AI-UI: building an AI-native desktop shell in Rust

A production-grade desktop shell replacement is buildable today in Rust. The COSMIC desktop, shipped by System76 in December 2025 WebProNews Rexai with ~200K lines of Rust, Rexai proves the entire stack works at scale. This guide provides every crate, configuration file, code pattern, and CI/CD pipeline needed to build AI-UI — a GPU-accelerated, AI-powered desktop shell targeting Windows, Linux, and macOS from a single Rust codebase. The recommended stack is **iced 0.14 + wgpu 28 + cosmic-text + Taffy 0.7**, backed by the Claude API with Ollama fallback, packaged via GitHub Actions CI into .msi, .deb, .AppImage, and .dmg installers.

---

1. Project structure and core dependencies

Workspace layout

```
ai-ui/
├── Cargo.toml                  # Workspace root
├── crates/
│   ├── ai-ui-shell/            # Main binary (desktop shell)
│   │   ├── Cargo.toml
│   │   └── src/
│   │       ├── main.rs
│   │       ├── app.rs          # Application state + iced Application impl
│   │       ├── taskbar.rs      # Taskbar widget
│   │       ├── launcher.rs     # App launcher
│   │       ├── command_bar.rs  # AI command bar
│   │       └── platform/       # Platform-specific code
│   │           ├── mod.rs
│   │           ├── windows.rs
│   │           ├── linux.rs
│   │           └── macos.rs
│   ├── ai-ui-ai/               # AI integration library
│   │   ├── Cargo.toml
│   │   └── src/
│   │       ├── lib.rs
│   │       ├── claude.rs       # Claude API client
│   │       ├── ollama.rs       # Ollama fallback
│   │       ├── mcp.rs          # MCP client
│   │       └── streaming.rs    # SSE streaming
```

```
│   └── ai-ui-system/           # System integration library
│       ├── Cargo.toml
│       └── src/
│           ├── lib.rs
│           ├── apps.rs         # App enumeration
│           ├── status.rs       # Battery, WiFi, volume
│           ├── hotkeys.rs      # Global hotkeys
│           └── windows.rs      # Window management
├── assets/
│   ├── icons/
│   ├── fonts/
│   └── ai-ui.desktop
├── wix/                        # Windows installer template
├── debian/                     # Debian packaging scripts
└── .github/workflows/
    └── release.yml
```

Root Cargo.toml

```toml
[workspace]
resolver = "2"
members = ["crates/*"]

[workspace.package]
version = "0.1.0"
edition = "2024"
license = "MIT"
authors = ["Developer <dev@example.com>"]

[workspace.dependencies]
# UI Framework (production-proven by COSMIC desktop)
iced = { version = "0.14", features = ["wgpu", "canvas", "image", "svg", "tokio"]

# Async runtime
tokio = { version = "1", features = ["full"] }
futures-util = "0.3"

# Serialization
serde = { version = "1", features = ["derive"] }
serde_json = "1"

# HTTP
reqwest = { version = "0.12", features = ["json", "stream"] }

# Error handling
thiserror = "2"
```

```
anyhow = "1"

# Logging
tracing = "0.1"
tracing-subscriber = "0.3"
```

Shell crate Cargo.toml (crates/ai-ui-shell/Cargo.toml)

```
[package]
name = "ai-ui-shell"
version.workspace = true
edition.workspace = true

[dependencies]
iced.workspace = true
tokio.workspace = true
serde.workspace = true
serde_json.workspace = true
tracing.workspace = true
tracing-subscriber.workspace = true
anyhow.workspace = true

ai-ui-ai = { path = "../ai-ui-ai" }
ai-ui-system = { path = "../ai-ui-system" }

# Layout engine (CSS Flexbox + Grid)
taffy = "0.7"

# Fuzzy search (used by Helix editor, 6x faster than skim)
nucleo-matcher = "0.3"

# System tray (by Tauri team)
tray-icon = "0.19"
muda = "0.16"

# Date/time
chrono = "0.4"

# Config
dirs = "6"
toml = "0.8"

# Global hotkeys
global-hotkey = "0.6"

# Platform-specific
```

```
[target.'cfg(windows)'.dependencies]
windows = { version = "0.62", features = [
    "Win32_Foundation",
    "Win32_UI_WindowsAndMessaging",
    "Win32_UI_Shell",
    "Win32_Graphics_Gdi",
    "Win32_System_Threading",
    "Win32_System_Power",
] }
winreg = "0.55"

[target.'cfg(target_os = "linux")'.dependencies]
zbus = { version = "5", default-features = false, features = ["tokio"] }
freedesktop-desktop-entry = "4"
x11rb = "0.13"

[target.'cfg(target_os = "macos")'.dependencies]
objc2 = "0.6"
objc2-app-kit = "0.3"
objc2-foundation = "0.3"

[package.metadata.deb]
maintainer = "Developer <dev@example.com>"
depends = "$auto, libvulkan1, libwayland-client0"
section = "x11"
assets = [
    ["target/release/ai-ui-shell", "usr/bin/", "755"],
    ["assets/ai-ui.desktop", "usr/share/applications/", "644"],
    ["assets/ai-ui.desktop", "usr/share/wayland-sessions/", "644"],
    ["assets/icons/256x256.png", "usr/share/icons/hicolor/256x256/apps/ai-ui.png"
]

[package.metadata.wix]
upgrade-guid = "A1B2C3D4-E5F6-7890-ABCD-EF1234567890"
path-guid = "B2C3D4E5-F6A7-8901-BCDE-F12345678901"
license = false
eula = false
```

---

2. Why iced is the right framework choice

The four major Rust UI frameworks occupy distinct positions. **iced 0.14** is the only viable option for a desktop shell in 2026 — the others either lack maturity or target different use cases.

iced (v0.14, released December 2025) uses the Elm Architecture (Model → Message → Update → View) with **wgpu** rendering and **cosmic-text** for shaping. System76 maintains a soft fork powering their entire COSMIC 1.0 desktop environment: Wikipedia compositor, file manager, terminal, settings, app store Wikipedia — all running on real hardware shipped to customers. ArchWiki It handles **~700MB idle memory** versus GNOME's ~1.4GB. Rexai The Elm Architecture provides clean separation between state and rendering, making complex multi-panel UIs manageable.

egui (v0.31) is immediate-mode and rebuilds every frame. AN4T-Lab Excellent for debug overlays and tools, but architecturally wrong for a retained-mode desktop shell with persistent state across hundreds of widgets. It lacks the layout sophistication needed for a taskbar + launcher + command bar. AN4T-Lab

Slint (v1.9) has the most stable API (semantic 1.x versioning) and commercial backing, but uses a proprietary DSL compiled to native code. Its "Making Slint Desktop-Ready" initiative is still adding rich text and system tray support. GPL licensing for open-source projects may constrain distribution. Aman Chourasia

Xilem + Masonry (Linebender ecosystem) has the most modern architecture — reactive views on a retained widget tree, GitHub rendered by Vello's GPU compute pipeline. GitHub But Masonry is at v0.2.0 GitHub and Xilem is pre-alpha. GitHub Production deployment is **12–18 months away**.

Basic iced application skeleton

```
use iced::{Application, Command, Element, Settings, Theme, Length};
use iced::widget::{column, row, text, text_input, container, scrollable};

pub fn main() -> iced::Result {
    AiUiShell::run(Settings {
        window: iced::window::Settings {
            size: iced::Size::new(1920.0, 1080.0),
            decorations: false,
            transparent: true,
            ..Default::default()
        },
        antialiasing: true,
        ..Default::default()
    })
}

struct AiUiShell {
    command_input: String,
    ai_response: String,
```

```
    installed_apps: Vec<AppEntry>,
    search_results: Vec<AppEntry>,
    taskbar_state: TaskbarState,
    is_command_bar_visible: bool,
}

#[derive(Debug, Clone)]
enum Message {
    CommandInputChanged(String),
    ExecuteCommand,
    AiResponseChunk(String),
    AiResponseComplete,
    ToggleCommandBar,
    LaunchApp(String),
    SystemStatusUpdate(SystemStatus),
    Tick(chrono::DateTime<chrono::Local>),
}

impl Application for AiUiShell {
    type Message = Message;
    type Theme = Theme;
    type Executor = iced::executor::Default;
    type Flags = ();

    fn new(_flags: ()) -> (Self, Command<Message>) {
        let app = Self {
            command_input: String::new(),
            ai_response: String::new(),
            installed_apps: Vec::new(),
            search_results: Vec::new(),
            taskbar_state: TaskbarState::default(),
            is_command_bar_visible: false,
        };
        // Load installed apps on startup
        let init_cmd = Command::perform(
            ai_ui_system::apps::enumerate_apps(),
            |apps| Message::AppsLoaded(apps.unwrap_or_default()),
        );
        (app, init_cmd)
    }

    fn title(&self) -> String { "AI-UI Shell".into() }

    fn update(&mut self, message: Message) -> Command<Message> {
        match message {
            Message::CommandInputChanged(input) => {
                self.command_input = input.clone();
```

```
                self.search_results = fuzzy_search(&self.installed_apps, &input);
                Command::none()
            }
            Message::ExecuteCommand => {
                let prompt = self.command_input.clone();
                self.ai_response.clear();
                Command::perform(
                    ai_ui_ai::claude::stream_response(prompt),
                    |chunk| Message::AiResponseChunk(chunk),
                )
            }
            Message::AiResponseChunk(chunk) => {
                self.ai_response.push_str(&chunk);
                Command::none()
            }
            Message::LaunchApp(exec_path) => {
                let _ = std::process::Command::new(&exec_path).spawn();
                Command::none()
            }
            _ => Command::none(),
        }
    }

    fn view(&self) -> Element<Message> {
        let taskbar = self.render_taskbar();
        let desktop = if self.is_command_bar_visible {
            self.render_command_bar()
        } else {
            self.render_desktop()
        };
        column![desktop, taskbar]
            .width(Length::Fill)
            .height(Length::Fill)
            .into()
    }

    fn subscription(&self) -> iced::Subscription<Message> {
        iced::time::every(std::time::Duration::from_secs(1))
            .map(|_| Message::Tick(chrono::Local::now()))
    }
}
```

---

3. Claude API integration with streaming and tool use

AI crate dependencies (crates/ai-ui-ai/Cargo.toml)

```
[package]
name = "ai-ui-ai"
version.workspace = true
edition.workspace = true

[dependencies]
reqwest = { workspace = true }
serde = { workspace = true }
serde_json = { workspace = true }
tokio = { workspace = true }
futures-util = { workspace = true }
thiserror = { workspace = true }

# SSE streaming (2.4M+ downloads)
reqwest-eventsource = "0.6"

# Ollama local fallback
ollama-rs = { version = "0.3", features = ["stream"] }

# MCP (official Rust SDK, 2.6K stars)
rmcp = { version = "0.8", features = ["client", "transport-child-process", "trans

# Retry with exponential backoff
backoff = { version = "0.4", features = ["tokio"] }

# Secure API key storage
keyring = "3"
```

Complete Claude API client with streaming

There is **no official Anthropic Rust SDK**. The recommended approach is `reqwest` + `serde` for full control. Community crates exist (`anthropic-ai-sdk` v0.2.27, crates.io `anthropic_rust` , crates.io `clust`), GitHub but raw HTTP gives maximum reliability and flexibility.

```
// crates/ai-ui-ai/src/claude.rs
use reqwest::header::{HeaderMap, HeaderValue, CONTENT_TYPE};
use serde::{Deserialize, Serialize};
use futures_util::StreamExt;
use reqwest_eventsource::{Event, EventSource};

const API_URL: &str = "https://api.anthropic.com/v1/messages";
const API_VERSION: &str = "2023-06-01";
```

```rust
#[derive(Serialize)]
pub struct MessageRequest {
    pub model: String,
    pub max_tokens: u32,
    #[serde(skip_serializing_if = "Option::is_none")]
    pub system: Option<String>,
    pub messages: Vec<Message>,
    #[serde(skip_serializing_if = "Option::is_none")]
    pub stream: Option<bool>,
    #[serde(skip_serializing_if = "Option::is_none")]
    pub tools: Option<Vec<Tool>>,
}

#[derive(Serialize, Deserialize, Clone)]
pub struct Message {
    pub role: String,
    pub content: serde_json::Value,
}

#[derive(Deserialize, Debug)]
pub struct MessageResponse {
    pub id: String,
    pub content: Vec<ContentBlock>,
    pub stop_reason: Option<String>,
    pub usage: Usage,
}

#[derive(Deserialize, Serialize, Debug, Clone)]
#[serde(tag = "type")]
pub enum ContentBlock {
    #[serde(rename = "text")]
    Text { text: String },
    #[serde(rename = "tool_use")]
    ToolUse { id: String, name: String, input: serde_json::Value },
    #[serde(rename = "tool_result")]
    ToolResult { tool_use_id: String, content: String },
}

#[derive(Deserialize, Debug)]
pub struct Usage { pub input_tokens: u32, pub output_tokens: u32 }

#[derive(Serialize)]
pub struct Tool {
    pub name: String,
    pub description: String,
    pub input_schema: serde_json::Value,
```

```
}

#[derive(Debug, thiserror::Error)]
pub enum AiError {
    #[error("API key not configured")]
    NoApiKey,
    #[error("Rate limited — please wait")]
    RateLimited,
    #[error("Network error: {0}")]
    Network(#[from] reqwest::Error),
    #[error("API error: {status} — {message}")]
    ApiError { status: u16, message: String },
    #[error("No AI backend available")]
    NoBackend,
}

pub struct ClaudeClient {
    client: reqwest::Client,
    api_key: String,
    model: String,
    system_prompt: Option<String>,
}

impl ClaudeClient {
    pub fn new(api_key: String) -> Self {
        Self {
            client: reqwest::Client::new(),
            api_key,
            model: "claude-sonnet-4-5-20250929".into(),
            system_prompt: Some("You are an AI assistant integrated into a deskto
                Help users launch apps, manage files, answer questions, and contr
                system. Be concise and actionable.".into()),
        }
    }

    fn headers(&self) -> HeaderMap {
        let mut h = HeaderMap::new();
        h.insert(CONTENT_TYPE, HeaderValue::from_static("application/json"));
        h.insert("x-api-key", HeaderValue::from_str(&self.api_key).unwrap());
        h.insert("anthropic-version", HeaderValue::from_static(API_VERSION));
        h
    }

    /// Non-streaming call — returns complete response
    pub async fn send(&self, messages: Vec<Message>) -> Result<MessageResponse, A
        let request = MessageRequest {
            model: self.model.clone(),
```

```
        max_tokens: 4096,
        system: self.system_prompt.clone(),
        messages,
        stream: None,
        tools: None,
    };
    let resp = self.client.post(API_URL)
        .headers(self.headers())
        .json(&request)
        .send().await?;
    if !resp.status().is_success() {
        let status = resp.status().as_u16();
        let msg = resp.text().await.unwrap_or_default();
        return Err(AiError::ApiError { status, message: msg });
    }
    Ok(resp.json().await?)
}

/// Streaming call — yields text chunks via callback
pub async fn stream(
    &self,
    prompt: &str,
    mut on_chunk: impl FnMut(String),
) -> Result<(), AiError> {
    let body = serde_json::json!({
        "model": self.model,
        "max_tokens": 4096,
        "stream": true,
        "system": self.system_prompt,
        "messages": [{"role": "user", "content": prompt}]
    });
    let request = self.client.post(API_URL)
        .headers(self.headers())
        .json(&body);
    let mut es = EventSource::new(request)
        .map_err(|e| AiError::Network(e.into()))?;

    while let Some(event) = es.next().await {
        match event {
            Ok(Event::Message(msg)) => match msg.event.as_str() {
                "content_block_delta" => {
                    if let Ok(data) = serde_json::from_str::<serde_json::Valu
                        if let Some(text) = data["delta"]["text"].as_str() {
                            on_chunk(text.to_string());
                        }
                    }
                }
```

```
                    "message_stop" => { es.close(); break; }
                    _ => {}
                },
                Err(_) => { es.close(); break; }
                _ => {}
            }
        }
        Ok(())
    }
}
```

Tool use for system actions

```
/// Define tools Claude can call to control the desktop
pub fn desktop_tools() -> Vec<Tool> {
    vec![
        Tool {
            name: "launch_app".into(),
            description: "Launch an installed application by name".into(),
            input_schema: serde_json::json!({
                "type": "object",
                "properties": {
                    "app_name": { "type": "string", "description": "Application n
                },
                "required": ["app_name"]
            }),
        },
        Tool {
            name: "system_command".into(),
            description: "Execute a system action (volume, brightness, wifi toggl
            input_schema: serde_json::json!({
                "type": "object",
                "properties": {
                    "action": { "type": "string", "enum": [
                        "volume_up", "volume_down", "mute",
                        "brightness_up", "brightness_down",
                        "wifi_toggle", "bluetooth_toggle"
                    ]}
                },
                "required": ["action"]
            }),
        },
    ]
}

/// Handle tool use responses from Claude
```

```
pub async fn handle_tool_call(
    name: &str,
    input: &serde_json::Value,
) -> String {
    match name {
        "launch_app" => {
            let app_name = input["app_name"].as_str().unwrap_or("");
            match ai_ui_system::apps::launch_by_name(app_name).await {
                Ok(_) => format!("Launched {}", app_name),
                Err(e) => format!("Failed to launch {}: {}", app_name, e),
            }
        }
        "system_command" => {
            let action = input["action"].as_str().unwrap_or("");
            ai_ui_system::status::execute_action(action).await
        }
        _ => "Unknown tool".into(),
    }
}
```

MCP client with rmcp (official SDK)

rmcp v0.8.5 is the official Model Context Protocol Rust SDK, maintained at `github.com/modelcontextprotocol/rust-sdk` with **2.6K stars**.

```
// crates/ai-ui-ai/src/mcp.rs
use rmcp::{ServiceExt, transport::TokioChildProcess};
use tokio::process::Command;

pub async fn connect_mcp_server(
    command: &str,
    args: &[&str],
) -> Result<impl rmcp::Service, Box<dyn std::error::Error>> {
    let client = ().serve(
        TokioChildProcess::new(
            Command::new(command).args(args)
        )?
    ).await?;

    // List available tools from the MCP server
    let tools = client.list_tools(Default::default()).await?;
    tracing::info!("MCP server provides {} tools", tools.tools.len());
    for tool in &tools.tools {
        tracing::info!("  - {}: {}", tool.name, tool.description.as_deref().unwra
    }
    Ok(client)
```

```
}
```

Ollama fallback with automatic detection

```
// crates/ai-ui-ai/src/ollama.rs
use ollama_rs::Ollama;
use ollama_rs::generation::completion::request::GenerationRequest;

pub async fn is_ollama_running() -> bool {
    reqwest::get("http://localhost:11434/api/tags")
        .await
        .map(|r| r.status().is_success())
        .unwrap_or(false)
}

/// Unified AI backend — Claude first, Ollama fallback
pub async fn generate_response(
    prompt: &str,
    claude_key: Option<&str>,
) -> Result<String, AiError> {
    // Try Claude first
    if let Some(key) = claude_key {
        let client = ClaudeClient::new(key.to_string());
        match client.send(vec![Message {
            role: "user".into(),
            content: serde_json::Value::String(prompt.into()),
        }]).await {
            Ok(resp) => {
                for block in &resp.content {
                    if let ContentBlock::Text { text } = block {
                        return Ok(text.clone());
                    }
                }
            }
            Err(e) => tracing::warn!("Claude failed: {}, trying Ollama", e),
        }
    }

    // Fallback to Ollama
    if is_ollama_running().await {
        let ollama = Ollama::default();
        let res = ollama.generate(
            GenerationRequest::new("llama3.2:latest".into(), prompt.into())
        ).await.map_err(|e| AiError::ApiError {
            status: 500, message: e.to_string()
        })?;
```

```
        return Ok(res.response);
    }

    Err(AiError::NoBackend)
}
```

API key management

```
use keyring::Entry;

pub fn store_api_key(key: &str) -> Result<(), keyring::Error> {
    let entry = Entry::new("ai-ui", "anthropic-api-key")?;
    entry.set_password(key)?;
    Ok(())
}

pub fn load_api_key() -> Option<String> {
    // Priority: 1) env var, 2) OS keyring, 3) config file
    if let Ok(key) = std::env::var("ANTHROPIC_API_KEY") {
        return Some(key);
    }
    if let Ok(entry) = Entry::new("ai-ui", "anthropic-api-key") {
        if let Ok(key) = entry.get_password() {
            return Some(key);
        }
    }
    // Fall back to ~/.config/ai-ui/config.toml
    let config_dir = dirs::config_dir()?.join("ai-ui");
    let config: toml::Value = toml::from_str(
        &std::fs::read_to_string(config_dir.join("config.toml")).ok()?
    ).ok()?;
    config["api"]["anthropic_key"].as_str().map(String::from)
}
```

---

4. Desktop shell replacement per platform

Windows: full-screen overlay with Win32 AppBar

True shell replacement (replacing `explorer.exe` via the registry key `HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\Shell`) gives you a black screen with only your process. This is viable but risky for end users. **The recommended approach is a full-screen overlay that coexists with explorer**, using the AppBar API to

reserve screen space like a taskbar.

```
// crates/ai-ui-shell/src/platform/windows.rs
#[cfg(windows)]
pub mod shell {
    use windows::Win32::UI::Shell::*;
    use windows::Win32::UI::WindowsAndMessaging::*;
    use windows::Win32::Foundation::*;

    /// Register the window as an AppBar (reserves screen space like a taskbar)
    pub unsafe fn register_appbar(hwnd: HWND, height: u32) {
        let mut abd = APPBARDATA {
            cbSize: std::mem::size_of::<APPBARDATA>() as u32,
            hWnd: hwnd,
            ..Default::default()
        };
        // Register
        SHAppBarMessage(ABM_NEW, &mut abd);

        // Set position (bottom of screen)
        abd.uEdge = ABE_BOTTOM;
        abd.rc = RECT {
            left: 0,
            top: GetSystemMetrics(SM_CYSCREEN) - height as i32,
            right: GetSystemMetrics(SM_CXSCREEN),
            bottom: GetSystemMetrics(SM_CYSCREEN),
        };
        SHAppBarMessage(ABM_QUERYPOS, &mut abd);
        SHAppBarMessage(ABM_SETPOS, &mut abd);
    }

    /// Enumerate all visible windows for the task list
    pub fn list_windows() -> Vec<(String, HWND)> {
        let mut windows = Vec::new();
        unsafe {
            EnumWindows(Some(enum_callback),
                LPARAM(&mut windows as *mut _ as isize)).ok();
        }
        windows
    }

    unsafe extern "system" fn enum_callback(hwnd: HWND, lparam: LPARAM) -> BOOL {
        let windows = &mut *(lparam.0 as *mut Vec<(String, HWND)>);
        if IsWindowVisible(hwnd).as_bool() {
            let mut title = [0u16; 256];
            let len = GetWindowTextW(hwnd, &mut title);
```

```
            if len > 0 {
                let title = String::from_utf16_lossy(&title[..len as usize]);
                if !title.is_empty() {
                    windows.push((title, hwnd));
                }
            }
        }
        BOOL(1)
    }
}
```

For full shell replacement (kiosk/power-user mode), set the registry during install:

```
pub fn set_as_shell(exe_path: &str) -> std::io::Result<()> {
    let hkcu = winreg::RegKey::predef(winreg::enums::HKEY_CURRENT_USER);
    let (key, _) = hkcu.create_subkey(
        r"Software\Microsoft\Windows NT\CurrentVersion\Winlogon"
    )?;
    key.set_value("Shell", &exe_path)?;
    Ok(())
}
```

Linux: Wayland layer-shell panel or full compositor

Two approaches exist. The **pragmatic path** is running as a layer-shell panel on existing compositors (Sway, Hyprland, COSMIC). The **ambitious path** is building a full Wayland compositor with **smithay**, as COSMIC does with `cosmic-comp`.

Layer-shell panel approach (works today on wlroots compositors):

- Use `wayland-client` + `wlr-layer-shell-unstable-v1` protocol
- Creates overlay surfaces anchored to screen edges
- No need to handle input devices, DRM, or display management

Full compositor approach (requires smithay):

- **smithay** provides Wayland protocol handlers, DRM/GBM backends, libinput handling Lib.rs
- The sample compositor **Anvil** serves as a starting template
- COSMIC's `cosmic-comp` (GPL-3.0) is the production reference
- Dependencies: libseat, libinput, libwayland, libxkbcommon, mesa/libEGL

Register as a desktop session for display managers:

```
# /usr/share/wayland-sessions/ai-ui.desktop
[Desktop Entry]
Name=AI-UI Desktop
Comment=AI-powered desktop environment
Exec=/usr/bin/ai-ui-shell --session
Type=Application
DesktopNames=AI-UI
```

macOS: full-screen overlay with hidden dock

macOS does **not** allow true shell replacement — the Dock, Finder, and WindowServer are baked into the OS. The viable approach is a borderless, always-on-top window that hides the Dock and menu bar.

```
#[cfg(target_os = "macos")]
pub mod shell {
    use objc2_app_kit::*;
    use objc2_foundation::*;

    pub unsafe fn configure_as_shell(window: &NSWindow) {
        // Borderless full-screen, above dock
        window.setLevel(NSWindowLevel(25)); // kCGMainMenuWindowLevel + 1
        window.setCollectionBehavior(
            NSWindowCollectionBehavior::CanJoinAllSpaces |
            NSWindowCollectionBehavior::Stationary
        );
        window.setOpaque(false);

        // Auto-hide the dock and menu bar
        let app = NSApplication::sharedApplication();
        app.setPresentationOptions(
            NSApplicationPresentationOptions::AutoHideDock |
            NSApplicationPresentationOptions::AutoHideMenuBar
        );

        // Don't show in Dock
        app.setActivationPolicy(NSApplicationActivationPolicy::Accessory);
    }
}
```

---

5. System integration across all three platforms

Application enumeration

The **`applications` crate (v0.3.1)** provides cross-platform app listing. For production robustness, supplement with platform-specific code:

```
// crates/ai-ui-system/src/apps.rs
use std::path::PathBuf;

#[derive(Debug, Clone)]
pub struct AppEntry {
    pub name: String,
    pub exec: String,
    pub icon_path: Option<PathBuf>,
    pub description: Option<String>,
}

pub async fn enumerate_apps() -> anyhow::Result<Vec<AppEntry>> {
    let mut apps = Vec::new();

    #[cfg(target_os = "linux")]
    {
        use freedesktop_desktop_entry::{default_paths, get_languages_from_env, It
        let locales = get_languages_from_env();
        for entry in Iter::new(default_paths()).entries(Some(&locales)) {
            if let (Some(name), Some(exec)) = (entry.name(&locales), entry.exec()
                apps.push(AppEntry {
                    name: name.to_string(),
                    exec: exec.to_string(),
                    icon_path: entry.icon().map(|i| PathBuf::from(i)),
                    description: entry.comment(&locales).map(|c| c.to_string()),
                });
            }
        }
    }

    #[cfg(windows)]
    {
        use winreg::enums::*;
        use winreg::RegKey;
        let hklm = RegKey::predef(HKEY_LOCAL_MACHINE);
        if let Ok(uninstall) = hklm.open_subkey(
            r"SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall"
        ) {
            for key_name in uninstall.enum_keys().filter_map(|k| k.ok()) {
```

```
                if let Ok(subkey) = uninstall.open_subkey(&key_name) {
                    if let Ok(name) = subkey.get_value::<String, _>("DisplayName"
                        let exec = subkey.get_value::<String, _>("InstallLocation
                            .unwrap_or_default();
                        apps.push(AppEntry { name, exec, icon_path: None, descrip
                    }
                }
            }
        }
        // Also scan Start Menu .lnk files for better coverage
    }

    #[cfg(target_os = "macos")]
    {
        for dir in &["/Applications", "/System/Applications"] {
            if let Ok(entries) = std::fs::read_dir(dir) {
                for entry in entries.flatten() {
                    let path = entry.path();
                    if path.extension().map_or(false, |e| e == "app") {
                        let name = path.file_stem()
                            .map(|s| s.to_string_lossy().to_string())
                            .unwrap_or_default();
                        apps.push(AppEntry {
                            name,
                            exec: path.to_string_lossy().to_string(),
                            icon_path: None,
                            description: None,
                        });
                    }
                }
            }
        }
    }

    Ok(apps)
}
```

Fuzzy search with nucleo (6x faster than skim)

nucleo-matcher powers the Helix editor and uses a Smith-Waterman algorithm with aggressive ASCII prefiltering:

```
use nucleo_matcher::{Matcher, Config};
use nucleo_matcher::pattern::{Atom, AtomKind, CaseMatching, Normalization};
use nucleo_matcher::Utf32Str;
```

```
pub fn fuzzy_search(apps: &[AppEntry], query: &str) -> Vec<AppEntry> {
    if query.is_empty() { return apps.to_vec(); }
    let mut matcher = Matcher::new(Config::DEFAULT);
    let atom = Atom::new(query, CaseMatching::Ignore, Normalization::Smart,
                         AtomKind::Fuzzy, false);
    let mut results: Vec<(i32, &AppEntry)> = apps.iter().filter_map(|app| {
        let mut buf = Vec::new();
        let haystack = Utf32Str::new(&app.name, &mut buf);
        atom.score(haystack, &mut matcher).map(|score| (score as i32, app))
    }).collect();
    results.sort_by(|a, b| b.0.cmp(&a.0));
    results.into_iter().map(|(_, app)| app.clone()).collect()
}
```

System status (battery, WiFi, volume)

```
// crates/ai-ui-system/src/status.rs

#[derive(Debug, Clone, Default)]
pub struct SystemStatus {
    pub battery_percent: Option<f32>,
    pub battery_charging: bool,
    pub wifi_connected: bool,
    pub wifi_ssid: Option<String>,
    pub volume_percent: Option<f32>,
    pub cpu_usage: f32,
    pub memory_used_gb: f32,
    pub memory_total_gb: f32,
    pub time: String,
}

pub async fn read_status() -> SystemStatus {
    let mut status = SystemStatus::default();

    // Battery (cross-platform via starship-battery)
    if let Ok(manager) = starship_battery::Manager::new() {
        if let Some(Ok(battery)) = manager.batteries().ok().and_then(|mut b| b.ne
            status.battery_percent = Some(battery.state_of_charge().value * 100.0
            status.battery_charging = matches!(
                battery.state(), starship_battery::State::Charging
            );
        }
    }

    // CPU/RAM (via sysinfo)
    let mut sys = sysinfo::System::new();
```

```
    sys.refresh_memory();
    status.memory_used_gb = sys.used_memory() as f32 / 1_073_741_824.0;
    status.memory_total_gb = sys.total_memory() as f32 / 1_073_741_824.0;

    // WiFi (platform-specific)
    #[cfg(target_os = "linux")]
    {
        // Via NetworkManager D-Bus
        if let Ok(conn) = zbus::Connection::system().await {
            // Query org.freedesktop.NetworkManager for connectivity
        }
    }

    // Time
    status.time = chrono::Local::now().format("%H:%M").to_string();
    status
}
```

Global hotkeys for the command bar

`global-hotkey` v0.6 (by the Tauri team) covers Windows, macOS, and Linux/X11. For Wayland, **`hotkey-listener` v0.3** reads from evdev directly.

```
use global_hotkey::{GlobalHotKeyManager, hotkey::{HotKey, Modifiers, Code}};
use global_hotkey::GlobalHotKeyEvent;

pub fn register_command_bar_hotkey() -> Result<GlobalHotKeyManager, Box<dyn std::
    let manager = GlobalHotKeyManager::new()?;
    let hotkey = HotKey::new(Some(Modifiers::CONTROL), Code::Space);
    manager.register(hotkey)?;
    Ok(manager)
    // Poll GlobalHotKeyEvent::receiver() in your event loop
}
```

Notifications, clipboard, file watching

Capability	Crate	Version	Platforms
Notifications	`notify-rust`	4	Windows, macOS, Linux
Clipboard	`arboard`	3 (+ `wayland-data-control` feature)	Windows, macOS, Linux

File watching	`notify`	7	Windows, macOS, Linux
D-Bus (Linux IPC)	`zbus`	5	Linux (pure Rust, no C deps)

---

6. Cross-compilation strategy and CI/CD pipeline

Do not cross-compile wgpu apps from WSL

Cross-compiling a wgpu/winit desktop application from WSL to other platforms is **impractical** for production. GPU-dependent apps need platform SDKs (Metal headers for macOS, Windows SDK for D3D12), and testing requires native GPU access. The correct approach is **native compilation on each platform via GitHub Actions**, which provides free runners for all three operating systems.

From WSL Ubuntu, you can still do development and testing using the Linux target directly. For quick Windows testing from WSL:

```
# Install MinGW for Windows GNU target
sudo apt install mingw-w64
rustup target add x86_64-pc-windows-gnu

# .cargo/config.toml
# [target.x86_64-pc-windows-gnu]
# linker = "x86_64-w64-mingw32-gcc"

cargo build --target x86_64-pc-windows-gnu --release
```

This works for CI verification but **D3D12 rendering must be tested on actual Windows**.

Windows .msi with cargo-wix (v0.3.9)

cargo-wix generates WiX manifests and builds .msi installers. The WiX toolset only runs on Windows, so this step happens in CI.

```
cargo install cargo-wix
cargo wix init          # generates wix/main.wxs
cargo wix --no-build    # builds .msi from existing binary
```

Edit `wix/main.wxs` to add shell replacement registry option and assets:

```
<Directory Id="INSTALLDIR" Name="AI-UI">
    <Component Id="binary" Guid="*">
        <File Id="exe" Source="target\release\ai-ui-shell.exe" KeyPath="yes"/>
    </Component>
    <Directory Id="AssetsDir" Name="assets">
        <Component Id="assets" Guid="*">
            <File Id="config" Source="assets\default-config.toml"/>
        </Component>
    </Directory>
</Directory>
```

Linux .deb with cargo-deb (v3.6.2) and AppImage

```
cargo install cargo-deb
cargo deb               # outputs target/debian/ai-ui-shell_0.1.0-1_amd64.deb
```

The `[package.metadata.deb]` section shown in the Cargo.toml above handles all packaging metadata, including the wayland-sessions entry.

For AppImage, use `linuxdeploy`:

```
wget -q https://github.com/linuxdeploy/linuxdeploy/releases/download/continuous/l
chmod +x linuxdeploy-x86_64.AppImage
./linuxdeploy-x86_64.AppImage \
    --appdir AppDir \
    --executable target/release/ai-ui-shell \
    --desktop-file assets/ai-ui.desktop \
    --icon-file assets/icons/256x256.png \
    --output appimage
```

macOS .dmg with cargo-bundle

```
cargo install cargo-bundle
cargo bundle --release --format osx    # creates .app bundle
hdiutil create -volname "AI-UI" \
    -srcfolder target/release/bundle/osx/AI-UI.app \
    -ov -format UDZO AI-UI.dmg
```

Add to shell Cargo.toml:

```
[package.metadata.bundle]
name = "AI-UI"
```

```
identifier = "com.ai-ui.shell"
icon = ["assets/icons/icon.icns"]
osx_minimum_system_version = "10.15"
category = "public.app-category.utilities"
```

Complete GitHub Actions release workflow

```
name: Release
on:
  push:
    tags: ["v*.*.*"]

permissions:
  contents: write

env:
  CARGO_TERM_COLOR: always
  APP_NAME: ai-ui-shell

jobs:
  build-linux:
    runs-on: ubuntu-22.04
    steps:
      - uses: actions/checkout@v4
      - name: Install system deps
        run: |
          sudo apt-get update
          sudo apt-get install -y libwayland-dev libxkbcommon-dev \
            libvulkan-dev libx11-dev libxrandr-dev libxi-dev libdbus-1-dev
      - uses: dtolnay/rust-toolchain@stable
      - uses: Swatinem/rust-cache@v2
        with: { key: linux }
      - run: cargo build --release -p ai-ui-shell
      - name: Package .deb
        run: |
          cargo install cargo-deb
          cargo deb -p ai-ui-shell --no-build
      - name: Package AppImage
        run: |
          wget -q https://github.com/linuxdeploy/linuxdeploy/releases/download/co
          chmod +x linuxdeploy-x86_64.AppImage
          ./linuxdeploy-x86_64.AppImage \
            --appdir AppDir \
            --executable target/release/ai-ui-shell \
            --desktop-file assets/ai-ui.desktop \
            --icon-file assets/icons/256x256.png \
```

```
          --output appimage
    - uses: actions/upload-artifact@v4
      with:
        name: linux
        path: |
          target/debian/*.deb
          *.AppImage

build-windows:
  runs-on: windows-latest
  steps:
    - uses: actions/checkout@v4
    - uses: dtolnay/rust-toolchain@stable
    - uses: Swatinem/rust-cache@v2
      with: { key: windows }
    - run: cargo build --release -p ai-ui-shell
      env: { RUSTFLAGS: "-C target-feature=+crt-static" }
    - name: Package .msi
      run: |
        cargo install cargo-wix
        cargo wix -p ai-ui-shell --no-build --nocapture
    - uses: actions/upload-artifact@v4
      with:
        name: windows
        path: target/wix/*.msi

build-macos:
  strategy:
    matrix:
      include:
        - { target: x86_64-apple-darwin, os: macos-13 }
        - { target: aarch64-apple-darwin, os: macos-14 }
  runs-on: ${{ matrix.os }}
  steps:
    - uses: actions/checkout@v4
    - uses: dtolnay/rust-toolchain@stable
      with: { targets: "${{ matrix.target }}" }
    - uses: Swatinem/rust-cache@v2
      with: { key: "macos-${{ matrix.target }}" }
    - run: cargo build --release -p ai-ui-shell --target ${{ matrix.target }}
      env: { MACOSX_DEPLOYMENT_TARGET: "10.15" }
    - name: Package .dmg
      run: |
        cargo install cargo-bundle
        cargo bundle --release -p ai-ui-shell --target ${{ matrix.target }} --f
        APP=$(find target/${{ matrix.target }}/release/bundle/osx -name "*.app"
        hdiutil create -volname "AI-UI" -srcfolder "$APP" -ov -format UDZO \
```

```
            "AI-UI-${{ matrix.target }}.dmg"
      - uses: actions/upload-artifact@v4
        with:
          name: macos-${{ matrix.target }}
          path: "*.dmg"

  release:
    needs: [build-linux, build-windows, build-macos]
    runs-on: ubuntu-latest
    steps:
      - uses: actions/download-artifact@v4
        with: { path: artifacts }
      - uses: softprops/action-gh-release@v2
        with:
          files: artifacts/**/*
          generate_release_notes: true
```

Code signing essentials

Windows requires a code signing certificate ($200–500/year from DigiCert/Sectigo, or **$9.99/month via Azure Trusted Signing**). In CI, import a Base64-encoded .pfx as a GitHub secret and sign with `signtool.exe`.

macOS requires an Apple Developer account ($99/year) with a "Developer ID Application" certificate. In CI: import the .p12 certificate into a temporary keychain, then `codesign --force --deep --sign "Developer ID Application: ..." --options runtime`, then `xcrun notarytool submit` + `xcrun stapler staple`. Code signing and notarization **can only happen on macOS runners**.

Linux uses GPG signing via `dpkg-sig -k KEY_ID --sign builder *.deb`.

---

7. Complete crate dependency map

This table summarizes every crate referenced in this guide with its role and current version:

Crate	Version	Role
`iced`	0.14	UI framework (Elm arch, wgpu rendering)
`wgpu`	28.0	GPU abstraction (Vulkan/Metal/D3D12)
`winit`	0.30	Cross-platform windowing

`vello`	0.6	GPU compute 2D renderer (alpha, for custom rendering)
`taffy`	0.7	CSS Flexbox + Grid layout engine
`cosmic-text`	0.12	Text shaping/rendering (used by iced)
`parley`	git	Text layout (Linebender, for Vello pipeline)
`reqwest`	0.12	HTTP client
`reqwest-eventsource`	0.6	SSE streaming for Claude API
`serde` / `serde_json`	1	Serialization
`tokio`	1	Async runtime
`ollama-rs`	0.3	Ollama local model client
`rmcp`	0.8	Official MCP Rust SDK
`backoff`	0.4	Exponential retry logic
`keyring`	3	OS-native credential storage
`nucleo-matcher`	0.3	Fuzzy search (Helix editor's matcher)
`global-hotkey`	0.6	Global hotkeys (Tauri team)
`tray-icon`	0.19	System tray icons (Tauri team)
`muda`	0.16	Cross-platform menus (Tauri team)
`notify-rust`	4	Desktop notifications
`arboard`	3	Clipboard access
`sysinfo`	0.38	CPU, RAM, disk, network stats
`starship-battery`	1.0	Battery status
`chrono`	0.4	Date/time
`notify`	7	File system watching
`zbus`	5	D-Bus (Linux, pure Rust)

`windows`	0.62	Win32 API (Microsoft official)
`winreg`	0.55	Windows registry
`objc2-app-kit`	0.3	macOS AppKit bindings
`x11rb`	0.13	X11 protocol (Linux)
`freedesktop-desktop-entry`	4	.desktop file parsing (Linux)
`cargo-deb`	3.6	.deb packaging
`cargo-wix`	0.3.9	.msi packaging
`cargo-bundle`	latest	macOS .app/.dmg packaging
`cargo-dist`	0.30	Automated release pipeline

Conclusion: a buildable path exists today

The single most important validation for this project is **COSMIC 1.0** — a complete, shipping Rust desktop environment built by System76 with under 20 engineers. Their stack (iced + wgpu + smithay + cosmic-text) proves that every layer works in production. AI-UI can follow the same architecture and add Claude integration as its differentiator.

The critical trade-off is between **compositor scope** and **time to ship**. Building a full Wayland compositor with smithay is a multi-month effort. The faster path is building a layer-shell panel on Linux, a Win32 overlay on Windows, and an NSWindow overlay on macOS — delivering the AI command bar and smart taskbar without replacing the entire windowing system. Graduate to a full compositor later once the core AI experience is validated.

Three decisions will determine whether this project reaches non-technical users. First, **use iced's Elm Architecture** rather than building a custom wgpu+Vello renderer — the framework handles accessibility, text input, widget lifecycle, and platform quirks that take months to reimplement. Second, **rely on GitHub Actions CI for all packaging** rather than cross-compiling from WSL — wgpu's platform SDK requirements make cross-compilation impractical. Third, **ship the Claude API integration with Ollama fallback from day one** — the `reqwest` + `reqwest-eventsource` streaming pattern and `ollama-rs` crate make both backends straightforward, and the fallback ensures the app works offline without any AI provider dependency.