Install and configure REA with Ghidra on my computer so this coding agent can perform real native-binary analysis.

Official sources:
- REA: https://github.com/morluto/rea (npm package: rea-agents)
- Ghidra: https://github.com/NationalSecurityAgency/ghidra
- Ghidra releases: https://github.com/NationalSecurityAgency/ghidra/releases

Read the current installation instructions and compatibility requirements from these official projects before choosing versions. Use a published REA release and a Ghidra/JDK combination that it explicitly supports. Before running the npm package, confirm that its repository field points to morluto/rea. The platform notes below reflect REA 6.4.0 and Ghidra 12.1.4; if the current docs differ, follow the docs and tell me what changed.

Complete this in one pass. Ask for my approval once, for the setup plan in step 3. After that, finish everything without further questions unless something fails or needs an action only I can take.

1. Identify my operating system, CPU architecture, shell, and coding client, and whether commands run locally or on a remote host. Keep REA, Ghidra, Java, and their paths in that same environment. On Windows, install and run everything natively on Windows. Don't enable Windows features or virtualization, and don't restart my computer.

2. Check for existing REA, Ghidra, Node.js/npm, and a full Java Development Kit, including javac. Reuse compatible installations. Identify missing prerequisites and any platform limitations.

3. Show me one concise setup plan covering downloads (with sizes), installation locations, required build tools, configuration changes, and anything that needs administrator rights. Prefer user-level installs that avoid admin rights. Wait for my approval, then complete the approved work.

4. Install missing prerequisites and Ghidra from official releases or trusted package managers. Verify published download checksums when available.
   - Check that Ghidra has a working native decompiler for my exact OS and CPU architecture, in Ghidra/Features/Decompiler/os/<platform>/.
   - If the decompiler needs building, follow the matching release's instructions with the build tools for my OS, and include those tools in the plan. Make sure the build targets my CPU architecture, and confirm the result (for example with `file` on macOS or Linux) before installing it.
   - If the operating system's security features block a binary (macOS Gatekeeper, Windows SmartScreen or Defender), stop and tell me instead of changing security settings.

5. Use REA's own setup workflow to register its MCP server and matching reverse-engineer-anything skill. Pick my client's ID from REA's supported-client list, for example

claude_code for the Claude Code desktop app or terminal, or codex for Codex desktop or terminal.
   - Set GHIDRA_INSTALL_DIR and JAVA_HOME in the same command that runs setup, so it saves them into the MCP registration. Desktop apps don't inherit shell variables, so the paths must live in the registration.
   - Setup's interactive prompts need a real terminal, so run it with --dry-run first and include its plan in step 3. After my approval, apply it with --yes and an explicit --client.
   - Prefer stable paths that survive package-manager upgrades, such as Homebrew's opt path on macOS rather than a version-numbered Cellar path. Where only a version-numbered path exists (common for Windows JDK installers), note what will need updating after an upgrade.
   - Raise REA_GHIDRA_STARTUP_TIMEOUT_MS (default 330000 ms) in the registration to allow for large binaries.
   - Check the client's own per-call MCP tool timeout, and raise it if it's shorter than Ghidra's analysis might take. Examples: Codex's tool_timeout_sec (default 300 seconds), or Claude Code's per-server timeout.
   - Verify the client's actual configuration locations and overrides. Preserve existing settings, keep setup's backups, and avoid duplicate registrations.

6. Run REA's client, skill, and Ghidra diagnostics. Run the Ghidra check with the same environment variables as the registration; from a plain shell it will wrongly report Ghidra as missing. Distinguish server startup from individual tool-call timeouts.

7. Verify the integration through this coding agent's actual REA MCP tools:
   - Compile a tiny fixture for my platform containing a function with known behaviour. If no C compiler is available, include a lightweight one in the plan (for example Zig or LLVM's clang) rather than a full IDE.
   - Open it with provider_id ghidra, list its functions, and decompile that function.
   - The first Ghidra-backed call waits for the full analysis. Never cancel or retry it partway through: cancelling shuts Ghidra down, and the next call starts over.
   - Show the expected behaviour next to the actual decompiled result, and preserve the evidence.
   - Close the analysis session, and confirm that no Ghidra process is left running.

Platform notes:
- macOS (Intel or Apple Silicon): Official Ghidra archives may not include a macOS decompiler. If so, build it from the source included in the release, using the Xcode Command Line Tools and make, then install it to os/mac_arm_64 or os/mac_x86_64. On Apple Silicon, the default macOS build may produce an x86_64 binary; it must be arm64.
- Linux x64: The official release includes the linux_x86_64 decompiler.
- Linux ARM64: The linux_arm_64 decompiler may need building (gcc/g++, make, and bison/flex if required).
- Windows x64: Install natively. REA's Windows Ghidra support is labelled experimental and is read-only. It requires Windows 10 or later and local NTFS drives, and it analyses Windows

x86/x64 programs and DLLs (.exe and .dll files). The official release includes the win_x86_64 decompiler.
- Windows on ARM: REA's Windows Ghidra support requires x64. Report this limitation instead of working around it.

Configuration changes only reach agent sessions started after the change. If a restart is required to expose the tools, save a short setup checkpoint and tell me exactly what to restart: a new Code session in the Claude Code desktop app, a restarted claude process in a terminal, or a restarted Codex session. When I return and say "I restarted—continue verification," resume from that checkpoint.

Finish with the installed versions and paths, configuration and backup locations, verification results, the disk space used and any downloaded archives you removed, and any remaining limitations, including which program formats this machine can analyse. Only mark the integration fully verified after a real MCP decompilation succeeds.

Use the reverse-engineer-anything skill and REA's Ghidra provider to fully decompile SuperTuxKart's physics from its shipped binary.
Read first: `supertuxkart/README.md` (target, measurements, gotchas) and `supertuxkart/analysis/initial-survey/physics_function_inventory.txt` (the 697 physics-related functions, grouped by subsystem).
Rules
- Do NOT use SuperTuxKart's source code: no stk-code repo, wiki, docs, forum posts or memory of the codebase. Evidence must come from `supertuxkart/target/` only: the binary plus the data files shipped in the .app (`kart_characteristics.xml` and `stk_config.xml` hold physics constants).
- General knowledge of the Bullet library's public API is fine. Treat stock Bullet (`bt*` classes) as a known library: name it, don't decompile it. `btKart` and `btKartRaycaster` are SuperTuxKart's own classes and ARE in scope.
- Never present a guess as fact. Label each reconstruction confirmed, inferred or unknown, cite the function address it came from, and record open questions as REA unknowns.
How REA + Ghidra behave here (tested on this exact binary)
- Open `supertuxkart/target/supertuxkart-arm64` with `open_binary` and `provider_id: "ghidra"`. The first Ghidra-backed call (use `binary_overview`) takes about 5 minutes while Ghidra analyses. Make that call once and let it finish.
- Never cancel or interrupt that first call. Cancelling it shuts Ghidra down, and the next call starts the full 5-minute analysis again.
- Keep one session open for the whole chat. `close_binary` and reopening re-analyses from scratch, and loading a saved snapshot doesn't skip it either.
- Names are mangled. Pass a mangled name or an address as the `procedure`; the demangled form (`Physics::update`) is rejected. For example, `Physics::update` is `__ZN7Physics6updateEi`.
- To find functions, use `search_procedures` with `mode: "regex"` and a mangled prefix: `^__ZN7Physics`, `^__ZN6btKart`, `^__ZN4Kart`, `^__ZN8MaxSpeed`, `^__ZN8Skidding`. Literal searches like `Physics::` find nothing.
- Use `procedure_pseudo_code`, `batch_decompile`, `procedure_callees`, `get_call_graph`, `xrefs` and `inspect_native_data_type`. These are all fast.
- Avoid `analyze_function`: one call on `Physics::update` returned 3.9 MB. Don't call `list_procedures` on the whole binary either; its reply is over REA's 10 MB limit.
- Renames and comments made in Ghidra (`annotate_native_function`, `set_address_name`) only last for this session, so keep the real record of names and layouts in your output files.
If the first Ghidra call fails, don't retry blindly; every retry costs another 5 minutes. Read the REA error's `details.diagnostics` (including `stderr`):
- `OutOfMemoryError`: Ghidra needs more Java memory than `GHIDRA_HEADLESS_MAXMEM` in the `rea` entry of `~/.claude.json` (currently `16G`).
- A REA startup-deadline error: raise `REA_GHIDRA_STARTUP_TIMEOUT_MS` in that same entry (currently `1800000`, which is 30 minutes).
- Claude Code itself stopped waiting for the tool call (a timeout with no REA error): add `"timeout": 3600000`, Claude Code's per-server tool timeout in milliseconds, to the `rea` entry under `mcpServers` in `~/.claude.json`.

For any of these, show me the exact change and ask before editing `~/.claude.json`. New settings only apply in a new chat, so update `PROGRESS.md` first, then tell me to start one.
Approach
1. Map the structure first: the call graph from `Physics::update` and the kart update path down through `btKart`, `Moveable`, `MaxSpeed`, `Skidding`, `SlipStream`, `TerrainInfo` and the collision handling.
2. Recover class layouts. Turn raw offsets like `param_1 + 0xf4` into named fields, and use the shipped XML constants to identify them.
3. Reconstruct readable C++ for each class, with comments explaining the maths.
Write everything to `supertuxkart/physics/` (large output goes in files, not chat):
- `PROGRESS.md`: a checklist of every inventory function (done / skipped as a trivial getter or library / blocked). Update it after each class so a later chat can pick up where this one stops. Tell me before the context gets full.
- `layouts/`: recovered struct and class layouts, with offsets and evidence.
- `reconstructed/`: `.hpp`/`.cpp` per class, each function commented with its address and confidence. Where practical, check that they parse with `clang++ -fsyntax-only` against stub headers.
- `PHYSICS_MODEL.md`: a plain-English explanation of how a kart moves: the physics step, suspension raycasts, engine and braking forces, friction, drifting and skidding, speed caps and boosts, slipstream, terrain effects, kart and kart-to-world collisions, and rescue and explosion handling.
- Export REA's evidence bundle (`export_evidence_bundle`) to `physics/evidence/` at each checkpoint.
Suggested order: `Physics` → `btKart` / `btKartRaycaster` → `Moveable` / `KartMotionState` → physics parts of `Kart` → `MaxSpeed` → `Skidding` → `SlipStream` → `TerrainInfo` → `TriangleMesh` / `PhysicalObject` → the kart animations → `KartControl` → `KartProperties` and the characteristics (mostly getters).
When everything is done, summarise what you're confident about and what's still unknown, then ask me before comparing anything against the real source code.

PROJECT: Reverse engineer Ditto's network clipboard protocol and build a native macOS companion

Use the installed reverse-engineer-anything skill and REA's Ghidra provider to investigate Ditto's network clipboard sharing. Then independently implement a native macOS companion from the protocol specification you recover.

REA is already being set up in this Codex environment. Reuse that installation and configuration. Do not start a second setup, reinstall REA, switch providers, or overwrite existing configuration unless a specific problem requires it and I approve the change.

GOAL

With unmodified Ditto running on Windows and the companion running on my Mac:

- Copy supported content on Windows, then paste it normally on the Mac.
- Copy supported content on the Mac, then paste it normally on Windows.
- No manual selection from Ditto's history should be necessary.
- Clipboard changes must not bounce endlessly between the computers.

The first milestone is reliable bidirectional Unicode text. Rich text, HTML, and images come afterward. Files are optional and last.

Receiving something into Ditto's history is not equivalent to updating the active Windows clipboard. Verify that distinction explicitly.

Do not promise that every possible clipboard format will work. Report the formats actually implemented and tested.

1. WORKSPACE AND EXISTING SETUP

Use the currently selected Codex workspace as the project root. Determine its actual absolute path from the environment.

Do not reuse a path from another project. Do not rename the workspace or automatically create another enclosing ditto/ directory.

Inspect applicable AGENTS.md instructions and relevant existing setup files first. Preserve existing work and do not overwrite unrelated files.

Create these paths beneath the project root when absent:

target/

analysis/
protocol/
protocol/fixtures/
tools/
captures/
mac-companion/
PROGRESS.md
README.md

Use target/ for downloaded binaries and separate extraction directories. Keep captures/ private and excluded from version control.

Add appropriate ignore rules for secrets, local configuration, raw captures, downloaded binaries, clipboard databases, and build artifacts. Sanitized synthetic test fixtures can be versioned.

Record the project root, execution host, operating system, architecture, and installed REA/Ghidra versions in PROGRESS.md.

Verify whether Codex is actually executing on the Mac Studio. Do not assume that a cloud container or another computer can access its GUI clipboard or local network.

2. BINARY-ONLY INVESTIGATION RULES

Treat this as binary-only protocol reverse engineering.

Allowed evidence:
- The pinned release's executable files, DLLs, PE resources, and matching shipped debug information.
- REA/Ghidra analysis of those artifacts.
- Ditto's actual user interface and observed behavior.
- Authorized network captures and controlled runtime observations using synthetic test content.

Do not read:
- Ditto's source code or source archives.
- Ditto's documentation, wiki, forum posts, issues, or discussions describing its implementation.
- Existing third-party implementations or descriptions of Ditto's protocol.

Release metadata may be used to locate and verify the specified binary assets. General documentation for REA, Ghidra, Codex, Apple APIs, Windows clipboard formats, and cryptographic libraries is allowed.

Do not use remembered Ditto implementation details as evidence. Every Ditto-specific protocol claim needs a traceable basis in this investigation.

Write the companion from the recovered protocol specification. Do not copy Ditto source code or mechanically turn decompiler output into application code.

For protocol findings, use:
- Confirmed: relevant static analysis and actual Ditto traffic or controlled runtime behavior agree.
- Inferred: plausible but not yet corroborated. State the evidence and missing test.
- Unknown: unresolved.

Distinguish what a tool directly observed from your interpretation.

For important findings, record:
- Artifact filename and SHA-256.
- Function address and image-relative offset where applicable.
- Relevant instructions, pseudocode, resources, or strings.
- Capture identifier, stream direction, and byte offset where applicable.
- Confidence and remaining questions.

Record unresolved questions through REA's available unknown-tracking tools and in the project files. If a requested REA feature is unavailable, document that rather than inventing tool calls.

3. DOWNLOAD AND VERIFY THE TARGET

Pin the investigation to Ditto 3.25.113.0 from the official sabrogden/Ditto GitHub release.

Download only these named assets into target/:

DittoPortable_3_25_113_0.zip
SHA-256:
c21dc05f27f744813e0f19d2d20842546baea160ab366382548aa283e2636216

DittoDebug_3_25_113_0.7z
SHA-256:
33d14929b4474309289259322564 2c6bd553e2cb5eb11561817f116617bab674

Use explicit asset filters with gh release download, or download the exact assets directly from the official release if gh is unavailable.

Verify each archive before extracting it. On a mismatch, stop using that asset and report the discrepancy. Never silently substitute another version or change the expected hash.

Preserve the original archives. Extract the portable and debug assets into separate directories.

Inspect PE headers to establish architecture. Prefer x64 when available. Record the exact Ditto.exe chosen and its own SHA-256.

Inventory the debug archive without reading any bundled source text. Shipped PDBs may be used for symbols and type information.

Before applying PDB addresses or names to a binary, verify that the PDB matches that binary, including its identifying debug information. Do not assume a debug-build PDB matches the portable release executable.

Use the installed provider's actual PDB capabilities. If necessary, inspect a matching PDB with a tool such as llvm-pdbutil. Do not fetch source through source-server information.

A different debug binary may supply investigation leads, but its addresses and layouts must not be treated as release-binary facts without independent validation.

4. REA AND GHIDRA OPERATING RULES

Read the installed reverse-engineer-anything skill and inspect the available tool schemas. Follow the installed version rather than assuming every behavior from a previous project still applies.

Confirm that REA's Ghidra provider is healthy using the existing installation.

Select the verified binary with open_binary and provider_id: "ghidra", using the actual installed schema.

The first Ghidra-backed query, such as binary_overview, may start import and full analysis. Allow healthy analysis to complete. Do not cancel it merely because it is slow.

Reuse the live session. Avoid unnecessary target switching, duplicate analysis sessions, and repeated whole-binary queries.

Use exact identifiers returned by REA, preferably function addresses. Do not assume a displayed demangled name is an accepted selector.

Useful focused operations may include:
search_procedures
search_strings
xrefs
procedure_pseudo_code
batch_decompile
procedure_callees

get_call_graph
read_bytes
inspect_native_data_type
inspect_pe_resources

Use bounded searches and small decompilation batches. Avoid full-binary procedure listings and large function dossiers when a narrower query will answer the question.

Do not assume an old response-size limit or snapshot limitation applies unchanged. Inspect the current implementation's capabilities.

Keep durable names, layouts, findings, and evidence in project files. Do not depend on session-only annotations surviving a restart. Export evidence through supported REA tools where useful.

Before expensive analysis, check that relevant deadlines are adequate:
- Codex MCP server initialization: startup_timeout_sec.
- Codex individual tool calls: tool_timeout_sec.
- REA's Ghidra import/analysis startup deadline: REA_GHIDRA_STARTUP_TIMEOUT_MS.
- Ghidra heap allocation: GHIDRA_HEADLESS_MAXMEM.

These address different failure modes. Do not indiscriminately increase all of them.

For a failure, inspect details.diagnostics and the relevant logs before retrying. A tool timeout is not automatically a Ghidra memory problem.

This is Codex, not Claude Code. Do not edit ~/.claude.json.

Locate the active Codex configuration before proposing changes. Ask before modifying Codex configuration, REA configuration, or persistent environment settings. Show the specific proposed change and preserve existing values and unrelated entries.

Do not kill unrelated processes or repeatedly restart a failing analysis without a diagnosis.

5. SECURITY AND CLIPBOARD PRIVACY

Use only my approved computers and local network interfaces. No cloud relay, public listener, port forwarding, or automatic network scanning.

Require a nonempty Ditto network password. Recover and preserve Ditto's actual password handling, encryption, and validation behavior. Never bypass checks or introduce an unencrypted fallback to make a test pass.

Do not assume that a password or encryption implies strong authentication or modern security. Investigate integrity protection, key derivation, nonce/IV handling, and replay behavior as far as the evidence permits.

If the scheme is weak or important protections remain unknown, explain that plainly. Keep testing limited to synthetic data and obtain approval before enabling ordinary clipboard syncing.

Enter passwords through a local secure prompt or app settings, not this chat. Store the companion's persistent password in macOS Keychain. Never put it in source files, command-line arguments, fixtures, or PROGRESS.md.

Before live testing, agree on synthetic test content. Do not inspect, transmit, or record my existing clipboard history.

Do not automatically send whatever is already on the clipboard at startup. Start paused and require explicit enablement.

Production diagnostics must not contain clipboard payloads, passwords, derived keys, or payload hashes. Debug payload capture must be explicitly enabled for synthetic tests only.

Keep real clipboard data out of Codex tool output, chat messages, screenshots, and committed files. Do not describe the entire AI-assisted development process as offline merely because the resulting clipboard transport is local.

Limit listeners and captures to the approved peers and ports. Ask before changing firewall rules. Do not disable the firewall.

Validate incoming content before replacing the clipboard. Bound payload sizes, allocations, image dimensions, and decompression output. Reject malformed input safely.

6. INVESTIGATE THE MINIMUM TEXT PROTOCOL FIRST

Do not try to fully reconstruct the whole application.

Locate the sending and receiving paths using:
- Winsock imports and their callers, including wrappers or asynchronous variants.
- Network-related strings and PE resources.
- Settings for peers, ports, passwords, automatic sending, and receiving.
- Cryptographic imports, linked libraries, tables, and constants.
- Clipboard serialization and deserialization paths.

Treat strings and crypto constants as leads, not proof of a complete protocol.

Create protocol/SPEC.md incrementally. Prioritize everything required for Unicode text:

- Which side connects and which side listens.
- Ports and whether they are configurable.
- Connection setup and teardown.
- Handshake, password handling, and peer identity.
- Message types, framing, headers, lengths, and version fields.
- Encryption details and the order of encryption, compression, and serialization.
- Clipboard format identification and encoding.
- Acknowledgments, error responses, retries, and duplicate handling.
- Byte order, integer widths, string encodings, terminators, and padding.

Document the Windows-side behavior needed for:
1. Automatically sending new copies.
2. Automatically placing received clips onto the active clipboard.

Derive the applicable settings from shipped resources and the running UI, not online Ditto documentation.

If unmodified Ditto cannot satisfy the second requirement, report that limitation before expanding the Mac app. Propose any Windows helper separately and ask before implementing one.

Add sections for RTF, HTML, images, and files as evidence becomes available. Unknown optional formats must not block the Unicode-text milestone.

7. VALIDATE WITH ACTUAL DITTO TRAFFIC

Before live testing, ask for any missing information about:
- The Windows PC or VM being used.
- Its current IP address and running Ditto version.
- The Mac's intended network interface/address.
- Permission to configure the required Ditto settings and run scoped captures.

Reuse information already supplied or reliably discovered. Do not ask me to repeat it.

Ensure the tested Windows binary matches the pinned target. Do not silently downgrade or replace an existing Ditto installation or risk its database.

Temporarily disable other clipboard-sync paths during verification, including Jump Desktop clipboard sharing, VM shared clipboards, or another clipboard-sync utility.

Use synthetic content such as:

REA-WIN-TO-MAC-001
REA-MAC-TO-WIN-001
REA Unicode: café 日本語 😀

Also test multiline content and repeated copies.

A raw TCP listener is only an initial diagnostic. Do not assume it can capture a complete transfer if Ditto expects a handshake or response.

If two suitable Windows instances are already available, a genuine Ditto-to-Ditto capture can provide a baseline. Do not require a second Windows machine unnecessarily.

Otherwise, build a minimal receiver/sender harness from the static findings and refine it against actual Ditto behavior.

Capture only the approved peers and ports. Reassemble TCP streams and distinguish protocol message boundaries from individual socket reads or network packets.

Reconcile captures with static analysis. Do not force bytes to fit a guessed structure.

Validate Windows-to-Mac and Mac-to-Windows independently. A successful round trip between two copies of our own prototype does not prove Ditto compatibility.

If a live test is blocked, mark it blocked or not run. Continue useful offline work where possible without claiming interoperability has been demonstrated.

8. BUILD THE COMPANION IN STAGES

Stage A: Protocol prototype

Build a small command-line test harness under tools/ with:
- A sender and receiver for the recovered protocol.
- Secure local password entry.
- Explicit peer and port configuration.
- Synthetic fixtures and focused automated tests.
- Clear errors without sensitive logging.

Prove Unicode text interoperability with real Ditto in both directions before spending substantial effort on the menu-bar interface.

Stage B: Native macOS app

Build a native Swift menu-bar app under mac-companion/ targeting my Mac Studio.

Separate the protocol codec, networking, clipboard integration, and interface so they can be tested independently.

Include:
- Configurable Windows address and port.
- Configurable local listening interface and port where needed.
- Password storage in Keychain.
- Explicit start/pause controls.
- Separate send and receive toggles.
- Connection/error status without displaying clipboard content.
- Controlled reconnect behavior and bounded retries.

Run clipboard integration in the intended logged-in macOS GUI user session. Do not assume a daemon or arbitrary SSH session is equivalent.

Watch clipboard changes without busy-looping. Preserve supported representations together rather than converting everything to plain text.

Prevent echo loops using local clipboard-write tracking and bounded in-memory deduplication. Account for round-trip format conversions. Do not permanently suppress a later intentional copy of identical content.

Do not invent protocol fields or identifiers that unmodified Ditto does not understand.

Define behavior for rapid copies, simultaneous changes, startup, sleep/wake, and reconnects. Do not replay an accumulated backlog of stale clipboard content after reconnecting.

Stage C: Additional formats

Implement and test in this order:
1. Unicode text.
2. RTF and HTML.
3. Images.
4. Optional files, only after separate approval.

For text, verify Unicode, terminators, and line-ending behavior.

For HTML, handle the Windows clipboard wrapper and byte-based offsets correctly. Do not fetch external resources.

For images, determine the actual Windows wire formats rather than assuming every bitmap is a BMP file. Test dimensions, row layout, alpha, and PNG/TIFF conversion where applicable.

Unsupported formats must be reported honestly and skipped safely. Do not silently replace an unsupported rich clip with empty content.

Keep file transfer disabled until specifically approved. Never automatically open received files or write to arbitrary paths.

9. ACCEPTANCE TESTS AND HANDOFF

The first milestone is complete only when, with other clipboard-sync utilities disabled:

- A new Windows copy can be pasted normally on the Mac.
- A new Mac copy can be pasted normally on Windows.
- Neither direction requires choosing an entry from clipboard history.
- No echo loop occurs.
- Unicode and multiline text survive correctly.
- Wrong-password and malformed-message cases fail safely.
- Startup and reconnect do not transmit old clipboard content.
- Pausing the companion stops its clipboard syncing.

Include a negative-control test: disable the companion's transport and confirm that the same test no longer syncs. This helps establish that the companion, not another utility, caused the successful result.

Test repeated copies, rapid changes, disconnect/reconnect, and sleep/wake. Test parser edge cases locally with synthetic fixtures.

Maintain a per-format, per-direction test matrix with:
- Test content or fixture identifier.
- Expected behavior.
- Actual result.
- Pass, fail, blocked, or not run.
- Relevant evidence.

Do not label unperformed manual tests as passing. Compilation and unit tests alone do not establish end-to-end success.

Update PROGRESS.md after each meaningful milestone and before any handoff. Include:
- Current stage and completed work.
- Exact artifact identities and tool versions.
- Important findings and evidence locations.
- Open protocol questions.
- Tests actually performed and their results.
- Current blockers.

- Reproduction commands.
- The next concrete action.

Keep the protocol specification and implementation synchronized.

Checkpoint proactively rather than relying on a last-minute warning that the context is full.

Finish with the project path, build/run instructions, tested Ditto version, supported formats in each direction, security limitations, and remaining unknowns.

Do not compare against Ditto's real source until I explicitly approve that separate final validation step.

Begin by inspecting the current workspace and existing REA setup, verifying provider readiness, and preparing the pinned artifacts. Then start focused static analysis. Do not stop at a plan.
[@Computer](plugin://computer-use@openai-bundled)