ChatGPT

Project Plan: simple_dev – AI-Orchestrated Eiffel Development

Background: The Simple Eiffel Ecosystem and AI-Assisted Workflow

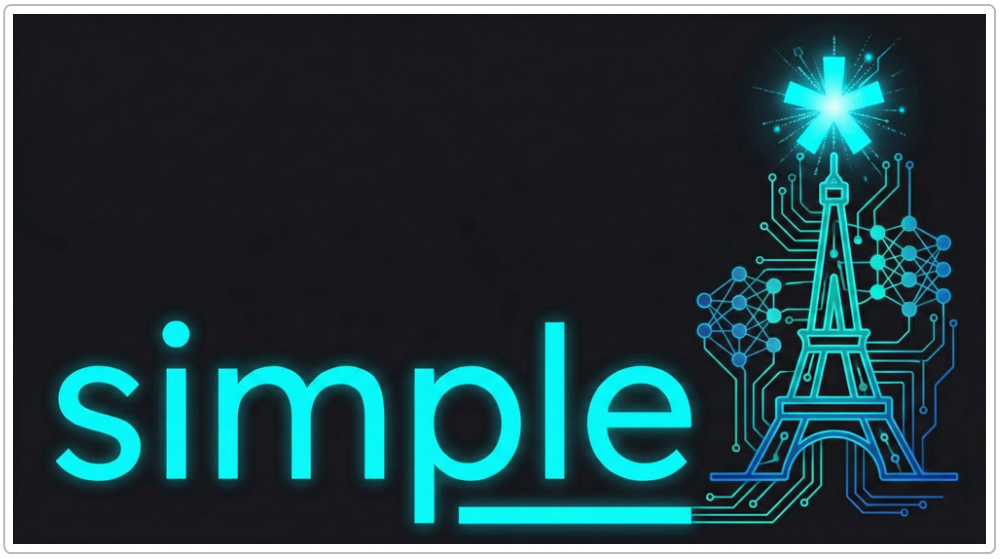

Figure: The Simple Eiffel initiative is building a suite of lightweight Eiffel libraries (each named `simple_*`) to modernize the Eiffel ecosystem. These libraries, developed rapidly with AI assistance, aim to equip Eiffel with features on par with modern tech stacks, from web development to cryptography.

Over the past months, the Eiffel community (spearheaded by the Simple Eiffel project) has launched a **"Christmas 2025 Challenge"** – creating a suite of Eiffel libraries to make Eiffel competitive with today's popular tech stacks [1]. This effort has produced libraries like **simple_json**, **simple_sql**, **simple_web**, **simple_htmx**, **simple_alpine**, **simple_uuid**, **simple_hash**, **simple_jwt**, and more – all built in a matter of weeks using an AI-assisted development approach [2]. Notably, an entire Eiffel-based web showcase site was recently developed in a single session by leveraging an AI (Anthropic's Claude) alongside Eiffel's design-by-contract checks [3]. In that workflow, the human provided direction ("build a hero section"), the AI generated Eiffel code, and then the Eiffel compiler and contract system immediately caught errors or contract violations, which were fixed in short iterative cycles [3]. This **experiment in Eiffel+AI** showed both the promise and the pitfalls of current AI coding assistants. It dramatically accelerated development, but also highlighted challenges that need to be addressed before AI-generated Eiffel code can be truly *"real-world"* in robustness and maintainability.

Current Limitations of LLM-Assisted Coding (Lay of the Land)

Despite the successes, using large language models (LLMs) like Claude or ChatGPT for coding still exhibits several **serious limitations**. Through experience, we've identified a few key pain points when using AI in the development of Eiffel libraries and applications:

- **Lack of Long-Term Memory:** Modern LLMs do **not** retain conversation context beyond a single session or beyond their fixed context window. They essentially have no true long-term memory [4]. This means if you close a session or the conversation grows too long, the AI may "forget" important details discussed earlier. Every new session starts fresh, requiring re-feeding of information. Even within one session, if the conversation is lengthy, older details can be dropped once the context window is exceeded [5] [4]. We've observed the AI often fails to recall decisions or information given just a day before. Any **"compaction"** or summarization of the conversation to save tokens can further cause loss of critical detail. In short, without explicit memory aids, the AI cannot reliably remember past instructions, leading to repetition and inconsistency.

- **Eager to Code, Reluctant to Design:** LLMs tend to jump straight into coding solutions without sufficient upfront design discussion. If given a feature request, the AI often outputs code immediately, treating the prompt as a spec – even when a higher-level architectural dialog is needed. This **lack of patience for planning** can result in suboptimal designs. In practice, we had to constantly force the AI to slow down and engage in design brainstorming. This aligns with observations by others: a common mistake is "diving straight into code generation with a vague prompt" instead of first *defining the problem and planning a solution* [6]. Best practices for AI-assisted engineering now emphasize writing a detailed specification with the AI and outlining the design **before** writing any code [6]. Our current AI partner, however, doesn't naturally do this – it needs to be guided to focus on design. Without that guidance, it may produce code that technically works but doesn't necessarily follow human-preferred design choices or misses edge cases that a design discussion would catch.

- **Myopic Implementation Choices:** Today's AI often makes implementation decisions **myopically**, without considering downstream effects or the broader project context. It will modify a module or class to satisfy the immediate request, but may not realize that this change breaks other components that depend on it. For example, we saw it update classes in the `simple_sql` library to satisfy a new requirement, but **failed to account for other libraries depending on** `simple_sql`, leading to regression in those libraries. This happens because an LLM works only on the snippet of context it's given (a "snapshot of input") and lacks a holistic view of the entire codebase or dependency graph [7]. As noted in an analysis of LLM coding pitfalls, *"LLMs operate on a snapshot of input... Without deep integration into the full codebase, they lack the context needed for consistent architecture. They often don't reason about scalability, modularity, or downstream impacts like a seasoned engineer would"* [7]. In practice, this means the AI might introduce tight coupling, break APIs, or choose non-idiomatic designs that increase technical debt [8]. It does not naturally perform impact analysis of a change. The burden falls on the human to detect these issues (via tests or code review) and then prompt the AI to fix inconsistencies across the project.

- **Lack of Persistent Specifications and Style Recall:** Another issue is that the AI does not inherently *remember* the project's established conventions, documentation style, or specifications from prior work – unless we remind it every time. For instance, each time we start a new `simple_*` library, we have to reiterate our standards (e.g. *"create a README.md and docs folder following the style of earlier simple_* libraries"*). The AI doesn't on its own recall these past instructions or infer the style guide. This is a side effect of the first limitation (no long-term memory) and the fact that the model only knows what's in the prompt. Expert LLM users have found that you must explicitly provide context and guidance for each new task – e.g. by supplying examples, project conventions, and requirements in the prompt [9] [10]. Tools like Claude's "Projects" mode or VS Code plugins attempt to include relevant files automatically, but the bottom line is:* LLMs are only as good as the context you provide – they will not reliably recall prior project knowledge on their own* [9]. If we forget to restate a crucial requirement (say, "include design-by-contract assertions" or "use the naming conventions we used before"), the AI's output may omit or violate it. This leads to the human getting "tired of having to do that" repeatedly. Maintaining consistency in style and specs across sessions is tedious and error-prone under the current ad-hoc prompting approach.

In summary, while our AI coding companion (Claude) has **enormous strengths** – speed, knowledge, ability to generate working Eiffel code – it also has clear **weaknesses** in context retention, design foresight, holistic reasoning, and consistency. These weaknesses are not due to "faults" in the AI per se, but inherent to how LLMs function and the limitations of prompt-based development. As one experienced engineer noted, using LLMs for programming is *"not a push-button magic experience – it's difficult and unintuitive, and getting great results requires learning new patterns"* [11]. We now realize that to harness the AI's strengths for serious software development, we need a more structured approach that compensates for these weaknesses. This is where **our vision for `simple_dev` comes in.**

Vision: A Guided AI Development Orchestrator (simple_dev)

simple_dev is envisioned as a new `simple_*` tool/library that will act as an **AI development orchestrator** for Eiffel projects. It's not meant to replace the AI (we still leverage LLMs like Claude, ChatGPT, Gemini, or local models), but to *use* the AI in a far more controlled, efficient, and context-aware manner. In essence, simple_dev will wrap around the AI, providing the "glue" of project knowledge, best practices, and workflow management so that the AI's contributions are consistently correct and aligned with our goals. The driving idea is to **put guardrails and a guiding framework around the prompt-response cycle**, making AI a truly effective pair-programmer integrated into Eiffel's software engineering methodology.

Our Goals for simple_dev:

- **Persistent Knowledge & Context:** *Never lose important information given to or produced by the AI.* simple_dev will maintain project knowledge in a structured form (likely via SQLite databases and files) that can be reloaded into context as needed. This includes requirements, design decisions, specifications, coding guidelines, and even past conversation summaries. By having an internal "memory" of the project, the orchestrator can re-inject relevant context into prompts, mitigating the AI's forgetting problem. For example, if a new session starts, simple_dev can automatically feed in the high-level spec and style guidelines for the project from its database. If the conversation gets too long, it can summarize and store older parts, and recall them on demand when relevant. This **long-term memory layer** ensures continuity across sessions and days, so the AI doesn't act like a blank slate every morning.

- **Adaptive Conversation Flow Across the Dev Lifecycle:** simple_dev will manage the interaction with the AI in **phases** that mirror a proper software development lifecycle. Rather than every prompt being an isolated Q&A, the tool will guide the AI through stages such as:

- **Ideation & Requirements Gathering:** In this stage, the developer can brainstorm with the AI. The orchestrator will prompt the AI to ask clarifying questions about the project idea and ensure all requirements and constraints are fleshed out. The output is a clear problem definition. This addresses the AI's tendency to skip straight to coding – instead, simple_dev *forces a preliminary dialogue* about *what* to build before *how* to build it. (As an analogy, experienced AI users now "begin by defining the problem and planning a solution" [6], and we'll encode that behavior.)
- **Design & Specification:** Next, simple_dev will lead the AI in producing a detailed **specification document** (e.g. a `spec.md` or design description). The AI will be prompted to outline the architecture, key classes/modules, their responsibilities, interfaces, and even a testing strategy, based on the requirements. The orchestrator can ensure this spec includes important Eiffel-specific considerations (Design by Contract expectations, inheritance structures, etc.). We'll likely prompt the AI to **iteratively refine the design**, possibly asking "Does this design handle all requirements and edge cases?" – effectively simulating a design review. This phase results in a comprehensive plan that both human and AI can reference. In the words of one engineer, it's like doing *"a waterfall in 15 minutes"* – a rapid upfront planning that makes subsequent coding much smoother [12]. By the end, *"we compile this into a comprehensive spec.md – containing requirements, architecture decisions, data models, and even a testing strategy. This spec forms the foundation for development."* [6]. In simple_dev, this spec will be saved (and possibly versioned) in the project's database and files, to be used throughout the development.
- **Task Planning:** With a high-level design in place, simple_dev will break down the implementation into **logical tasks or milestones**. Essentially, it will create a to-do list or ticket list for the AI (and human) to tackle one by one [12]. The AI can assist in this breakdown as well – e.g. generating a list of steps ("Implement class X", "Then add feature Y", "Write test for Z", etc.). The orchestrator ensures the plan is coherent and complete before proceeding. Each task can be relatively small and self-contained.
- **Incremental Implementation (Coding Stage):** Here's where the AI actually writes Eiffel code, but under strict guidance. simple_dev will take one task at a time (e.g. "Implement the `SIMPLE_XYZ` class with these features..."), gather the necessary context (relevant part of spec, relevant existing code, coding conventions), and prompt the AI to generate the code for that task. Crucially, *we will limit the scope of each prompt* to a focused task – aligning with the idea that *"LLMs do best when given focused prompts: implement one function, fix one bug, add one feature at a time"* [13]. By **iterating in small chunks**, we reduce the chance of the model going off-track or producing a large incoherent blob of code [14]. After each chunk, simple_dev can insert a validation step (compiling the code, running tests for that unit).
- **Testing & Verification:** After or alongside implementation, simple_dev will ensure the new code is tested. Eiffel's design by contract will automatically provide some runtime verification (e.g. postconditions checking outcomes), but we will also integrate any unit tests or use Eiffel's testing frameworks (like EUnit or manual tests). In fact, as part of the spec phase, we plan to have a testing strategy – possibly even auto-generating some basic tests or at least identifying test cases. simple_dev can prompt the AI to generate unit tests for a module after implementing it, or use existing tests. Then it will compile and run these tests. Any failures or contract violations will be caught in this phase.

- **Debugging & Refinement:** If tests or the compiler reveal issues, simple_dev will capture those error messages or failing assertions and feed them back to the AI for debugging. This creates a **tight feedback loop** where the AI can suggest a fix, apply it, and re-run the test – repeatedly until the issue is resolved. Such rapid iteration is something AI excels at, as long as we provide the feedback. For instance, if a contract fails or an Eiffel compilation error arises, simple_dev will format that information (perhaps as JSON or plain text) and include it in the next prompt like: *"The test for X failed with this output... or The compiler error is Y... please fix."* This mirrors workflows reported elsewhere, where an AI agent is given failing test results and can quickly suggest corrections [15] [16]. Indeed, in our earlier manual process we often did this by copying compile errors into Claude; simple_dev will automate it. The goal is that *the AI "won't consider the task done until all tests pass,"* effectively using the tests/contracts as a quality gate [17].
- **Integration & Regression Checks:** Because Eiffel projects often consist of multiple libraries (as in the Simple Eiffel ecosystem), a change in one library may affect others. simple_dev will be aware of project dependencies. After making changes to (or adding) a library, it can run a **regression build** of the whole suite (using something like `simple_ci`). The idea is to catch any downstream compilation errors or test failures in dependent projects early. Thanks to the existing **Simple CI** tool, we can compile multiple Eiffel projects in one go and produce a machine-readable report [18] [19]. simple_dev could invoke such a CI run and then parse the results. If, say, modifying `simple_sql` caused `simple_web` to no longer compile, the orchestrator will catch that and prompt the AI with those specific errors (rather than discovering it much later manually). This ensures the AI considers *system-wide consequences*, not just the local code. It effectively forces the AI to "think with a holistic mindset" by confronting it with any integration issues it caused.
- **Documentation & Artifacts:** At appropriate points, simple_dev will also prompt for non-code outputs that are part of best practices – for example, updating the `README.md`, generating API documentation comments, or preparing deployment scripts. These often get neglected by AI unless asked. The orchestrator can enforce that when a new library or major feature is finalized, the AI generates the README and usage examples (using the templates and style we've established in the ecosystem). It can also ensure things like version numbers are updated, changelogs written, etc., based on the patterns from earlier libraries. By automating these prompts, we won't have to manually remind the AI to do them each time.
- **Release & Delivery:** Finally, simple_dev can assist in packaging the project – for Eiffel, this could include generating final C compilation (finalized build), producing binaries (e.g. DLLs or executables), and even creating installer scripts if needed (the user mentioned possibly Inno Setup installers or similar for deliverables). While these steps are more tooling-oriented, the AI could help write an Inno Setup script or an installer README if provided with context. The orchestrator ensures the AI has all necessary info (file paths, version numbers, etc.) to do this correctly.

In essence, simple_dev will **scale the AI's involvement across the entire dev lifecycle** – from initial idea discussions to final deployment – while maintaining appropriate structure at each level. It will know when to switch the AI's role from *brainstorming partner* to *code writer* to *debugger* to *documenter*, etc., following a logical progression. This should feel like a more **continuous interaction model** where the AI is always engaged but in different modes depending on the stage.

Equally important, simple_dev's interaction model scales in **granularity**: the AI will handle everything from single-line code fixes up to high-level architectural suggestions, but always within a scoped context. The tool will smartly zoom in or out: it can ask the AI to focus on a tiny detail (like renaming a variable or fixing a single contract) or to consider the big picture (like reviewing whether the overall design meets the spec), by feeding it the appropriate amount of context for each. Managing this scope is key to using tokens efficiently and keeping the AI on track.

Key Components and Features of *simple_dev*

To realize the above vision, `simple_dev` will incorporate several components and techniques. Below we outline the major features and how they address the earlier limitations:

- **Unified AI Client with Multi-Provider Support:** We will leverage the existing `simple_ai_client` library as the backbone for connecting to various AI models. This library already supports multiple AI providers – for example, local models via **Ollama**, and cloud models like **Anthropic Claude** and **OpenAI GPT** [20] . simple_dev can use this to flexibly switch or use whichever AI the user configures (e.g. use Claude for code generation, or a local Qwen-7B for smaller tasks to save costs). The user will provide API keys or local endpoints, and simple_dev will route requests accordingly through simple_ai_client's unified interface. This also means we can use different models for different subtasks: perhaps a big model for complex code generation and a smaller one for running quick embedding queries or summarization. abstracting the model specifics away.

- **Persistent Project Database (Knowledge Base):** A core piece will be an SQLite-backed knowledge store where **specifications, design decisions, context embeddings, and past conversation summaries** are recorded. This addresses the memory problem by giving the orchestrator its own memory. For example, after the spec is created, it's saved in the DB (and possibly as a file `spec.md`). If the conversation is resumed later, simple_dev will fetch the spec and remind the AI of it. We can store structured data too – perhaps a table of requirements, a table of key architecture decisions, etc., which can be selectively included in prompts. The DB can also track what has been done (task status) so that if a session is interrupted, we know where to resume. Essentially, the project DB becomes the source of truth that persists beyond the ephemeral AI chat context.

- **Vector Embedding Knowledge and Semantic Search:** Building on `simple_ai_client`'s embedding capabilities, simple_dev can implement a semantic memory of things like error messages and code snippets. In fact, `simple_ai_client` already provides an `AI_EMBEDDING_STORE` that uses SQLite to store error texts and their resolutions as vectors, enabling similarity search [21] [22] . We will use this feature (and extend it) to allow the system to recall solutions to known problems. For example, if the Eiffel compiler throws a certain error (say a common one about a missing creation procedure or a type mismatch), and we solved it before, simple_dev can retrieve that from the store and either automatically apply the fix or prompt the AI with *"This looks similar to a past error; the fix then was XYZ"*. This reduces repeated work and helps the AI not re-invent fixes. Additionally, we might embed other text, such as portions of the codebase or documentation, to enable semantic lookup ("find relevant context for this prompt"). Before asking the AI a question, simple_dev could vector-search the spec or code to pull in only the most relevant snippets, rather than always sending everything. This technique keeps prompts concise (saving tokens) while still giving the AI what it needs.

- **Structured Prompt Templates and Guidance:** Rather than freeform chat, simple_dev will use *prompt templates* for different situations – ensuring the AI always gets clear instructions and necessary context. For instance, a template for implementing a new class might include: a brief of the class purpose from the spec, a reminder of coding conventions (like *"follow Eiffel naming conventions and include contracts"*), and the exact task description. A template for bugfix might include the error message and relevant code snippet, then ask for a corrected code diff. These structured prompts act as guardrails, preventing the AI from going off-topic or ignoring important factors. We will encode Eiffel **best practices** into the prompts. For example, at the top of prompts we might insert a system message like *"You are an Eiffel expert assistant. Always enforce Design by Contract and produce code consistent with Eiffel style. If designing a new class, ensure it has a creation procedure* `make`*, a full class-level comment, and relevant examples."* By doing this systematically, we relieve the human from repeatedly typing the same guidance. Moreover, as we accumulate more project knowledge, these prompt templates can evolve (perhaps even dynamically incorporating lessons learned from earlier tasks).

- **Enforcing Eiffel Design by Contract & Standards:** Because Eiffel's philosophy is centered on correctness and clear contracts, simple_dev will bake those standards into every step. That means new code generated should have appropriate **preconditions, postconditions, class invariants** where applicable. If the AI omits them, the orchestrator can prompt it to add contracts (we can detect if a routine has no postcondition and ask for one). Eiffel is also big on documentation; each class should have a header note, each feature a brief description. simple_dev can ensure the AI includes these by default. This not only yields better code but also uses the contracts as additional verification (if the AI writes them, it clarifies the intended behavior, and then we can catch if they're violated). Essentially, we want the AI to act in the spirit of Eiffel methodology – something it won't do unless explicitly told. By maintaining a store of Eiffel best practices (or even a static prompt segment with guidelines), we ensure *consistent application of Eiffel's Method and style*. This addresses the inconsistency issue: the AI will be reminded of the "past experience" (like always create a README for a new lib, always include examples in docs, etc.). Over time, we could even encode these as a library of *"Claude Skills"*-like modules (a concept where repeatable procedures and domain expertise are packaged for the AI [23]). For instance, a "Create Eiffel Library Template" skill could automatically inject the procedure to set up the directory, README template, docs skeleton, etc., whenever a new library creation task is invoked. By moving these from ad-hoc human reminders to programmed behavior, we get more **reliable and repeatable results** [23].

- **Integration with Eiffel Build and Test Tools:** simple_dev will deeply integrate with the Eiffel toolchain to provide the AI with immediate feedback from the compiler and tests. This is where existing tools like **simple_ci** come in. simple_ci can compile multiple projects and generate a JSON report of success/failure [24] [25]. In our orchestrator, after the AI writes or modifies code, we can automatically run `ec` (the Eiffel compiler) on the affected project(s). If compilation fails, we capture the error output. We may use simple_ci's JSON output mode to get a structured list of errors and projects status [24] [26]. That JSON can be directly fed to the AI (which is easier for it to parse than raw console text) [25]. The same for test results – if we have a test suite, we run it and collect any failures. By programmatically looping the AI's output through compile/test and then through AI analysis of errors, we achieve an **autonomous fix loop**: *Code -> Compile -> (if errors) Explain & Fix -> Code (updated) -> Compile again… until clean.* This is analogous to what some advanced AI coding agents do with CI pipelines, turning bug-fixing into a collaborative loop [27]. With simple_dev, a developer could hit "go" on a high-level task and watch the tool and AI iterate until the code compiles and all tests pass – all the while logging the conversation and actions for transparency.

- **Token Efficiency and Cost Awareness:** A design priority for simple_dev is to **minimize token usage** (and by extension API costs and latency) during AI interactions. That means we won't just dump the entire codebase or spec into every prompt – we'll be selective. Techniques to achieve this include:

- Using the aforementioned **vector search** to only include relevant context snippets (functions or classes likely related to the task at hand, or just the diff of code that changed, etc.).
- Summarizing or abstracting large context when detail isn't needed. For example, instead of including an entire module's code in prompt, we might include a summary like "Module X provides A, B, C functionality" or just its interface.
- Caching AI responses (if we ask the AI to summarize the spec, we store that summary for reuse instead of asking again).
- Driving the AI in smaller steps (though it seems counterintuitive, breaking work into smaller pieces can be more token-efficient because each prompt+response focuses on one thing rather than a giant multi-thousand-line output that may need regeneration multiple times if it's wrong).
- Possibly using local models for certain tasks: For instance, a lightweight local model could handle retrieving documentation or doing a first-pass code completion, saving calls to expensive models for the harder problems. `simple_ai_client` already supports calling local Ollama models [28]. We could even have heuristic: e.g., use a local model for simple Q&A or boilerplate generation, and only call the big model if the output doesn't pass tests or for complex logic.
- Monitoring token usage and providing tools for the user to configure limits (like a dry-run mode to estimate tokens, or a cost cap that if reached, prompts the user).

By being efficient, we allow longer sessions within context windows (some models have up to 100k token context which is huge, but cost scales with usage, so efficiency still matters).

- **Interactive CLI/TUI for the Developer:** Initially, simple_dev will run as a **CLI (Command-Line Interface)** application, likely with a text-based UI to guide the developer through the process. The idea is that a user can launch `simple_dev` in a terminal, and engage in a continuous interactive session where the tool prints information and questions, and the developer can provide input or confirmations. We might incorporate a terminal-based UI (a Text UI, possibly leveraging a library like `simple_tui` if available) to present menus or formatted outputs. For example, after generating a spec, the tool could show a diff or allow the user to edit the spec if desired. Or when multiple fixes are possible for a bug, it might list them and ask the user to choose. While initially a CLI/TUI is simpler to implement and sufficient for a single developer workflow, we foresee later possibly integrating with an IDE or GUI. (Notably, the Eiffel community is exploring VS Code integration; one could imagine simple_dev hooking into an Eiffel VS Code extension down the line to provide AI-assisted features in-editor. However, for now, focusing on a robust CLI logic is the priority, which can later be wrapped in other interfaces.)

- **Generality and Configurability:** Though we are tailoring simple_dev for Eiffel development (because we believe Eiffel's strong-contract paradigm pairs excellently with AI-assisted coding), we plan to keep the system flexible so it could, in theory, be used with other languages or workflows. Users will be able to configure which AI model to use, what their API keys are, and possibly tweak the prompt templates or add custom guidelines. The goal is to make the tool **useful to any Eiffel developer** (and maybe broader audience) – someone should be able to download the simple_dev tool, set their API key, and immediately use AI to generate and maintain Eiffel code with confidence. It should *"limit the cost of using AI"* by optimizing prompts, and maximize utility by encoding a lot of

Eiffel expertise into the interactions (so that even a newcomer to Eiffel can benefit from built-in best practices).

Embracing Eiffel’s Strengths: Design by Contract meets AI

One of the motivations for building simple_dev is the observation that **Eiffel’s philosophy and toolchain can significantly enhance AI-assisted development** – and vice versa. Eiffel is uniquely positioned to take advantage of AI-generated code because of several factors:

- **Design by Contract as Automatic Validator:** Eiffel’s built-in contracts (require, ensure, invariants) act like executable specifications. When the AI writes code with contracts, those contracts immediately provide a way to **validate the AI’s output against the intended behavior**. In our Eiffel+AI experiments, we saw that postconditions could catch incorrect output that *appeared* plausible. For example, a postcondition checking that an HTML page contains a certain substring caught a missing element that the AI had overlooked [29]. This tight feedback is something we will leverage heavily. simple_dev will always encourage the AI to specify contracts for important properties. This way, if the AI misunderstands something, it’s often revealed by a contract failing at runtime. It’s much cheaper to catch an error through a contract or test and have the AI fix it, than to have silent logical bugs. As Addy Osmani noted, *“those who get the most out of coding agents tend to be those with strong testing practices… an agent like Claude can fly through a project with a good test suite as safety net”* [30]. In Eiffel, contracts are part of that safety net.

- **Compiler Errors as Guidance, Not Obstacles:** Eiffel is a compiled language with strict type checking. For an AI, hitting a compiler error is not a failure; it’s feedback for learning. The Simple Eiffel approach has been to compile early and often – essentially using the compiler to tell the AI what’s wrong. For instance, if the AI calls a feature that doesn’t exist, the compiler will throw an error that we can feed back. The AI then knows to implement or correct that feature. This is analogous to test-driven development but at the compilation level. With simple_dev, after each code generation step, we’ll compile immediately, capture any errors, and present them to the AI. The AI, having been trained on lots of code, can usually interpret compiler errors quite well and suggest fixes. This automated compile-fix loop continues until no errors remain. In effect, the Eiffel compiler (and any static analysis tools) become part of the AI’s feedback loop, **reducing hallucinations and type mistakes**. It’s worth noting that, unlike dynamic languages, Eiffel won’t let many errors slip by – which is good when an AI that sometimes produces minor mistakes is writing the code. We anticipate that this approach will yield correct-by-construction code by the time the process is done.

- **High-level Abstractions and Clear Syntax:** Eiffel’s language design (e.g. explicit type declarations, relatively verbose and self-documenting syntax) might actually help the AI stay on track. The clarity of Eiffel code can make it easier for the model to reason about and modify it without confusion (compared to languages with lots of subtle gotchas). Also, Eiffel encourages thinking in terms of contracts and classes first – exactly the kind of thinking we want the AI to adopt through prompting. We will instruct the AI to follow Eiffel idioms (like command-query separation, immutability of functions, etc.), which in turn produces code that is easier to verify. In short, Eiffel’s strong structure complements the AI’s need for explicit guidance.

- **Rich Legacy of Specifications:** Eiffel has a tradition of formal specifications and even tooling like AutoProof (for static verification). In the future, simple_dev could integrate such tools as well – for example, running AutoProof on critical classes to formally verify correctness, and then having the AI fix any proof failures. While that may be beyond the initial scope, it aligns perfectly with the idea of *provable software* built with AI assistance. The more we can harness Eiffel's verification tools, the more we can trust the AI-produced code. (A long-term vision might even compile Eiffel to WebAssembly for an online playground, or integrate other analysis to catch things contracts might not – simple_dev provides a framework to add these steps when ready.)

- **Consistency and Maintainability:** The Simple Eiffel libraries have placed emphasis on clean, uniform documentation and licensing (MIT License for all libs). By ensuring the AI always incorporates these elements (like adding the MIT license header, using the same documentation generator format), we maintain a **professional consistency** across the ecosystem. This was already noted by Eiffel community reviewers – the Readme files and docs of simple_ *libs look* "very nice and professional"* [31]. We want to keep that standard high even as more code is generated. The orchestrator will thus not only generate code, but also the peripheral pieces (docs, examples, build scripts) that make a library truly usable and polished.

Inspiration from Similar Projects and Industry Trends

Our approach with simple_dev is ambitious, but we're not working in isolation – we are inspired by and will improve upon patterns emerging in other AI-driven development tools:

- **AI "Conductor" vs. Freeform Chat:** Instead of treating the AI as an autonomous agent given an open-ended goal (as some projects like AutoGPT or GPT-Engineer have tried), we are essentially creating an AI *conductor* that coordinates the AI's efforts through a predetermined workflow. This philosophy is echoed by experienced practitioners: treat the LLM as a powerful **pair programmer** that *"requires clear direction, context, and oversight rather than autonomous judgment."* [32] In other words, the human remains in charge of the process (and ultimately accountable for the result [33]), but the heavy lifting is delegated to the AI in a controlled manner. This yields a better outcome than expecting the AI to magically handle high-level planning itself. Projects such as **GPT-Engineer** and **MetaGPT** have highlighted the need for multi-step planning (GPT-Engineer, for instance, generates a plan and then code) – we take a similar concept but tailor it to Eiffel and embed it in a continuous interactive loop rather than a one-shot generation. The **Addy Osmani** workflow we cited is a real-world validation of this approach: spec first, break into tasks, iterative development, test frequently [6] [14] [16]. We are essentially codifying those best practices into software.

- **Continuous Integration Loops:** Big tech is already experimenting with AI agents that integrate with CI pipelines. For example, Google's **Jules** or GitHub's **Copilot Agent** can clone a repo, run tests, make fixes, and even open pull requests autonomously [34]. They operate asynchronously in the cloud, acting as an AI DevOps engineer. simple_dev is conceptually similar but currently aimed at a local developer's workflow. We draw inspiration from those tools in how they use CI feedback: our use of simple_ci and feeding its report to the AI is akin to what those agents do with GitHub Actions results [15]. The key difference is simple_dev will run locally (or on the user's machine) and give the *developer* a chance to intervene at each step. This provides a reassuring level of control (the developer can inspect each AI output, run additional tests, etc., before moving on). We believe a semi-automated, human-in-the-loop approach is safer in the context of generating critical software like system libraries.

- **Context Window Workarounds:** Many tools are coming out to address the context limitation of LLMs. For instance, the **Augment Code** engine (as described in ByteByteGo) indexes entire codebases so the AI can retrieve relevant pieces on the fly [35]. There are also utilities like **gitingest** or **repo2txt** which dump relevant code parts to feed into GPT [36]. Our use of an embedding store and selective context inclusion is in line with these trends. Essentially, simple_dev serves as an **intelligent retrieval-augmented generation** system: it will retrieve the necessary code or docs for the AI instead of expecting the AI's own limited memory to suffice. This is a proven approach to scaling AI to work on large projects without hitting the "lost in the middle" problem of long context [37]. We'll continue to monitor advances here (for example, if Claude's 100k token context or tools like LangChain's agents can be leveraged, we might integrate those capabilities).

- **Skill-Based AI Responses:** The concept of AI "skills" or tool usage is gaining ground (e.g., **Claude Skills**, OpenAI function calling, etc.) [23]. Our design for simple_dev inherently gives the AI certain "skills" (like the ability to call the compiler, run tests, etc., albeit mediated by the orchestrator). As AI platforms allow more programmatic interaction, we can imagine simple_dev instructing the AI to output not just code, but possibly structured commands that the tool can interpret to perform actions (like a mini agent loop). For now, we'll likely keep things straightforward (the tool decides when to call compiler or tests), but in the future, a more agentive AI could be integrated, that knows how to invoke simple_dev's functions directly when needed. This is an area of innovation we'll watch – for instance, packaging Eiffel-specific expertise as a skill module that can be reused across projects [23]. The architecture of simple_dev should be amenable to updates as the AI APIs evolve (e.g., if an API lets the AI reference code files by name, we could utilize that instead of injecting code text).

- **Human Oversight and UI/UX:** A crucial insight from others is that **keeping a human in the loop is vital**. Simple_dev will not remove the developer from the process – rather, it amplifies the developer's productivity while continuously requiring their guidance and review. We anticipate the UX to involve the tool presenting AI outputs and asking for confirmation or adjustments. The developer might choose among options or edit the AI's code themselves at times. This interactive way of working is echoed by many who caution that *"AI will happily produce plausible-looking code, but you are responsible for quality – always review and test thoroughly"* [38]. Our system is designed to make that reviewing easier by breaking changes into small pieces and automating the testing. The end goal is that using simple_dev, a single developer can accomplish what would normally take a small team, but **without sacrificing quality**. All the traditional software engineering rigor – design discussions, version control, continuous testing, documentation – are not discarded; rather, they're baked into the workflow and often facilitated by the AI. As one commentary put it, *the classic practices – "design before coding, write tests, use version control, maintain standards – not only still apply, but are even more important when an AI is writing half your code."* [33]. We fully embrace that mindset in simple_dev's design.

Conclusion

In summary, **simple_dev** aims to be a comprehensive AI-assisted development partner tailored for Eiffel. It will address the major shortcomings of raw AI chat-based coding by introducing memory, structure, and guardrails grounded in sound software engineering principles. By guiding the AI through a structured lifecycle – from brainstorming to design to implementation, testing, and deployment – we ensure that the resulting code is correct, maintainable, and aligned with Eiffel's high standards of reliability.

This project is both a natural extension of the Simple Eiffel ecosystem and an innovation that could significantly boost developer productivity. It takes the hard-earned lessons from our recent AI coding experiments (and industry best practices) and codifies them into a repeatable process. The end result will be a tool that any Eiffel developer (or team) can use to **collaborate with AI efficiently** – harnessing the speed and creativity of LLMs, while the tool mitigates the forgetfulness, impatience, and narrow focus that LLMs often exhibit.

By prioritizing strong design, continuous verification, and knowledge preservation, simple_dev will allow Eiffel programmers to confidently generate new libraries and applications at a blazing pace without losing control of quality. It's about making AI a true engineering assistant rather than a stochastic code generator. If successful, this could position Eiffel as a frontrunner in demonstrating how classic *"provable software"* ideals can merge with cutting-edge AI capabilities. Eiffel's 2026 renaissance could very well be powered by such human+AI co-development workflows – delivering software that is not only built faster, but built right.

Proceeding from here, the next steps will involve prototyping the components described (especially the conversation flow manager and integration with `simple_ai_client` and `simple_ci`), and incrementally testing this approach on a real Eiffel project. With each iteration, we expect to refine the prompts, databases, and algorithms that constitute simple_dev. The vision laid out is broad, but by breaking our own work into manageable pieces (just as we will instruct users to do), we'll build up this capability step by step. The excitement around this project is palpable – *"we are DOING THIS!!!"* [1] – and with careful design, simple_dev could become an empowering tool for Eiffel developers worldwide, turning the dream of AI-augmented software creation into a practical reality.

Sources:

1. Liberty Lover (Larry) – *Eiffel Users Mailing List*, discussion on Simple Libraries challenge [1] [2].
2. *Simple Showcase project notes* – Eiffel + AI web development example [3] [2].
3. ByteByteGo Newsletter – *The Memory Problem: Why LLMs Sometimes Forget* [4].
4. Metabob Blog – *Pitfalls of Using LLMs in Software Development* [7].
5. Addy Osmani – *"My LLM coding workflow going into 2026"*, best practices for AI-assisted coding [6] [14] [9] [10].
6. Simple Eiffel GitHub – **simple_ai_client** documentation (AI providers, embeddings) [20] [22].
7. Simple Eiffel GitHub – **simple_ci** documentation (CI tool and Claude integration) [24] [25].
8. Addy Osmani – on integrating testing and CI feedback in AI coding loops [16] [15].
9. Eiffel Users Mailing List – feedback from Eric Bezault on Simple libraries docs [31].
10. Addy Osmani – reinforcing spec-first, human oversight, and standards in AI development [33] [32].

---

[1] [31] Simple Lib's - First Two ...
https://groups.google.com/g/eiffel-users/c/GalH2I0GY-A

[2] [3] [29] Simple Showcase (SSC) Website Plan
https://groups.google.com/g/eiffel-users/c/bqsabMSygzM

[4] [5] [35] The Memory Problem: Why LLMs Sometimes Forget Your Conversation
https://blog.bytebytego.com/p/the-memory-problem-why-llms-sometimes

[6] [9] [10] [11] [12] [13] [14] [15] [16] [17] [23] [27] [30] [32] [33] [34] [36] [38] My LLM coding workflow going into 2026 - by Addy Osmani
https://addyo.substack.com/p/my-llm-coding-workflow-going-into

[7] [8] The Hidden Pitfalls of Using LLMs in Software Development - Why Language Models Aren’t the Silver Bullet You Might Think
https://metabob.com/blog-articles/the-hidden-pitfalls-of-using-llms-in-software-development---why-language-models-arent-the-silver-bullet-you-might-think.html

[18] [19] [24] [25] [26] GitHub - simple-eiffel/simple_ci
https://github.com/simple-eiffel/simple_ci

[20] [21] [22] [28] GitHub - simple-eiffel/simple_ai_client
https://github.com/simple-eiffel/simple_ai_client

[37] The context window problem or why LLM forgets the middle of a ...
https://bdtechtalks.com/2025/02/05/the-context-window-problem-or-why-llm-forgets-the-middle-of-a-long-file/