize.

- `(drag LOCATOR FROM-X FROM-Y TO-X TO-Y)`: Press a canvas or a split's divider at one point, move to another with the button held, and release. Coordinates are integer logical pixels from the painted canvas's origin, or from the divider's. The window runner presses, moves, and releases the window's own pointer, and GPUI hit-tests the press; the semantic runner delivers what that gesture delivers. A canvas receives its three pointer phases, each a `drag` cycle. A divider asks for the size the move measures from the press, one `drag` cycle, and nothing when the divider already shows that size.
- `(drop LOCATOR "PATH"...)`: Drop files on a drop target, each named relative to the application directory and staying inside it, in the order a person dragged them. The window runner delivers the drop through GPUI's own file-drop events — the files enter the window over the target, are held there, and are dropped — so GPUI hit-tests the drop and the target's production listener takes it. The semantic runner admits the same paths against the target's types and takes the same route that listener calls. Either way the drop is one `drop` cycle, and the files it grants are grants the application holds.
- `(pointer-move LOCATOR X Y)`: Move an unpressed pointer to a point of a canvas, in canvas coordinates. The window runner moves the window's own pointer to that point of the painted surface; the semantic runner delivers the event that movement produces. A move to the point already delivered delivers nothing. The canvas must handle `on_hover`.
- `(pointer-move ITEM)`: Rest an unpressed pointer on the centre of the one canvas primitive a canvas-item locator names, through the same route, so a mark is hovered by what it means rather than where the data put it. As with `click`, the step fails when the canvas's hit test would give that point to a primitive drawn over it.
- `(pointer-leave LOCATOR)`: Take a hovering pointer off a canvas. The canvas receives `Leave` at the last point delivered, and nothing when it was not hovered.
- `(system-theme light)` **and** `(system-theme dark)`: The desktop reports a new appearance, as a person changing it would; `reduced-motion` may follow the scheme. Adaptive colours resolve by it unless the application has chosen a scheme over the system's, and a `Gui.Appearance.next_change!` wait answers it. The window runner presents one frame after it, drawn in the new scheme.
- `(wheel LOCATOR X Y DX DY)`: Scroll a wheel by `DX` and `DY` pixels over a point of a canvas that handles `on_wheel`. Positive distances scroll towards the content's end.
- `(focus LOCATOR)`: Give semantic keyboard focus to exactly one enabled button or checkbox. Focus remains valid only while that mounted node is live. Focus entering a popover's anchor presents it at once; focus leaving closes it unless the pointer rests there. The window runner activates the window first, as a person moving focus there would; focus arriving in a region is reported only in the active window.
- `(key "CHORD")`: Press one key chord, such as `"ctrl-k"`, `"down"`, or `"shift-f3"`. The window runner sends a real keystroke through the window's production keymap, so the host's own keys, a focused control, and a focused text field take it exactly as they take a person's. The semantic runner offers it to the application's shortcuts through the same resolution the window's root uses, from the control `focus` last named, and runs the shortcut that answers as a `key` cycle; a chord nothing answers does nothing, as in a window. A chord the window would give to the host or to the focused control instead — Tab, Escape, Enter on a button, a character typed into a text field — fails the

step in the semantic runner, which does not imitate what they do; use `focus`, `press-key`, `replace-text`, or a window run.
- `(mark-metrics)`: Separate benchmark setup from the operation being measured.
- `(press-key Enter)`: Activate the focused button through the normal Roc event route.
- `(press-key Escape)`: Dismiss the active dialog through its `on_dismiss` route. With no dialog, close every presenting popover; with neither, the step fails.
- `(press-key Space)`: Activate the focused button, or toggle the focused enabled checkbox, through the normal Roc event route.
- `(remove-file "NAME")`: Delete the direct child `NAME` of the `(directory copy ...)` grant, as a person deleting a file does. It changes only the private copy, and a specification without that grant is refused.
- `(replace-file "NAME" "SOURCE")`: Replace the direct child `NAME` of the `(directory copy ...)` grant with a copy of `SOURCE`, a file named relative to the application directory. The copy is written beside the folder and renamed over the child in one step, as a person moving a new file into place does, so the application sees one name bound to a different file. It changes only the private copy, and a specification without that grant is refused.
- `(replace-text LOCATOR "value")`: Replace the complete controlled value of one editable text input or textarea through its production Roc input route. A text input delivers `Gui.Event.TextChange`; a textarea delivers `Gui.Event.Input`. A disabled text input consumes no edit and dispatches nothing. A disabled or read-only textarea fails as non-editable.
- `(submit LOCATOR)`: Deliver an enabled text input's current controlled value as `Gui.Event.TextSubmit` through the normal Roc event route.

Assertions

Steps that check the application, in alphabetical order.

- `(expect-before FIRST SECOND)`: Require a match for `FIRST` to occur before a match for `SECOND` in mounted-tree order.
- `(expect-asset-counters OPENS REFUSED CHECKS READS READS-REFUSED BYTES)`: Require the asset owner's six totals, in that order: opens, refused opens, manifest checks, reads, refused reads, and bytes read. No asset path, file name, or content becomes evidence.
- `(expect-app-access open|closed)`: Require the trusted App access surface to be showing, or not. Window-only, and satisfied only once the surface has actually drawn: it is not a mounted node — an application must not be able to find it, style it, or notice it is open — so no locator names it and a claim about the application would be true whether or not it drew.
- `(expect-grant-counters RECORDED RELEASED REVOKED DENIED)`: Require the grant registry's four totals, in that order. Released and revoked are counted apart because they are different facts: an application that stopped holding a handle did not have its authority taken away.
- `(expect-grants DESCRIPTION...)`: Require exactly these grants, in registry order, each written as `KIND ORIGIN/ENFORCEMENT LINEAGE RIGHTS` with `revoked` appended once its authority has been taken away — for example `"directory provisioned/consent-only root read,list,derive"`. A grant the application remembered, and every grant derived

from it, carries `remembered` before `revoked`. `ORIGIN` is `trusted-selection`, `drop`, `provisioned`, or `automatic`; `ENFORCEMENT` is `brokered` when the broker returned authority the process could not obtain for itself and `consent-only` when a person chose in a trusted surface but nothing outside the process enforces the scope. `LINEAGE` is `root` or `derived`. No step means the application holds no authority at all. Identifiers are deliberately not part of a description: they are allocation order, which is a fact about the run rather than about the authority.

- `(expect-canvas-size LOCATOR WIDTH HEIGHT)`: Require the size last reported to one canvas that handles `on_size` to be `WIDTH` by `HEIGHT` logical pixels. A canvas without the handler, or one that has heard no size, fails. The window runner reports the size GPUI laid the canvas out at; the semantic runner the size it derives from the window the application asked for. Both answer from the size the mounted graph recorded.
- `(expect-canvas-primitives LOCATOR N)`: Require one canvas owner to report exactly `N` retained primitives. The expected and observed values are stored as deterministic scale evidence.
- `(expect-clipboard-counters LIVE ACQUIRE READ WRITE)`: Require the owner-recorded live handle count and attempted clipboard operations. A field may be _ when concurrent production work makes that particular expectation nondeterministic. Its observed value is still recorded; the expected evidence column is absent rather than zero.
- `(expect-component-work :rendered N :compared N :skipped N :mounted N :retired N :registry-visits N :ancestor-invalidations N :projection-gets N :projection-sets N :keyed-order-visits N :keyed-snapshot-items N)`: Require exact component-runtime work in the most recently committed action turn, including its nested rendering and lowering. Supply one or more named counts in any order; omitted counts are unconstrained. Both runners read the same production observation, including when capture is disabled. A missing observation fails as unavailable. Assertions do not start a new action turn. Projection counts record configured getter and setter calls, including calls that return `Removed`; they exclude arbitrary application reads and background worker execution. They measure adapter work separately from boundary renders.
- `(expect-count LOCATOR N)`: Require exactly `N` matches and record expected and observed counts.
- `(expect-rows LIST :count N :first N :mounted N)`: Require one virtual list to hold `:count` rows, to have mounted its rows from index `:first`, and to have mounted `:mounted` of them. Supply one or more of the three. A list given its items mounts all of them from index zero; a list whose rows are produced on demand holds a count and mounts only a window of it, so this is how a specification proves a length the graph does not contain. `:count` is recorded as expected and observed deterministic scale evidence.
- `(expect-device-connections N)`: Require exactly `N` host-owned device connection handles.
- `(expect-device-transactions N)`: Require exactly `N` completed host-owned device transactions.
- `(expect-document-counters PICKS CHOSEN CANCELED REFUSED READS LIVE)`: Require the document owner's totals for single-file grants: `pick_file!` calls, chosen files, cancellations, refused picks of any reason, whole-file reads, and then the document handles the application still holds.

- `(expect-drop-counters DROPS GRANTED REFUSED)`: Require the document owner's totals for drops: drops delivered to a drop target's route, files they granted, and dropped items they refused. No name or path becomes evidence.
- `(expect-recent-counters REMEMBERED REOPENED REFUSED FORGOTTEN ENTRIES)`: Require the recent list's totals: grants remembered, entries reopened, reopens and remembers refused, and entries forgotten, then the entries it holds now. No name or path becomes evidence.
- `(expect-theme light)` **and** `(expect-theme dark)`: The scheme adaptive colours now resolve to: the system's, or the one the application chose with `Gui.Appearance.prefer!`. An `expect-background` on an adaptive colour compares the value it resolves to.
- `(expect-watch-counters STARTED CHANGES CANCELED REVOKED LIVE)`: Require the watch owner's totals: watches started, changes delivered, watches cancelled, and watches ended because their grant was withdrawn, then the watches the application still holds. `_` leaves one unconstrained, for a count that follows another process's writing rather than the application.
- `(expect-task-counters ISSUED COMPLETED DELIVERED SUPERSEDED CANCELLED INTERRUPTED)`: Require the task owner's totals: tasks issued, completions their workers published, completions delivered to the application, tasks superseded by a newer task with the same key, tasks cancelled explicitly or with their component, and capability waits interrupted by a task ending. `_` leaves one unconstrained, for a count that depends on whether a worker reached its wait before it was superseded. A benchmark's `:scale` is verified by a superseded count equal to it. Semantic runs only.
- `(await-task-waits N)`: Wait until exactly `N` running tasks hold a worker that only their end, or their capability, can move on: blocked in a wait that ending them would interrupt (a timer tick, a watch, or a SQLite statement), or ended and still unwinding. It delivers nothing and records no cycle. With a positive count it lets a later cancellation find a task that is certainly blocked; with zero it waits until every ended task has released what it held. Semantic runs only. `(expect-file-picks N)`, `(expect-file-lists N)`, `(expect-file-opens N)`, and
- `(expect-file-reads N)`: Require exact cumulative counts recorded by the Files owner.
- `(expect-focused LOCATOR)`: Require semantic focus to be on exactly one matching enabled control. This proves the host's dialog focus policy, not a compositor-delivered focus event.
- `(expect-http-counters LIVE ACQUIRE SEND DENIED)`: Require the owner-recorded live client count, operation attempts, and authority denials.
- `(expect-hash-counters HASHED REFUSED BYTES)`: Require the file hash owner's three totals, in that order: files hashed, hashes refused, and bytes hashed. No file name, path, or digest becomes evidence.
- `(expect-image-bytes LOCATOR N)`: Require an image node to own exactly `N` encoded bytes, and record expected and observed deterministic counts without persisting image content.
- `(expect-keyboard-counters OFFERED MATCHED COMPARED FOCUSED)`: Require the mounted graph's keyboard work since the run began: keystrokes offered to the application's shortcuts, keystrokes a shortcut answered, declared shortcuts compared against a keystroke, and focus requests that moved focus. Resolving a keystroke visits the focused control's ancestors and the elements that declare window shortcuts, never the

rest of the graph, so `COMPARED` is the cost a scaling case bounds. Both runners answer from the same graph.

- `(expect-not-visible LOCATOR)`: Require no matches.
- `(expect-patch :kind KIND :staged N :removed N)`: Check the host patch produced by the preceding click. `KIND` is `mount`, `replace`, or `no_change`.
- `(expect-popover-counters OPENED CLOSED DISMISSED)`: Require the mounted graph's popover transitions since the run began: surfaces that began presenting, surfaces closed because pointer and focus both left, and surfaces closed by Escape. Both runners answer from the same graph.
- `(expect-processes N)`: Require exactly `N` live host-owned PTY child processes.
- `(expect-sqlite-counters LIVE OPEN QUERY)`: Require the owner-recorded live connection count and attempted SQLite operations. `QUERY` counts every statement the host was asked to run, whether whole, with parameters, or as one page.
- `(expect-subscriptions N)`: Require exactly `N` active host timer handles. This assertion reads the timer registry that owns cancellation state.
- `(expect-system-samplers N)`: Require exactly `N` active host-owned system samplers.
- `(expect-system-samples N)`: Require exactly `N` completed host-owned system samples.
- `(expect-tcp-counters LIVE CONNECT READ WRITE CLOSE)`: Require the complete owner-recorded TCP resource and attempted-operation vector.
- `(expect-tcp-streams N)`: Require exactly `N` live host TCP stream handles.
- `(expect-selected LOCATOR)` **and** `(expect-not-selected LOCATOR)`: Require exactly one matching tab, and that it is, or is not, its strip's selected tab.
- `(expect-value LOCATOR "value")`: Require an exact controlled text-input or textarea value, the exact line a canvas text primitive sets when the locator names a canvas item, or a divider's value, its size in logical pixels or `collapsed`, when the locator names a separator. The value is compared in memory and is not copied into captures or diagnostics.
- `(expect-value-bytes LOCATOR N)`: Require the text-input or textarea value to contain exactly `N` UTF-8 bytes, and record the expected and observed deterministic counts.
- `(expect-visible LOCATOR)`: Require at least one match.

Each counter assertion reads the production owner of the resource or component work, and records its expected and observed values as ordinary deterministic step evidence.

Prefer semantic results—visible text, counts, and order—for application behaviour. Use `expect-patch` when the amount or absence of host work is itself part of the contract, such as proving that an idempotent press returns `no_change` and stages no nodes.

What the semantic runner proves

The semantic runner queries the same mounted graph the native host uses. It does not construct a separate test tree. Because it opens no GPUI window, it suits deterministic semantic checks; pixels, layout, and presentation are outside its evidence. Those claims belong to the window runner.

For a virtual list, text and control locators inspect all logically mounted items, including rows outside an interactive viewport. A headless `expect-visible` therefore proves logical

presence, not viewport intersection. Only GPUI records native visible-range and materialisation evidence.

A list of rows produced on demand mounts a window of its rows chosen by the viewport. The semantic runner takes the viewport to be as tall as the window the application asked for, so it mounts the rows such a viewport shows and one viewport below them, and an application's scroll request moves that viewport as it would move the real one. Visible-range events come only from GPUI frames.

`focus` and `press-key` model the production controls' accepted activation-key matrix and invoke their normal Roc event route. In the semantic runner they inject no native GPUI key event and verify neither compositor delivery, tab traversal, focus painting, nor operating-system accessibility.

Window specifications

A window specification is an ordinary `(test ...)` file that uses window-only steps. It lives beside every other specification for its application, because the steps a file contains are what decide which runner takes it. Nothing about its location or its name carries that meaning.

Your application runs one itself:

```
./my-app --host-run-window-spec specs/on-screen.scm \
    --host-window-report=.test-out/report.json \
    --host-window-shot-dir=.test-out
```

The process opens the production GPUI window, drives it through the same render path and event route an interactive session uses, and writes a JSON report. In this repository `scripts/run_specs.py` builds the host, builds each example, and routes each case to the runner its steps require:

```
python3 scripts/run_specs.py 'examples/counter/specs/*.scm'
python3 scripts/run_specs.py --only window
```

`--host-describe-specs PATH...` prints one JSON line per file giving the runner it needs and the capability flags its grants become, which is how the driver routes and wires a case without keeping a second copy of the vocabulary.

One vocabulary, split by capability

There is one specification language. Every step declares where it can be honest, and each runner refuses the other's steps by name rather than skipping them silently:

- `semantic-only`: `mark-metrics`, `expect-patch`, and `benchmark` clauses, which depend on the semantic runner's per-iteration mount, and the `(file recording)` grant, since only the

semantic runner records a capture. Also every step the window runner does not implement: `replace-text`, `submit`, `await-task-waits`, `expect-task-counters`, and the repository fixture steps the Contributing chapter describes.

- `window-only`: `settle`, `screenshot`, `type`, `resize`, `scroll`, `drag-files`, `expect-on-screen`, `expect-rendered-count`, and `expect-bounds`. None of these mean anything without a presented frame: without layout there is no fold for content to be below, and no size a window could be resized from.
- `both`: `click`, `drag`, `drop`, `pointer-move`, `pointer-leave`, `wheel`, `focus`, `press-key`, `key`, `await-task`, `await-count`, `clipboard-text`, `system-theme`, `expect-theme`, `await-ticks`, `expect-visible`, `expect-focused`, `expect-not-visible`, `expect-count`, `expect-component-work`, `expect-canvas-size`, and the value and ordering assertions — `expect-value`, `expect-selected`, `expect-not-selected`, `expect-value-bytes`, `expect-image-bytes`, `expect-before`, `expect-rows`, and `expect-canvas-primitives`. The fixture clipboard is one process-wide store, so changing the granted source and waiting for the application's own timer to observe it mean the same thing under either runner; without them a windowed case could not put a single item into a clipboard-driven application, and so could not photograph one. The value and ordering assertions read the mounted graph, which both runners hold, and both call the same code to answer them, so one window case can assert a semantic truth and photograph it in the same run.

Also `revoke-file-grants`, which withdraws every chosen folder and file, `replace-file` and `remove-file`, which change what a copied folder holds, and the live-resource and capability counter assertions. The host owns one files registry, one clipboard, one audio device table; both runners read them through the same shared implementation, and revocation acts on that one registry. A windowed case can therefore photograph a granted-authority readout and assert the counters that produced it in the same run. File operation counts are differences from the start of the run's lifecycle, of which the window runner has exactly one.

A step is classified by what a runner actually does, so a specification is refused before the window opens rather than part-way through a run. `expect-visible` asserts the same claim on either runner: presence in the mounted graph. The stronger pixel claim is `expect-on-screen`, so one word never means two different things.

Window-only steps

Input steps first, then waiting, then assertions in alphabetical order.

- `(drag-files LOCATOR "PATH"...)`: Drag files over a drop target and hold them there without dropping, through the same GPUI file-drop events `drop` begins with, so a screenshot shows how the target answers files it would take. A following `drop` of the same files lets go of them.
- `(type "TEXT")`: Send each character as a real keystroke into the focused element, through GPUI's own keystroke dispatch. The semantic `replace-text` sets a value directly and exercises none of the editing path.
- `(resize W H)`: Resize the production window to `W` by `H` logical pixels, each 64 to 8192. A layout is only proved at the size it was tested at, and `main.roc` asks for one size, so this

is how a specification claims a narrow or a wide window rather than assuming one. Follow it with `(settle)` before asserting, as with any other input step.

- `(scroll REGION :by N)`: Move a scroll region or a virtual list by `N` logical pixels. Positive moves towards the end of the content, which is the direction a wheel-down gesture moves it; negative moves back. The request is clamped against the real content size, so an overlarge `N` settles at the end rather than past it, which is how a specification returns a list to its start without knowing how tall it is.
- `(scroll REGION :to LOCATOR)`: Move `REGION` only as far as it must to bring `LOCATOR` inside it. A target already in view does not move the region at all. The target must be inside the region, and a target taller than the region is aligned to its leading edge, since no offset shows all of it.
  For a scroll region the distance is real geometry: every child is laid out, below the fold included, so the travel is measured rather than estimated. A virtual list has no such rectangle — a row below the fold has no element at all — so the list is positioned by that row's index among the entries the application mounted, and the row materialises as a result. This is what brings an off-screen target on screen, and it is the only handle an unrendered row offers. A list of rows produced on demand counts that index from its first mounted row; a row it has not mounted has no locator, and is reached with `:by` or by the application's own scroll request.
- `(settle)` / `(settle :frames N :timeout-ms N)`: Wait for `N` consecutive presented frames during which no host task started or finished and no list of rows produced on demand was rendered again for its viewport. Defaults are `:frames 2 :timeout-ms 2000`. A timeout fails the step; it never proceeds quietly, because an assertion against a half-settled frame is not evidence.
- `(expect-bounds LOCATOR :min-width N :max-width N :min-height N :max-height N)`: The laid-out size falls within the given bounds. At least one bound is required. Deliberately minimum and maximum only, never exact: exact pixel geometry is font- and DPI-dependent and would rot the suite.
- `(expect-on-screen LOCATOR)`: The node was painted in the last completed frame and its rectangle survives clipping by the viewport and by every scrolling ancestor. This is the claim the semantic runner cannot make.
- `(expect-rendered-count LOCATOR N)`: Exactly `N` matching nodes took part in the last frame. For a virtual list this is materialisation evidence; `expect-count` counts logically mounted nodes instead.

Screenshots

- `(screenshot NAME)`: Photograph the window's content area.
- `(screenshot NAME :region LOCATOR)`: Crop to a located element's visible bounds. These are the bounds `expect-on-screen` uses, so a shot frames what was asserted, and an element half below the fold of a scroll region is photographed where it has pixels rather than where it was laid out.
  A canvas-item locator works here too. A primitive has no element and no recorded bounds of its own, so its rectangle is the canvas's painted surface offset by the coordinates the owner drew at — the mapping `canvas_target` inverts to decide what a press landed on — clipped to that surface. A line is expanded by half its stroke, since its

coordinates describe its centre. Give it `:pad`: a marker a few points across is unreadable cropped to itself.

- `(expect-ink LOCATOR)`: Photograph a located element, keep the photograph beside the shots, and fail unless it drew ink: at least eight pixels that differ from the region's commonest colour by more than 48 in some channel. Laid-out text that draws no glyphs passes every semantic assertion; this is the check that does not. A capture the platform cannot take is reported as unavailable, as a screenshot's is, never as a pass.
- `(screenshot NAME :region (rect X Y WIDTH HEIGHT))`: Crop to an explicit content-relative rectangle.
- `:pad N`: Expand the region by `N` points on every side, then clamp to the content area. A control cropped to its exact bounds is usually unreadable out of context.

`NAME` is required and becomes the file stem, so it is limited to lowercase letters, digits, and hyphens. Files are written as `NN-NAME.png`, numbered by step, and indexed in the report's `screenshots` array. A `:region` locator that does not match exactly one node fails the step, as `click` does.

The host knows where each node was laid out, so it computes the crop itself, relative to the content area. On Linux the renderer reads back the next frame it presents and the host crops those pixels: the image is the frame the application flew, whatever the compositor or desktop, with no capture tool or permission involved and nothing drawn over the window in the picture. On Windows the window renders itself into a bitmap with `PrintWindow`. On macOS the host adds its window origin and asks `screencapture` for that screen rectangle, so cropping is part of the request.

Capture is honest about not happening. A missing tool, a denied Screen Recording permission, or a region with no area on screen is reported as `unavailable` with a reason and a hint, never as a pass and never as an empty file. By default an unavailable screenshot fails the run; `--host-window-allow-missing-shots` downgrades it to a report entry, which is how continuous integration runs these specifications on machines that cannot grant screen recording. Every behavioural assertion still gates there.

Screenshots taken before a failing step are kept. The image of the state an assertion stopped at is usually the most useful evidence in the run.

What input is real

Keyboard steps send real keystrokes through the window's production keymap. `scroll` writes the offset cell of the scroll handle the production element tracks — the same cell GPUI's own wheel handler writes, and the same one that keeps a list where the person left it across a rerender. What it does not synthesise is a wheel event, so hit testing, momentum, and rubber-banding are not exercised; those belong to the pointer seam this host does not have. Pointer steps are simulated at the production handler, but are refused unless the control was laid out, is unclipped, is enabled, and is not behind a modal dialog, so a `click` a real user could not have performed fails rather than passing quietly. `drag` presses, moves, and releases the window's pointer through GPUI's own event dispatch, so the press is hit-tested and the production listeners follow the move. `hover-enter` and `hover-exit` dispatch mouse-move events into the live GPUI window, exercising hit testing, production views, and the normal event route. They do not move the operating-system

pointer. The specification owns its simulated pointer across window activation and resizing; the physical cursor cannot change its hover target. GPUI tests also verify replacement continuity and disabled or stale routes. The Testing your application chapter explains why, and what each guarantee is worth.

What the probe cannot see

Bounds come from the production render path, through the same `canvas` prepaint closure the host already uses for canvas hit testing. Recording is off unless a window specification asked for it, gated by a runtime check rather than a compile-time one, so the shipped binary stays the tested binary.

- A node that never renders has no bounds. That is the intended signal rather than a gap: an unmaterialised virtual-list row is genuinely absent from the frame.
- Only a canvas node's own rectangle is recorded, never its primitives. A canvas-item locator has no bounds of its own to assert against, so `expect-bounds` and `expect-on-screen` do not take one; a screenshot region does, because a primitive's rectangle is derivable from its canvas's painted surface and the coordinates the owner drew at.
- Bounds are recorded for nodes scrolled out of view, because they are laid out and merely clipped. `expect-on-screen` intersects against scrolling ancestors for this reason; asking only whether bounds exist would call a clipped element visible.

Bounds persist until a node leaves the mounted graph. GPUI caches views, so a control whose state did not change is not re-rendered and records nothing that frame; treating that as "not laid out" would report a plainly visible control as missing. `expect-rendered-count` therefore counts nodes that have been laid out while mounted, which is the materialisation evidence a virtual list needs.

Because bounds describe the tree as it stood when a frame drew it, they answer nothing about a change no frame has drawn yet. The runtime counts a generation for every applied patch and each frame stamps the generation it drew, so a painted step waits for the window to catch up before it reads anything, and a graph the window has not drawn is reported as exactly that rather than as an element that is missing. Waiting is not left to the specification: an input step returns once the window has drawn what it did, and `await-count` ends on a frame carrying the nodes it waited for. Assertions that read the mounted graph, such as `expect-visible` and `expect-count`, need no frame and take none.

Selecting and repeating cases

Patterns match repository-relative specification paths. Multiple patterns may be supplied. Use `--skip-host-build` only to reuse an already staged local host while iterating:

```
python3 scripts/run_specs.py \
  'benchmarks/rows/specs/create-*.scm' \
  'benchmarks/rows/specs/append-*.scm' \
  --skip-host-build
```

The default is one case at a time. `--jobs N` runs cases concurrently, which is useful for semantic validation but marks timing-environment evidence partial. Use serial runs for performance comparisons. `--aa` repeats each passing case with the same executable to show run-to-run noise.

The runner prints a UTC start, UTC finish, and elapsed wall time for every case, including timeouts and process failures. Timing is diagnostic evidence only; specification assertions and deterministic counters decide correctness. CI runs ordinary semantics, serial scaling cases, native smoke checks, and real-window specifications as independent jobs. On failure it retains bounded JSON status, partial `.rgstats` evidence, and a sanitized textual Windows debugger report when one is available. Raw process memory is never retained because it can contain application data and machine identity.

Measuring scaling

A scaling benchmark is a realistic Roc application driven by the same semantic steps as a behaviour specification. Scale comes from application data: more rows, more text, a deeper tree, or a change farther from a render boundary. A benchmark never receives a special element or alternate renderer unavailable to an ordinary application.

Every scaling investigation answers one question: "Is our integration invalidating, rebuilding, or handing GPUI substantially more work than the application change requires?" A scale ladder separates application rendering and lowering, the mounted-graph transaction, native entity work, and GPUI frame work. Semantic results and deterministic counters recorded by those owners are the gates; timings diagnose the remaining work but do not define correctness.

The specification runner builds Roc applications with the native development backend by default. Compare captures from the same backend and executable before interpreting timing differences; the Rust host uses its independent release build profile.

Benchmark policy

Add a `benchmark` clause before `steps`:

```
(test "Rows: create 10,000"
  (benchmark :warmups 2 :samples 7 :iterations 1
             :scale 10000 :initial-size 0 :change-size 10000)
  (steps
    (mark-metrics)
    (click (role button :name "Create 10,000 rows"))
    (expect-patch :kind replace :staged 40015 :removed 15)
    (expect-count (text-prefix "Row ") 10000)
    (expect-visible (text "Rows: 10000"))))
```

- `:warmups`: Complete runs discarded from timing summaries. Warmups exercise the same setup, operation, and assertions as samples.
- `:samples`: Runs retained for measurement.
- `:iterations`: Measured runs per sample policy. Each lifecycle starts from a fresh application state.
- `:initial-size`: Items present before the marked operation.
- `:change-size`: Items the operation is intended to affect.
- `:scale`: Resident workload proved by an `expect-count` with the same value, or, for a list of rows produced on demand, by an `expect-rows` whose `:count` has that value.

Every benchmark has exactly one `mark-metrics`. Put state preparation before the mark and the user operation plus its assertions after it. The runner attributes cycles to setup or measured phases; locator resolution and assertion scans are harness work, not part of the operation duration.

The declared scale is a label until the application proves it. Requiring a matching `expect-count` prevents a fast but broken “10,000 row” case from being reported when it created fewer rows.

Read a scale ladder

One large case cannot distinguish fixed cost from growth. Measure the same operation at three or more sizes, then compare columns in the scaling summary:

```
python3 scripts/run_specs.py 'benchmarks/rows/specs/create-*.scm'
python3 scripts/summarize_suite.py .test-out/specs/TIMESTAMP
```

Look across sizes rather than treating one duration as a score:

- the Roc work view separates routing, application update, application render, and platform lowering time and allocator activity;
- validation and graph-application growth point to host graph work;
- staged and removed node counts explain how much structural work was asked for;
- allocation totals and process resources expose memory cost across samples;
- `NULL` means that component did not provide that measurement. It is not zero.

Timing is diagnostic evidence, not a pass/fail threshold. Semantic assertions, patch expectations, scale verification, and capture health are deterministic gates.

Verify depth with a process-stack budget

Tree-shape specifications exercise ordinary construction, lifted structural containers, translation ownership, current-state delegation, and replacement by a shallow tree. Run the same production executable and specifications with an explicit process-stack budget as well as the ordinary environment. Preserve the selected compiler and build mode in the comparison evidence.

The structural regression reaches 10,000 lifted containers and exercises two leaf updates before teardown. The translation regressions build and retire 1,000 owners, including alternating state types, current-state task resolution, removal before completion, and veto at an intermediate boundary. Run each with a 1 MB stack to verify its headless construction, adaptation, lowering, and release path. Record the compiler backend with these results. A successful case establishes that particular path; it does not measure arbitrary application recursion or GPUI layout and paint depth. Record failures and their owning call paths in the backlog rather than widening the process stack to make the result pass.

Bound recorder detail overhead

Use the same realistic scaling case and isolated job count for both recorder policies:

```
python3 scripts/run_specs.py 'benchmarks/rows/specs/select-10k.scm' --jobs 1 --
detail summary --output .test-out/observer-summary
.test-out/observer-summary/bin/rows --host-run-spec benchmarks/rows/specs/
select-10k.scm --host-stats-output=.test-out/observer-full.rgstats --host-stats-
job-count=1 --host-stats-detail=full
```

```
python3 scripts/analyze_stats.py .test-out/observer-summary/benchmarks/rows/specs/
select-10k.rgstats
python3 scripts/analyze_stats.py .test-out/observer-full.rgstats
```

Invoking one executable for both policies keeps executable identity fixed. The reference run recorded seven samples per policy. Summary recorded 18 cycles and a 3089.204 ms median; full recorded nine additional setup cycles and a 3103.423 ms median. The full-policy median increase was 0.46%, and its complete sample range remained inside the summary range. Treat 1% as the observed detail-policy overhead bound for this case. Timing remains report-only, and a performance decision reruns the bound on its comparison machine.

Workload families

The suite varies operations and shapes that applications naturally produce:

- **Scroll**: Populate a named scroll region with 100, 1,000, and 10,000 ordinary text rows. This measures eager graph construction and host materialisation; it does not substitute a benchmark-only list or renderer.
- **Virtual list**: Populate the production fixed-row virtual list with 100, 1,000, and 10,000 stable-keyed text rows. Semantic scale proves the complete logical graph; interactive captures separately report viewport-owned materialisation and recycling without treating offscreen rows as visible native entities. The same list with its rows produced on demand holds 10,000, 100,000, and 1,000,000 rows: `expect-rows` proves the count, and the mounted rows, graph work, and render time stay those of one window at every size.
- **Settings search**: Populate the text-input benchmark's virtual catalogue with 100, 1,000, and 10,000 categorised settings, then edit its controlled search field to retain one category. Exact semantic counts prove both the initial catalogue scale and the filtered result while the measured turn uses the ordinary text-change route.
- **Query supersession**: Run the Database Browser's statement 11, 101, and 1,001 times in a row over a 10,000-row result already on screen. Each run supersedes the one in flight, so 10, 100, and 1,000 queries are interrupted where they run or never start, and only the last result is delivered. The task owner's superseded count proves the scale, and its delivered count proves that no superseded result reached the application.
- **Rows**: Create, append, sparse update, select, idempotent reselect, delete, clear, swap, and repeated selection at 100, 1,000, and 10,000 rows, with a 100,000-row sparse-update point for confirming large replace-with-removal scaling. Row actions are ordinary action-button nodes, so this ladder also exercises styled button lowering, native materialisation, dispatch, and replacement at realistic application sizes.
- **Row boundaries**: Update the first, middle, and last item when each row owns its own render boundary, and exercise selection, deletion, swaps, and sparse updates whose invariants are owned by the collection. Persistent row `Index` storage keyed by stable IDs is separate from display order, so resolving or replacing a row follows a bounded radix path without scanning or copying the collection. Local edits render one row and

replace four nodes: its boundary marker, row, text, and button. Collection-owned changes enumerate the boundary frontier and, when memoization is enabled, retain rows whose complete input is equivalent. Dynamic SHA-256 keys are stored when entries are created. The compact-view toggle removes per-row management controls and puts keyed components directly beside each other under the list container; its local-edit cases exercise direct sibling identity without per-row wrapper scopes at 100, 1,000, and 10,000 rows in both memo modes. Component, patch, and validation counts establish local work independently of unrelated sibling count. Captured timings and allocation spans report only the work measured by their owners; they do not establish copying, retained memory, or unmeasured native work.

- **Incident queue / keyed collections**: Exercise local item updates, front insertion, middle removal, far movement, identity replacement, and nested child creation and destruction in a live queue at 100, 1,000, and 10,000 incidents. The application uses `KeyedSeq` and `Gui.keyed_col`; it does not substitute a benchmark-only collection or renderer. Component and keyed-order counters prove the Roc work requested by one edit, graph counters prove the canonical mounted transaction, and native counters prove the entities handed to GPUI. Exact journal edits materialise no full snapshot; the deliberate stale-journal case records the complete fallback it consumes. These results establish proportional integration work, not constant GPUI layout or paint: an operation that changes row geometry can still require GPUI to lay out the affected scene.
- **Hover trail**: Move across a 125 by 80 grid of native buttons, lighting each cell and leaving a delayed reset trail. Every cell is an unmemoized keyed translation with persistent indexed state. Entry, exit, and completion ladders at 100, 1,000, and 10,000 cells assert one boundary rendered, no comparison, and two mounted nodes replaced. Reentry tokens and fresh grid keys protect against stale completions. A 32-cell burst exercises queued work, and a window case checks both grid corners and native GPUI mouse-move delivery. Native GPUI mouse-movement tests exercise the same production views and event route. Callback durations exclude waiting for reset tasks; cycle durations include the wait. Neither headless captures nor injected window events measure physical pointer sampling or compositor presentation latency.
- **Text rows**: Vary short, long, and metadata-rich text across row counts.
- **Textarea editing**: Edit the production multiline text control at 1,000, 10,000, and 100,000 initial bytes through an ordinary append action. The same application selects memoized or unmemoized keyed translations; exact value lengths, component counts, and patch counts establish semantic equivalence. Root refresh also distinguishes retained input from unconditional rendering. Append derives its value from dispatch-time state. Same-length late edits change the final byte to exercise comparison beyond the early length check. Policy switching keeps the same key and mounted lifetime while resetting the cache. Allocation spans report work observed by their owners; they do not establish snapshot-retained bytes or identify copy-on-write as a cause. Serial A/A captures characterize timing noise without turning timing into a correctness gate.
- **Checkboxes**: Lower and replace 10,000 real styled controls through their ordinary check event route.
- **Tree shape**: Compare deep narrow trees and balanced trees so total node count is not the only scaling dimension.

Keep each family as a usable application with specifications beside it. Add an operation to an existing family when it uses the same state and interaction model; create a separate application when the data model or UI shape is materially different.

Compare two captures

Generate captures serially on the same machine and compare matching cases:

```
python3 scripts/analyze_stats.py BEFORE.rgstats --compare AFTER.rgstats
```

The comparison checks capture identity and workload compatibility before showing a ratio. Use `--aa` to establish the natural spread of an unchanged executable. Supply that validated capture to the scaling view to expose the A/A spread bound and whether the measured spread exceeds it:

```
python3 scripts/analyze_stats.py CAPTURE.rgstats --view scaling --aa-bound CAPTURE-
aa.rgstats
```

Without `--aa-bound`, `aa_bound_ns` and `spread_exceeds_aa_bound` are `NULL`: the capture contains no A/A evidence from which to infer them. Treat a change smaller than an available bound as noise.

Compare different scales only when both captures came from the same executable, application, target, machine, recorder detail, and isolated serial policy:

```
python3 scripts/analyze_stats.py SCALE-10K.rgstats --view scaling --scale-against
SCALE-100K.rgstats
```

The cross-scale report gives separate ratios for the Roc callback, native-tree validation, canonical graph application, complete sample-lifecycle allocation, and allocation attributed to measured Roc work spans. Lifecycle allocation includes setup outside the marked operation. Span allocation includes only the instrumented work attributed to measured callbacks. These scopes are not interchangeable and the report labels both. A component whose evidence status is not complete is `NULL` independently of the other components. Mechanically incompatible captures make every ratio unavailable. Ratios are diagnostic, report-only timing and resource evidence; semantic and capture-health results remain the correctness gates.

Understanding captures

Every specification run produces a versioned `.rgstats` SQLite database. The database keeps the semantic result, attributed operation records, resource counters, recorder health, and privacy-safe comparison identity together. This supports several focused reports without inventing a second measurement format.

Start with the summary

```
python3 scripts/analyze_stats.py CAPTURE.rgstats
```

The summary validates the schema and capture finalisation, then reports the case identity, run and cycle counts, and the distribution of measured operation duration. A capture with lost events, writer failure, output truncation, or an unclean shutdown is rejected as incomplete.

A capture's verdict follows from its own records. Until `metadata.final_state` is `complete` it has none: it is *withheld*, because the records a verdict rests on are decided at finalisation, and a capture still being recorded reads the same as one whose recorder stopped. A finalised capture is *untrusted* when `clean_shutdown` is not `1`, `recorder_health` reports omitted events, a writer failure, or a reached output limit, `recording_gaps` holds any row, or any measurement family is still `unfinalized`. It is *partial* when any family is `partial`. Otherwise it is *complete*: a `not_recorded` or `unavailable` family is a declared absence, not a loss. The Observatory example applies this rule to every capture it lists.

The analyser opens databases read-only. You can also inspect them with any SQLite client; application-facing reports should use the supplied views so the evidence-status rules remain visible.

Choose the question you are asking

```
python3 scripts/analyze_stats.py CAPTURE.rgstats --view semantic_health
python3 scripts/analyze_stats.py CAPTURE.rgstats --view roc_work
python3 scripts/analyze_stats.py CAPTURE.rgstats --view component_work
python3 scripts/analyze_stats.py CAPTURE.rgstats --view cycle_targets
python3 scripts/analyze_stats.py CAPTURE.rgstats --view host_gpui_work
python3 scripts/analyze_stats.py CAPTURE.rgstats --view gpui_frame_spans
python3 scripts/analyze_stats.py CAPTURE.rgstats --view native_work
python3 scripts/analyze_stats.py CAPTURE.rgstats --view gpui_frame_work
python3 scripts/analyze_stats.py CAPTURE.rgstats --view
virtual_list_materialization
python3 scripts/analyze_stats.py CAPTURE.rgstats --view timeline
python3 scripts/analyze_stats.py CAPTURE.rgstats --view process_resources
python3 scripts/analyze_stats.py CAPTURE.rgstats --view capture_health
python3 scripts/analyze_stats.py CAPTURE.rgstats --view scaling
python3 scripts/analyze_stats.py CAPTURE.rgstats --view spec_results
```

- `semantic_health`: Did runs and steps pass, and is their evidence complete?
- `roc_work`: How much callback time belonged to platform routing, application update, application render, component comparison, and platform lowering, and how much Roc allocation activity occurred inside each owned span?
- `component_work`: How many renders, comparisons, skips, component mounts and retirements, registry visits, ancestor invalidations, keyed order-tree visits, and keyed fallback snapshot items did the component runtime perform in marked sample turns?
- `cycle_targets`: Which element did each cycle reach, by node kind and structural identity, how many cycles reached it, and how long was the slowest?
- `host_gpui_work`: How much time did graph validation, graph application, and available GPUI materialisation consume, and how many native nodes were retained or visited during structural validation?
- `gpui_frame_spans`: For each drawn frame, how long did the host's own root element spend requesting layout, prepainting, and painting the application subtree, and what is the status of the frame stages the host does not own?
- `native_work`: How many native `NodeView` render calls and host-created `AnyView` elements occurred in completed GPUI frames, grouped by node kind?
- `virtual_list_materialization`: For interactive GPUI frames, how many logical items intersected each viewport and how many native entities were materialised, recycled, and retained? Each list records one row per drawn frame, after GPUI has drawn it; a patch that rebuilds part of a row, or retires rows with their list, records the entities it built or recycled in rows of its own. For a list of rows produced on demand, the intersecting items include rows not yet mounted, which are drawn as blank places and materialise no entity. The Roc turns that move such a list's window are ordinary cycles with the trigger `viewport`. Semantic-headless captures report this family as `not_recorded` because they have no viewport.
- `timeline`: How many drawn frames record the cycles they were first to draw, how many list passes record the frame or cycle that produced them, and what span of the capture's clock do cycles, frames, and passes cover?
- `process_resources`: How did process CPU time, peak RSS, and Roc allocation counters change across runs?
- `capture_health`: Did batching, queue pressure, omissions, limits, or writer errors weaken the capture?
- `scaling`: What workload sizes, deterministic work counts, and timings belong to each measured cycle?
- `spec_results`: Which individual steps passed, and what counts or diagnostics did they record?

Audio counter assertions retain their complete expected and observed vectors in `audio_counter_assertions`. The columns cover live output and track resources, then acquire, load, play, pause, seek, status, and stop operations. Each row is owned by the assertion step that sampled the production audio host.

Clipboard and SQLite assertions likewise retain complete vectors in `clipboard_counter_assertions` and `database_counter_assertions`. Clipboard rows record live handles and acquire, read, and write attempts. SQLite rows record live connections and open and query attempts. These are correctness counters sampled directly from the capability owners; unavailable evidence is absent.

HTTP assertions retain live clients plus acquire, send, and denied-operation counters in `http_counter_assertions`. Transport cancellation is not recorded because the HTTP capability does not expose it; stale reducer suppression does not fabricate cancellation evidence.

TCP assertions retain live streams and connect, read, write, and close attempts in `tcp_counter_assertions`. Endpoint, Redis key, and Redis value text are not capture fields.

Schema 18 records component work in `component_work_counts`, keyed by cycle and numeric kind. Kinds are `0 rendered`, `1 compared`, `2 skipped`, `3 mounted`, `4 retired`, `5 registry_visits`, `6 ancestor_invalidations`, `7 projection_gets`, `8 projection_sets`, `9 keyed_order_visits`, and `10 keyed_snapshot_items`. The Roc runtime reports each count at the operation that owns it. A turn publishes its observation when the mounted graph accepts its transaction. Rejected turns do not replace the last committed observation.

`rendered` includes the application root renderer as well as component renderers; `mounted` and `retired` count component instances, not that implicit root. `compared` counts comparisons against valid saved input, and `skipped` counts successful comparisons. `registry_visits` counts logical event-route lookups and emitted-component instance lookups, not every internal radix-index step, state projection, or ancestor walk. `ancestor_invalidations` counts strict ancestor invalidation operations, including ancestors whose snapshot was already unknown. `projection_gets` and `projection_sets` count actual calls to configured state adapters during the UI callback, including reads for existence and memo checks. They count invocation even when a fallible adapter returns `Removed`. They exclude arbitrary application reads and background worker work. `keyed_order_visits` counts persistent order-tree nodes visited while updating mounted keyed-owner metadata. `keyed_snapshot_items` counts items materialised only for initial mount or stale-revision recovery; an exact journal update records zero. Neither count stands for host graph work, native edits, layout, or paint. The iterative executor accumulates these counts and publishes them before the same turn commits. They are not inferred from component depth, rendered nodes, or elapsed time. These are owner-defined events, not estimates of total work.

Capture cycles use the difference between the owner's totals at their start and end. Each action turn in a gesture has its own cycle; the last-turn specification assertion describes its final committed turn.

`cycles.component_work_recorded` distinguishes a measured turn with no component work from a turn with no observation. Positive counts have rows; an absent kind means zero only for a cycle whose recorded flag is set. The component report withholds totals when observation coverage is incomplete. `component_work_assertions` records numeric kind, expected count, and observed count for each requested assertion field, without component keys or application state. An unavailable observed count is `NULL`.

The graph owner supplies `cycles.retained_nodes` and `cycles.validation_visits`. Retained nodes are mounted nodes preserved by explicit subtree retention. Validation visits count payload examinations in fresh-node shape validation, rooted fresh/frontier traversal, retirement discovery, and retained-root liveness checks. They do not stand for metadata index lookups, identity ancestry work, or all host operations.

Keyed graph transactions additionally report `cycles.keyed_graph_visits`, `cycles.keyed_original_reads`, and `cycles.keyed_first_touches`. The graph owner counts validated or retired fragment nodes; the keyed-order owner counts original link reads and distinct first-touched keys. Ordinary patches report zero because they perform no keyed graph transaction. `cycles.keyed_native_edits` and the three keyed item-entity columns record the native application owner's accepted edits and item entities created, retired, or moved. Native-work kind 16 records keyed-container renders and view-element construction. Layout, prepaint, and paint remain frame-wide owner spans; no keyed-container share is inferred from them.

Each focused view includes `evidence_status` and `evidence_reason`. Read those before interpreting numeric columns. Missing measurements are `NULL` rather than a fabricated zero.

What semantic runs measure

`scripts/run_specs.py` invokes the executable with `--host-run-spec`. The Roc application still renders through the platform, events still use its installed dispatcher, and patches still enter the host's canonical mounted graph. The runner does not open a GPUI window. Consequently its captures can attribute platform routing and lowering, application update and render, component comparisons, allocation activity within each span, the enclosing Roc callback, host validation, graph application, CPU, RSS, and deterministic graph work, while GPUI layout, paint, and presentation are unavailable.

`cycles.roc_callback_ns` measures time inside one Roc dispatch call. Each drag phase and task completion has its own cycle, including steps that await several completions. A canvas pointer phase records a `drag` cycle, a delivered canvas hover a `hover` cycle, and a canvas wheel scroll a `wheel` cycle, whether a specification or a person in the interactive window drives it. Each size a split's divider asks for while it is dragged records a `drag` cycle, and each of its keys a `key` cycle. Each size a canvas hears it was laid out at records a `resize` cycle, after the frame that laid it out. Waiting for workers and applying graph patches are outside the callback span. The enclosing cycle duration includes the wait.

Schema 18 records task completions delivered by the interactive window loop as well as those driven by specifications. Each delivered completion records its own callback, application spans, allocation activity, and graph work.

This distinction matters: headless graph application is useful host evidence, but it is not a proxy for rendering a frame.

What a drawn frame measures

Interactive task completions pass through the recorded callback path just as input events do. Their delivery cycles begin after the worker result arrives; neither callback duration nor delivery-cycle duration includes queue or worker waiting. Each accepted completion records its own `task` cycle, including completions that produce no graph change.

A capture taken with a GPUI window records one `gpui_frames` row per drawn frame. The application's element tree hangs below a single host-owned element, and each row holds

that element's own three stages for the whole subtree: `layout_request_ns` builds the subtree's styles and layout nodes, `prepaint_ns` resolves bounds, text, and hitboxes once layout is solved, and `paint_ns` produces the frame's scene. Each duration belongs to the call that performed it.

Two frame stages are performed by GPUI outside any host-owned element and are therefore reported as unavailable rather than attributed. Taffy solves layout once for the window from GPUI's own root element, between the host element's layout request and its prepaint. Presentation and frame completion are private to GPUI; a compositor frame callback would be the seam that owns them.

Place work on one clock

Every interval in a capture is measured on one process-relative monotonic clock, whose origin is fixed when recording starts. `runs.started_ns` and `runs.ended_ns`, and the `start_ns` and `end_ns` of `cycles`, `gpui_frames`, and `virtual_list_frames`, are nanoseconds on that clock. They order and place work; they carry no wall-clock time or machine identity.

Each interval is recorded by the component that performs the work. A cycle's interval is its owner's: from the instant it began handling the event or completion to the instant its patch was applied, the same reading its `duration_ns` is taken from. A frame's interval runs from the start of the host element's layout request to the end of its paint. It therefore encloses GPUI's layout solve, whose own duration remains unavailable; the interval places the frame without attributing that time. A list pass's interval runs from the first range GPUI asked of the list in a frame to the moment its entities were settled after that frame, or, for a pass performed while a patch was applied to native views, across that application.

Causal links are recorded by the owner that knows them, never inferred from the clock. Two intervals that touch are not thereby related.

Table or column	Link
`gpui_frame_cycles(frame_id, cycle_id)`	The recorded cycles whose patches reached native views after the previous frame began drawing and before this one did: the changes this frame is the first to draw. An event loop may coalesce several cycles into one frame, which then has several rows. A frame that follows no new cycle, such as one drawn for hover or a list's own scrolling, has none. A cycle whose patch changed nothing is not linked, because it gave a frame nothing to draw. A frame abandoned before paint hands its cycles to the frame that does draw them.
`virtual_list_frames.origin`	`paint` for a pass settled after GPUI drew the list, `patch` for a pass performed while a patch was applied to native views.

Table or column	Link
`virtual_list_frames.frame_id`	For a `paint` pass, the drawn frame that asked for the list's rows. `NULL` when the frame that asked is not the frame the host recorded as painted.
`virtual_list_frames.cycle_id`	For a `patch` pass, the recorded cycle whose patch performed it. `NULL` for a patch the host applied outside any recorded cycle.

Two measurement families describe this linkage. `frame_cycle_linkage` is complete when every drawn frame carries the cycles its owner handed it; a frame with no row is then a recorded "no new cycle", not a missing one. `virtual_list_linkage` is complete when every `paint` pass was settled against the frame that drew it. Both are `unavailable` when no frame was drawn or no list entered a viewport, and `partial` when the recorder omitted events, since an omitted cycle or frame has no row to link. Semantic-headless captures report both as `not_recorded`: they draw no frame and have no viewport, although their cycles still carry intervals.

Count native rendering work

Schema 18 stores native work in `gpui_native_work`, keyed by `frame_id`, numeric `metric`, and numeric `kind`. Each `frame_id` references `gpui_frames.id`. Metric `0`, `node_view_renders`, counts actual calls to the production `NodeView::render` implementation. Metric `1`, `view_elements_created`, counts the host helper constructing an `AnyView` element to hand to GPUI, including elements for retained native entities.

Native counters are always active, including when capture recording is disabled. Production-window tests can therefore observe the same operations without enabling the recorder. The host snapshots these counters at the start of `Runtime::render` and, when recording is enabled, publishes their deltas when the root element finishes painting that frame. The `native_work` report gives totals, maxima, and means per completed frame for each metric and node kind. Its observation family is `gpui_native_work`.

The `gpui_frame_work` report separates fresh frame construction from cached subtree replay. GPUI records counts at the operations that own them: prepaint and paint cache replay, hitbox and listener insertion, scene insertion and replay, element-state access and finish-time movement, and cached view-range rebasing. Counts describe operations, not bytes or elapsed time. A missing completed frame is unavailable rather than zero.

Kind	Node
0	Canvas
1	Button
2	Checkbox
3	Textarea

Kind	Node
4	Image
5	Column
6	Dialog
7	Panel
8	Row
9	Scroll
10	Virtual item
11	Virtual list
12	Text input
13	Text
14	Styled text
15	Boundary
16	Keyed container
17	Popover
18	Split divider
19	Drop target

Schema 20 adds kind 17. A popover's view renders when its surface opens or closes as well as when the popover itself changes; its anchor and content are counted under their own kinds. The region that carries hover handlers, shortcuts, or a focus request for an element is a popover with no content, and is counted here too.

Schema 24 adds kind 18. A split's divider renders when the size it carries changes; the panes beside it are columns, counted under kind 5.

Schema 25 adds kind 19, and a cycle may target a `drop_target`. A drop target renders as a column does; a drop on it is one cycle with the trigger `drop`.

Rows contain positive counts. A missing metric/kind row means measured zero only within a recorded frame with complete native-work evidence. A capture with no completed frames reports unavailable counts. A semantic-headless capture reports this family as `not_recorded`; it has no native render path. Incomplete observations withhold counts rather than publishing partial totals as complete evidence.

Roc component renders, native view renders, and native element construction measure distinct operations. A local Roc update can perform one component render while a subsequent frame renders many native views. Native counts identify this additional work without estimating it from Roc counters or elapsed time. They do not identify GPUI cache hits, layout-solver work, presentation, retained memory, or GPU work. Counts are frame-

scoped. A frame's recorded cycles in `gpui_frame_cycles` may be several when the event loop combines updates, and a frame's work is never divided among them.

Native subtree reuse follows layout ownership. Non-root rows, columns, and panels whose width and height are fixed by matching pixel size, minimum, and maximum values, with growth disabled, have outer geometry independent of child measurement. Baseline-aligned subtrees retain ordinary layout. Retained off-path containers reuse their native subtree; a changed descendant invalidates its ancestor path. Intrinsic and flexible containers preserve their content-driven layout behavior.

The hover grid declares its existing five-pixel cells and one-pixel gaps as exact row and grid dimensions. A local cell update reaches its button, translation boundary, row, grid column, and outer application column. Rebuilding those containers can construct wrappers for their immediate children. Compare both native metrics across 100, 1,000, and 10,000 cells: a bounded set of rendered views does not imply the same bound on constructed wrappers. The row and column fanout is part of the expected work.

The nested hover-grid companion uses the same leaf counts and cell geometry as the flat grid. Recursive quadrant translations project persistent child indices with at most four entries. Local hover updates render the leaf; delegated clicks travel to the depth-one quadrant, which accepts the candidate and renders its subtree. Reordering sibling quadrants within their native row preserves their keyed lifetimes; removing a quadrant retires its descendants and their task deliveries. Compare owner callback spans as well as native counts across both layouts. Ancestor-invalidations count the affected ownership path; they do not count composed getter calls or establish linear dispatch cost.

A native-work assertion belongs to a controlled interaction and frame window. Its expected counts follow the affected patch nodes and ancestor containers in that case. Input, focus, and layout invalidation can cause additional native work in other interactions; these assertions are specific evidence rather than an unconditional limit on rendering for every action.

Record an interactive application

Recording is disabled by default. Enable it on an ordinary application with:

```
./counter --host-stats-record
```

Choose a destination explicitly when captures are part of an experiment:

```
./counter --host-stats-output=counter-session.rgstats
```

Additional controls are:

- `--host-stats-detail=summary|full`: Select the recorder detail policy. Summary omits setup cycles while retaining measured, initialisation, and interactive cycles together

with all semantic evidence. Full additionally records setup cycles, making the extra observer work explicit.
- `--host-stats-buffer-mib=N`: Set the bounded in-memory admission buffer.
- `--host-stats-max-mib=N`: Limit database and write-ahead-log output.

The recorder batches writes on a separate thread. When recording is disabled, the event hot path skips clocks and event construction behind a predictable inactive branch. Deterministic component and native-work counts remain available to specifications independently of capture recording. When it is enabled, recording failure is evidence failure; it does not change application behaviour.

Read a capture while it is recorded

A capture is a write-ahead-logged SQLite database, and the recorder commits batches of events as it goes, so a reader may open it while the application is still running. Open it read-only and not immutable, as the platform's SQLite capability does, and each query sees exactly the transactions committed so far. Every transaction leaves every foreign key satisfied: a reader never sees a cycle without its run, or a span without its cycle.

What a reader may trust before `metadata.final_state` is `complete`:

- `metadata`: Every key except the finalisation keys below is written once, when recording starts, and never changes. `capture_id` among them identifies this capture.
- `steps`, `cycles`, `roc_work_spans`, `component_work_counts`, `gpui_frames`, `gpui_native_work`, `gpui_frame_cycles`, **and** `virtual_list_frames`: Append-only. A committed row is never updated or deleted, and ids only grow, so a reader that remembers the largest id it read finds exactly what arrived since with `id > ?`. A cycle arrives with its spans and component work, and a frame with its native work and the cycles it drew, in one transaction.
- `runs`: A run's row arrives when it starts, with outcome `running` and every end column `NULL`, and is updated once, when it ends.
- `gpui_frame_work`: Recorded after its frame, possibly in a later transaction, so a frame with no work rows yet may still receive them.

What exists only once `final_state` is `complete`, and must not be read as evidence before then:

- `measurement_status`: Every family is `unfinalized` until finalisation decides its status, reason, and row counts.
- `recorder_health`: Absent until finalisation writes its one row.
- `recording_gaps`: Written at finalisation.
- `final_state`, `clean_shutdown`, **and the drain keys of** `metadata`: `final_state` is `recording` and `clean_shutdown` is `0` until the recorder finalises, then `complete` and, as the last write, `1`.

A verdict, a comparison, or a scaling judgement rests on those, so a reader withholds it until the capture is finalised. A capture whose recorder died reads the same as one still being written; neither has a verdict.

The Observatory example follows this contract. A capture it opens before finalisation is marked *Recording* and watched through the platform's database watch, which reports each commit to the capture or its write-ahead log; when a commit wrote rows past the

largest ids it read, or ended a run, or finalised the capture, it reads the capture again and shows the new rows where the person is. Its verdicts read *capture not yet finalised* until then.

`capture_id` is a random 128-bit identifier drawn when recording starts. It carries no machine, user, path, or clock. A reader that sees a file's `capture_id` change knows the file was replaced by another capture rather than grown, and reads it again from the start.

Cycle targets

A cycle an element caused records that element in `cycles.target_kind` and `cycles.target_identity`. Both are present or both are null: a cycle no element caused, such as initialization, a task completion, or a timer tick, has no target.

`target_kind` is the node's kind in the mounted graph: `button`, `checkbox`, `text_input`, `textarea`, `canvas`, `region` (a hover or shortcut region with no surface of its own), `popover`, `split`, `drop_target`, and the containers. A shortcut's target is the region that declares it; a canvas gesture's is the canvas; a divider's drag, and its keys, target its split; a drop targets the drop target it landed on. Schema 24 admits `split`, and schema 25 `drop_target`.

`target_identity` is sixteen hexadecimal digits: a 64-bit FNV-1a hash of the node's structural path, the kind and display position of each node from the root down to the target. The path holds no label, name, text, value, key, or component instance, so the identity carries no application text and can be recorded in any capture. It depends only on the shape of the mounted graph, so the same control in the same place has the same identity in every run, on every machine, and across rebuilds that do not move it. A keyed or virtual row is named by where it is shown, never by its key, because an application derives its keys from its data. An identity is a place, not a promise: a control that moves has a new one.

Comparison identity and privacy

A capture stores comparison-relevant build, executable, workload, recorder, operating-system, architecture, and CPU properties. It never stores usernames, home-directory paths, hostnames, environment variables, command lines, application text or state, native pointers, or machine identifiers.

The capture identity, `capture_id`, is random and names only the capture; it is excluded from every comparison, because two captures are never the same one.

Grant tokens, selected resource names and paths, trusted-dialog contents, network destinations, device identifiers, clipboard values, and capture-source identities are likewise excluded. Resource owners record bounded categories and counters; unavailable authority or enforcement evidence remains absent rather than being reconstructed from diagnostics.

Compiler and application revisions are represented as unavailable unless they are supplied by their owning build process. Missing identity weakens a comparison; it is never guessed.

The Observatory example reads this identity from two captures, each through a connection of its own, and gates a comparison on it key by key: a key absent or

unavailable in either capture fails the gate rather than being assumed equal. A second run of the same executable, an A/A capture, bounds the noise of every difference it shows.

Troubleshooting

roc build cannot find libhost.a

Run `python3 build.py` from the repository root. It creates `platform/targets/x64glibc/libhost.a`, the archive selected by the platform's `x64glibc` target. Rebuild it after Rust host changes.

The Windows build rejects the SDK's shader tools

`build.py` compiles GPUI's shaders with the Windows SDK's `fxc.exe` and its `d3dcompiler_47.dll`. CI and release builds admit only the reviewed pair, by hash and signing certificate, recorded in `scripts/windows_gnu_build.py`. A development machine receives SDK servicing updates that re-sign and re-hash that pair, so a local build admits any validly Microsoft-signed pair instead and records the identities it loaded; it says so as it builds. Set `ROC_GUI_PINNED_SHADER_TOOLS=1` to require the reviewed pins locally as well. Either way an unsigned or altered pair is refused, and a refusal names the SDK directory it examined.

An application stops building after a Roc update

The platform follows the latest Roc nightly. A checkout builds with a current nightly; `roc version` names the one on the machine. A released platform names the nightly it was built and tested with in its release notes, and a newer nightly can reject code that an older one accepted. Update the platform to its latest release, or build with the nightly its notes name.

The executable rejects an argument

Host arguments belong after the executable name. Supported options include `--host-help`, `--host-run-spec PATH`, `--host-run-window-spec PATH`, `--host-smoke`, `--host-stats-record`, and the `--host-stats-*=VALUE` recording controls described in the observatory chapter. `--host-help` lists them all. The host rejects unknown arguments.

A click locator is missing or ambiguous

Check the `label` field passed to `Gui.button`. A role locator uses this semantic name, not the visible text. Names must be unique among the buttons present at that step. If a preceding click replaced a boundary, locate the new button by name rather than retaining an old host node identity.

A specification passes but performance columns are NULL

Run `semantic_health` and `capture_health` first. If both are complete, `NULL` means the selected backend did not perform or instrument that work. Semantic headless runs do not open a window, so GPUI layout, paint, and presentation cannot be inferred from them.

A comparison is rejected

Compare captures for the same specification and benchmark policy, built for a compatible target, backend, profile, recorder policy, and machine properties. Generate both serially and use `--aa` to measure normal variation. The analyzer rejects incomplete captures before calculating ratios.

The documentation build cannot render diagrams

Run `python3 scripts/prepare_docs_diagrams.py` with Mermaid CLI 11.16.0 and its Chromium runtime after changing a Mermaid block. Commit its prepared PNG, SVG, and manifest together with the source. The shared renderer checks their digests and rejects a missing or stale preparation. Run `blueprint run docs` to build the manual and API reference with the updated inputs.

How Roc GUI works

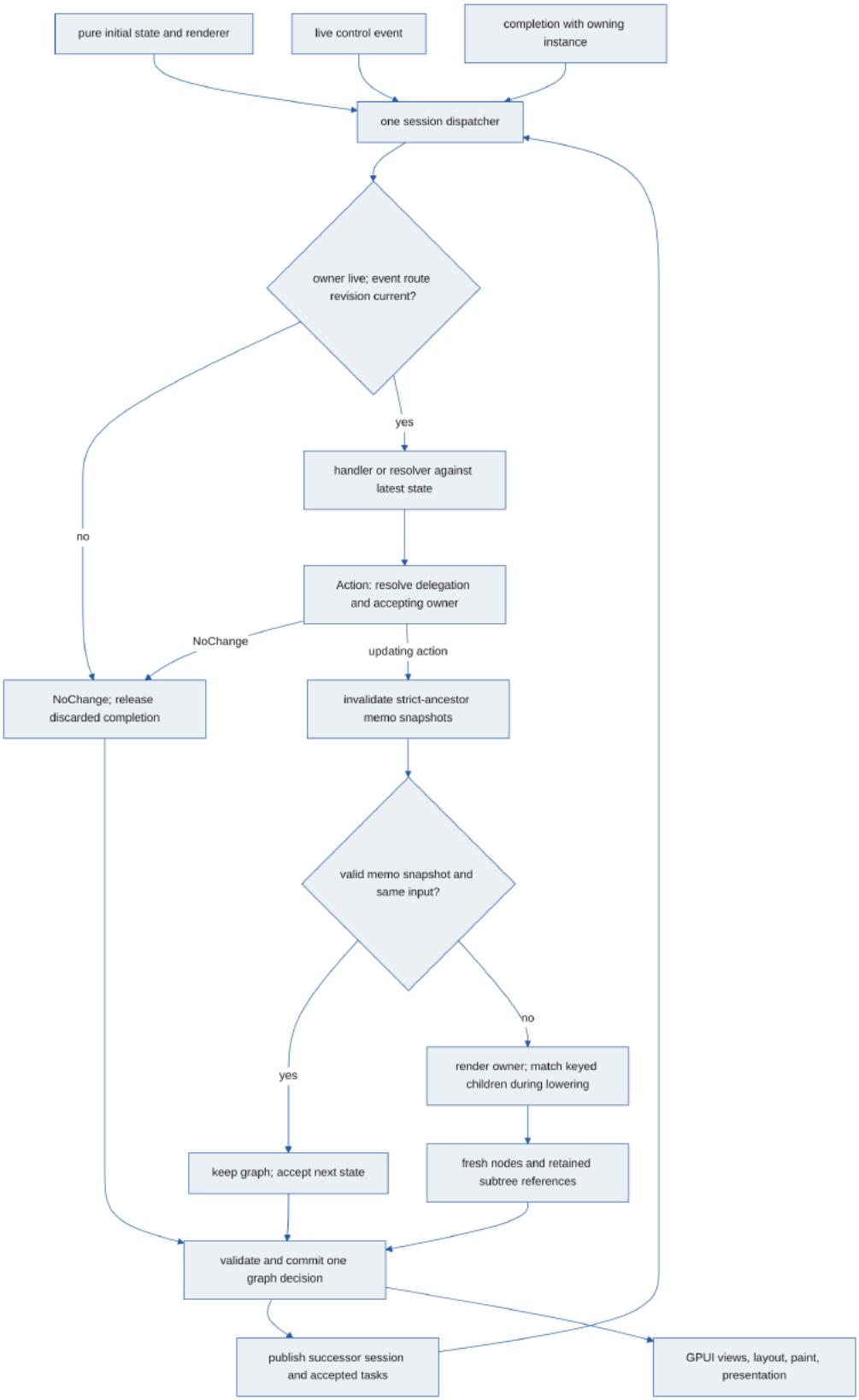
This chapter is the map of roc-gui for advanced users and contributors. It explains the path from application state to pixels, the path from input back to state, and the replacement model that joins them. You do not need to know GPUI internals to follow it.

The crucial detail is that roc-gui does not diff two virtual trees. Explicit component boundaries select the scope of an update. Within that scope, keyed memoized components can retain their complete mounted subtrees; the remainder is rendered and lowered into fresh nodes. Native controls retain their interaction state according to a separate, scoped identity contract.

The model in one page

roc-gui has three representations of a UI:

- `Elem` **value**: A short-lived Roc description produced by a renderer. It carries text, layout, buttons, checkboxes, controlled text inputs, handlers, and explicit render boundaries.
- **Mounted graph**: The host's canonical, validated tree of native nodes. It is the source of truth shared by the interactive window and semantic specifications.
- **GPUI entities**: Retained native views materialised from mounted nodes. GPUI uses them to perform its rendering-system work and deliver input.



The application never mutates the mounted graph or GPUI entities directly. It returns an `Action`; the platform turns that action into exactly one patch; and the host applies it through the same graph implementation in every GUI host and the semantic runner.

Resource security has a separate boundary from graph ownership. The application process is confined by the operating system; trusted broker services own protected selection, persistent identity, and revocation. Opaque Roc handles cross that boundary, while raw paths and platform persistence mechanics do not. The Resource access chapter describes that boundary.

From state to pixels

The application renders a value

`Gui.run` supplies pure initial state, a pure renderer, and optional resizable-window metadata. Calling the renderer produces an immutable `Gui.Elem(State)` description. Composition stays in Roc: styled and semantically named `Gui.row`, `Gui.col`, `Gui.panel`, `Gui.text`, `Gui.button`, and named `Gui.scroll` regions build the tree. `Gui.translate` introduces a local boundary with ordinary state adapters; `Gui.translate_with` and `Gui.try_translate` add explicit keys and optional policies. They are pure descriptors, with no setup-time registration or function-identity comparison. A translation adds no layout surface.

An `Elem` is not a long-lived widget object, and rendering one does not draw pixels. It is an intermediate value consumed immediately by platform lowering. The application renderer runs initially and after an updating event; it does not run for every display frame.

The platform lowers the description

The platform lowers the `Elem` description into native nodes with host-issued identities. The result is staged separately from the visible graph so it can be validated as a complete change before commit.

Framework-owned element-descriptor traversal, native-node lowering, and boundary retirement use explicit work stacks. Enter and finish items preserve child order, native scopes, delegation ownership, and measurement boundaries while keeping unfinished traversal in managed memory. Finish items hold shallow container properties or owner metadata, allowing processed descriptions to be released incrementally. Native entity materialization and retirement use the same iterative traversal principle.

Traversal and callback execution have distinct depth contracts. Descriptor mapping suspends container reconstruction in heap work items; render adapters compose typed state projections and result adaptation. Verify these owners independently, including both a deeply nested structural tree and a chain of translation boundaries. Native layout is a third execution path with its own verification requirements.

Typed state adapters use a fixed-result continuation protocol. A renderer, projection, memo predicate, or action adapter returns a work item instead of calling an arbitrarily deep adapter chain to completion. Both descent into a child and ascent through parent setters suspend before invoking the next continuation. Adjacent states can have different types; their values remain in typed Roc closure captures. The work executor has no application-state payload and the host does not decode one.

Protocol builders and continuations are pure. Effects execute in deferred work items, including the terminal item that commits the transition or delivers a task resolver. One iterative executor drives those items. Configured getters and setters have distinct work tags, so their invocation counts measure actual projection callbacks rather than the number of adapter wrappers. A missing projection short-circuits its remaining chain, and a rejected setter discards the candidate before delegation handling or task launch.

During the same walk, the platform builds event routes and records each boundary's instance, renderer, native root, parent, and route revision. A route connects a native event identity to a Roc handler and its owning boundary. Routes and boundaries live in indexed registries; each boundary owns lists of its direct routes and child boundaries, so replacing it need not filter all routes in the application.

Application keys retain all 256 SHA-256 digest bits. String keys hash a byte `0` followed by UTF-8 bytes; numeric keys hash a byte `1` followed by eight little-endian bytes. This separates numeric and string identities without a short-name limit or truncation. String literals use `Gui.Key.from_quote` at compilation; computed strings use `Gui.Key.from_str` at runtime.

Lowering resolves a keyed boundary using its surviving owner, native-container scope, and full SHA-256 key digest. The host validates the 32-byte digest and assigns a separate, monotonically increasing mounted-instance ID. A matching key preserves the lifetime across parent rebuilding and reordering within that scope. Duplicate keys in one scope are rejected. Unkeyed translations allocate fresh lifetimes when reconstructed by their parent; a local update preserves its own boundary. Key names and digests never enter observatory captures.

A keyed column carries a persistent ordered sequence and its exact one-generation journal into lowering. The Roc owner applies that journal to its persistent keyed-instance metadata, then publishes one atomic mounted-graph transaction and one native-container transaction. Inserts and sets render the affected item; moves retain its component and native identity; removals retire that identity and its routes. If the mounted owner has an older revision, the column lazily enumerates its complete ordered snapshot and reconciles surviving keys. Any rejected operation publishes neither graph nor native state. The Roc runtime, graph, native container, and GPUI frame each record only the work they own, so a bounded graph edit is not used as evidence for unmeasured layout or paint.

A keyed boundary can opt into memoization. A matching instance with the same policy and valid snapshot compares the saved input with its input in latest state. A hit retains the complete marker subtree, routes, and nested boundaries without rendering or lowering its contents. A miss builds a fresh marker and contents, giving nested keyed translations the same opportunity to survive. Enabling or disabling memoization rebuilds and resets the cache while retaining the instance and its tasks. Unmemoized boundaries keep no input snapshot.

Lowering ends with exactly one `Mount`, `Replace`, or `NoChange` commit. Staging alone cannot modify the visible graph.

The host validates and commits

The host validates a staged tree before accepting it. References and ownership must be internally consistent, the result must describe one valid tree, and a replacement target must still belong to the mounted graph. Retained roots must be live component markers from the subtree being replaced. They form a disjoint forest: none can be retained twice, shared with another owner, or made its own ancestor. Validation visits fresh nodes and the retained frontier, using saved subtree summaries for whole-graph constraints instead of traversing retained contents.

The canonical `MountedGraph` then applies the patch:

- `Mount` installs the first root.
- `NoChange` leaves the graph untouched, including when an accepted state change needs no new UI after memo comparison.
- `Replace` atomically substitutes a staged tree containing fresh nodes and explicitly retained subtrees for a mounted subtree.

The graph decision is the transaction's acceptance boundary. Only after it succeeds does the host publish the successor session, component registry changes, and submitted jobs. Rejection releases staged ownership without publishing a new dispatcher or starting a task against an uncommitted tree.

Node IDs and component mount tokens are distinct and never reused. A node ID removed by replacement is stale. A component's mount token survives a matching rerender, but retirement ends its lifetime even if the same key is mounted again later.

The host materialises GPUI views

Ordinary newly mounted graph nodes become GPUI `Entity<NodeView>` values. Rows and columns become flex containers, panels become styled surfaces, dialogs become centred modal layers with input-blocking scrims, popovers become an anchor wrapper with a floating surface, splits become two pane holders either side of a focusable divider, drop targets become flex columns that take GPUI's external file drops, text becomes a text child, and enabled action buttons add focus, keyboard, and pointer listeners. A boundary marker materialises as a transparent single-child view, not a synthetic row, column, or styled surface. Retained marker subtrees retain their native views without rebinding their unchanged content. Disabled action buttons add no event listener or tab stop. Text input entities retain native focus, caret, selection, and IME preedit state while committed values cross the same controlled Roc replacement boundary. IME preedit remains transient in the editor and dispatches one change when the composition commits. A virtual list is the deliberate exception: its complete keyed item subtrees remain in the canonical graph, but GPUI's viewport callback creates entities only for intersecting fixed-height rows and recycles entities that leave the range. GPUI asks for one row to measure and then for the rows it shows, so a list's frame is settled once, after GPUI has drawn it: rows that left the view are recycled, the measured row is kept, and the frame's materialisation is recorded. The host connects the new native subtree to retained entities around its boundary and releases replaced views.

A list of rows produced on demand goes further: the canonical graph holds only a window of its rows. The list lowers inside its own transparent render boundary, and lowering asks the host which rows to build, because the host owns the viewport. The host keeps that viewport per boundary instance, since the list node is replaced whenever its window moves. It places the rows the viewport shows and one viewport either side, applies a scroll request once per serial, and, before GPUI has drawn the list, takes the viewport to be as tall as the window. GPUI's callback draws the list's full length and holds the place of a row that is not mounted. When a settled frame shows rows within half a viewport of the mounted edge, or shows a new range the application asked to hear of, the host dispatches the list's own route. That turn renders the list boundary again against unchanged state, which invalidates no ancestor's memo snapshot, and the new window replaces the old one in one ordinary patch. An action a row raises passes through the transparent boundary to the list's owner.

A canvas that asks for its size is settled the same way, after the frame. Its prepaint records the bounds GPUI laid it out at and defers the report until the frame is drawn. The mounted graph holds the size last reported to each canvas and carries it by identity across a rebuild, as it carries hover and focus, so a size is dispatched through the canvas's route only when it is new to that canvas. Both runners ask the graph, which is what makes a size reported once whichever runner lays the canvas out.

The host enables GPUI's native view cache for non-root button leaves, rows, columns, and panels whose pixel width and height equal their minimum and maximum dimensions and whose `grow` is false. Transparent single-child boundary chains share that eligibility. Baseline-aligned subtrees use ordinary layout so that descendant content supplies the correct alignment baseline.

An unchanged eligible container reuses its subtree without constructing its children's native view elements. A changed descendant invalidates the ancestor path; rebuilding an ancestor permits independently unchanged children to reuse their caches. Changed geometry, clipping, inherited text style, and explicit window refresh invalidate affected cached content. Reused subtree frame ranges are relocated when the frame completes, preserving later child updates and native event routing.

Intrinsic and flexible dimensions follow ordinary GPUI layout. Changing a retained node's dimensions or baseline context reevaluates cache eligibility. This cache is independent of Roc boundary memoization. Rebuilding containers still visits their immediate children, and GPUI scene replay and native input dispatch have their own scaling costs.

A node ID identifies a node, not a control. Because it is never reused, a patch that rebuilds a subtree gives the same control a new ID, and keying the GPUI element by it would hand that control a new element on every patch and discard the element state GPUI holds — the pending mouse-down that pairs a press with a release, the scroll offset, the focus. The host therefore keys each element by the node's **element identity**: the path of sibling keys from the mounted root, each key the node's own semantic name where it has one and its position among its siblings where it has not, with repeated sibling names separated by occurrence. A staged node claims back the view of a retired node with the same identity and the same kind. Component instance tokens also scope these paths: two instances may contain controls with the same local name without sharing native editor or interaction

state. Within an unchanged structural scope, a control can survive repeated rerenders under a held press. Nothing of the retired node's content survives with it: handlers are rebuilt from the staged node, so a click still routes to the live node ID, and a control whose name changed or that left the tree has a different identity, gets a different element, and drops the press rather than having it misrouted to whatever took its place.

Keys do not promise arbitrary reparenting. Moving a component within its keyed parent scope preserves its instance; changing an ancestor's kind, name, or structural path remounts it. Removing and recreating the same key also remounts it. Native focus, caret, IME preedit, scroll offsets, and pending presses belong to these scoped lifetimes, not to a globally reusable application key. A virtual row leaving the materialised viewport does not remove its canonical component instance or revoke its tasks. A row produced on demand that leaves the mounted window leaves the graph, and its components retire with it like any removed subtree; state that must outlive the row belongs to the list's owner.

An ordinary patch keeps focus on the control that had it, by its stable role and semantic name, even where the node carrying that control is a new one. A control that leaves the graph entirely hands focus to whatever now occupies its place in the tab order — for a deleted row, the row that followed it, and for the last control, the new last one. Focus is dropped only when a graph has nothing focusable left, because then there is nowhere for it to go.

While a dialog is mounted, controls outside its graph subtree do not install pointer listeners or tab stops. The runtime focuses the first enabled dialog control, routes Escape to the dialog node, and restores the opener by its stable role and semantic name after the dialog leaves the graph.

A popover mounts its anchor and its content together, and the graph decides when the content presents. The pointer resting on the anchor arms the popover's delay, the host waits it out on the window's clock and asks the graph again, and focus entering the anchor presents it at once. The surface is drawn as a deferred, window-anchored layer beside the anchor, so no scroll region clips it. Escape that no dialog takes reaches the window root and closes every presenting popover. The graph carries hover and presentation state across a rebuild by identity, so an application that rerenders the anchor under the pointer neither closes the surface nor sees a second enter. A newly mounted dialog retires background popover state as it does background hover state. Opening and closing a surface runs no Roc: it is presentation, owned by the host, and counted by the graph that decides it.

A split mounts both of its panes, and the graph decides which it presents: a collapsed split's sized pane stays mounted, with its component lifetime and its tasks, but it is neither drawn nor presented to locators. The host measures a drag of the divider from the press, against the bounds the split carries, in one rule both runners call, and asks Roc only for a size the divider does not already show. The divider's keys are shortcuts of the split, live only while the divider itself holds focus, so they reach Roc through the same shortcut route every region uses.

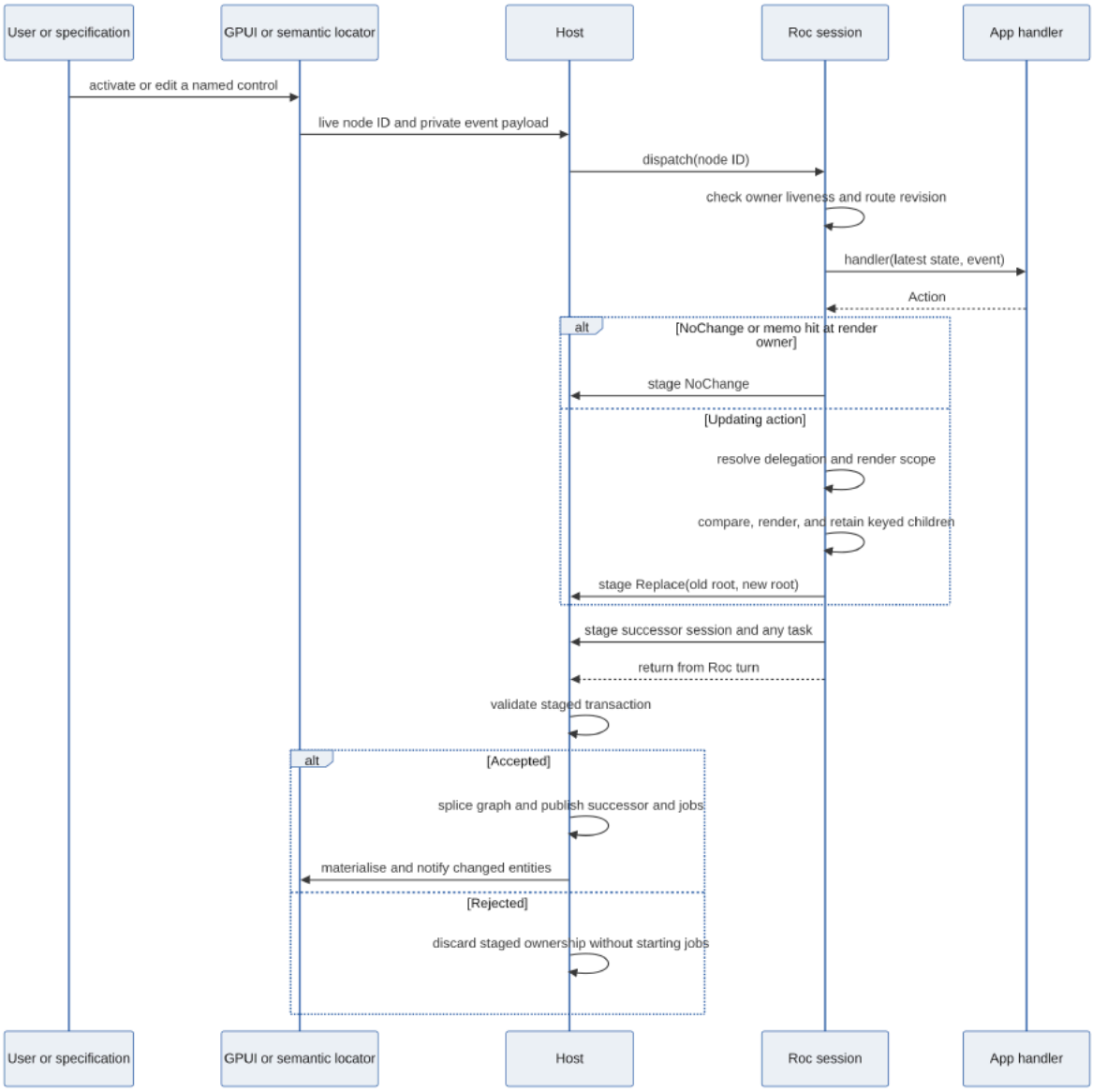
A drop target carries the file types it accepts, so the host admits a drop without asking Roc: GPUI delivers the operating system's file drop to the target it hit-tests, the host checks each dropped item as it checks a chosen file, and the target's route asks for the

grants as it runs. A window specification drops through those same GPUI events; the semantic runner admits the same paths and takes the same route. The files a drop grants are made only when that route asks, so a drop that reaches no live route grants nothing.

Shortcuts are declared by elements, so the graph knows every live one and the component that owns it. GPUI offers a keystroke to its keymap's actions along the focus path first: the host's Tab, Escape, and App access chords at the root, a focused control's activation keys, and a focused text input's editing keys. A keystroke no action took, and that a focused field is not typing, reaches one listener at the window root, which asks the graph for the shortcut it reaches. The graph walks up from the focused control to find the regions enclosing it, innermost first, then asks the regions holding window shortcuts; it never visits the rest of the tree, and it counts every comparison. The matched region's single route runs the shortcut's handler as a `key` cycle. A focus request is a serial on a region: the graph carries each region's serial across a rebuild by identity, so only a new serial asks, and after applying a patch the host focuses the first enabled control inside the region that asked. The semantic runner resolves a `key` step and honours a request through the same graph calls.

The host notifies GPUI of the changed entity. GPUI owns subsequent rendering, layout, prepaint, paint, and presentation. roc-gui does not calculate geometry or issue GPU commands in Roc, and one application event does not necessarily correspond to one presented frame.

One event turn



The session is one-shot state: one boxed dispatcher owns the state, routes, and boundaries valid for the mounted graph. Events and worker completions enter that same dispatcher through a typed Roc message, and each accepted turn installs its successor. The ABI erases boxed callables, not an unboxed generic state union. There are no separate installed event and completion closures forcing duplicate captures of the root state.

Handlers receive the latest state rather than a render-time snapshot. The host checks that native event IDs remain live; Roc also checks the owning instance and its route revision. Rebuilding a live component advances its revision, so a callback from an old route cannot address a new handler in that same instance.

Boundaries determine update scope

`Gui.translate(render, get, set)` relates child state to parent state and creates a rendering, delegation, and task-ownership boundary. The application renderer is the root boundary. `Gui.translate_with` supplies a key and optional policies; `Gui.try_translate` uses fallible getters and setters for removable state. Adapters operate on state rather than decoding keys. The `lift` method only adapts a prebuilt element's handlers: it adds no lifetime or boundary and does not consume delegation.

`Gui.Action.update` commits through the adapters and renders only the originating boundary. It neither renders nor compares unrelated siblings. Parent summaries respond through explicit delegation or composition within one render boundary. The platform does not infer dependencies by tracking state reads.

`Gui.Action.delegate(next_child)` constructs a candidate parent and passes it to the nearest translation's `on_delegate` callback. Its parent action accepts, revises, vetoes, starts parent-owned work, or delegates further. `Gui.Action.none()` vetoes the entire candidate. Delegation reaching the root becomes a root update; the default callback accepts with `Gui.Action.update`.

A key promises a stable logical component and state projection for its mounted lifetime. Rebinding either requires a different key. Removal, a changed key or scope, and owner remount all end a lifetime. Missing projections cannot recreate deleted state, and emitting a boundary with a missing projection is an application error. Parent-owned data survives reconstruction unless changed by an action.

Memoization is an explicit behavioral contract

Memoization is opt-in through `memo: Some(compare)` on a keyed translation. Ordinary translations require no equality and retain no comparison snapshot. Comparison is performed only at boundaries reached by rendering. The input must include all variable rendering and captured-handler dependencies, including labels, layout choices, and parent properties. Renderer captures and comparators remain semantically stable for the keyed lifetime; no function or pointer equality establishes identity.

Returning `True` promises that keeping the old subtree and its handlers is observationally equivalent to rebuilding with the new input. Ordinary equality is suitable only when it satisfies that promise. Custom equality can omit a relevant field, and floating-point equality can equate positive and negative zero even when a renderer distinguishes them. A false negative costs work; a false positive violates the application contract. The availability of equality does not establish this behavioral law. Deterministic floating-point calculations do not imply representation-sensitive equality: inputs whose signed zero matters need an exact wrapper or a custom comparator. No hashing or comparison of renderer functions is used.

The saved snapshot belongs to the mounted output. A local descendant update invalidates every strict ancestor's memo snapshot, even when those ancestors are not rendered. That prevents an ancestor's old `A` snapshot from incorrectly skipping after descendants changed it through `A -> B -> A`.

Typed child snapshots are erased into boxed comparison closures over the root state, not stored in an existential record field. Retaining such snapshots can increase reference counts and prevent Roc's in-place updates. The single session avoids an unnecessary duplicate root capture; it does not guarantee unique ownership of lists, strings, resources, or indexed registry nodes.

Asynchronous action turns

A `Gui.Action.task` proposes its pending state and a boxed worker. After graph acceptance, the host queues the worker on the native pool. The worker runs its continuations and hands a boxed reducer to the host completion slot. The UI thread delivers that reducer through the latest session, applies its action with the same ownership and memoization rules as an event, and commits one graph decision.

A task belongs to the boundary instance that accepts the task action. A matching ancestor rerender or keyed reorder preserves that owner. Semantic removal retires it permanently. Before invoking a completion or any captured child getter, the session verifies that the owner and its ancestor projections still exist. A removed owner's completion is discarded; it never recreates child state and never falls back to the root. To outlive a child, work must be started by a surviving ancestor, including through `on_delegate`.

Task lifetime, supersession, and cancellation

The host issues each task when the transaction that proposed it commits, and records it against its owner and an optional key. Every issued task ends in exactly one way:

- **delivered**: its worker published a completion and the UI thread applied it;
- **superseded**: the same owner committed a newer `Gui.Action.keyed_task` with the same key;
- **cancelled**: the owner committed a `Gui.Action.cancel` for its key, the owner was removed from the mounted graph, or the session ended.

Supersession and cancellation are staged with the turn that asks for them and take effect only when its transaction commits, like the task launches of the same turn: a removed owner's tasks end first, then the turn's cancellations, then its new tasks are issued. A task that ends undelivered never reaches the application. If a worker has not started it, it never runs. If it is running, a capability wait it is blocked in is interrupted: a timer's `next!` and a watch's `next!` return `Canceled`, leaving the timer or watch itself running for a later task, and a SQLite statement stops through its connection's `sqlite3_interrupt`, backed by a progress handler for a statement that had not yet begun to step. If its completion is already queued, the UI thread drops it where it takes it. No undelivered completion enters Roc, so none records a cycle.

The task owner keeps six deterministic counters: tasks issued, completions published, completions delivered, tasks superseded, tasks cancelled, and capability waits interrupted. Issued tasks minus delivered, superseded, and cancelled ones are the work still outstanding; that difference is what a window run's settling waits on. A gauge of the running tasks blocked in an interruptible wait, or ended and still unwinding, lets a

specification cancel a task that is certainly blocked rather than one still queued, and then wait until it has released what it held.

Keys are scoped to the owning boundary instance, so two instances of one component never supersede each other. An unkeyed `Gui.Action.task` supersedes nothing and is cancelled only with its owner. Discarding a completion releases its result and captured resources through normal reference counting. Roc values cross threads only as owned erased callables; generic state and actions remain boxed across the ABI.

Periodic subscriptions use the same route. A timer has at most one waiting `Gui.Timer.Handle next!` task. Its wait parks on a worker and returns a normal boxed reducer; the UI thread applies that reducer through the installed session. Cancellation wakes the waiter, and cadence advances from the completion of the preceding pull, so a slow consumer cannot create an unbounded event backlog.

HTTP requests also use `Gui.Action.task` and the canonical worker/completion route. An opaque `Box(U64)` client names a host registry entry for one explicitly granted origin. The initial URL and every redirect retain its scheme, host, and effective port. Application-data file operations use that route as well. Roc retains only an opaque typed read-write directory handle; the host confines flat child names to the granted application root, bounds values, and commits each atomic write by synchronising a private temporary file before an in-root rename. Application request identities decide whether a completion is still current on the UI thread. Clipboard observation follows the same capability and task boundary. GPUI's UI context owns operating-system reads and writes; the host publishes only the latest bounded text and a monotonic sequence to capability-checked worker calls. The application timer supplies backpressure and cancellation. Neither the shared snapshot nor clipboard values are observatory fields. The host applies explicit scheme, timeout, redirect, header, request-body, and response-body limits before returning an owned reducer to the UI thread. It buffers one bounded byte response rather than exposing a streaming callback. Applications suppress obsolete completions by comparing their request identity with latest state. URLs, headers, bodies, and response text never enter host captures or deterministic diagnostics.

Directory, HTTP, and timer handles are typed nominal wrappers over `Box(U64)`. Each token names a host registry entry containing a `cap_std::fs::Dir`; final box deallocation removes that entry. The phantom resource kind leaves room for other handle families without allowing their types to be confused. The trusted resource broker creates a grant only for the selected project. Listing and child acquisition operate relative to the stored handle, and child acquisition uses the no-follow operation from `cap-fs-ext`.

There is no node-level diff

roc-gui compares declared component inputs and matches explicit component keys; it does not diff arbitrary old and new `Elem` descriptions or mutate equivalent ordinary nodes in place. Fresh output outside retained component subtrees gets fresh node IDs, even if its text and position are unchanged.

This makes the costs and semantics direct:

- the boundary structure exposes replacement and invalidation scope;

- key scope and memo equivalence are explicit application contracts;
- removed routes and node IDs become unambiguously stale; and
- work is attributed separately to comparison, rendering, lowering, graph validation, retirement, and native materialisation.

An ancestor render still enumerates its immediate output and the component frontier it emits. Memoization does not make that enumeration free. Registry indexing avoids global route scans, but immutable index updates can copy paths, and an application's keyed getter can itself perform a linear collection search. Scaling claims must account for each of those owners instead of attributing every allocation or callback cost to rendering.

The scaling suite exercises full-root collections, local boundaries, text density, tree depth, and tree breadth so the consequences of replacement scope remain visible as the implementation evolves.

Origin of the Action-State model

The Action-State model this platform implements originates with Richard Feldman, who described it in *Design Idea: Action-State*. That write-up proposes a user interface API for Roc in which a renderer is a pure function from state to an `Elem` value, event handlers return an `Action` rather than a message, and a translation function relates a component's state to its parent's state while marking the unit of re-rendering. roc-gui is an development of that design. `Gui.Action.update`, `Gui.Action.none()`, the shared `state` type variable on `Elem` and `Action`, and adapting child renderers through getters and setters follow its central idea: state ownership can identify the subtree an action needs to update. Pure translations express that boundary; explicit keys preserve logical identity through parent rebuilding, while the host owns mounted lifetimes.

The motivation is worth restating, because it explains the shape of the API more than any implementation detail does. A renderer takes state and returns a description; it performs no effect, invokes no capability, and observes no clock, so calling it twice with the same state yields the same description and calling it is never a way to make something happen. Everything that can make something happen is an `Action` returned from a handler, and an `Action` is a value: it describes a transition rather than performing one. Constructing `Gui.Action.update(next)` changes nothing. The platform changes state only when it receives that value back from a handler and accepts its graph transaction.

That division is what makes the concurrency tractable. Application state lives on one thread, and every state transition is applied there, one at a time, by the dispatcher installed for the currently mounted graph. Each turn takes the state, runs one handler, applies the single action it returned, renders the required scope, commits one graph decision, and installs its successor. Two transitions cannot interleave, no handler can observe a half-applied update, and a handler always receives the state that precedes it rather than a snapshot captured when the tree was rendered. A handler's first argument is therefore the state to build the next state from.

`Gui.Action.task` extends the same discipline to work that must not block the interface. It commits its `pending` state before handing the effect body to a worker thread, which returns a reducer rather than a state value. The UI thread applies that reducer to whatever the latest state is by then, so events that arrived while the task was in flight are not

overwritten by a stale snapshot. The effect runs off the UI thread; the state transition it produces still happens on the UI thread, in the same serialised order as every other transition. Effects thus stay out of rendering entirely, and concurrency is confined to the body of the work rather than spread through the state model.

Several ideas in the early write-up are useful independently of its proposed surface syntax. Pure descriptions leave execution to the platform. Delegation lets the parent decide whether to accept a child's proposal. Fallible state projections make removal explicit. Deferred rendering enables memoized subtrees. roc-gui gives those ideas one coherent ownership model through pure translations, `Gui.Action.delegate`, host-owned keyed lifetimes, and explicit memo equivalence.

Periodic work uses `Gui.Action.task` around a timer handle in application state, with explicit cadence and cancellation. There is no string-keyed global effect tracker. Instance liveness answers whether an owner still exists; application request identity answers whether a result remains relevant to that live owner. Those are distinct questions, and neither replaces the other.

How this compares

Model	What persists	How application changes arrive	Identity on update
roc-gui	Mounted graph and GPUI entities outside rebuilt output and inside memo hits	A state-derived subtree and explicit retained roots become one graph decision	Scoped keyed component lifetimes; fresh ordinary IDs except in retained subtrees
Virtual-tree reconciliation	A native/render tree, often plus a previous description	Old and new descriptions are diffed, commonly using keys	The reconciler may preserve matching nodes
Immediate mode	Rendering-system state rather than an application widget tree	Application code describes the interface repeatedly, commonly each frame	Usually not expressed as persistent application widget identity

Model	What persists	How application changes arrive	Identity on update
Traditional retained widgets	Application-visible widget objects	Application code mutates long-lived widget properties and children	Widget objects ordinarily retain identity until explicitly removed

roc-gui combines a declarative Roc front end with retained native state. It is not immediate mode because the application does not redraw the interface every frame. It is not virtual-DOM reconciliation because render boundaries, and declared memo equivalence, not a general tree diff, choose what survives. GPUI may perform its own internal rendering invalidation; that is separate from roc-gui's application graph replacement model.

Semantic tests use the production graph

The semantic runner stops after the mounted-graph layer because it needs no display server to locate semantic controls or assert text. It invokes the real Roc renderer and dispatcher, consumes the real patches, and uses the same `MountedGraph` implementation for validation, replacement, event routing, and node accounting. It does not reproduce those behaviours in a test-only graph.

The interactive host continues by materialising GPUI entities. Consequently, semantic specifications prove application behaviour and mounted topology, but do not exercise `NodeView`, display frames, layout, paint, GPU work, or presentation. Those require the interactive production path.

Threading and observability

Application callbacks, graph commits, GPUI entity updates, and window work stay on the interactive application path. Work owners maintain deterministic component counts even when capture recording is disabled, so specifications observe the same facts in both runners. Clock reads and capture records remain behind recorder enablement.

With recording enabled, work owners emit bounded records. A separate writer thread drains them in batches and commits transactions to the `.rgstats` SQLite capture. UI behaviour never waits for a report query. If the recorder cannot accept evidence, the capture records that loss rather than presenting partial data as complete.

A timer means only the phase that owns it. Host application timing is not a substitute for later layout, paint, GPU execution, or presentation. The Understanding captures chapter covers availability and report views, and the Measuring scaling chapter covers representative workloads.

Correctness boundaries

The platform and host maintain these invariants across every event turn:

- one initial mount and one canonical mounted root;
- one completed action and graph decision per dispatch;
- unambiguous native identity for every live node;
- structurally valid changes applied only to mounted targets;
- stable logical projections and unique full-digest keys within each scope;
- retained subtrees keep their routes, descendants, and native identities;
- rebuilt routes advance their owner's revision; retired instances never revive;
- ancestor memo snapshots invalidated after descendant updates;
- completion owner liveness checked before captured state projections run;
- successor state, routes, and tasks published only after graph acceptance; and
- mounted-graph commit before GPUI materialisation.

A malformed ABI value or structurally invalid patch is a platform defect, not application input to recover from. The host rejects it at the ownership boundary rather than displaying a partially mutated graph.

Rows and columns stream their lowered child identities into nesting-safe, host-owned builders. The closing node effect consumes its builder and stages the same ordered graph node used everywhere else. This ownership boundary avoids allocating a temporary Roc child-identity list and then copying it into host storage. A patch cannot commit while a child builder remains unfinished, and builders must close in nesting order.

Responsibilities when extending the platform

Concern	Primary owner
Public state, element, action, and layout API	Roc platform
Lowering, routes, boundaries, and dispatcher succession	Roc platform
Hosted effects and ABI ownership	Platform and Rust host boundary
Canonical graph, patch validation, and accounting	Host graph
Native appearance, click input, and entity materialisation	GPUI host
Tab order, visible focus, and Enter/Space control activation	GPUI host over the canonical button and checkbox nodes
Headless semantic driving and locators	Semantic runner using production structures
Capture lifecycle, batching, SQLite schema, and reports	Observatory

A new feature crosses every owner it needs: public API, lowering and ABI, graph representation and validation, GPUI behaviour, semantic locators and specifications, scaling coverage, and documentation. The governing rule is one mounted graph, one

event route, and one production path: tests and measurements drive those structures; they do not imitate them.

Contributing

Read `AGENTS.md` before changing the platform. The central constraint is “test what we fly”: tests and measurements drive the production implementation and never reproduce it.

Development environment

`Blueprint.roc` declares the environment and `Blueprint.lock` fixes it. Commit both; `.blueprint/` holds generated files. Install the released `blueprint` binary and a current Roc nightly as `roc`, then run tasks from the repository root:

Task	What it does
`blueprint run check`	Runs `scripts/check_all.roc`: the compiler pins, the bindings root, tracked machine code, the scripts and their tests, and `roc check` and `roc test` on every application and benchmark. Errors, warnings and failed expectations all fail it.
`blueprint run build`	Builds the native host.
`blueprint run test`	Builds every application and runs its specifications through the production host.
`blueprint run lint`	Runs Clippy over the host with warnings treated as errors.
`blueprint run regenerate-glue -- <roc source>`	Regenerates the host bindings from a nightly’s source; see Generated bindings.
`blueprint shell`	Starts a shell in the development environment.

The `check` task needs no Nix: it fetches `roc-stable` and expects `roc` and `git` on the machine. The native tasks take their toolchain and libraries from the pinned Nix package set.

Two compilers

The platform, its applications and its benchmarks name no compiler. They are written for the latest Roc nightly, which is bare `roc`, and CI resolves that nightly once per run. A nightly that breaks them fails the scheduled compatibility workflow, and the fix is a change to the platform.

`roc-stable` is the one pinned compiler. `Blueprint.roc` fetches it with `Recipes.roc_nightly`, and every script under `scripts/` names the same tag in its `roc` header field, which `check` enforces. It runs the repository’s Roc scripts only. To move it, change the tag in `Blueprint.roc` and in every script header, run `blueprint update`, format the scripts with `roc-stable fmt scripts`, and merge once CI passes.

Install the repository hooks with `pre-commit install` and run them inside `blueprint shell`. The Roc hooks run `scripts/check_all.roc`; with `--staged` it checks only what the next commit records for machine code. Specification builds also reject a warning exit even when the compiler produced an executable.

The development environment scopes native build dependencies, including Fontconfig and Wayland, to Linux. The build helper prepends its verified ALSA interface to the shell search paths. Executables use the native desktop runtime libraries selected by the system ELF loader.

Development builds take the unified linker-input release when its producer inputs match the checkout. A missing or stale release stops the build rather than selecting other inputs. `--source-inputs` selects the recipe inputs a target still has: generated interfaces on macOS and the released GNU runtime on Windows; Linux has none outside a producer run. Invalid locks and failed artifact verification stop the build. Release packaging requires a matching unified lock.

CI runs the same Clippy command as the local pre-commit hook, including test targets, with `-D warnings`. Install the `clippy` component from the repository Rust toolchain when running the hook outside the Blueprint environment.

CI checks that every maintained example and benchmark has adjacent specifications and runs the semantic suites. A build or semantic failure fails the gate; it is not silently skipped. Record unresolved defects in `wip/issues-backlog.md`.

Release packaging records the compiler that bundled the release in its manifest, and the release notes name it with `roc-stable`. Archived examples name only the released platform, like the branch sources.

Validate a change

```
python3 scripts/run_cargo.py test --locked --package roc-gui-host
roc check examples/counter/main.roc
python3 scripts/run_specs.py 'examples/counter/specs/*.scm'
```

`roc check` answers whether a program type-checks, which is not the same question as whether it compiles. A change to the shape of a platform type can leave every example checking cleanly while code generation for one of them fails to terminate, so a change to `platform/` is validated by building an application and not only by checking one. The specification runner builds every application it runs, which is why it is the gate that matters for a platform change.

The specification runner verifies and stages the independently released native host when its recorded source fingerprint matches the checkout. A host-source change automatically builds and exercises the changed production host instead.

Run the affected benchmark family when a change touches rendering, routing, graph work, allocation, recording, or a scaling-sensitive data structure. Run the complete specification suite before integrating a cross-cutting change.

When changing an effect, update `wip/resource-access-inventory.md` after reading the public API, hosted ABI, native owner, configuration source, and verifying specifications. Mark missing enforcement or evidence `Unverified`; never infer security from an absent public function or an opaque Roc type.

Complete a platform feature

A new element, property, event, or state mechanism is complete when it has:

1. a Roc API and hosted ABI representation;
2. behaviour in the single host graph and GPUI path;
3. semantic locators, operations, and assertions where users need to observe it;
4. deterministic counters and timing spans recorded by the work owner;
5. behaviour, ambiguity, and failure specifications;
6. a realistic scaling case for every dimension that can grow;
7. focused SQLite reporting for evidence that production code populates;
8. application-facing documentation.

A resource feature additionally requires trusted authorisation, OS-enforced scope, derivation and lifetime rules, revocation semantics, privacy review, and platform-boundary tests. The Resource access chapter describes them.

Exercise recorder-disabled, enabled, saturated, output-limited, interrupted, and writer-failure paths when changing capture infrastructure. Timing is not a correctness assertion.

Fixture-backed steps

The steps below are available to any specification, and this repository's deterministic fixture data is what makes their counts stable. All of them are numeric owner evidence: no name, path, URI, identifier, or application text is recorded. What a case may reach is declared by its own `(grants ...)` clause, described in the specification language reference.

- `expect-audio-counters`: Live outputs, live tracks, and the seven operation-attempt counters.
- `expect-file-access`: Chooser, provisioned, and revoked root counts.
- `expect-file-lifecycle-counters`: Root allocations, child allocations, live handles, revocation requests, successful revocations, and post-revocation operation rejections.
- `expect-file-selection-counters`: Seven host-owned totals, in order: chooser requests, chosen results, cancellations, denials, unavailable results, suppressed prompts, and development provisions.
- `clipboard-text`: Changes the clipboard source granted to a fixture, and is rejected without an explicit fixture grant. The next application timer pull observes the new value through the production capability and host read route; the value is neither copied into capture evidence nor emitted as a diagnostic.
- `expect-image-owner-counters`: The production image-data owner's staged-node, staged-byte, inspection, and failure totals. Image names, paths, encoded values, and dimensions are not capture evidence.

- `expect-asset-counters`: The production asset owner's six totals, in order: opens, refused opens, manifest checks, reads, refused reads, and bytes read. Asset paths, file names, and asset content are not capture evidence.
- `expect-document-counters`: The document owner's picks, chosen files, cancellations, refusals, and reads, then its live handles.
- `expect-hash-counters`: The file hash owner's files hashed, hashes refused, and bytes hashed, across directory and single-file grants. File names, paths, and digests are not capture evidence.
- `revoke-file-grants`: Invokes the trusted-host revocation seam for every chosen folder and file. It is not an application event or a Roc capability API.

The Settings Center's 100-, 1,000-, and 10,000-byte profile-note cases exercise ordinary controlled editing followed by the same worker-backed atomic save the application uses. `expect-value-bytes` supplies deterministic, content-free scale evidence; task and patch counters remain owned by their production paths.

Generated bindings

Regenerate checked-in Roc ABI bindings whenever hosted functions or `platform/Host.roc` change:

```
blueprint run regenerate-glue -- /path/to/roc-source
```

The source is that of the nightly `roc` is, such as the nightly release's source archive unpacked. CI regenerates the bindings from the resolved nightly's verified source archive and fails when they differ from the committed files.

`platform/main-glue.roc` is an ABI-generation root. Applications build against `platform/main.roc`.

Documentation

AsciiDoc sources live in `docs/`; `docs/index.adoc` is the complete book. Build the HTML site, standalone PDF, static Markdown, and compiler API reference with:

```
blueprint run docs
```

Blueprint downloads the released `roc-docs` executable for the host and verifies its digest against `Blueprint.lock`, including restored cache bytes. The shared renderer supplies the HTML and PDF theme and source highlighting. The `roc` on PATH generates the platform API reference; it is independent of the renderer's tooling compiler. Output is written beneath `.docs-out/`, with the complete publishable site in `.docs-out/site/` and the manual at `.docs-out/roc-gui.pdf`. Pass renderer options after `--`, for example `blueprint run docs -- --docs-version main --roc-command /path/to/roc`. Use `--output-directory` to select another output directory inside the repository.

Mermaid source stays beside the prose. After editing a diagram, install `@mermaid-js/mermaid-cli@11.16.0` and its Chromium runtime, then run `python3 scripts/prepare_docs_diagrams.py`. Commit the source, PNG and SVG inputs under `docs/images/diagrams/`, and `docs/prepared-diagrams.json` together. The manifest binds the source and prepared assets by SHA-256; the renderer rejects stale assets. Diagram preparation is an authoring step. Ordinary manual builds consume those inputs without a browser or external converter.

Headings receive IDs automatically. Use those IDs in cross-references and reserve explicit IDs for intentional permalinks. The shared authoring profile is documented in the shared tooling manual. CI renders with read-only permissions and adds the tested example gallery before handing inert site files to the Pages deployment job. Release assembly builds the same documentation before the separate publishing job receives its archives.

Documentation describes the product and its enduring contracts. Put defects, missing capabilities, and implementation tasks in `wip/issues-backlog.md`, then remove entries when they are resolved.

Native linker inputs and releases

Generated objects, libraries, TBD files, host archives, and platform bundles do not belong in Git. Reviewed recipes, catalogs, provenance, locks, and license texts do.

There are two kinds of release and three things that are built:

1. The linker-input producer builds one content-addressed release containing the complete `x64glibc`, `arm64mac`, and `x64mingw` sets. This includes every `platform/main.roc` input the host does not produce; the Windows resource therefore belongs to this release, not to the host.
2. The host, `libhost.a` for each target, is bound to the exact host source, Cargo lock, configuration, profile, and toolchain inputs. The Windows host also produces `windows-imports.lib`, described below. The host has no release of its own: it ships in the platform release.
3. Platform release assembly builds every target's host on its own system, bundles it with the locked linker inputs, runs the supported consumer matrix, and publishes those same tested bytes under Roc's content-addressed bundle name, together with each host's source companion.

Continuous integration

CI produces each target's native host once per run and exercises those exact bytes everywhere. The reusable host workflow fingerprints the host sources. When an earlier run sealed a host with the same fingerprint and linker-input lock, it is restored from that run's cache and nothing native is compiled. Otherwise the job runs the host unit tests once and builds with the release producer, `scripts/prepare_host_build.py`, or, when the linker inputs are not yet released, against development inputs. The job seals the staged `platform/targets/<target>` with its source fingerprint and the digest of every file, and caches it under that fingerprint.

Every specification, smoke, window, and gallery job admits that sealed host with `scripts/ci_host.py admit`, which refuses a different source fingerprint or any changed byte, and runs with `--skip-host-build`. No test job builds a host. Each target's tests wait only for that target's host.

When a host is built rather than restored, a candidate job composes the notice payload, source companion, and host archive from the same sealed build, so the archive a release would bundle is the one the specifications exercised. `release.yml` builds and composes every host the same way, from the commit it releases.

Host source companions

Host notice selection comes from Cargo's successful build messages and is checked against the committed lock. Registry sources retain their published crate archive and checksum. Git sources require an exact repository, revision, and package path in `dependencies/gui-host-notices/git-sources.json`.

The build captures Git package sources from immutable Git objects, checks them against the compiler's checkout, and retains original workspace declarations and notices. In-repository symbolic links are resolved; escaping links and submodules are rejected. Evidence records the repository, revision, package path, Git tree identity, and a separate SHA-256 digest of the deterministic source archive. Notice composition and source-companion validation verify that archive digest; a Git revision is never used as a crate archive checksum.

The Windows import library

A Windows link needs an import record for every DLL function the program calls. The platform ships exactly those records in one library, `windows-imports.lib`, rather than the complete export lists of every DLL a dependency could name. Only the host and the GNU runtime call DLLs, and what they call is fixed when the host is built, so the host build derives it:

1. rustc reports the native libraries the host archive leaves to the final link, and the build scripts report where they are.
2. `scripts/windows_link_imports.py` follows the link the way the linker does: from the host's objects and the runtime's startup object, it pulls a runtime or import member only when a symbol it defines is still undefined.
3. An import rustc emitted is taken with the DLL and type rustc gave it. A name referenced without one is resolved through rustc's native libraries in rustc's order. Anything left must be defined by the linker or by the Roc application; any other name fails the build.
4. The complete exports of `kernel32` are added as the application ABI: code the Roc compiler emits calls it as C code calls the UCRT, and a later compiler may call more of it than this host does.
5. Zig's `dlltool` writes one import library per DLL from a module-definition file, and the members are merged into one archive. Every member is read back and compared with the requirement it was made from.

The host's own import members are then separated from `libhost.a` against the host's own inventory. The normalization receipt records both steps, the native libraries, and every import by DLL, name, and type; a host archive is admitted only when the packaged library matches that receipt.

Linker-input production runs only when explicitly dispatched for an open pull request whose branch contains a material recipe, catalogue, resource, or symbol change. The unprivileged producer builds each input twice, validates it, and uploads exact candidate bytes. The trusted default-branch publisher admits the requested branch and commit, verifies the successful producer run and its attestations, and publishes those same bytes under the SHA-256 of the aggregate manifest. The release remains permanent after the pull request merges. The publisher adds a GitHub-signed, lock-only commit to that pull request, so the review adopts the produced release without a bootstrap build on the default branch.

`link-inputs.lock.json` records the immutable release, target archive digests and sizes, source revision and branch, signer workflow, source fingerprint, and aggregate manifest digest. Ordinary pull requests have no linker-input build job. They restore the archive cache, verify its locked size and SHA-256 even on a cache hit, and download the immutable release only on a miss.

Schema-2 unified locks also carry FreeType's reviewed component build identity, selected snapshot binding and original component archive digest. Linux admission CI verifies the component bytes and compares the exact prepared build identity. It may refresh the binding only when that identity is unchanged. A different published artifact requires an explicit repository admission decision.

Consumers use `blueprint plan build freetype --json` for the selected snapshot, without Nix preparation. Missing, incompatible or changed evidence cannot pass freshness checks. These checks require the `blueprint` executable of the release `Blueprint.roc` names on `PATH`; CI installs that release's binary on every supported system after verifying it against its reviewed SHA-256. The other producers retain their shared-file and standalone package-pin fingerprint; only FreeType-exclusive paths move to component admission.

Consumers verify locked bytes before extraction and on cache hits. A digest, size, manifest, target, provenance, or attestation mismatch is a hard failure and must never fall back to a local build. Development builds require current reviewed authority. `./build.py --source-inputs` selects the macOS source interfaces explicitly. A Linux host build takes its build-time ALSA interface from the admitted linker inputs: the unified release, or the producer's own candidate through `./build.py --skip-inputs --candidate-inputs <directory>` while it validates fresh components; the producer then stages its exact candidate archives before final-link validation. Final platform releases accept only locked, released, and attested inputs.

Change a linker input through its owning recipe and tests, then dispatch the trusted linker-input publisher with that pull request number. The resulting signed lock commit makes the new symbols available to the same branch. Ordinary platform and host changes reuse the locked release and never invoke a producer.

Linux producers with Nix

FreeType uses Blueprint's managed Build contract declared by `dependencies/FreeType.roc`. Its dedicated environment names its compiler, headers, libraries and tools. `dependencies/freetype.json` is the reviewed upstream URL/checksum authority. Selected files exclude the whole project configuration, whole Lock and unrelated component producers. The Python script retains compilation, packaging and its native glyph probe.

The other four Linux component producers use `dependencies/linux/default.nix` and the reviewed `dependencies/nixpkgs-pin.nix`. Blueprint uses the same reviewed revision for its shared package source. Zig 0.16.0 and the build tools come from that package set. Meson uses the recipe's fixed-hash 1.12.0 source with the same nixpkgs packaging. Upstream source downloads are fixed-output derivations; compilation and native probes run inside the network-isolated Nix sandbox.

Run a component producer with an absent output directory, then force a rebuild:

```
python3 scripts/build_freetype.py --output /tmp/candidate
python3 scripts/build_freetype.py --rebuild --output /tmp/rebuild
cmp /tmp/candidate/freetype-x64glibc.tar /tmp/rebuild/freetype-x64glibc.tar
```

`--rebuild` executes Nix's reproducibility check, even when the first result is cached. CI requires sandboxing and repeats this for every Linux component. Schema-5 FreeType archives include selected native reproduction evidence: package revision/NAR hash, original-file hash, captured source identity and tool roots. Its native format 1 also explicitly lists source trees, dependency-build outputs and Roc package paths. This FreeType profile requires all three lists to be present and empty; unexpected inputs are rejected rather than silently omitted from its reproduction contract. The evidence is checked against its checksum and shipped corresponding source files. Selected source modes are recorded for faithful reconstruction. The final build identity is external, in `freetype-x64glibc.tar.blueprint.json`, bound to the measured archive digest and Blueprint's selected snapshot. It is never embedded into its own recipe inputs.

`python3 scripts/check_linux_reproduction.py /tmp/candidate` rebuilds a minimal configuration around the shipped unchanged FreeType module, verifies resolved pins against shipped evidence and prepares the exact same build identity. Reproduction explicitly permits fetching the immutable pinned inputs; ordinary consumers do not run the producer.

The other Linux producers use schema-4 archives with builder derivation, standalone package-pin checksum and Nix recipe. FreeType accepts only its Blueprint schema-5 contract; other producers retain their own native archive contracts. Producers do not embed the whole Blueprint Lock; unrelated lock edits do not alter their input bytes.

Published shared libraries have no build search paths or store-qualified runtime dependencies. Producer probes use the sandbox's private loader and providers. A separate Ubuntu validation job admits the exact candidate archives, final-links the

production host, and runs the GPUI smoke test outside Nix before those archives can be assembled and attested.

Windows inputs from Nix and native Windows

On Linux x86-64, the Windows runtime and resource producers use `dependencies/windows-gnu-runtime/default.nix`, with the same nixpkgs pin. The runtime compiles with the recipe's hash-verified official Zig distribution; the resource compiler comes from pinned nixpkgs. Downloads are fixed-output derivations and builds run in the network-isolated sandbox.

```
python3 scripts/build_windows_gnu_runtime.py --output /tmp/windows-candidate
python3 scripts/build_windows_gnu_runtime.py --rebuild --output /tmp/windows-
rebuild
python3 scripts/build_windows_resource.py --output /tmp/windows-resource
python3 scripts/check_linux_reproduction.py /tmp/windows-candidate --target
x64mingw
```

On native Windows x86-64, Python 3.12 or newer runs the same runtime compilation and packaging code using the recipe's hash-pinned Windows Zig distribution. The resource command uses Zig 0.16.0 on `PATH`. Set `TEMP` to a directory without spaces, as required by the verbose linker diagnostics used to identify inputs.

```
python scripts/build_windows_gnu_runtime.py --output candidate
python scripts/build_windows_resource.py --output resource
```

Native builds use fresh compiler caches on every invocation; `--cache` stores only the verified Zig download. Schema-3 native archives record the Windows toolchain digest and reproduction source digests. Schema-4 Nix archives record the pinned Nix build environment. Neither archive records the builder's local paths or machine identity.

CI requires two identical Nix builds, two identical native Windows builds, and matching resource bytes across both hosts. Windows executes the runtime probes against each host's candidate, including CRT startup and teardown, thread-local destructors, C++ and Rust unwinding, and UBSan failure. The published runtime is the Nix candidate that passed native execution; cross-host runtime byte equality is not required because the build provenance differs.

Generated system-library interfaces

When an application links to a library supplied by the target environment, its producer emits only the reviewed symbols and the runtime library's stable name. The archive does not contain an implementation or an absolute provider path. Production verifies the exact exported symbol inventory and stable library name, links a probe through the generated

interface, and runs that probe against a native provider. The release carries the interface recipe, probe, producer, and archive verification inputs needed to reproduce it.

The target system or package environment supplies the implementation at runtime. This keeps linker inputs independent of a distribution's filesystem layout while preserving normal runtime library resolution.

Generated macOS interfaces

Read `dependencies/macos-interfaces/README.md` and `PROVENANCE.md` before changing the macOS catalogue. No Apple SDK binary, header, TBD, framework binary, dylib, object file, symbol table, generated SDK metadata, or other installed macOS software may supply interface data.

The project-built `libhost.a` may be inspected only to inventory unresolved references. That inventory does not establish library ownership, public API status, weak-link behaviour, or final-link reachability and never populates the catalogue automatically. Every selected symbol and owning library requires reviewed public documentation or identified public open-source evidence. Prefix-based classification and installed system artifacts do not establish library ownership.

Generation is deterministic and offline from the reviewed catalogue. Native macOS may then final-link and run applications to validate the candidate. This uses the operating system as the implementation under test; it does not derive the interfaces from installed Apple software.

Release notes

Each release's notes, newest first. A release candidate's notes stay here after its final release.

roc-gui 0.1.0-rc1

The first published release of Roc GUI, a platform for native desktop applications in Roc rendered by GPUI. An application is pure `init` and `render`; there is no virtual DOM, no diffing step to manage, and no component lifecycle to learn. Every executable also carries a specification runner, so behaviour, layout and appearance are tested against the same binary a person runs.

This is a release candidate. Try it, report what does not behave as the manual says, and expect a `0.1.0-rc2` if anything needs fixing before `0.1.0`.

Use this release

```
app [State, main] { pf: platform "{package-url}" }

import pf.Gui

State : { clicks : U64 }

render : State -> Gui.Elem(State)
render = |state| Gui.col(
    { label: "Counter", gap: 8, padding: 12 },
    [
        Gui.text("Clicked ${state.clicks.to_str()} times"),
        Gui.button({ caption: "Click me", label: "Click me", on_press: |s, _|
Gui.Action.update({ clicks: s.clicks + 1 }) }),
    ],
)

main : Gui.Program(State)
main = Gui.run({ init: |_access| { clicks: 0 }, render })
```

```
roc build --opt=dev --output=counter main.roc
./counter
```

Which Roc to use

The platform names no compiler. Build with a current Roc nightly; the "Compilers" section below names the nightly this release was built and tested with. A scheduled workflow builds the release's examples on every new nightly and opens an issue when one stops building.

If you built against a checkout

Applications written against the repository before this release need these changes:

- `Gui.Action.none`, `Gui.Action.refresh` and `Gui.KeyedSeq.empty` are functions: write `Gui.Action.none()`. Roc now gives an annotated value one type, so a value that every application uses at its own state type has to be a function.
- `Gui.Audio.Output` and `Gui.Audio.Track` carry their operations as methods, like every other resource: `output.load!(directory, name)` and `track.play!()`, `pause!`, `seek!`, `status!` and `stop!`.
- Application headers and the examples archive name no compiler.

Targets

- `x64glibc`: Linux x86-64 (glibc)
- `arm64mac`: macOS on Apple silicon
- `x64mingw`: Windows x86-64

Every example and benchmark is rebuilt from the published bundle on each target, and every specification not named under Known issues runs there, before the release is published. Linux needs `libfreetype6`, `libxkbcommon0` and `libxkbcommon-x11-0` at run time.

Known issues

This candidate ships with four open defects. Each is tracked in the repository's backlog and is expected to close before `0.1.0`.

- Observatory does not build with the default `--opt=speed` backend: the compiler crashes or exhausts memory on every recent nightly. Build it with `--opt=dev`, as the release tests do.
- The compiler intermittently crashes while type-checking on Windows. A build that stops with "Access violation in the Roc compiler" succeeds when run again.
- Observatory's window Timeline specification fails on hosted macOS and Windows runners, where the window opens smaller than requested; it passes on Linux at 1920x1080.
- Observatory's folder-rewatch specification sometimes counts a transient change beside a capture it opens, so the release tests skip it.

Examples

The examples archive holds sixteen applications, from a two-counter demo to an HTTP client, a read-only SQLite browser, a terminal workspace, a Redis explorer, an image library, a music player, a system monitor, an animation editor and Observatory, which reads the performance captures the others record. Each carries its specifications beside it.

Verify what you downloaded

Every asset carries a GitHub build-provenance attestation:

```
gh attestation verify <asset> --repo lukewilliamboswell/roc-gui
```

`release-manifest.json` records the bundle's digest and size, the compiler that bundled it, and the release identity of every host archive and linker input inside it.