AI-Augmented Lean

AI-Augmented Lean 4 Proof Development System for *OperatorKernelO6*

Introduction & Project Constraints

OperatorKernelO6 is a highly constrained Lean 4 formalization project with ambitious goals: proving *strong normalization* (SN), *confluence*, arithmetic properties, and Gödel's theorems in a **procedural, axiom-free** manner*[1][2]*. The core of this project is a custom inductive *Trace* type (with constructors like *delta*, *merge*, *recΔ*, *eqW*, etc.) and a well-founded *ordinal μ-measure* used to prove termination of rewrite rules*[3][4]*. All logic, arithmetic, and even Gödel encoding are built *internally* via this trace-based system – **no Peano axioms, no truth tables, no external logical frameworks***[1]*.

Strict rules govern development, enforced by project guides *AGENT.md* and *ordinal-toolkit.md*. In particular:

- **No new axioms or unchecked assumptions:** All proofs must be by construction. The *Prime Directive* for AI assistants is *"Don't hallucinate lemmas or imports. Don't add axioms."[5]*. If the AI is unsure about a required fact, it should **raise a constraint blocker** (flag it for human attention) rather than inventing one*[5]*.
- **Whitelisted library usage:** Only specific Lean 4 modules and lemmas are allowed. The **ordinal toolkit** consolidates *all* permitted facts about ordinals and arithmetic (e.g. *omega0_pos*, *Ordinal.mul_lt_mul_of_pos_left*, etc.)*[6][7]*. If a needed lemma isn't in this catalogue (or in *AGENT.md §8*), the AI must follow a strict 4-point check (no new axioms, correct type-class, small import footprint, kernel-safe proof) before using it*[8][9]*. New lemmas passing this test are added to *ordinal-toolkit.md* for future use*[10]*.
- **Ordinal tactic legality:** When reasoning about ordinals, the proofs must use *ordinal-specific monotonicity lemmas* and follow a structured pattern. For example, one must call *Ordinal.opow_le_opow_right fully qualified* (not the generic version) and prefer *Ordinal.mul_lt_mul_of_pos_left* over generic *mul_lt_mul_of_pos_right[11][12]*. Certain "red zone" lemmas (e.g. *add_lt_add_right* for ordinals) are forbidden because they break the monotonicity conditions*[13][14]*. The AI assistant must **enforce these patterns** – every proof of $\mu\ t > \mu\ u$ for a rewrite step should follow the standard 7-step *μ-measure playbook* (assert positivity, lift through exponents, rewrite sums via *opow_add*, apply product inequalities, absorb finite addends, bridge successors, then simplify)*[15][16]*.
- **Kernel code immutability:** The six constructors and eight rewrite rules of the kernel are fixed*[17]*. The AI should never alter or "simplify" them, and must treat any attempt to do so as a serious violation. Proofs should work with the kernel as given.
- **Meta-level freedom (with caution):** Outside the kernel, we *are* allowed to use Lean's meta tools (classical logic, well-founded recursion, tactics like *simp*, *linarith*, *ring*) as needed*[18][19]*. However, *any* tactic or lemma use must respect the whitelist and not introduce inconsistency. For instance, *ring* and *linarith* are explicitly whitelisted for arithmetic side conditions*[19][20]*. The AI should avoid any *unsafe* or *irreversible* tactics (no *unsafe* definitions, no *sorry/admit*, etc.). Every suggestion must be *kernel-checkable* by Lean.

Given these strict constraints, an **AI-augmented development environment** must assist the user in **structuring complex proofs, debugging Lean syntax, selecting correct ordinal lemmas**, and iterating through compile-check-fix loops *without ever violating the rules*. Moreover, the user has **limited Lean experience**, so the setup should be as automated and user-friendly as possible, catching mistakes early and providing guidance at each step. The user prefers VS Code, but is open to alternatives if they dramatically improve results. They also have substantial AI resources: enterprise ChatGPT (GPT-4) access, Anthropic Claude (Pro tier), Perplexity AI, Google's upcoming *Gemini* model, GitHub Copilot, Sourcegraph **Amp**, and Continue.dev. Additionally, they have a budget of cloud credits ($25k on GCP) but little cloud ops experience.

Our task: Design an end-to-end solution – a toolkit of local and cloud-based AI and dev tools – that integrates with Lean 4 in VS Code to provide: (1) automated **μ-measure checks** and ordinal proof assistance, (2) interactive **error feedback loops** (AI suggests fixes when Lean errors occur), (3) strict **enforcement of project rules** (no illicit lemmas or tactics), and (4) a streamlined **model orchestration** (using the right AI model for the right subtask to minimize hallucination and maximize math accuracy). We will present a complete operational plan with ranked tool options, setup instructions, and rationale for each choice.

Environment Setup: Lean 4 and VS Code

Before integrating AI, it's essential to set up a robust Lean 4 environment for interactive proof development. We recommend using **Visual Studio Code** with the Lean 4 plugin – this is the officially supported, "rich" dev environment for Lean*[21][22]*. The Lean VS Code extension provides syntax highlighting, on-the-fly type-checking, and an interactive goal state panel, which are invaluable for formal proof work*[21]*. Steps to get started:

1. **Install Lean 4** – If not already installed, the simplest method is via the VS Code extension (it can install Lean automatically via the *elan* tool). In VS Code, open the Extensions panel and search for "Lean 4", then install the official Lean 4 extension*[21][22]*. The first time you open a Lean file, the extension will guide you through installing Lean (and possibly Mathlib). This yields a working Lean compiler and standard library on your system.
2. **Set up the project** – Open the *OperatorKernelO6* project folder in VS Code. Ensure the Lean toolchain version matches the project's needs (Lean ≥ 4.6 is likely required, since *termination_by* is used*[23]*). The project should have a *lakefile.lean* or *lakefile.toml*; run *lake update* to fetch Mathlib dependencies if needed. Mathlib 4 (v4.8 or later) is required for the ordinal library*[24]*, so ensure your mathlib is up to date (the Lean extension can do this or run *lake exe cache get* to download compiled mathlib).
3. **Verify Lean is working** – Open a Lean source file (for instance, *Termination.lean*) in VS Code. You should see Lean's **infoview** panel showing the goal state or errors as you place the cursor in a theorem. If the environment is correct, existing proofs in the project should compile without error. If you see *unknown import: Mathlib...*, it means mathlib isn't downloaded – run *lake update*. If you see other errors, resolve them now (or use the AI assistants once configured).

Tip: Lean's extension adds a ∀ button in VS Code's top-right – clicking it opens a documentation pane and the Lean **Setup Guide***[25][26]*, which can help troubleshoot installation issues. Also, if on Windows, ensure the **Lean server is on the PATH** or let the extension manage it. With Lean up and running in VS Code, you now have

immediate feedback on proofs: as you edit, errors will be flagged and you can hover to see messages. This tight feedback loop is crucial for the AI integration to work effectively.

AI Integration Overview

With the Lean environment ready, we now augment it with AI assistance. The goal is to have **AI "co-pilots"** that can: propose theorem proofs or outlines, suggest lemmas, fix syntax errors, and enforce the project's strict constraints. We recommend a **multi-tiered AI toolkit** in VS Code, each component serving different needs:

- **Autonomous Coding Agent (AI-as-developer):** A tool that can take high-level instructions ("prove this theorem" or "fix this error") and autonomously edit code, compile it, and iterate until the task is done. For this, **Sourcegraph Amp** is our top choice (Rank #1), given its ability to use advanced models and perform iterative code edits and compilation checks*[27]*. As a strong alternative (Rank #2) or supplement, **Continue.dev** offers an open-source approach to similar agentic coding, with more customizability to inject our specific rules.
- **Interactive Chat/Editing Assistant:** for step-by-step guidance, explanation, or fine-grained edits, we suggest using an AI chat extension. This could be **Continue.dev's chat mode** (if installed), or **GitHub Copilot Chat** (if you have access), or an OpenAI ChatGPT extension. These allow you to highlight code and ask questions like "What does this error mean?" or "Suggest a proof for this lemma." They're less autonomous than Amp, but give you control over each step.
- **Inline Code Completion:** We recommend keeping **GitHub Copilot** enabled for real-time code completion as you type. Copilot (Rank #3 in our list) can intelligently suggest the next line or two of a proof, which is useful for boilerplate like pattern-matching or repetitive tactic sequences. It won't ensure compliance with project rules (it might propose a lemma that's not allowed), but when used carefully in conjunction with the project docs, it can speed up routine typing.
- **External QA Assistants:** Sometimes you may need to step back and ask conceptual questions or search the broader web. For this, you can use **ChatGPT (GPT-4)** in the browser (ChatGPT Teams interface) or **Anthropic Claude** in their console/Slack integration, as well as **Perplexity AI** for web queries. These are not integrated into VS Code, but they are useful for tasks like "explain ordinal normalization in simple terms" or "find references on confluence proof techniques". They are ranked lower (#4) for direct coding help because of the friction of copy-paste and their tendency to hallucinate code, but they excel at high-level guidance and large-context analysis (Claude, for example, can take in the entire *ordinal-toolkit.md* and answer questions about it given its 100k-token context window).
- **Full-project context:** Amp can **see your entire repository** at once and use it as context for the AI. It will read and follow the *AGENT.md* rules file by design*[28]*, which is perfect for enforcing the OperatorKernelO6 guidelines. (Ensure that your *AGENT.md* in the repo root contains all the rules – which it does – because Amp expects a single rules file*[28]*.) Amp's ability to incorporate the whole codebase means it knows about your custom *Trace* type, the specific lemmas in *ordinal-toolkit.md*, and existing proofs, minimizing the chance of suggesting non-existent or banned lemmas.

- **Autonomous compile-edit cycle:** Amp doesn't just generate code once – it can **run commands like compiling the project, running tests, etc., and then adjust the code based on the results**[27]. In our context, this means Amp can attempt to fill in a proof, notice from Lean's output that an error occurred (e.g. "unknown identifier" or "type mismatch"), and then *automatically* make further edits to fix the issue, looping until Lean passes or a logical blocker is hit. This aligns exactly with the "recursive compilation/fix loops" the user needs. Essentially, Amp will simulate what a diligent developer would do: write a proof, run *lake build*, see error, fix, and so on.
- **High-quality AI models:** Amp uses "frontier models" under the hood[29]. As of mid-2025, it reportedly leverages **Anthropic Claude** (which has excellent understanding of instructions and large context) for its reasoning[28]. This is advantageous because Claude's 100k context can hold *AGENT.md*, *ordinal-toolkit.md*, and the current Lean file content together, so it always remains aware of the ordinal rules and tactic whitelist. (Amp's model choice might evolve – possibly integrating GPT-4 or others – but it abstracts that detail away. It focuses on *outcome quality*[30], so you don't manually switch models – it will use whatever is best to follow your request.)
- **Automated rule adherence:** Amp is explicitly designed to **"adhere to coding rules/philosophy defined"** by the user[31]. In practice, after loading *AGENT.md*, Amp will avoid disallowed patterns. For example, if your *AGENT.md* says "never use *mul_le_mul_left* (unqualified) for ordinals," Amp should recognize that and instead use the allowed *Ordinal.mul_le_mul_iff_left* or primed variants. If Amp is unsure about a lemma (say it wants an ordinal lemma not in the toolkit), it will likely either do a code search or ask for confirmation (this might appear as it leaving a note or *CONSTRAINT BLOCKER* in its thread, per the defined protocol[32]).
- **Integration into VS Code:** Amp provides a VS Code extension with a chat panel and the ability to apply edits. You interact with it by typing commands or questions. For example, you can ask: *"Amp: Prove the theorem mu_lt_rec_zero following the μ-measure playbook."* Amp might then output a plan and directly start editing *Termination.lean* to insert the proof, step by step. It will show a live commentary of what it's doing (e.g. "Adding have h0 : …", "Applying Ordinal.le_mul_right lemma") as it works. By default Amp auto-applies edits to your files[33][34], but you can always review changes via VS Code's Git diff. This auto-apply speeds up the workflow, letting you review after the fact rather than confirming each edit (though if you prefer stepwise approval, you could instruct Amp to present a plan for approval first[35]).
- **Compilation and test running:** You can instruct Amp to **"compile the project and ensure all proofs complete"**, or even to run any fuzz tests you have. Amp can execute terminal commands, so it can run *lake build* or a custom test script. For example, after writing all SN proofs, you might say: "Verify that there are no sorrys or axioms and run the random fuzz tests." Amp will run those and report back. It's able to make code adjustments if a test fails or if a forbidden axiom slipped in, etc., making it a truly continuous integration assistant.
- *"Prove the merge cases of strong normalization: for each rule involving merge, prove μ decreases (use the toolkit lemmas)."*
- **User-controlled model selection:** Continue allows you to use *any* model via API – OpenAI (GPT-4, GPT-3.5), Anthropic, local models, etc.[39][40]. Given you have GPT-4 and Claude API access, you can configure Continue to use GPT-4 for certain tasks and Claude for others. For example, you might use GPT-4 (which is excellent at formal logical tasks) as the primary for generating proofs, but use Claude for a "referee" agent that double-checks for rule compliance (due to Claude's larger context window for reading *ordinal-toolkit.md* in full). Continue makes such model-switching possible, whereas Amp currently abstracts model choice away.

- **Custom rules and tools:** Continue is built to let developers define their own *agent behaviors and rules*. You can write extensions or config files that encode things like: "On file save, if any Lean errors, trigger an agent to fix them." or "Before accepting an AI-suggested lemma, call a custom script to run *#print axioms* on the file to ensure no new axioms were introduced." The Continue documentation highlights that you can integrate **MCP (Multi-Command Pipeline) tools** and define rules for various scenarios[41][42]. For instance, one could create a **"Lean Proof Checker" agent** that monitors the repository and whenever a proof is completed, it automatically runs a Lean linter to check for forbidden patterns (like usage of disallowed ordinal lemmas) and alerts if any are found. This would effectively automate *μ-checks and ordinal-helper enforcement* beyond what Lean's own typechecker does. Setting this up might involve writing a small Python script or YAML config that Continue's CLI can run in the background, leveraging events (Continue supports background agents triggered on events like file changes or PRs[41]).
- **Interactive workflow in-editor:** Continue offers multiple modes in VS Code – Chat, Edit, and Autocomplete[43][44]. In **Chat mode**, you can converse with the AI similarly to Amp's chat, asking for explanations or next-step suggestions. In **Edit mode**, you can highlight a specific piece of code and give an instruction (e.g. "inline this lemma usage" or "fix the type error in this tactic"), and it will modify the code directly. In **Autocomplete mode**, Continue can also provide inline completions as you type (much like Copilot, though Copilot is generally more trained for this purpose). This multi-modal approach is helpful: for a tough proof, you might start in chat to discuss strategy ("How do I prove confluence? What lemmas will I likely need?"), then use edit to apply a particular fix it suggests.
- **No vendor lock-in / privacy control:** Since Continue is open-source and can be self-hosted, you have full control over data. You could, for example, run Continue's backend on a GCP VM using your credits, ensuring all code stays on that machine. It even supports running local LLMs via integrations like Ollama or LM Studio[45][46]. While the current task probably benefits from the power of GPT-4 (so using OpenAI's API), the option to switch to a self-hosted model in the future might appeal if sensitive code or cost control becomes an issue.
- **Community and Extensibility:** Continue has a concept of community-contributed "tools" and configurations for various languages (they mention Rust, React, etc. rules)[47]. It's plausible to either find or build a **Lean-specific agent config** that encapsulates known best practices. For example, one could encode the μ-measure playbook as a sequence of prompt instructions: "whenever solving *mu x < mu y*, ensure to do steps A through G." You could incorporate the content of *ordinal-toolkit.md* directly into the system prompt of the agent. Continue's flexibility means you can fine-tune the prompts and behavior more than with Amp.
- **Boilerplate generation:** e.g. writing the structure of inductive definitions or instances. If you start typing *theorem strongNorm_{something} : ... := by*, Copilot might auto-complete a skeleton proof using the patterns it learned from similar contexts (it might have seen some Lean mathlib proofs or even (if public) earlier versions of this project). It can fill in obvious steps like case splits (*cases h* etc.) or the repetitive ordinal arithmetic steps after seeing a few examples.
- **Reducing syntax errors:** Copilot often suggests correct syntax which can help a non-expert avoid small mistakes. For instance, if you write *cases h <;> simp [size]*, Copilot might know to add *; linarith* if it saw that pattern. It's not always correct, but if Lean's error messages confuse you, Copilot's guess might sometimes circumvent a few iterations.

- **VS Code ecosystem integration:** Copilot works seamlessly in the editor without any extra prompts – suggestions just appear as you type. This low-friction interaction means it's always there, even when you're focusing on something minor that you wouldn't bother Amp/Continue with. For example, when writing a new lemma in *ordinal-toolkit.md*, Copilot might suggest the Lean code for it if it recognizes the math.
- **ChatGPT (GPT-4) via Teams**: Since you have ChatGPT Teams, you can have shared chats to collaborate on. You might use this to discuss high-level strategy: e.g., *"How does one typically prove confluence in a rewrite system? Can you outline the steps?"* GPT-4 can give you a clear explanation or even pseudo-code of a proof, which you can then implement in Lean with the help of the coding agents. Another use is for **translating Lean errors or tactics**: if a Lean error message is confusing, paste it into ChatGPT and ask for interpretation. GPT-4, with its training likely including some Lean knowledge, can often explain the cause (though always double-check its answer with Lean's docs).
- **Anthropic Claude**: Claude's strength is its huge context window. You can literally paste the entirety of *ordinal-toolkit.md* and *Termination.lean* into Claude and ask: *"Do you see any use of a non-whitelisted lemma or tactic in this code?"* This kind of large-scale audit is something GPT-4 8k cannot do in one go, but Claude can*[27]* (and GPT-4 32k might, if available to you). Claude might point out, for example, "In *Termination.lean* line 7, *Mathlib.Algebra.Order.Monoid.Defs* is imported, which includes *mul_le_mul_left* – ensure it's not used for ordinals." This cross-checks your enforcement. Similarly, if you have a *draft paper* (*draft_paper.md* perhaps in your files) describing the theory, Claude can ingest it and answer questions to clarify the theory for you.
- **Perplexity (or Bing with browsing)**: These tools are essentially search engines with integrated LLMs. They can be used to find if someone has solved a similar problem or to get quick answers with sources. For example, you might search "Lean 4 strong normalization ordinal measure example" to see if any blog or mathlib example exists. Perplexity will return some results with citations. Given your project is quite unique, you might not find exact matches, but sometimes learning how Lean's standard lib does well-founded recursion on ordinals (if anywhere) could be insightful. You could then share that info with ChatGPT to see how to adapt it.
- **Lean Linter and Axiom Checks:** Lean 4 has a built-in way to check that no new axioms are introduced: after building, run *lean --print-axioms <File>* or use *#print axioms <decl>* in a file to see if it depends on any axioms. As a habit, after a proof is done, do *#print axioms MyTheorem* to ensure it's axiom-free (the CI can automate this for all theorems). You should also add a **project CI step** that greps for forbidden patterns (like *mul_lt_mul_of_pos_right* or use of *sorry*). This can be done with a simple script and run in GitHub Actions or a GCP Cloud Build. With $25k credits, spinning up a lightweight VM or using GitHub's included CI minutes to run these checks on each commit is feasible. This CI won't directly involve AI, but it's a safety net to catch anything that slips past.
- **Fuzz Testing Harness:** The project deliverables mention *fuzz tests* for deep rewrites*[52]*. If you have a Lean file that generates random *Trace* terms and normalizes them to check properties, make sure to integrate that with your build or test process (*lake test* could run it if set up as a test). Amp or Continue can be instructed to run these fuzz tests after major changes. If a random test fails (meaning likely a counterexample to confluence or a bug in a rule), that's a big deal – the AI might not automatically solve it since it could indicate a missing case or lemma. However, you can use the AI to analyze a failing case: feed the counterexample into ChatGPT and ask for insight, or ask Amp "explain why this trace didn't normalize as expected." This might guide you to the fix.

- **Mathlib Documentation & Search:** Bookmark **Mathlib4 documentation** site (or use *#help* and *#search* in Lean). For instance, *#find _ < omega0* might show lemmas about ω (omega0) if you forget their names. The AI will rely on these too (library_search tactic, etc., though your project might not allow arbitrary *library_search* if it suggests out-of-toolkit lemmas). Use **Loogle***[49]* for any lemma names that the toolkit lists (it can show the source and usage examples).

Crucially, all these AI systems must be funneled through the lens of our project's constraints. We will set up **shared knowledge** (the content of *AGENT.md*, *ordinal-toolkit.md*, etc.) to inform the AI agents, and use Lean's feedback as the ultimate arbiter of correctness. The following sections detail each major AI tool, how to configure it, and how they work together. We also provide a **ranked recommendation** with rationale for each.

Rank #1 – Sourcegraph **Amp**: Autonomous Agent with Rule Enforcement

Sourcegraph Amp is a cutting-edge AI coding agent that behaves like a "junior developer" working on your codebase autonomously*[27]*. We recommend Amp as the primary assistant for this project due to several key strengths:

Setup & Usage: To set up Amp in VS Code, install the **Amp extension** from the VS Code Marketplace (or Sourcegraph's site). After installation, you'll sign up or log in (GitHub or Google OAuth is typically used). Once logged in, open your project in VS Code. Amp should detect the repository and load context (note: ensure you've saved any unsaved *AGENT.md* changes, as it reads from disk). You can then open the Amp panel (often a tab or sidebar) and see a prompt area.

Start by asking Amp a simple question to warm it up, e.g. "Explain the goal of this project" or "List the rewrite rules in the kernel." This tests that it has read the code. Given Amp's design, it likely has already parsed the files. Next, you can try a real task, like:

Amp will reason about what files/functions to open (perhaps it knows *mu_lt_merge_void_left* is one target, etc.), then make the changes. Monitor its suggestions and the Lean infoview; Amp should catch errors itself, but you should still verify that the final proof is logically what you intended. If Amp proposes anything suspicious (like introducing an import not in the whitelist, or using a lemma you don't recognize), pause and double-check. Amp is generally good at not introducing unauthorized changes (as long as rules are clear) – *"Don't hallucinate imports"* was literally in the project's Prime Directive*[5]*, which Amp will heed.

Rationale: Amp is ranked #1 because it dramatically **boosts productivity and safety**. It essentially acts as an expert Lean developer who never gets tired: it will try many small fixes until the proof works, sparing the user the frustration of manual trial-and-error. It also has the discipline to follow project rules thanks to the context. As evidence of its capabilities, users report Amp *"can do non-trivial changes across a codebase – e.g. modify a model and update all references – and compile/test, making adjustments until done"[27]*. This level of autonomy is particularly valuable for OperatorKernelO6, where changes in one theorem (say a new lemma in ordinal toolkit) might require updates in multiple files – Amp can handle that cascade. The main caveats to note: Amp currently might not allow choosing which underlying model to use (it manages that internally) and it can consume a lot of tokens (thus API usage) for big tasks*[36]*. However, since you have enterprise subscriptions, and given Amp's

free trial credits, this is manageable. Always review Amp's final output; treat it like a junior engineer's work – usually correct, but occasionally needing oversight[37]. With that in mind, Amp offers the best balance of **power and rule-adherence** for this project.

Rank #2 – Continue.dev: Customizable Open-Source Agent

As a second option, **Continue** (by Continue.dev) provides an *open-source* framework to achieve many of the same goals as Amp, with greater flexibility and control[38]. We rank it #2 because it may require a bit more configuration to reach the seamless experience Amp gives out-of-the-box. However, for a user willing to tinker (or as a backup if Amp's closed platform is limiting), Continue can be extremely powerful:

Setup & Usage: Install the **Continue VS Code extension** (there are versions for VS Code and JetBrains; use VS Code here). After installing, open the Continue panel. You'll need to provide API keys for the models you want to use (e.g., your OpenAI key, Anthropic key). This is typically done in a Continue config file or through the VS Code settings for the extension. For example, you might set GPT-4 as the default model for chat completions. If Continue supports OpenAI's function calling, you could even integrate a function that calls *lean --make* on demand to get error output (though an easier way is just instructing the agent to read the Lean output from the editor).

Start in chat mode with a simple test, like asking, "What is the Trace inductive type defined in this project?" to ensure it has loaded the context. If it hasn't, you may need to "prime" it by opening relevant files or copy-pasting key content (some versions of Continue may not automatically read the entire repo, in which case provide *AGENT.md* content manually as a system message or user message at start).

One powerful pattern is to use Continue's **Background Agent** feature for automation. For example, you can create a background agent that watches for file saves. Pseudo-configuration: "On Lean file save -> run *lake build* for that file -> if build fails, parse the error -> invoke the fix agent with error message." This way, the moment you save a file with an incomplete proof, the agent might catch the error and offer a fix. Since setting this up might be non-trivial for a newcomer, you can initially use Continue more interactively: manually copy any Lean error into the chat and ask for a solution. Over time, as you become comfortable, you can script these triggers.

Rationale: We rank Continue #2 because it *can* achieve everything Amp does (and more in terms of customization), but it's not as plug-and-play. The user's limited background means Amp's simplicity is a big plus; however, Continue's openness is a huge advantage for long-term maintainability. Continue shines in letting you incorporate new models (e.g. if **Google Gemini** proves to be strong at coding, you could configure Continue to use Vertex AI's Gemini model via API when it becomes available – given Google's roadmap, *Gemini 2.5 Pro* should be coming to Vertex AI around mid-2025[48]). Continue would let you leverage those GCP credits directly for model inference, whereas Amp uses its own service (Amp's costs would be separate from your GCP credits).

Another scenario: if you want an AI agent to run **continuous integration checks** on your repository (for example, a nightly run that attempts to find any pattern that violates the project rules or tries random rewrites as fuzz testing), Continue's CLI could be run in a headless mode on a GCP VM to do that. You could schedule a

job that uses an AI to write new random *Trace* terms and verify normalization (just as an idea to use your credits for quality assurance). This level of customization is unique to Continue.

In summary, Continue is the best choice for a **bespoke AI assistant** tailored to your project's evolving needs. Initially, you might use it just for chat/edit help if Amp is handling most heavy lifting. But it's reassuring to have a fallback if Amp's closed ecosystem ever falls short or if you want to directly use GPT-4's reasoning on some problem (you can prompt GPT-4 via Continue with the exact context you want). With Continue configured properly, you have a second pair of AI eyes – and given formal proof is a complex domain, having diversity (Claude via Amp *and* GPT-4 via Continue) could catch mistakes one model might miss.

Rank #3 – **GitHub Copilot & VS Code Extensions**: Inline Assistance

While the autonomous agents take center stage, we also recommend **GitHub Copilot** as a supplementary tool for day-to-day coding. Copilot is ranked #3 because it doesn't understand your custom project rules, but it excels at providing quick suggestions and completing patterns. In Lean 4 development, Copilot can be useful for:

To set up Copilot, install the **GitHub Copilot** extension in VS Code. You'll need a GitHub account with Copilot enabled. (It sounds like you have this subscription already.) Once logged in, Copilot will start suggesting in any code file. You might also consider enabling **Copilot Chat** (GitHub's chat extension, currently in beta for enterprise users). Copilot Chat provides an in-IDE chat where you can ask questions similar to ChatGPT, but it has read access to your code open in the editor. This can be handy to query, for example: "Copilot, what does this Lean error mean and how might I fix it?". It's not as knowledgeable as GPT-4, but it sometimes gives useful pointers. Given you have GPT-4 via other means, Copilot Chat isn't essential, but it's integrated nicely with VS Code's UI if you want a quick question answered without swapping windows.

Usage Tips: Keep in mind Copilot's suggestions are drawn from its training data (which might not include the latest state of your project). It might occasionally suggest an approach that uses an illegal tactic or a different style. **Always verify Copilot's completion against the project rules.** Lean's real-time feedback will immediately tell you if a suggestion is wrong (red underline errors). For example, if Copilot suggests using *Nat* or numeric literals in the kernel (forbidden), Lean will error or your own knowledge should flag it. Treat Copilot as a fast typist who knows general Lean, but *not* a project-specific expert. It's great for small-scale tasks: finishing a *by simp* proof, or filling arguments to a lemma once you've recalled its name.

Rationale: We include Copilot because it *dramatically improves ergonomics*. Formal proof writing can be tedious; Copilot's ability to complete an induction template or suggest the next tactic saves time and keystrokes. It's ranked below the specialized agents because it doesn't actively enforce rules or do multi-step reasoning. However, it's complementary: Amp/Continue might handle the heavy, complex proofs, while you use Copilot for quick lemmas or definitions. Many Lean developers successfully use Copilot to help with library code, etc., so it's a proven productivity booster. Given that you have the subscription, it's worth leveraging.

One caution: Copilot can sometimes produce **subtly incorrect proofs** that "look" plausible but are not actually correct (this is a form of hallucination). Lean will reject those, but it might not be obvious why a suggestion fails.

If you encounter this, you might need to fall back to an AI chat to get a proper explanation. This is where Copilot Chat or ChatGPT could help by explaining the error log or why a certain approach doesn't work.

Rank #4 – **ChatGPT (GPT-4) and Anthropic Claude (100k)**: Out-of-IDE Brainstorming

For conceptual understanding, research, or when you hit a wall, using **ChatGPT** or **Claude** outside the IDE can be very valuable. We rank these at #4 for direct coding purposes, but they play an important supporting role:

Integration with the Workflow: Using these external QA assistants is more manual. However, it's good to know you have them when Amp or Continue says "CONSTRAINT BLOCKER: needed lemma X"*[32]*. If Amp can't find a lemma, you have a couple of choices: - Search in mathlib documentation (there is **Loogle**, a Lean lemma search engine*[49]* – you can go to loogle.leanprover.community and type a query for a lemma statement or name). - Ask ChatGPT or Claude if they know of such a lemma (sometimes GPT-4 might "remember" a mathlib fact by name if it was famous, but be cautious: verify via mathlib docs). - Use VS Code's global search in Mathlib (if Mathlib source is cached locally, searching for keywords like "omega0" could find relevant lemmas).

Once you identify a candidate lemma, you can feed it to the AI agent. For example, Amp said we need *Ordinal.opow_le_opow_right (a := omega0)* and suggests importing it*[32]* – if that wasn't in the import list, you'd confirm via mathlib docs that it exists and then allow Amp to add the import*[50][51]*.

Rationale: While not the primary development tools, GPT-4 and Claude are your safety net for understanding and research. They help ensure **no single point of failure**: if Amp/Continue are confused, you can consult a second opinion in ChatGPT, which might phrase things differently or catch a subtlety the agent missed. Additionally, having a conversational partner to **explain proofs in plain English** can deepen your understanding (e.g., you could copy a completed Lean proof and ask ChatGPT to walk you through it step by step). This is important given your limited background – it turns the formal gibberish into something intelligible, which helps you supervise the AI's work meaningfully.

Claude's large context can also allow you to do a **full-project review**: you can paste significant chunks of code or the entire list of theorems and ask if anything stands out as potentially problematic. It might not be 100% accurate, but it could, for example, notice if one rewrite rule's μ-decrease proof is missing a case or if a theorem is stated but never proved. This kind of overview is hard to get from Lean alone without manually tracking to-dos.

Additional Tools and Automation

Beyond the main AI assistants, here are some additional toolkit items and best practices to round out the system:

- **Gemini and Future Models:** As Google's **Gemini** model becomes available (e.g., via Vertex AI with *gemini-2.5-pro*[53]), you could experiment to see if it offers any improvements in formal reasoning or not. You could do this by configuring Continue to call Vertex AI endpoints with your GCP credits. That said, formal math is a niche skill for LLMs; GPT-4 and Claude have some known training on it (OpenAI trained GPT-f on Lean, etc.), whereas Gemini's prowess in this area is untested in public as of 2025. Use it as a complementary trial – for example, run the same prompt ("prove X theorem") with GPT-4 and with Gemini, and see which output is more Lean-correct. Continue's model flexibility[38][39] would make this A/B testing possible.
- **Team Collaboration via AI:** Since you have ChatGPT Teams, if multiple people are involved, you can share ChatGPT conversation links. For instance, if you use ChatGPT to develop a plan for Gödel's theorem encoding, you can share that with colleagues or even with the AI agents (you can paste the plan into Amp as a starting point). Also consider setting up a **shared knowledge base**: e.g. OneNote or a markdown file where you jot down tricky lemma names or pitfalls discovered, which you then feed to the AI so it doesn't repeat mistakes. Amp's memory per thread is ephemeral, but you can always remind it: "Recall the guidelines from AGENT.md about not assuming $\mu(s) \le \mu(\delta\, n)$[54] – apply that here." This ensures continuity of context even if the underlying AI doesn't perfectly remember state between requests.
- **Monitoring and Logging:** Both Amp and Continue likely keep logs of their interactions (Amp stores threads on their server[55]; Continue might log locally). It's wise to keep a log of the AI's suggestions and your code changes. This not only provides an audit trail (important for trust – you can trace why a proof was written a certain way), but it can help debug issues. If the AI made an incorrect change, you can review the thread to see the reasoning. Amp shares threads by default among your team (if you have multiple users, they can see what was done and reuse successful prompts)[56].

Workflow: Putting It All Together

To illustrate a typical session with this AI-augmented setup, consider the task of proving **strong normalization** for the *OperatorKernelO6* rules:

1. **Theorem structuring:** You write a theorem statement in Lean, e.g. *theorem mu_decreases_R_merge_void_left ... : μ t < μ (merge void t)*. Immediately, VS Code underlines it as incomplete. You invoke Amp: *"Plan the proof of mu_decreases_R_merge_void_left."* Amp returns a **PLAN** (as per the format in AGENT.md[57]) outlining steps: *1. Start with positivity of ω. 2. Lift through exponent using opow_le_opow_right. 3. Split products with opow_add. 4. Apply Ordinal.mul_le_mul_iff_left… etc.* This matches the μ-measure playbook[15]. Satisfied, you say "Go ahead and implement." Amp switches to **CODE** mode and begins editing the file, inserting the proof.
2. **AI proof search & compile:** Amp uses the known lemmas from the toolkit. Suppose it initially tries a lemma not imported (e.g. it attempts *Ordinal.mul_lt_mul_of_pos_right*, which is on the "red" list). Lean errors out ("unknown identifier or violates monotonicity rule"). Amp catches this from the compile output and, referencing *AGENT.md*/toolkit, realizes that's disallowed. It backtracks and uses the correct approach (perhaps adding a padding term and using *le_add_left* as in the toolkit example). This time Lean accepts the proof. Amp marks the task done.

3. **User review:** You quickly scan the proof. Thanks to consistent enforcement, you see it used *Ordinal.mul_lt_mul_of_pos_left* and not the forbidden right-version, and it added a helpful comment or at least followed the structure you expected (it might have even added the *@[simp]* attribute or other formatting as per project style). You run *#print axioms mu_decreases_R_merge_void_left* in the Lean REPL (or ask Amp to do it) – it returns **none**, meaning the proof introduced no axioms (good!).
4. **Edge-case query:** You recall the pitfall noted in the docs: not to assume *μ s ≤ μ (δ n)* for recursor. You highlight the relevant part of the *mu_lt_rec_succ* proof that Amp provided and ask in Continue's chat: *"Does this proof anywhere implicitly assume μ s ≤ μ(δ n)?"*. Continue/GPT-4 analyzes the proof steps and confirms that the proof treats *μ s* and *μ(δ n)* independently (perhaps it sees they never tried to compare them directly, which is correct*[54]*). This double-check gives you confidence no forbidden assumption slipped in.
5. **Confluence proof planning:** For confluence, you need a different strategy (normalize-join). You ask ChatGPT (outside VS Code) to explain the *normalize-join* method. It describes defining a normalize function that reduces a trace to normal form (using strong normalization) and then showing any two reduction paths lead to a common successor. It might even cite a known result. Armed with that, you instruct Amp: *"Define a normalize function using well-founded recursion on μ, and prove it yields a normal form (hint: use well-founded induction on μ)."* Amp creates a new file *Normalize.lean*, writes the definition with *WellFounded.fix* or *WellFounded.recursion* (importing *Init.WF* which was allowed*[58]*) and proves termination by referencing the proven well-foundedness of < on μ (from SN). If Amp hits a wall (maybe well-founded recursion in Lean can be tricky), it might output a *CONSTRAINT BLOCKER* asking for a hint or a specific lemma (like *InvImage.wf* usage). You then either provide the hint yourself or ask Continue's chat, which might quickly recall the pattern to use *measure_wf* or *InvImage.wf* on μ. You feed that to Amp, and it completes the definition.
6. **Testing and iteration:** After major developments, you run *lake build* and *lake test* (fuzz tests). Suppose a fuzz test finds a specific trace that doesn't normalize properly – indicating maybe a missing rewrite rule or a bug. You take that counterexample, put it into a ChatGPT prompt with the context of the semantics, and ask what could be wrong. GPT-4 might suggest that the *merge* cancellation isn't being applied because the trace structure is such and such – leading you to realize you missed a lemma about merging nested traces. You then go back to Amp or Continue, formulate that new lemma, prove it similarly, and rerun tests.
7. **Documentation and final checks:** As you approach completion, you use Amp one more time to **audit the project**: "List any uses of forbidden lemmas or suspicious tactics in the entire project." Amp (with full context) scans and reports nothing violating the rules (since we maintained discipline). It might flag one thing: *"I see Mathlib.Algebra.Order.Monoid.Defs is imported but none of its content is used – consider removing it to avoid confusion"*. This is a good catch; you remove that import to ensure no one accidentally uses *mul_le_mul_left* generic in the future. Finally, you commit the changes. Your CI (or a final Amp command) runs *#print axioms* on all new theorems and confirms **axiom-free** proofs*[59]* as required.

Throughout this process, the AI agents handled the heavy formalization labor, but **you remained in control** – guiding the AI with high-level instructions, verifying critical adherence to rules, and providing mathematical insight when needed. The result is a Lean 4 development workflow that is *dramatically more efficient* than manual coding, yet remains *trustworthy and aligned* with the project's foundational philosophy.

Conclusion and Rationale for the Recommended Setup

In summary, our optimal AI-augmented Lean 4 proof system consists of:

- **Lean 4 + VS Code** for the interactive proving environment[21].
- **Sourcegraph Amp** (Rank #1) as the primary autonomous coding agent, leveraging its ability to incorporate the entire project context (especially the *AGENT.md* rules) and perform iterative compile-fix cycles[27][28]. Amp serves as the tireless executor of proofs and enforcer of project conventions.
- **Continue.dev** (Rank #2) as a secondary, highly configurable AI assistant, giving access to GPT-4, Claude, and other models with custom rule integration[38][41]. Continue provides flexibility for special tasks and future-proofing (e.g. integrating new models or custom automation like file-watching agents).
- **GitHub Copilot** (Rank #3) for inline code completions and minor suggestions to streamline writing, acknowledging that it speeds up routine tasks but must be overseen for compliance.
- **ChatGPT (GPT-4) and Anthropic Claude** (Rank #4) outside the IDE for brainstorming, conceptual Q&A, and large-context analysis, ensuring that the human-in-the-loop can always query the AI for understanding or cross-check the agents' outputs.

Why this combination? It covers all bases: Amp and Continue handle **generation and modification of proofs**, reducing the barrier of formal syntax; Copilot handles **micro-level convenience**; and ChatGPT/Claude handle **macro-level insight and verification**. Importantly, this setup emphasizes **no-hallucination and math-checking** at every step. Lean itself is the final arbiter (any hallucinated lemma simply fails to typecheck). Amp's design explicitly forbids hallucinating imports/lemmas[5], and with the provided rules, it will prefer a *constraint blocker* over making something up, which is exactly the behavior we want. Continue allows us to codify additional checks (like automatically scanning for blacklisted tactics). Thus the AI not only *generates* solutions but also *polices* itself to some extent using the knowledge base we provide.

By following this operational plan and utilizing the toolkit recommended, the user – even with limited background – should be able to develop the OperatorKernelO6 proofs end-to-end. The AI will serve as a knowledgeable guide that suggests correct approaches (e.g. ordinal induction steps), fixes low-level issues (missing parentheses, incorrect names), and relentlessly respects the project's unique framework. This leaves the user to focus on the creative and conceptual aspects (like how to encode provability in the Trace system or how to diagonalize for Gödel) while the AI handles the tedious formalization details within the constraints.

Rationale Recap (Ranked Options):

1. **Amp** – best for automated heavy-lifting and built-in rule enforcement[27][28]. Use it for major proofs and multi-file refactorings.
2. **Continue** – best for customization, using specific models, and setting up advanced workflows (event-driven agents, custom checks)[41][39]. Use it for fine control and when you need to inject specific knowledge or try different AI models.
3. **Copilot** – best for on-the-fly code completion that accelerates writing. Use it for small lemmas, definitions, and to avoid typos. Always verify its suggestions via Lean.
4. **ChatGPT/Claude** – best for understanding, verifying, and researching. Use them to clarify doubts, get high-level proof sketches, and ensure you (and the AI) haven't missed anything logically.

Rigorously applying this toolkit, you will achieve an AI-augmented workflow that is **efficient**, **comprehensive**, and – most importantly – **compliant** with the strict procedural and foundational requirements of the OperatorKernelO6 project. The final deliverable will be a Lean 4 formalization of SN, confluence, arithmetic, and Gödel's theorems that is *fully machine-checked*, with the AI having been an invaluable assistant every step of the way (but never a hallucinating gremlin!). All of this comes with a detailed setup and usage manual (as above), enabling even a relative newcomer to harness state-of-the-art AI in the service of deep formal mathematics.

*[1][2][3][4][5][7][8][9][10][11][12][17][18][19][20][23][32][50][51][52][54][57][58][59]*agent.md

file://file-3em2TmjHZhiNEHzp959mpm

*[6][13][14][15][16][24]*ordinal-toolkit.md

file://file-2FmV8hSr4hvU6BYEE3a6BP

[21][22][25][26][49] Install — Lean Lang

https://lean-lang.org/install/

[27][28][30][31][33][34][35][36][37][55] Tried Amp, Sourcegraph's new AI coding agent — here's how it stacks up vs Cursor : r/cursor

https://www.reddit.com/r/cursor/comments/1kpin6e/tried_amp_sourcegraphs_new_ai_coding_agent_heres/

[29][56] Amp - an AI coding agent built by Sourcegraph

https://sourcegraph.com/amp

[38][39][40][41][42][43][44][45][46][47] Continue - Ship faster with Continuous AI

https://www.continue.dev/

[48] Gemini 2.5: Our most intelligent AI model - Google Blog

https://blog.google/technology/google-deepmind/gemini-model-thinking-updates-march-2025/

[53] Model versions and lifecycle | Generative AI on Vertex AI

https://cloud.google.com/vertex-ai/generative-ai/docs/learn/model-versions