ChatGPT

Integrating Ralph Loop Workflow and PRD System into LightwaveOS

Ralph Loop Skill Integration (`.claude/skills/ralph-loop/`)

The project's design allows adding new Claude Code **skills** without conflict. The repository already contains dozens of specialized skills under `.claude/skills/` (about 28 currently) [1]. Each skill resides in its own directory with a unique name (e.g. `dispatching-parallel-agents`, `test-driven-development`, etc. [2]), so introducing a new `ralph-loop` skill folder is straightforward. No existing skill uses "ralph," so there's no naming collision. The Claude Code environment will simply load it as another capability, provided its `SKILL.md` is defined properly.

Caveats: Ensure the new skill's frontmatter has a unique `name` and description to avoid any internal identifier overlap [3]. Also consider the trigger conditions: the **Ralph Loop** skill should auto-invoke only when appropriate (e.g. when the user explicitly requests an autonomous development loop or mentions the Ralph Wiggum technique). Review existing skills to ensure the `ralph-loop` triggers won't inadvertently overlap with other keywords. There is no conflict in terms of file structure, but clarity in its purpose and triggers will prevent confusion with existing workflows (for instance, distinguish it from parallel-agent dispatching or other automation skills).

Next Steps: - Create the directory `.claude/skills/ralph-loop/` with a `SKILL.md` defining the skill. Include a clear description (e.g. "Autonomous development loop executor") and specify any necessary tools or model settings (e.g. allow shell commands if it needs to call scripts). - Define **trigger keywords** in the skill (or usage guidance) so that Claude recognizes when to activate the Ralph loop. For example, mentions of "continuous loop" or a special command could invoke it. - If the Ralph loop will execute external scripts (like a `ralph_loop.sh` or harness commands), add those to the skill's `allowed-tools` and ensure the skill is tested in a safe environment. - Test the skill in a development session: confirm it activates only on the intended cue and that it coexists with the Pre-Task Agent Selection protocol. Document its usage so team members know when to use the Ralph loop vs. standard single-step agent operation.

Extending `feature_list.json` for Ralph Loop and PRD

Adding new fields to `.claude/harness/feature_list.json` is feasible and should not disrupt existing functionality. The harness code that loads and validates this JSON does not enforce a strict schema beyond a few required fields (`id`, `title`, `status`) [4]. It will ignore additional keys it doesn't recognize. This means we can introduce fields like `"ralph_loop"` and `"prd_reference"` in each feature item without causing errors. All current items will remain valid (they simply won't have those new keys, which is fine since the harness treats missing optional fields leniently). In short, the Domain Memory Harness will **load** and **save** the JSON with extra fields intact, and continue focusing on the known fields unless we program it otherwise.

Caveats: While no runtime conflict exists, it's important to maintain consistency. We should decide the data types and meanings of the new fields: - `ralph_loop`**:** likely a boolean flag (`true/false`) indicating if a feature is meant to be tackled via the Ralph autonomous loop. This could guide the agent or harness to handle it differently (e.g. skip Pre-Task protocol and directly engage the loop). If unset or false, the feature would proceed normally. Make sure to document this new field's purpose. - `prd_reference`: a string linking to the PRD file name for that feature. For example, an item with `"id": "NEW-FEATURE-X"` might have `"prd_reference": "NEW-FEATURE-X.json"` to point to `.claude/prd/NEW-FEATURE-X.json`. Consistency in naming is key: use feature IDs as filenames so it's easy to find the corresponding PRD. Also, consider case sensitivity – since some feature IDs are uppercase with hyphens, ensure the filesystem (and any code accessing it) handles that appropriately. - The top-level schema metadata in `feature_list.json` (`"$schema": "domain-memory-harness-v2"`) is just an identifier, but if a formal JSON Schema is used for validation elsewhere, it would not know about the new fields. In practice no strict validation is done except by our harness code, so there's no breakage. We might bump a schema version or at least note in the description that version 2.1 (for example) includes optional `ralph_loop` and `prd_reference` fields.

Next Steps: - Add the `ralph_loop` and `prd_reference` keys to the relevant feature items. For example:

```
{
  "id": "FEATURE123",
  "title": "Some Feature",
  "description": "...",
  "status": "FAILING",
  "priority": 2,
  "ralph_loop": true,
  "prd_reference": "FEATURE123.json",
  ...
}
```

For existing backlog items where it doesn't apply, you can omit the fields or set `"ralph_loop": false` (the harness doesn't require uniform fields in all items). - Consider updating the `"schema_version"` in the JSON (if you treat that field formally) or adding a comment in the file to note the schema extension. This is mostly for human/agent clarity since our code won't break. - Update harness validation or add warnings as needed. For instance, you might enhance `harness.py` to log a warning if `prd_reference` is set but the file is missing, or if `ralph_loop` is true but no PRD is provided. This would proactively catch inconsistencies. - Ensure any **agents or scripts** that read `feature_list.json` (outside of `harness.py`) are adjusted to handle the new fields if necessary. If, for example, some agent logic lists keys or tries to parse items, they should simply ignore the new fields (which is usually the default). A quick search in the repo for usage of `feature_list.json` outside the harness would confirm if any changes are needed (likely none, since harness is the main accessor).

Adding a `scripts/verify.sh` Validation Script

Introducing a `scripts/verify.sh` script will not interfere with the current build or CI processes. In the repository, the `scripts/` directory contains only a few targeted scripts (like `build.sh` and `upload.sh` for PlatformIO) and nothing named "verify" exists yet. We can safely add `verify.sh` as a new utility script. It will be invoked manually (or via a git hook/CI step), so simply having it in the repo does not change any behavior by itself. The CI (GitHub Actions) workflows do not automatically run all scripts in that folder; they only execute what is explicitly configured. Therefore, adding `verify.sh` will **not** break CI unless we choose to integrate it. It also won't conflict with similarly named scripts elsewhere (the only vaguely similar script is `firmware/Tab5.encoder/scripts/validate.sh` used in a different context, which is isolated in that sub-folder) [5].

Caveats: We should design `verify.sh` to complement the existing checks: - The `.claude/harness/init.sh` script already performs a "boot ritual" verification of the project's health (toolchain installed, code compiles, etc.) [6] [7]. Our new `verify.sh` likely has a different focus – it should validate **project metadata and standards** (e.g., check that all JSON files are well-formed and conform to schemas, ensure no stray errors in domain files, etc.). There's no direct overlap, but we should avoid redundancy. For example, we probably **don't** need to compile the firmware in `verify.sh` since `init.sh` or CI covers build testing. Instead, focus `verify.sh` on things that might not be caught by a normal build, such as documentation/schema consistency. - Ensure the script exits with a proper code (0 for success, non-zero for failure) because we intend to use it in gating mechanisms (git hook or CI job). It should be as platform-independent as possible (write in Bash that works on Mac/Linux, which is fine here). - No existing CI job calls `verify.sh` yet. If we plan to integrate it later (for example, as a step in GitHub Actions or a condition for merging), we should test it locally on all supported environments to avoid surprises in automation. - Make sure `verify.sh` has execute permissions (`chmod +x`) in git so that it can be run directly. This sounds trivial, but forgetting that bit can cause the hook or CI to fail to run it.

Next Steps: - **Implement** `verify.sh` with the intended checks. For example: - Validate **feature list**: run `python .claude/harness/harness.py validate` and capture its output. This will catch issues like missing required fields or duplicate IDs in `feature_list.json` [8]. - Validate **PRD JSON schema**: for each `.claude/prd/*.json` file, use a JSON schema validator or a Python snippet to ensure required sections are present. We could write a small Python call (using `json` module and simple asserts) within the script, or leverage a tool like `jsonschema` if available. Since we control the environment, a lightweight approach is to iterate over each PRD file and check that it contains certain keys (e.g., `"overview"`, `"requirements"` arrays, etc. as per our schema design). - Check for **orphan references**: cross-verify that every feature with a `prd_reference` field actually has a corresponding file, and conversely that every PRD file is referenced by some feature (to avoid stale docs). - Lint for **common mistakes**: e.g., ensure no trailing commas in JSON (which would break parsing), or that all `status` fields in feature_list are one of the allowed values (harness does this already and would flag it as a warning). - Summarize results: print a clear PASS/FAIL summary similar to `init.sh`. For consistency, perhaps use green/red output as done in `init.sh`. - After writing the script, **test it locally** with various scenarios: - All good (it should exit 0). - Introduce a deliberate error (e.g., put a syntax error in a PRD JSON or make `feature_list.json` malformed) to see that it catches the issue and exits non-zero. - Ensure it doesn't produce false failures on a clean repo. - Keep `verify.sh` **efficient**. It should run in a matter of seconds (parsing a few JSON files and running a Python validation is quick) so that it can be used frequently (e.g., before every push). Avoid doing heavy builds or long network calls in this script. - Optionally, plan for CI integration: we could add a GitHub Actions step to run `scripts/verify.sh` on each pull request as a safety net. This can be done once the script is stable, and it would help maintain codebase hygiene (catching issues early).

Pre-Push Hook Integration and Developer Workflow

Implementing a Git pre-push hook that calls `verify.sh` is a viable way to enforce these checks without disrupting normal development. In fact, the project's CI/CD documentation already encourages **local testing before pushing** [9], which is exactly what this hook would accomplish. Since Git hooks are not version-controlled by default, we won't be adding anything in the repository's code that could interfere with others – each developer would set up the hook locally. This means there's no direct conflict with any repository files or settings by introducing a pre-push hook; it's an optional client-side enhancement.

Caveats: To ensure this doesn't hinder developers: - **Performance:** The `verify.sh` must run quickly (as noted). If it takes too long, developers might be tempted to skip it. Given our planned checks, this should be fine (a few seconds at most). We should avoid running the full firmware build here – that is better left to CI or explicit local actions, because a full compile could take minutes. - **Opt-out:** Developers should know that in urgent cases they can bypass the hook (with `git push --no-verify`). This is standard, but documenting it prevents frustration. The hook should be seen as a helpful guardrail, not an unbreakable gate. - **Setup:** Because hooks don't travel with the repo, we'll need to guide the team to install it. We might include instructions in `README.md` or a dedicated `CONTRIBUTING.md` on how to enable the pre-push hook. For example, instruct "to activate pre-push checks, run: `ln -s ../../scripts/verify.sh .git/hooks/pre-push` in your local repo". We could also supply a script (perhaps `scripts/setup-hooks.sh`) to automate this symlink creation. - **No disruption to existing flow:** Currently, developers might be using `init.sh` or manual checks; the hook just adds an automatic step. As long as our script is reliable, it shouldn't produce false positives. However, if the project has developers working on multiple OSes, ensure the shebang (`#!/bin/bash`) works universally (Windows users may need WSL or Git Bash, which is typical in such projects). - Verify that nothing else is using pre-push hooks already. It's unlikely (no mention in the repo), but if someone has a personal hook, they'd need to merge the functionality.

Next Steps: - **Document the Hook:** Add a section in the project documentation (e.g., "Development Workflow") explaining the pre-push hook. Include how to set it up and why it's beneficial (it will run `scripts/verify.sh` to catch schema or harness issues before code is pushed). - **Provide the Hook Script:** We can either instruct to symlink to `scripts/verify.sh` (treating `verify.sh` itself as the hook script), or write a small `.git/hooks/pre-push` script that calls `scripts/verify.sh`. The latter approach could allow adding a friendly message if verification fails. For example, the hook script could call `scripts/verify.sh`; if exit status is non-zero, echo "Push aborted: fix the above issues or use --no-verify to override." - **Team Adoption:** Communicate to the team that this hook is available. Perhaps not everyone will enable it immediately, but encouraging its use will improve consistency. Over time it could even be made mandatory by CI (i.e., CI runs `verify.sh` too, so even if someone skips the hook, the CI will catch issues). - **Monitor & Refine:** After enabling, gather feedback. If developers find the hook too chatty or blocking trivial things, we might refine what `verify.sh` checks or how it reports (e.g., maybe some checks become warnings that don't fail the push). The goal is to integrate quality checks seamlessly into dev flow, not to create friction. So treat the first iteration as something we can tweak.

`.claude/prd/` JSON PRD Files and Schema Alignment

Adding a new directory for PRD files (Product Requirement Documents) under `.claude/prd/` is a sensible way to integrate structured specs into the project. This will not conflict with any existing paths – currently PRDs (if any) are in unstructured formats like Markdown (e.g. `docs/PRD_WEBAPP_DASHBOARD.md` is a narrative PRD for the web dashboard [10]). By using JSON, we make the requirements machine-readable for Claude or validation scripts. Each feature can have a file like `.claude/prd/<feature-id>.json` containing its detailed requirements or design spec.

Feasibility & Schema: We should establish a JSON schema for these PRD files so they are consistent. For example, a PRD JSON might include fields such as: - `"feature_id"` or a reference to link it to the feature list item (though if the filename encodes this, it might be redundant). - `"title"` or `"name"` of the feature. - `"overview"` or `"description"`: a high-level description of the feature (could mirror the short description in `feature_list.json` but expanded). - `"requirements"`: possibly an object with sub-sections like functional vs non-functional requirements, or a list of user stories/criteria. For instance, we might structure it as:

```
{
  "overview": "...",
  "functional_requirements": [ "...", "..." ],
  "nonfunctional_requirements": [ "...", "..." ],
  "acceptance_criteria": [ "...", "..." ]
}
```

This is just an example – the actual schema can be tailored to what information we need. The key is to capture the same kind of info currently found in our PRD Markdown docs (which include an Overview, Must-have/Should-have features, constraints, etc.) but in JSON form. - We might also include `"version"` or `"last_updated"` metadata in each PRD file to track currency.

There's no technical barrier to storing these files. The `.claude` directory is already used for AI-related config, so it's a logical place for PRDs that the AI (Claude) will reference. Keeping them in JSON ensures Claude can parse them systematically (Claude Code can read JSON easily), and our `verify.sh` can validate them.

Caveats: - **Schema Definition:** We will need to formally define what fields are expected. This could be in the form of a JSON Schema document (for developer reference or even programmatic validation). For example, define that each PRD JSON *must* have an `"overview"` and a list of `"requirements"`, etc. Without a clear schema, different contributors might structure PRD files differently, defeating the purpose of standardization. - **Data Duplication:** Some information may exist both in `feature_list.json` and the PRD file (e.g., the title or acceptance criteria). We should decide how to handle that. One approach is minimal duplication: the feature_list entry holds the basic info and maybe a pointer to PRD, and the PRD JSON holds extended details. For instance, acceptance criteria could live in either place – if they are high-level, keeping them in the feature list might suffice, but if they expand significantly, the PRD might be better. We need to ensure the agent and developers know where to look for what. Perhaps use the PRD for broader context and detailed requirements, and keep the feature_list for the summary and status tracking. - **Maintaining Sync:** If a feature's scope changes, we must update both the feature_list entry and its PRD JSON. This introduces an **ownership question** (who is responsible for updating PRDs? – more on that below in fragility). For now, just note that we'll need process discipline to keep the two in sync (the `prd_reference` in feature_list ensures a link, but the content needs human upkeep). - **File Naming:** Using the exact `feature_id` as filename is straightforward, but ensure the naming rules are clear (IDs here contain hyphens and uppercase letters; that's fine on most filesystems but could be an issue on case-insensitive systems if someone isn't careful. It's okay as long as we consistently refer to them in the correct case). - **Claude Integration:** We should verify how Claude (in Claude Code) will access these PRDs. Likely, the agent can simply open the JSON file with the `Read` tool. Claude is pretty good at reading JSON, but we might need to format prompts or provide guidance on how to interpret the PRD content. For example, if a skill is tasked with using the PRD, it should know the JSON structure (hence the importance of standard schema). - **Example Alignment:** We can take cues from our existing PRD for the web dashboard (which is human-written in markdown). That document has sections like "Must-Have Features (P0)" etc. [11]. In JSON, this could be an array under `functional_requirements` categorized by priority. We might not encode priorities in the JSON and instead rely on just listing them and letting context (or a priority field) indicate importance. The exact modeling is flexible, but should cover: objectives, features, constraints, and acceptance criteria.

Next Steps: - **Define the JSON schema** for PRDs. It might help to write a template JSON for a hypothetical feature, including all desired sections. For instance:

```
{
  "$schema": "lightwave-prd-v1",
  "feature_id": "NEW-FEATURE-X",
  "title": "New Feature X",
  "overview": "High-level summary of what this feature is and why it's
needed.",
  "functional_requirements": [
    "Capability A must do ...",
    "Capability B should allow ...",
    "... etc ..."
  ],
  "nonfunctional_requirements": [
    "Performance target ...",
    "Security considerations ...",
    "... etc ..."
  ],
  "acceptance_criteria": [
    "Criterion 1: ...",
    "Criterion 2: ...",
    "... etc ..."
  ]
}
```

This is just a starting point – tailor it to the kinds of features we have. Some features might not need both functional and non-functional sections (e.g., a small bugfix might just have an objective and criteria). The schema could allow certain sections to be optional or empty. - **Implement schema checks in `verify.sh`:** Once the schema is set, update `verify.sh` to validate each `.json` against it. If we use a formal JSON Schema definition, we could integrate a tool (perhaps a Python library `jsonschema`) to do this. Or, for simplicity, hard-code checks like "does this JSON have an `overview` field (critical) and is `functional_requirements` an array (critical)?" etc. The script can report if any PRD file is missing required fields or has type mismatches. - **Create initial PRD files:** Start by writing PRD JSON for one or two existing features to serve as examples. Perhaps choose a larger feature or recent project (maybe the WiFi dashboard or an effect system upgrade) to pilot the format. This will let us see if the JSON format adequately captures the info or if we need additional fields. - **Review and Iterate:** Have the team (or product stakeholders) review the JSON PRD format to ensure it's human-friendly as well (JSON is a bit less readable than Markdown, but it's structured – we could always auto-generate a pretty Markdown or HTML from it if needed for human readers). Adjust the schema if needed based on feedback. - **Update usage guidelines:** In `CLAUDE.md` or a relevant doc, note that detailed feature specifications may reside in `.claude/prd/`. Instruct agents (through prompts or skill behaviors) to check for a `prd_reference` and, if present, load that JSON for full context before coding. This will improve the agent's understanding of the task at hand.

Potential Fragility, Conflicts, and Ownership Considerations

Integrating the Ralph loop and PRD system introduces new moving parts. It's important to identify who/what "owns" each part of the workflow and where things might break or conflict. Below are key areas to watch:

- **Concurrent Agent Execution:** The Domain Harness includes a lock mechanism to prevent simultaneous modifications to `feature_list.json`, but it is advisory (non-blocking) [12]. If one agent (or the Ralph loop) acquires the lock on a feature, the harness will warn but not strictly stop another from interfering. This could lead to race conditions if, say, someone manually runs an agent on Feature A while the Ralph loop is also working on Feature A. To avoid this fragility, establish a practice that Ralph's loop runs exclusively. When the `ralph_loop` flag is true for a feature, maybe treat that feature as under "autonomous development" and don't manually assign it to other agents simultaneously. We might later enhance the lock to be stricter (fail a second lock attempt unless forced) if needed. But for now, coordination is key – team members should know a Ralph loop is in progress (perhaps via log output or status in `agent-progress.md`) and refrain from parallel changes on that item.

- **Harness vs. Loop Status Updates:** We need clarity on **ownership of feature status updates**. Currently, after an agent finishes a task, someone (or the agent itself via the harness API) should mark the feature as PASSING and record evidence. Who will do this in the Ralph loop scenario? The ideal approach is that the **Ralph loop process itself updates the** `feature_list.json` via harness functions when a feature is completed. For example, if Ralph solves "FEATURE123", it should call `harness.record_attempt(... result="PASSED" ...)` and `harness.update_status(... new_status="PASSING" ...)` with appropriate evidence [13] [14]. This ensures the single source of truth (feature_list.json) is always current. If the loop were to stop after completing a feature but not mark it, the harness would still consider it unfinished, causing confusion. Therefore, part of integrating Ralph is possibly writing a small hook or extending the skill to perform these JSON updates. The **ownership** of marking tasks done thus lies with the automation (Ralph) rather than a human, in these cases. We should test this mechanism thoroughly to avoid the loop "thinking" it's done while the harness remains unaware (an inconsistency that would be fragile). Also, decide what happens if Ralph partially implements a feature but cannot finish – does it mark it BLOCKED with a reason? Does it leave it FAILING? Having the loop communicate status changes back to the harness in all outcomes will be important for transparency.

- **PRD Content Ownership:** Who writes and maintains the PRD JSON files? This is an **ownership question** between the human team and the AI agents. Two models:

- A **human (product manager or lead developer)** writes the PRD JSON for a feature before development starts. In this case, the PRD is the authority on what needs to be built. The AI (Claude) should strictly adhere to it, and if implementation deviates, that's a bug. Here, verify scripts and possibly the harness could check that each acceptance criterion in the PRD is addressed by tests or code (though automating that fully is complex). The risk of fragility here is low as long as the human-written PRD is clear – the AI will just follow it.
- Alternatively, the **AI agent itself could draft the PRD** (perhaps via a specialized "PRD generation" skill) based on a high-level request. That draft would then ideally be reviewed by a human. If we go this route, we must ensure the agent-generated PRDs are validated (both against schema and for content sanity) before trusting them. An unclear ownership scenario would be if the AI generates a PRD and immediately starts coding to it without human approval –

this could lead to going down a wrong path. To mitigate that, build a review step: e.g., the agent creates `.claude/prd/FEATURE123.json`, then perhaps our workflow or harness could mark the feature as "pending approval" until a human sets a flag or edits the PRD to confirm it. This is more of a process design, but it's worth considering to avoid the AI confidently implementing an incorrect spec.

- In summary, **recommendation:** PRDs should be treated as authoritative specifications. Ideally a human should sign off on them (given they represent "what to build"). The AI (Ralph loop or any agent) should then use them as input. We should update team process docs to reflect this: e.g., "Before running Ralph loop on a complex feature, ensure a PRD exists and is approved."

- **Verification Responsibility:** Now that we have a `verify.sh` script, what happens when it finds an issue? For example, a missing PRD file or a schema violation. In CI, that will fail the build – meaning it's the developer's responsibility to fix (which is correct). During development, if the pre-push hook fails, the dev must resolve it. There is a potential unclear area if the AI agent itself encounters a schema validation need – e.g., could we ever have the AI attempt to fix its PRD to satisfy the schema? Possibly, but that's probably overkill. It's safer to assume structural checks are a human/dev concern. However, if an agent is writing JSON (like a PRD or updating feature_list), it might be worth having the agent call `harness.validate` or a schema check on its output as part of its workflow (this way it catches mistakes in the moment). In the **Ralph loop skill implementation**, we can incorporate a step where after each iteration or before finalizing, it runs `harness.validate` to ensure it didn't produce an invalid structure. This would catch, say, forgetting to add evidence for a PASSING status – something our harness validation would flag [15]. So, the fragility of data integrity can be addressed by making either the agent or the hook double-check, rather than relying on a human noticing later.

- **Documentation and Process Changes:** With multiple ways to develop (manual agent-by-agent vs. autonomous Ralph loop), clarity in project documentation is key. Currently, `CLAUDE.md` enforces a rigorous agent selection protocol for each task [16] [17]. We should update these docs to explain where Ralph loop fits in. For example, we might add: "If the feature is marked for autonomous implementation (`ralph_loop: true`), you may use the Ralph Loop skill to have Claude attempt the entire development cycle. In this mode, the Pre-Task Agent Selection Protocol may be bypassed or altered, since the loop will handle planning and coding iteratively." This ensures that an AI agent following the old protocol doesn't get confused by the new fields or workflows. Also, clearly state that the presence of a PRD (`prd_reference`) means detailed requirements are available and **must** be adhered to by any agent working on that feature. Unclear ownership could arise if agents aren't instructed properly – for instance, an agent might think it should come up with acceptance criteria when a PRD already defines them. So we must bake into the prompts that PRD files, when present, are the source of truth for requirements.

- **Edge Cases & Fragility:** Consider error conditions for the Ralph loop:

- What if the loop fails to converge? (E.g., Claude gets stuck in a loop or the task is too hard to complete autonomously.) We should define how to stop gracefully. Perhaps we set a maximum number of iterations or a time limit. The Ralph implementation we're basing this on has "intelligent exit detection" to prevent infinite loops [18] [19]. We should ensure those triggers are in place. If the loop decides it cannot finish, it might mark the feature as BLOCKED and provide a `failure_reason` in the attempts. That way the harness knows the feature wasn't completed and why [20] [21]. This scenario needs to be handled so that it doesn't just hang. Ownership of deciding "I give up" in this case is with the loop logic.

- What if multiple features are marked `ralph_loop: true`? Is the intention to run them one after the other, or even in parallel (with multiple Claude instances)? Running in parallel would raise more concurrency issues (multiple locks, etc.) and likely overwhelm resources or context windows. It's probably best to do one at a time. So maybe have the harness or a wrapper ensure that only one loop runs at once. The harness `lock` helps on a per-feature basis; if we attempted two loops on two different features, it might technically work but could hit rate limits with Claude's API. It's a use-case to be cautious about. Define that clearly: e.g., "Don't run more than one Ralph loop simultaneously unless you're confident in resource limits," or better, have an orchestrator that sequences them.
- Make sure the verify/CI process accounts for PRD changes: If a PRD is updated, do we require re-running the loop? Not automatically, but at least ensure any changes to PRDs are also checked by `verify.sh` (which we have covered). We might include PRD JSON in code reviews like any other artifact.

In summary, **assign clear ownership** as follows: - **Feature Implementation**: If `ralph_loop` is true, the Ralph skill/agent "owns" driving that feature to completion (or deciding it can't). If false, a human or single-run agent will own it. - **Spec (PRD) Creation**: Humans should own initial creation and approval of PRDs. Agents can assist or draft, but final responsibility lies with a human to ensure correctness. - **Maintaining** `feature_list.json`: The harness (through agents or manually) continues to own the truth of feature status. Even in Ralph mode, we ensure the harness is updated programmatically by the agent – so the data ownership stays centralized. - **Validation**: `verify.sh`/CI owns catching consistency errors (schema, missing files, etc.). When it flags something, a human must resolve it (either by fixing the JSON or guiding an agent to). - **Process & Protocols**: Update the documentation (owned by project maintainers) to reflect these new practices so everyone (including new agents spun up) understand the workflow.

By addressing these points, we minimize the fragility introduced by the new system and keep the development loop running smoothly. Each component – PRD files, harness, Ralph loop, verify script, and human oversight – will have a defined role in the overall workflow. This clarity will help prevent conflicts and ensure the project's AI-assisted development process remains reliable and effective.

---

1 2 16 17 CLAUDE.md
https://github.com/synqing/Lightwave-Ledstrip/blob/f81303268c79c422c9ffbf7c5cc4727956ae1fd4/CLAUDE.md

3 SKILL.md
https://github.com/synqing/Lightwave-Ledstrip/blob/f81303268c79c422c9ffbf7c5cc4727956ae1fd4/.claude/skills/claude-command-builder/SKILL.md

4 8 12 13 14 15 20 21 harness.py
https://github.com/synqing/Lightwave-Ledstrip/blob/f81303268c79c422c9ffbf7c5cc4727956ae1fd4/.claude/harness/harness.py

5 validate.sh
https://github.com/synqing/Lightwave-Ledstrip/blob/f81303268c79c422c9ffbf7c5cc4727956ae1fd4/firmware/Tab5.8encoder/scripts/validate.sh

6 7 init.sh
https://github.com/synqing/Lightwave-Ledstrip/blob/f81303268c79c422c9ffbf7c5cc4727956ae1fd4/.claude/harness/init.sh

9 README.md
https://github.com/synqing/Lightwave-Ledstrip/blob/f81303268c79c422c9ffbf7c5cc4727956ae1fd4/docs/ci_cd/README.md

10 11 PRD_WEBAPP_DASHBOARD.md
https://github.com/synqing/Lightwave-Ledstrip/blob/f81303268c79c422c9ffbf7c5cc4727956ae1fd4/docs/PRD_WEBAPP_DASHBOARD.md

18 19 GitHub - frankbria/ralph-claude-code: Autonomous AI development loop for Claude Code with intelligent exit detection
https://github.com/frankbria/ralph-claude-code