RustyJack Network Interface Selection & Isolation: Root Cause Analysis and Fix Plan

Prepared: January 10, 2026 | Audience: senior Rust developers

Executive summary

Hardware Detect → “Set Active Interface” currently conflates two different concerns:

1) selecting which interface should be the *only* admin-UP interface (isolation / policy), and
2) successfully obtaining a working L3 network configuration (carrier + DHCP + routes + DNS).

Because these are coupled, routine “disconnected” states (Ethernet with no carrier; Wi-Fi selected but not associated) are treated as *hard errors*. The result is exactly what you observed:
- eth0 selection fails immediately with “No carrier detected…”
- wlan0 selection “fails” because the UI waits for a condition that is not valid for Wi-Fi in the “selected but not connected yet” state, and because core code sometimes swallows bring-up errors.

Battle-tested managers (NetworkManager, systemd-networkd) explicitly model intermediate states like “no-carrier/unavailable” and “disconnected” instead of failing selection. This report maps that model onto RustyJack and proposes concrete changes (file/function/line-level) to make selection always succeed, keep the chosen interface admin-UP, and make Ethernet auto-connect on carrier events while keeping Wi-Fi manual-connect only.

Target behavior (requirements)

The product requirements implied by your report:

- When a user selects an interface in Hardware Detect, that interface becomes the *single selected* interface and all other interfaces are administratively down (isolation).
- Selecting Ethernet:
  - MUST immediately attempt to establish a network connection (DHCP + routes + DNS) if a carrier is present.
  - MUST NOT fail selection if there is no carrier or if DHCP fails.

- MUST keep the Ethernet interface admin-UP so that plugging in a cable later can transition to “connected” without re-selecting.
- Selecting Wi-Fi:
  - MUST bring the Wi-Fi interface admin-UP (and unblock rfkill if needed).
  - MUST NOT automatically connect/associate to a network.
  - Connecting to Wi-Fi is a separate explicit user action later.

Observed errors

You reported two UI-level failures:

• wlan0: "Interface: wlan0 Admin stat: down Operational: down Interface did not come UP ... Note: config saved"

• eth0: "internal error: failed to set interface 'eth0': failed to activate preferred interface: No carrier detected for eth0 - check cable connection"

Current flow (as implemented)

High-level call chain during Hardware Detect → Set Active Interface:

UI (rustyjack-ui)
  → daemon RPC (rustyjack-daemon)
    → core operation set_active_interface() (rustyjack-core/src/operations.rs)
      → IsolationEngine::enforce_passive() (rustyjack-core/src/system/isolation.rs)
        → IsolationEngine::enforce_with_mode(Passive)
          - bring_down + route cleanup for all non-selected interfaces
          - activate_interface() for the selected one

The word “passive” is misleading here: for Ethernet the “Passive” path still does carrier detection and DHCP, and treats “no carrier” and “DHCP failure” as fatal. For Wi-Fi the core does not require IP, but the UI assumes it must come “up” within 30s and shows a cable-related message even for Wi-Fi.

Key code references (current)

Core operation entry point (set_active_interface):

```
use std::{
    collections::{HashMap, VecDeque},
    env,
    fs,
    io::Write,
    net::{Ipv4Addr, SocketAddr, ToSocketAddrs},
    path::{Path, PathBuf},
    process::Command,
```

```
 9 |     time::{Duration, SystemTime, UNIX_EPOCH},
10 | };
11 |
12 | use anyhow::{anyhow, bail, Context, Result};
13 | use chrono::Local;
14 | use ipnet::Ipv4Net;
15 | use regex::Regex;
16 | use rustyjack_ethernet::{
17 |     build_device_inventory, connect_tcp_with_source, discover_hosts, discover_hosts_arp,
18 |     quick_port_scan, DeviceInfo, LanDiscoveryResult,
19 | };
20 | use rustyjack_evasion::{
21 |     MacAddress, MacGenerationStrategy, MacManager, MacMode, MacPolicyConfig, MacPolicyEngine,
22 |     MacStage, StableScope, VendorOui, VendorPolicy,
23 | };
24 | use rustyjack_portal::{start_portal, stop_portal, PortalConfig};
25 | use rustyjack_wireless::{
26 |     arp_scan, calculate_bandwidth, discover_gateway, discover_mdns_devices, get_traffic_stats,
27 |     capture_dns_queries, hotspot_disconnect_client, hotspot_set_blacklist, scan_network_services,
28 |     start_hotspot, status_hotspot, stop_hotspot, HotspotConfig, HotspotState,
29 | };
30 | use serde_json::{json, Map, Value};
31 |
32 | use walkdir::WalkDir;
33 | #[cfg(target_os = "linux")]
34 | use rustyjack_netlink::{TxPowerSetting, WirelessManager};
35 |
36 | use crate::cli::{
37 |     BridgeCommand, BridgeStartArgs, BridgeStopArgs, Commands, DiscordCommand, DiscordSendArgs,
38 |     DnsSpoofCommand, DnsSpoofStartArgs, EthernetCommand, EthernetDiscoverArgs,
39 |     EthernetInventoryArgs, EthernetPortScanArgs, EthernetSiteCredArgs, HardwareCommand,
40 |     HotspotBlacklistArgs, HotspotCommand, HotspotDisconnectArgs, HotspotStartArgs, LootCommand,
41 |     LootKind, LootListArgs, LootReadArgs,
42 |     MitmCommand, MitmStartArgs, NotifyCommand, ProcessCommand, ProcessKillArgs, ProcessStatusArgs,
43 |     ReverseCommand, ReverseLaunchArgs, ScanCommand, ScanDiscovery, ScanRunArgs,
44 |     StatusCommand, SystemCommand, SystemFdeMigrateArgs, SystemFdePrepareArgs, SystemUpdateArgs,
45 |     UsbMountArgs, UsbMountMode, UsbUnmountArgs, WifiBestArgs, WifiCommand, WifiCrackArgs,
46 |     WifiDeauthArgs, WifiDisconnectArgs, WifiEvilTwinArgs, WifiKarmaArgs, WifiMacRandomizeArgs,
47 |     WifiMacRestoreArgs, WifiMacSetArgs, WifiMacSetVendorArgs, WifiPmkidArgs, WifiProbeSniffArgs,
48 |     WifiProfileCommand, WifiProfileConnectArgs, WifiProfileDeleteArgs, WifiProfileSaveArgs,
49 |     WifiProfileShowArgs,
50 |     WifiReconArpScanArgs, WifiReconBandwidthArgs, WifiReconCommand, WifiReconDnsCaptureArgs,
51 |     WifiReconGatewayArgs, WifiReconMdnsScanArgs, WifiReconServiceScanArgs, WifiRouteCommand,
52 |     WifiRouteEnsureArgs, WifiRouteMetricArgs, WifiScanArgs, WifiStatusArgs, WifiSwitchArgs,
53 |     WifiTxPowerArgs,
54 | };
55 | #[cfg(target_os = "linux")]
56 | use crate::netlink_helpers::netlink_set_interface_up;
57 | use crate::anti_forensics::perform_complete_purge;
58 | use crate::mount::{MountMode, MountPolicy, MountRequest, UnmountRequest};
59 | use crate::system::{
```

```
60 |     active_uplink, acquire_dhcp_lease, append_payload_log, arp_spoof_running, backup_repository,
61 |     backup_routing_state, build_scan_loot_path, build_manual_embed, build_mitm_pcap_path,
62 |     cached_gateway, compose_status_text, connect_wifi_network, default_gateway_ip,
63 |     delete_wifi_profile, detect_ethernet_interface, detect_interface, disconnect_wifi_interface,
64 |     dns_spoof_running, enable_ip_forwarding, enforce_single_interface, find_interface_by_mac,
65 |     git_reset_to_remote, interface_gateway, kill_process,
66 |     last_dhcp_outcome, lease_record, list_interface_summaries, list_wifi_profiles,
67 |     load_wifi_profile, log_mac_usage, pcap_capture_running, ping_host, preferred_interface,
68 |     process_running_exact, randomize_hostname, read_default_route, read_discord_webhook,
69 |     read_dns_servers, read_interface_preference, read_interface_preference_with_mac,
70 |     read_interface_stats, read_wifi_link_info, restart_system_service, restore_routing_state,
71 |     sanitize_label, save_wifi_profile, scan_local_hosts, scan_wifi_networks,
72 |     select_active_uplink, select_best_interface, select_wifi_interface, send_discord_payload,
73 |     send_scan_to_discord, set_interface_metric, spawn_arpspoof_pair, start_bridge_pair,
74 |     start_dns_spoof, start_pcap_capture, stop_arp_spoof, stop_bridge_pair, stop_dns_spoof,
75 |     stop_pcap_capture, write_interface_preference, write_wifi_profile, HostInfo, KillResult,
```

Isolation engine activation path (activate_interface + Passive Ethernet behavior):

```
296 |         }
297 |
298 |         warn!("No operational interfaces found");
299 |         Ok(None)
300 |     }
301 |
302 |     fn activate_interface(&self, iface: &str, mode: EnforcementMode) -> Result<()> {
303 |         info!("Activating interface: {} ({:?})", iface, mode);
304 |
305 |         // Check interface exists before starting
306 |         if !self.ops.interface_exists(iface) {
307 |             bail!("Interface {} does not exist", iface);
308 |         }
309 |
310 |         if let Err(e) = self.ops.bring_up(iface) {
311 |             // Check if interface disappeared
312 |             if !self.ops.interface_exists(iface) {
313 |                 bail!("Interface {} disappeared during activation", iface);
314 |             }
315 |             warn!("Interface {} may already be up: {}", iface, e);
316 |         }
317 |
318 |         let is_wireless = self.ops.is_wireless(iface);
319 |
320 |         if is_wireless {
321 |             self.ops
322 |                 .set_rfkill_block(iface, false)
323 |                 .context("failed to unblock rfkill")?;
324 |
325 |             if mode == EnforcementMode::Passive {
326 |                 self.ops
327 |                     .apply_nm_managed(iface, false)
328 |                     .context("failed to set NM unmanaged")?;
329 |                 info!("Interface {} activated (passive, no DHCP)", iface);
330 |                 return Ok(());
```

```
331 |         }
332 |
333 |         // For wireless interfaces, check if we're connected to an AP
334 |         // If not, try to auto-connect using a saved profile
335 |         if self.ops.get_ipv4_address(iface).ok().flatten().is_none() {
336 |             info!("Wireless interface {} has no IP - checking for auto-connect profiles", iface);
337 |
338 |             match self.try_auto_connect_wifi(iface) {
339 |                 Ok(true) => {
340 |                     info!("Auto-connected to WiFi network on {}", iface);
341 |                     // WiFi connection includes DHCP, so we're done
342 |                     return Ok(());
343 |                 }
344 |                 Ok(false) => {
345 |                     info!("No auto-connect profile found for {}, will try DHCP anyway", iface);
346 |                 }
347 |                 Err(e) => {
348 |                     warn!("Auto-connect failed for {}: {}", iface, e);
349 |                     info!("Attempting DHCP anyway (may fail if not connected to AP)");
350 |                 }
351 |             }
352 |         }
353 |     }
354 |
355 |     self.ops
356 |         .apply_nm_managed(iface, false)
357 |         .context("failed to set NM unmanaged")?;
358 |
359 |     if mode == EnforcementMode::Passive {
360 |         if !is_wireless {
361 |             // Ethernet interface - retry DHCP acquisition with carrier detection
362 |             // Carrier check first to fail fast if no cable (~1s vs 15-25s)
363 |
364 |             // Pre-check: verify physical link/carrier is detected
365 |             let carrier_detected = self.interface_has_carrier(iface);
366 |             if !carrier_detected {
367 |                 error!("No carrier detected on {} - cable not plugged in?", iface);
368 |                 bail!("No carrier detected on {} - check cable connection", iface);
369 |             }
370 |
371 |             // Cable is plugged in - now retry DHCP in case of transient issues
372 |             // (slow DHCP server, switch STP delays, etc.)
373 |             const MAX_RETRIES: usize = 3;  // 3 retries × 5s = 15s total max
374 |             const RETRY_DELAY_SECS: u64 = 5;
375 |
376 |             let mut last_error = None;
377 |             let mut lease_acquired = false;
378 |
379 |             for attempt in 1..=MAX_RETRIES {
380 |                 info!("Attempting DHCP for {} (attempt {}/{})", iface, attempt, MAX_RETRIES);
381 |
382 |                 match self.ops.acquire_dhcp(iface, Duration::from_secs(30)) {
383 |                     Ok(lease) => {
384 |                         info!(
385 |                             "DHCP successful for {} on attempt {}: ip={}, gateway={:?}",
```

```
                    iface, attempt, lease.ip, lease.gateway
                );

                if let Some(gw) = lease.gateway {
                    let metric = if is_wireless { 200 } else { 100 };
                    self.routes
                        .set_default_route(iface, gw, metric)
                        .context("failed to set default route")?;
                } else {
                    warn!("No gateway in DHCP lease - link-local only");
                }

                if !lease.dns_servers.is_empty() {
                    self.dns
                        .set_dns(&lease.dns_servers)
                        .context("failed to set DNS")?;
                } else {
                    warn!("No DNS in DHCP lease, using fallback");
                    self.dns
                        .set_dns(&[
                            Ipv4Addr::new(1, 1, 1, 1),
                            Ipv4Addr::new(9, 9, 9, 9),
                        ])
                        .context("failed to set fallback DNS")?;
                }

                lease_acquired = true;
                break;
            }
            Err(e) => {
                last_error = Some(e);

                if attempt < MAX_RETRIES {
                    warn!(
                        "DHCP attempt {}/{} failed for {}: {}. Retrying in {}s...",
                        attempt, MAX_RETRIES, iface, last_error.as_ref().unwrap(), RETRY_DELAY_SECS
                    );
                    std::thread::sleep(Duration::from_secs(RETRY_DELAY_SECS));
                } else {
                    // All retries exhausted
                    error!(
                        "DHCP failed for {} after {} attempts: {}",
                        iface, MAX_RETRIES, last_error.as_ref().unwrap()
                    );
                }
            }
        }
    }

    if !lease_acquired {
        bail!(
            "Failed to acquire DHCP lease for {} after {} attempts: {}",
            iface,
            MAX_RETRIES,
            last_error.unwrap()
```

```
                );
            }
        }
        info!("Interface {} activated (passive)", iface);
        return Ok(());
    }

    // Try DHCP with graceful fallback
    match self.ops.acquire_dhcp(iface, Duration::from_secs(30)) {
        Ok(lease) => {
            info!(
                "DHCP lease acquired: ip={}, gateway={:?}",
                lease.ip, lease.gateway
            );

            if let Some(gw) = lease.gateway {
                let metric = if self.ops.is_wireless(iface) { 200 } else { 100 };
                self.routes
                    .set_default_route(iface, gw, metric)
                    .context("failed to set default route")?;
            } else {
                warn!("No gateway in DHCP lease - link-local only");
            }

            if !lease.dns_servers.is_empty() {
                self.dns
                    .set_dns(&lease.dns_servers)
                    .context("failed to set DNS")?;
            } else {
                warn!("No DNS in DHCP lease, using fallback");
                self.dns
                    .set_dns(&[Ipv4Addr::new(1, 1, 1, 1), Ipv4Addr::new(9, 9, 9, 9)])
                    .context("failed to set fallback DNS")?;
            }
        }
        Err(e) => {
            warn!("DHCP failed for {}: {}", iface, e);
            warn!("Continuing with manual IP configuration (if present)");

            // Set fallback DNS even without DHCP
            self.dns
                .set_dns(&[Ipv4Addr::new(1, 1, 1, 1), Ipv4Addr::new(9, 9, 9, 9)])
                .context("failed to set fallback DNS")?;

            // Check if interface has manual IP - if not, this is a failure
            if self.ops.get_ipv4_address(iface).ok().flatten().is_none() {
                bail!("DHCP failed and no manual IP configured");
            }

            info!("Interface has manual IP, continuing without DHCP");
        }
    }

    info!("Interface {} fully activated", iface);
    Ok(())
```

```
496 |     }
497 |
498 |     fn try_auto_connect_wifi(&self, iface: &str) -> Result<bool> {
499 |         use crate::system::{list_wifi_profiles, load_wifi_profile, connect_wifi_network};
500 |
501 |         // Load all profile records
502 |         let profile_records = match list_wifi_profiles(&self.root) {
503 |             Ok(p) => p,
504 |             Err(e) => {
505 |                 debug!("Failed to list WiFi profiles: {}", e);
506 |                 return Ok(false);
507 |             }
508 |         };
509 |
510 |         if profile_records.is_empty() {
511 |             debug!("No WiFi profiles found");
512 |             return Ok(false);
513 |         }
514 |
515 |         // Filter to auto-connect profiles matching this interface
516 |         let mut candidates: Vec<_> = profile_records
517 |             .into_iter()
518 |             .filter(|p| {
519 |                 p.auto_connect && (p.interface == iface || p.interface == "auto")
520 |             })
```

Carrier helper (note the differing semantics vs system/mod.rs):

```
560 |     }
561 |
562 |     fn interface_has_carrier(&self, iface: &str) -> bool {
563 |         // Check if physical link/carrier is detected on the interface
564 |         // Returns true if carrier = 1 (cable plugged in) or if carrier file doesn't exist
565 |         let carrier_path = format!("/sys/class/net/{}/carrier", iface);
566 |         match fs::read_to_string(&carrier_path) {
567 |             Ok(val) => {
568 |                 let carrier = val.trim();
569 |                 carrier == "1"  // 1 = carrier detected (cable plugged in)
570 |             }
571 |             Err(_) => {
572 |                 // If carrier file doesn't exist, assume interface might work
573 |                 // (some virtual/wireless interfaces don't have carrier detection)
574 |                 true
575 |             }
576 |         }
577 |     }
578 |
579 |     fn block_interface(&self, iface: &str) -> Result<()> {
580 |         debug!("Blocking interface: {}", iface);
581 |
582 |         // Delete all routes for this interface
583 |         if let Err(e) = self.routes.delete_default_route(iface) {
584 |             debug!("No default route to delete for {}: {}", iface, e);
```

```
585 |         }
586 |
```

UI selection flow and error generation:

```
4166 |         self.config.settings.discord_enabled = !self.config.settings.discord_enabled;
4167 |         self.save_config()?;
4168 |         // No message needed as the menu label will update immediately
4169 |         Ok(())
4170 |     }
4171 |
4172 |     fn show_hardware_detect(&mut self) -> Result<()> {
4173 |         self.show_progress("Hardware Scan", ["Detecting interfaces...", "Please wait"])?;
4174 |
4175 |         match self
4176 |             .core
4177 |             .dispatch(Commands::Hardware(HardwareCommand::Detect))
4178 |         {
4179 |             Ok((_, data)) => {
4180 |                 let eth_count = data
4181 |                     .get("ethernet_count")
4182 |                     .and_then(|v| v.as_u64())
4183 |                     .unwrap_or(0);
4184 |                 let wifi_count = data.get("wifi_count").and_then(|v| v.as_u64()).unwrap_or(0);
4185 |                 let other_count = data
4186 |                     .get("other_count")
4187 |                     .and_then(|v| v.as_u64())
4188 |                     .unwrap_or(0);
4189 |
4190 |                 let ethernet_ports = data
4191 |                     .get("ethernet_ports")
4192 |                     .and_then(|v| v.as_array())
4193 |                     .cloned()
4194 |                     .unwrap_or_default();
4195 |                 let wifi_modules = data
4196 |                     .get("wifi_modules")
4197 |                     .and_then(|v| v.as_array())
4198 |                     .cloned()
4199 |                     .unwrap_or_default();
4200 |
4201 |                 // Build list of detected interfaces (clickable)
4202 |                 let mut all_interfaces = Vec::new();
4203 |                 let mut labels = Vec::new();
4204 |
4205 |                 let active_interface = self.config.settings.active_network_interface.clone();
4206 |
4207 |                 for port in &ethernet_ports {
4208 |                     if let Some(name) = port.get("name").and_then(|v| v.as_str()) {
4209 |                         let label = if name == active_interface {
4210 |                             format!("* {}", name)
4211 |                         } else {
4212 |                             name.to_string()
4213 |                         };
4214 |                         labels.push(label);
4215 |                         all_interfaces.push(port.clone());
4216 |                     }
4217 |                 }
4218 |                 for module in &wifi_modules {
4219 |                     if let Some(name) = module.get("name").and_then(|v| v.as_str()) {
```

```
                    let label = if name == active_interface {
                        format!("* {}", name)
                    } else {
                        name.to_string()
                    };
                    labels.push(label);
                    all_interfaces.push(module.clone());
                }
            }

            // If nothing to show, just present summary
            if all_interfaces.is_empty() {
                let summary_lines = vec![
                    format!("Ethernet: {}", eth_count),
                    format!("WiFi: {}", wifi_count),
                    format!("Other: {}", other_count),
                ];
                self.show_message(
                    "Hardware Detected",
                    summary_lines.iter().map(|s| s.as_str()),
                )?;
            } else {
                // Present clickable list and show details on selection
                loop {
                    let Some(idx) = self.choose_from_menu("Detected interfaces", &labels)?
                    else {
                        break;
                    };

                    let info = &all_interfaces[idx];
                    let interface_name = info
                        .get("name")
                        .and_then(|v| v.as_str())
                        .unwrap_or("unknown")
                        .to_string();

                    // Build detail lines
                    let mut details = Vec::new();
                    details.push(format!("Name: {}", interface_name));
                    if let Some(kind) = info.get("kind").and_then(|v| v.as_str()) {
                        details.push(format!("Kind: {}", kind));
                    }
                    if let Some(state) = info.get("oper_state").and_then(|v| v.as_str()) {
                        details.push(format!("State: {}", state));
                    }
                    if let Some(ip) = info.get("ip").and_then(|v| v.as_str()) {
                        details.push(format!("IP: {}", ip));
                    }
                    details.push("".to_string());
                    details.push("[OK] Set Active".to_string());

                    self.display.draw_menu(
                        "Interface details",
                        &details,
                        usize::MAX,
                        &self.stats.snapshot(),
                    )?;
                    // Wait for action
                    loop {
```

```
                let btn = self.buttons.wait_for_press()?;
                match self.map_button(btn) {
                    ButtonAction::Select => {
                        self.show_progress(
                            "Setting Interface",
                            &[
                                &format!("Activating {}", interface_name),
                                "Blocking others...",
                                "Please wait",
                            ],
                        )?;

                        let mut lines = Vec::new();
                        match self.core.set_active_interface(&interface_name) {
                            Ok(data) => {
                                let allowed = data["allowed"]
                                    .as_array()
                                    .map(|a| {
                                        a.iter()
                                            .filter_map(|v| v.as_str())
                                            .map(String::from)
                                            .collect::<Vec<_>>()
                                    })
                                    .unwrap_or_default();
                                let blocked = data["blocked"]
                                    .as_array()
                                    .map(|b| {
                                        b.iter()
                                            .filter_map(|v| v.as_str())
                                            .map(String::from)
                                            .collect::<Vec<_>>()
                                    })
                                    .unwrap_or_default();

                                self.config.settings.active_network_interface =
                                    interface_name.clone();
                                let config_path = self.root.join("gui_conf.json");
                                let config_msg = match self.config.save(&config_path) {
                                    Ok(()) => Some("Config saved".to_string()),
                                    Err(e) => {
                                        Some(format!("Config save failed: {}", e))
                                    }
                                };

                                self.show_progress(
                                    "Confirming Interface",
                                    [
                                        format!("Waiting for {}", interface_name),
                                        "Acquiring DHCP...".to_string(),
                                    ],
                                )?;
                                let is_up = self.wait_for_interface_up_with_ip(
                                    &interface_name,
                                    Duration::from_secs(30),  // Carrier check fails fast; ~15s for DHCP retries
                                );
                                if !is_up {
                                    let state = if self.interface_admin_up(&interface_name)
                                    {
                                        "up"
```

```
                        } else {
                            "down"
                        };
                        let oper_state = self.interface_oper_state(&interface_name);
                        let mut err_lines = Vec::new();
                        err_lines.push(format!(
                            "Interface: {}",
                            interface_name
                        ));
                        err_lines.push(format!("Admin State: {}", state));
                        err_lines.push(format!(
                            "Operational: {}",
                            oper_state
                        ));

                        // Provide helpful error context
                        if state == "down" {
                            err_lines.push("Interface did not come UP".to_string());
                            err_lines.push("Check hardware/cable".to_string());
                        } else if oper_state == "down" {
                            err_lines.push("Interface UP but DHCP failed".to_string());
                            err_lines.push("or no carrier detected".to_string());
                        } else {
                            err_lines.push("Timeout waiting for full setup".to_string());
                        }

                        if let Some(msg) = config_msg.clone() {
                            err_lines.push(format!("Note: {}", msg));
                        }
                        self.show_message(
                            "Interface Setup Failed",
                            err_lines.iter().map(|s| s.as_str()),
                        )?;
                    } else {
                        if let Some(msg) = config_msg {
                            lines.push(msg);
                        }
                        if allowed.is_empty() {
                            lines.push(format!(
                                "Active: {}",
                                interface_name
                            ));
                        } else {
                            lines.push(format!(
                                "Active: {}",
                                allowed.join(", ")
                            ));
                        }
                        if !blocked.is_empty() {
                            lines.push(format!(
                                "Blocked: {}",
                                blocked.join(", ")
                            ));
                        }
                        lines.push(
                            "Connect via Saved Networks".to_string(),
                        );

                        self.show_message(
```

```
                                    "Active Interface",
                                    lines.iter().map(|s| s.as_str()),
                                )?;
                            }
                        }
                        Err(e) => {
                            // Show full error chain for better debugging
                            let error_msg = format!("{:#}", e);
                            self.show_message("Interface Error", [error_msg])?;
                        }
                    }
                    // Refresh the labels to show new active indicator
                    labels.clear();
                    all_interfaces.clear();
                    let active =
                        self.config.settings.active_network_interface.clone();
                    for port in &ethernet_ports {
                        if let Some(name) =
                            port.get("name").and_then(|v| v.as_str())
                        {
                            let label = if name == active {
                                format!("* {}", name)
                            } else {
                                name.to_string()
                            };
                            labels.push(label);
                            all_interfaces.push(port.clone());
                        }
                    }
                    for module in &wifi_modules {
                        if let Some(name) =
                            module.get("name").and_then(|v| v.as_str())
                        {
                            let label = if name == active {
                                format!("* {}", name)
                            } else {
                                name.to_string()
                            };
                            labels.push(label);
                            all_interfaces.push(module.clone());
                        }
                    }
                    break;
                }
                ButtonAction::Back => break,
                ButtonAction::MainMenu => {
                    self.menu_state.home();
                    break;
                }
                ButtonAction::Reboot => {
                    self.confirm_reboot()?;
                }
                _ => {}
            }
        }
    }
}
Err(err) => {
```

```
4456 |                 let msg = vec![format!("Scan failed: {}", err)];
4457 |                 self.show_message("Hardware Error", msg.iter().map(|s| s.as_str()))?;
4458 |             }
4459 |         }
4460 |         Ok(())
4461 |     }
4462 |
4463 |     fn toggle_dns_spoof(&mut self) -> Result<()> {
4464 |         // Check current status
4465 |         let is_running = self.status_overlay().dns_spoof_running;
4466 |
4467 |         if is_running {
4468 |             self.stop_dns_spoof()
4469 |         } else {
4470 |             self.start_dns_spoof()
```

Daemon interface status calculation (is_up derived from operstate):

```
508 |                     carrier: None,
509 |                     ip: None,
510 |                     capabilities: None,
511 |                 });
512 |             }
513 |
514 |             let oper_state = fs::read_to_string(sys_path.join("operstate"))
515 |                 .unwrap_or_else(|_| "unknown".to_string())
516 |                 .trim()
517 |                 .to_string();
518 |             let is_up = oper_state == "up";
519 |             let carrier = fs::read_to_string(sys_path.join("carrier"))
520 |                 .ok()
521 |                 .and_then(|val| match val.trim() {
522 |                     "0" => Some(false),
523 |                     "1" => Some(true),
524 |                     _ => None,
525 |                 });
526 |             let is_wireless = sys_path.join("wireless").exists();
527 |
528 |             let ops = RealNetOps;
529 |             let ip = ops
530 |                 .get_ipv4_address(iface)
531 |                 .ok()
532 |                 .flatten()
533 |                 .map(|addr| addr.to_string());
534 |
535 |             let capabilities = ops.get_interface_capabilities(iface).ok().map(|caps| {
536 |                 InterfaceCapabilities {
537 |                     is_wireless: caps.is_wireless,
538 |                     is_physical: caps.is_physical,
539 |                     supports_monitor: caps.supports_monitor,
540 |                     supports_ap: caps.supports_ap,
541 |                     supports_injection: caps.supports_injection,
542 |                     supports_5ghz: caps.supports_5ghz,
543 |                     supports_2ghz: caps.supports_2ghz,
544 |                     mac_address: caps.mac_address,
545 |                     driver: caps.driver,
```

```
                    chipset: caps.chipset,
                }
            });

            Ok(InterfaceStatusResponse {
                interface: iface.to_string(),
                exists,
                is_wireless,
                oper_state,
                is_up,
                carrier,
                ip,
                capabilities,
            })
        })
        .await;

        match result {
            Ok(status) => ResponseBody::Ok(ResponseOk::InterfaceStatus(status)),
            Err(err) => ResponseBody::Err(err),
        }
```

Daemon “netlink_watcher” (currently a one-shot dump, not a subscription):

```
use std::sync::Arc;
use std::time::{Duration, Instant};
use tokio::sync::Mutex;
use tokio::time::sleep;
use tracing::{debug, info, warn};

use crate::state::DaemonState;

#[cfg(target_os = "linux")]
pub async fn run_netlink_watcher(state: Arc<DaemonState>) -> anyhow::Result<()> {
    info!("Starting netlink watcher for hardware isolation enforcement");

    let last_event: Arc<Mutex<Option<Instant>>> = Arc::new(Mutex::new(None));
    let debounce_duration = Duration::from_millis(250);

    loop {
        match watch_netlink_events(Arc::clone(&state), Arc::clone(&last_event), debounce_duration).await {
            Ok(_) => {
                info!("Netlink watcher stopped normally");
                break;
            }
            Err(e) => {
                warn!("Netlink watcher error: {}, restarting in 5s", e);
                sleep(Duration::from_secs(5)).await;
            }
        }
    }

    Ok(())
}

```

```rust
#[cfg(not(target_os = "linux"))]
pub async fn run_netlink_watcher(_state: Arc<DaemonState>) -> anyhow::Result<()> {
    info!("Netlink watcher disabled on non-Linux platform");
    futures::future::pending::<()>().await;
    Ok(())
}

#[cfg(target_os = "linux")]
async fn watch_netlink_events(
    state: Arc<DaemonState>,
    last_event: Arc<Mutex<Option<Instant>>>,
    debounce_duration: Duration,
) -> anyhow::Result<()> {
    use futures::stream::StreamExt;
    use rtnetlink::new_connection;

    let (connection, handle, _) = new_connection()?;
    tokio::spawn(connection);

    let mut link_stream = handle.link().get().execute();
    let mut address_stream = handle.address().get().execute();

    loop {
        enum Event { Link, Address, End }

        let event = tokio::select! {
            biased;
            link_result = link_stream.next() => {
                if link_result.is_some() { Event::Link } else { Event::End }
            }
            addr_result = address_stream.next() => {
                if addr_result.is_some() { Event::Address } else { Event::End }
            }
        };

        match event {
            Event::Link => {
                debug!("Netlink link event");
                schedule_enforcement(Arc::clone(&state), Arc::clone(&last_event), debounce_duration).await;
            }
            Event::Address => {
                debug!("Netlink address event");
                schedule_enforcement(Arc::clone(&state), Arc::clone(&last_event), debounce_duration).await;
            }
            Event::End => {
                debug!("Netlink stream ended");
                break;
            }
        }
    }

    Ok(())
}

#[cfg(target_os = "linux")]
```

```
async fn schedule_enforcement(
    state: Arc<DaemonState>,
    last_event: Arc<Mutex<Option<Instant>>>,
    debounce_duration: Duration,
) {
    let now = Instant::now();

    {
        let mut last = last_event.lock().await;
        if let Some(prev) = *last {
            if now.duration_since(prev) < debounce_duration {
                *last = Some(now);
                return;
            }
        }
        *last = Some(now);
    }

    let state_clone = Arc::clone(&state);
    tokio::spawn(async move {
        sleep(debounce_duration).await;

        let _lock = state_clone.locks.acquire_uplink().await;

        let root = state_clone.config.root_path.clone();
        tokio::task::spawn_blocking(move || {
            use rustyjack_core::system::{IsolationEngine, RealNetOps};
            use std::sync::Arc;

            let ops = Arc::new(RealNetOps);
            let engine = IsolationEngine::new(ops, root);

            match engine.enforce() {
                Ok(outcome) => {
                    info!("Netlink event enforcement: allowed={:?}, blocked={:?}",
                        outcome.allowed, outcome.blocked);
                    if !outcome.errors.is_empty() {
                        warn!("Enforcement had {} errors:", outcome.errors.len());
                        for err in &outcome.errors {
                            warn!("  {}: {}", err.interface, err.message);
                        }
                    }
                }
                Err(e) => {
```

NetworkManager integration is currently a no-op (so NM can race/override isolation):

```

    fn is_wireless(&self, interface: &str) -> bool {
        use std::path::Path;
        Path::new("/sys/class/net")
            .join(interface)
            .join("wireless")
            .exists()
    }
```

```
208 |
209 |     fn interface_exists(&self, interface: &str) -> bool {
210 |         use std::path::Path;
211 |         Path::new("/sys/class/net").join(interface).exists()
212 |     }
213 |
214 |     fn apply_nm_managed(&self, interface: &str, managed: bool) -> Result<()> {
215 |         let _ = (interface, managed);
216 |         Ok(())
217 |     }
218 |
219 |     fn add_default_route(&self, iface: &str, gateway: Ipv4Addr, metric: u32) -> Result<()> {
220 |         use std::net::IpAddr;
221 |         crate::netlink_helpers::netlink_add_default_route(IpAddr::V4(gateway), iface, Some(metric))
222 |     }
223 |
224 |     fn delete_default_route(&self, _iface: &str) -> Result<()> {
225 |         crate::netlink_helpers::netlink_delete_default_route()
226 |     }
227 |
228 |     fn list_routes(&self) -> Result<Vec<RouteEntry>> {
229 |         use crate::netlink_helpers::netlink_list_routes;
230 |
231 |         // Refresh interface cache before querying routes
232 |         self.refresh_interface_cache().ok();
233 |
234 |         let routes = netlink_list_routes()?;
235 |
```

Root cause analysis

RC1 — Selection is treated as success-only-if-connected

The isolation engine's "activation" step returns Result<()> and treats "no carrier" and "DHCP failed" as *errors* in multiple branches.

Concrete failure:
- set_active_interface() uses enforce_passive() unconditionally.
- In activate_interface(), the Passive + Ethernet branch checks carrier and returns Err("No carrier detected...") if carrier==0.
- That error propagates up through enforce_with_mode() and becomes the UI error you saw for eth0.

This violates the required model: "selected interface" is a policy decision and must succeed even when the link is physically disconnected. Link state should be reported as a status, not an error.

RC2 — bring_up errors are swallowed, allowing wlan0 to remain admin-DOWN

In activate_interface(), bring_up() errors are caught and only logged as a warning:

```
if let Err(e) = self.ops.bring_up(iface) { warn!(...); }
```

Because the error is ignored, the function can proceed and ultimately return Ok(()) even if the interface never became IFF_UP. The UI then polls /sys/class/net/<iface>/flags and eventually errors with “Admin stat: down … did not come UP”.

Any error from RTM_SETLINK to set IFF_UP is meaningful (permissions, driver state, rfkill hard-block, interface disappearing). If we want “best effort”, we must still verify the post-condition and return a structured failure when admin-UP cannot be achieved.

RC3 — Passive mode is asymmetric (Wi-Fi: no connect; Ethernet: strict carrier + DHCP)

The “Passive” mode is used for “set active interface”, but it is not actually “passive” for Ethernet:
- It enforces carrier precondition and requires DHCP success (or manual IP) even in Passive mode.
- For Wi-Fi it returns early (no DHCP), which is correct for your requirement, but the mismatch means the mode name (and caller expectations) are wrong.

This is a design smell: the activation policy should be expressed explicitly per interface type, not by overloading a global “mode” enum.

RC4 — The daemon reports interface up/down using operstate (wrong signal)

In rustyjack-daemon/src/dispatch.rs, InterfaceStatusResponse::is_up is computed as:

```
let is_up = oper_state.trim() == "up";
```

This is incorrect. Kernel operstate is an RFC2863-style operational state that depends on lower-layer conditions (carrier, dormant, etc.). An interface can be administratively UP but operstate “down” (common for Ethernet with no carrier; also common for Wi-Fi before association). Using operstate conflates “admin up” with “link usable”.

This matters because:
- It causes UI/clients to interpret an admin-UP but disconnected interface as “down”.
- It encourages logic that fails selection when the link is merely disconnected.

The correct admin signal is IFF_UP. Carrier is IFF_LOWER_UP (or sysfs carrier).

RC5 — The netlink watcher is not actually watching (no RTNLGRP subscriptions)

rustyjack-daemon/src/netlink_watcher.rs uses:

```
handle.link().get().execute()
```

That stream is a dump of current links and then completes; it does not subscribe to multicast notifications. As a result, plugging/unplugging an Ethernet cable after selection will not trigger a DHCP retry or route/DNS updates.

Kernel guidance is explicit: subscribe to RTNLGRP_LINK to receive updates (and you only receive them while the interface is admin-UP). The project already depends on rtnetlink; it just needs to wire up the actual message stream and group membership.

RC6 — "NetworkManager integration" is stubbed out, risking interference

IsolationEngine calls ops.apply_nm_managed(iface, false) to prevent NetworkManager from reconfiguring interfaces and breaking "only one interface up".

But RealNetOps::apply_nm_managed() is currently a no-op.

If NetworkManager (or another DHCP/connection manager) is running on the target OS, it can:
- bring blocked interfaces back up (autoconnect),
- race DHCP lease acquisition (port 68 bind conflicts),
- rewrite routes and /etc/resolv.conf.

Even if NM isn't installed today, the code is *written* as if it is controlling NM, which hides the real constraints and makes failures non-deterministic across OS images.

Reference behavior: NetworkManager and the kernel state model

Kernel: admin vs operational vs carrier

Linux explicitly distinguishes:
- Administrative state (IFF_UP; what the admin requested),
- Lower-layer carrier (IFF_LOWER_UP; driver signals physical link),
- Operational state (RFC2863 operstate; derived and policy-influenced).

A manager that wants "selected interface stays ready even when unplugged" should keep the interface administratively UP and treat lack of carrier as a *normal state* (not an error). The kernel also notes that subscribing to RTNLGRP_LINK only delivers updates while the interface is admin-UP, which matches your "ready when cable is connected" requirement.

Primary reference: Linux kernel documentation on operational states (admin vs carrier vs operstate). [KERN-OPER]

NetworkManager: device states explicitly model 'no carrier' as 'unavailable'

NetworkManager models device lifecycle with a state machine, including:
- UNMANAGED: recognized but not managed

- UNAVAILABLE: managed but not usable (explicitly includes “no ethernet carrier” among reasons)
- DISCONNECTED: usable but idle (not connected)
- … PREPARE/CONFIG/IP_CONFIG …
- ACTIVATED: connected

This is the key design point: the device being “not available right now” is not treated as an exceptional failure to select or manage the device. It is a first-class state with well-defined transitions on carrier/link events.

Primary reference: NetworkManager libnm ‘NMDeviceState’ enum documentation. [NM-STATE]

Proposed design for RustyJack

Core idea: separate 'selection' from 'connectivity'

Split the current monolithic “activate preferred interface” into two layers:

1) Selection/Isolation (policy):
  - Persist user choice as the preferred interface.
  - Bring chosen interface admin-UP.
  - Bring all other interfaces admin-DOWN + rfkill block for Wi-Fi.
  - Do not require carrier, DHCP, or a default route.

2) Connectivity (best effort, event-driven):
  - If the selected interface is Ethernet and carrier is present → attempt DHCP immediately.
  - If carrier is absent → keep interface admin-UP and wait for RTNLGRP_LINK events.
  - On carrier up → attempt DHCP, configure routes and DNS.
  - DHCP failure is recorded as “Disconnected/DHCP failed” but does not revoke selection.

Wi-Fi selection stops at step (1). Wi-Fi connectivity is a distinct explicit operation (connect_wifi_*).

State model (NetworkManager-inspired)

Introduce an internal state enum for the *selected* device, persisted/observable via IPC:

- Selected { admin_up: true } (baseline; no assumptions about link)
- Unavailable { reason: NoCarrier | Rfkill | MissingFirmware | … } (selected but L2 not ready)
- Disconnected { reason: NoLease | UserDisconnected | … } (L2 ready; no L3)
- Configuring { method: Dhcp } (L3 in progress)
- Activated { ip, gateway, dns } (ready)

For Ethernet: Selected → (carrier? yes) Configuring → Activated OR Disconnected(DhcpFailed)
For Ethernet with no cable: Selected → Unavailable(NoCarrier) and wait for carrier events.
For Wi-Fi: Selected → Disconnected (until user explicitly connects, then Configuring/Activated).

Non-goals / explicit constraints

- Do not shell out to nmcli/ifconfig/ip; use netlink/sysfs directly.
- Avoid requiring NetworkManager/systemd-networkd/dhcpcd at runtime; if they are present, either disable them during install or explicitly mark devices unmanaged to avoid races.
- Keep the "only one interface admin-UP" invariant across all enforcement paths.

Fix plan: concrete code changes

1) rustyjack-core: make set_active_interface selection-only + best-effort Ethernet DHCP

File: rustyjack-core/src/operations.rs
Function: set_active_interface()

Current: writes preferred interface then calls isolation.enforce_passive() which can error on "no carrier".
Fix: call a new API, e.g. isolation.enforce_selection(&ifname), that guarantees:
- returns Ok if isolation and admin-UP succeeded,
- returns a structured outcome containing connection status/warnings (not fatal errors).

Implementation sketch (new API surface):

```
// rustyjack-core/src/system/isolation.rs
pub enum EnforcementMode {
    Selection,          // isolation + admin-up; never requires connectivity
    ConnectivityBestEffort, // tries DHCP/routes; never fails on no-carrier or DHCP
    StrictConnectivity, // optional: fail if carrier+DHCP cannot be achieved
}

pub struct ActivationReport {
    pub admin_up: bool,
    pub carrier: Option<bool>,      // None for unknown / unsupported
    pub ipv4: Option<std::net::Ipv4Addr>,
    pub dhcp: DhcpReport,           // { NotAttempted | Succeeded | Failed(String) }
    pub notes: Vec<String>,
}

pub struct IsolationOutcomeV2 {
    pub selected: String,
    pub blocked: Vec<String>,
    pub activation: ActivationReport,
```

```
    pub warnings: Vec<String>,
}
```

Minimal-change option (avoid IPC churn): keep the existing SetActiveInterfaceResponse as-is, but:
- never return an Err from set_active_interface() for NoCarrier/DHCPFailed,
- instead populate response.errors with warnings, and have the UI query InterfaceStatus right after.
That keeps wire format stable but still fixes the UX and invariants.

2) rustyjack-core: stop swallowing bring_up errors; verify post-conditions

File: rustyjack-core/src/system/isolation.rs
Function: activate_interface()

Fixes:
- Treat bring_up failure as fatal unless we can prove the interface is already admin-UP.
- For Wi-Fi, unblock rfkill *before* attempting bring_up, and retry bring_up once after unblocking.
- After bring_up, read IFF_UP (via netlink or /sys/class/net/<iface>/flags) and abort if still down.

Suggested replacement pattern:

```
// Pseudocode inside activate_interface()
if iface.is_wireless {
    self.ops.set_rfkill_block(iface, false).ok(); // unblock first
}

self.ops.bring_up(iface)
    .context("failed to set IFF_UP (RTM_SETLINK)")?;

// Verify admin-UP (IFF_UP) post-condition (do not use operstate)
if !self.ops.is_admin_up(iface).unwrap_or(false) {
    bail!("interface {} still admin-DOWN after bring_up()", iface.name);
}
```

3) rustyjack-core: remove Wi-Fi auto-connect from enforcement

File: rustyjack-core/src/system/isolation.rs
Function: activate_interface()

Currently, in EnforcementMode::Connectivity, Wi-Fi calls try_auto_connect_wifi() and may establish a network connection automatically.
This violates the requirement (“Wireless network interfaces should not automatically connect”).

Fix:
- Delete or hard-disable try_auto_connect_wifi() in the enforcement path.
- Keep connect_wifi_network / connect_wifi_from_saved as explicit user-driven operations only.

4) rustyjack-core: treat 'no carrier' and DHCP failure as non-fatal states

File: rustyjack-core/src/system/isolation.rs
Function: activate_interface(), verify_enforcement()

Changes:
- In Selection mode: never check carrier and never require DHCP.
- In ConnectivityBestEffort:
  - if carrier==0 → skip DHCP and record status Unavailable(NoCarrier)
  - if carrier==1 → attempt DHCP; if it fails → record Disconnected(DhcpFailed) and continue
- In verify_enforcement(): only require a default route when carrier is present and DHCP succeeded (or a manual IP config is applied).

Suggested logic for Ethernet best-effort:

```
let carrier = self.ops.read_carrier(iface).unwrap_or(None);
if carrier == Some(false) {
   report.dhcp = DhcpReport::NotAttempted;
   report.notes.push("no carrier; waiting for link".into());
   return Ok(report); // success: interface selected + admin-UP
}

match self.ops.acquire_dhcp(iface) {
   Ok(lease) => { /* configure routes + DNS */ report.dhcp = DhcpReport::Succeeded; }
   Err(e) => {
      report.dhcp = DhcpReport::Failed(e.to_string());
      report.notes.push("DHCP failed; will retry on carrier change".into());
      // DO NOT return Err
   }
}
```

5) rustyjack-daemon: report admin-up vs operstate correctly

File: rustyjack-daemon/src/dispatch.rs
Branch: Request::InterfaceStatus

Fix:
- Return admin_up from /sys/class/net/<iface>/flags (IFF_UP) or from netlink RTM_GETLINK flags.
- Keep operstate as a separate field if you want it for diagnostics, but do not map it to “is_up”.
- Return carrier separately (sysfs carrier or IFF_LOWER_UP).

Drop-in Rust snippet for flags parsing:

```
// flags file contains hex like "0x1003"
let flags_hex = std::fs::read_to_string(format!("/sys/class/net/{}/flags", ifname))?;
let flags = u32::from_str_radix(flags_hex.trim_start_matches("0x").trim(), 16)?;
let admin_up = (flags & 0x1) != 0; // IFF_UP
let running  = (flags & 0x40) != 0; // IFF_RUNNING (legacy-ish)
```

6) rustyjack-daemon: fix netlink watcher to subscribe to RTNLGRP_LINK and trigger DHCP on carrier-up

File: rustyjack-daemon/src/netlink_watcher.rs
Current code performs a one-time dump and exits.

Fix options:
A) Use a netlink socket subscribed to RTNLGRP_LINK and parse RTM_NEWLINK messages.
B) Poll carrier periodically (simpler, but less responsive and less efficient).

Option A is recommended. The kernel explicitly supports RTNLGRP_LINK subscriptions while the interface is admin-UP, which matches the requirement that the selected Ethernet interface stays ready when unplugged.

On carrier transition 0→1 for the selected Ethernet interface:
- call core's "ConnectivityBestEffort" path (DHCP + route + DNS) for that interface only.
- do not auto-connect Wi-Fi.

Event-driven watcher sketch (using netlink groups):

```
// Pseudocode: create netlink socket with RTNLGRP_LINK membership
// and drive an async loop that reacts to RTM_NEWLINK.
loop {
   let msg = link_event_stream.next().await?;
   if let Some((ifname, carrier)) = parse_link_change(&msg) {
      if ifname == selected_iface && is_ethernet(&ifname) && carrier == Some(true) {
         // best-effort DHCP attempt
         engine.enforce_with_mode(EnforcementMode::ConnectivityBestEffort).await?;
      }
   }
}
```

7) rustyjack-ui: stop treating 'no IP / no carrier' as a selection failure

File: rustyjack-ui/src/app.rs
Function: show_hardware_detect()

Fixes:
- Do not display "Acquiring DHCP..." when the user selected a Wi-Fi interface (Wi-Fi selection is isolation-only).
- For Ethernet: attempt DHCP, but if it fails or there is no carrier, show success ("Selected eth0") with a status badge ("No carrier / waiting for cable" or "DHCP failed; will retry").
- Replace the polling in wait_for_interface_up_with_ip() with a single call to daemon InterfaceStatus so the UI reflects the daemon's canonical state (and so both components agree on admin_up vs operstate).

UI outcome mapping suggestion:

```
// After set_active_interface(ifname):
let status = self.core.interface_status(ifname).await?;
if status.admin_up {
   self.hardware_detect_state.status = Some("Selected".into());
   // show secondary status:
   // - Ethernet: if !status.carrier => "Waiting for cable"
   // - Ethernet: if carrier && ip.is_none() => "DHCP in progress/failed"
   // - Wi-Fi: "Not connected (manual connect required)"
} else {
   self.hardware_detect_state.error = Some("Failed to bring interface admin-UP".into());
}
```

Testing and validation plan

Unit tests (fast)

- Parse /sys/class/net/<iface>/flags (hex) and verify IFF_UP extraction.
- Carrier parsing helper: handle missing files and non-0/1 values robustly.
- State machine transitions: Ethernet Selected→Unavailable(NoCarrier), CarrierUp→Configuring→Activated, DHCP failure paths.

Integration tests (Linux network namespaces)

In CI (where permitted), create a veth pair in a network namespace to simulate:
- carrier up/down transitions (by setting peer down),
- DHCP server presence/absence (dnsmasq in a namespace, or a minimal in-process DHCP responder),
- route/DNS changes.

Assertions:
- Selecting ethX succeeds even when peer down (no carrier).
- When peer transitions to up, daemon watcher triggers DHCP and configures default route.
- Selecting wlanX does not attempt association (no wpa interaction, no DHCP).

Manual validation on Raspberry Pi

Checklist:
1) With no Ethernet cable connected, select eth0:
  - UI should show “Selected eth0”, status “Waiting for cable”; no error.
2) Plug Ethernet cable into active network:
  - Within 1–3 seconds, DHCP attempt should run; UI should show an IPv4 address and default route.
3) Select wlan0:
  - wlan0 should become admin-UP; eth0 should be admin-DOWN; no Wi-Fi association occurs.
4) Trigger explicit Wi-Fi connect:

- Only then does association + DHCP occur.

Operational/deployment notes

If the OS image includes NetworkManager, dhcpcd, or systemd-networkd, you must ensure only one “network authority” is active. Otherwise, your in-process DHCP client (UDP/68) and route/DNS writes will conflict with other services.

Recommendation (aligned with “no third party software”):
- Disable/stop other network managers in install scripts.
- Or, if you must coexist, implement apply_nm_managed() (D-Bus) and/or avoid binding to port 68 when another DHCP client is active (but this is much harder to make reliable).

References

[NM-STATE] NetworkManager libnm Reference Manual — NMDeviceState enum (UNAVAILABLE includes “no ethernet carrier”).
https://networkmanager.dev/docs/libnm/latest/libnm-nm-dbus-interface.html

[KERN-OPER] Linux kernel documentation — Operational States (admin vs carrier vs operstate; RTNLGRP_LINK subscription).
https://www.kernel.org/doc/html/latest/networking/operstates.html