ChatGPT

Executive Summary

Gemini Code Assist is Google's AI-powered coding assistant integrated into VS Code, providing advanced code completion, generation, and even **agentic** multi-step task automation. With just a **Gemini Pro** model (included in Google One AI Premium) and the free VS Code extension – **no paid Code Assist subscription, no WSL, no separate API or CLI usage required** – you can already leverage powerful features for Python development on Windows 10. In this guide, we detail how to maximize Gemini Pro's capabilities in a **Conda-based, multi-environment** Python workflow on Windows 10, and explain what (if anything) is gated behind the paid *Gemini Code Assist* plan. Key takeaways include:

- **Gemini Pro (Free Tier) Capabilities:** Full access to Gemini's code-aware chat and completions in VS Code, with generous daily usage limits, multi-file generation in **Agent Mode**, integrated tools (file read/write, search, shell commands, etc.), and up to a million-token context window for understanding large codebases [1] [2] – all **without** a paid subscription.

- **Agentic Coding in VS Code:** Using the Gemini Code Assist extension's **Agent Mode** (available as a preview to all users [3]), you can describe high-level goals and get a step-by-step **plan for complex tasks**, then approve or refine the plan before execution [4]. The agent can then perform multi-step operations across **multiple files**, run commands (e.g. tests or build scripts) in an integrated terminal, and apply code changes with your oversight [5]. This **human-in-the-loop** workflow ensures you stay in control.

- **GitHub & External Integration:** Even without a paid plan, Gemini can incorporate structured context from your development artifacts. You can feed in **GitHub Issues, design docs, TODO comments,** etc. as part of your prompts, or even configure open-standard **MCP (Model Context Protocol) connectors** to let the agent fetch issues/PRs or use other tools (e.g. for documentation or data) directly [6] [7]. Additionally, the free **Gemini Code Assist for GitHub** app can be installed to get automatic AI code reviews on pull requests [8].

- **CLI vs Extension:** The VS Code extension already bundles **Gemini CLI** capabilities (Agent Mode in VS Code is "powered by the Gemini CLI" [9]). This means most agentic features are accessible within VS Code's UI. A standalone **Gemini CLI** (Node.js-based) can be installed if you want to run the AI assistant in a terminal or automate tasks outside VS Code, but it is optional for normal usage. All core features – from code generation to running tests via shell – **work inside VS Code** without needing to manually invoke the CLI.

- **Conda Environment Integration:** In a Windows 10 + Conda setup (with no system Python), Gemini can still execute code and tooling by leveraging VS Code's terminal. You'll want to ensure your Conda environment is activated in VS Code's integrated terminal so that any **"Run tests"** or shell commands the agent executes use the correct Python interpreter and dependencies. This guide provides tips (like using VS Code tasks or shell profile) to seamlessly integrate Conda with Gemini's agentic workflow.

Below we present a detailed **capabilities matrix**, an in-depth breakdown of features (what you get with Gemini Pro alone versus what *Code Assist* subscriptions add), and concrete **workflow recipes** for Python development on Windows (including environment setup, prompting techniques, multi-module project management, and integrating GitHub context). We also demystify the role of the CLI and MCP servers for those considering advanced integration, and provide step-by-step guidance to set up and start coding with Gemini on your specific environment. By the end, you'll have a clear playbook for mastering Gemini Pro's AI coding assistance within VS Code – maximizing productivity under the given constraints and knowing exactly when (or if) you might need additional tools or subscriptions.

Capabilities Matrix: Free vs Paid Gemini Features

The table below summarizes which features are available with **Gemini Pro (free, included in Google One AI Premium)** using only the VS Code extension, and which features would require a paid *Gemini Code Assist* **Standard/Enterprise** plan or additional tools. We also note if any aspect requires the CLI or other setup. In general, the free tier is very full-featured – paid plans mostly raise usage limits or add enterprise-specific capabilities.

Feature or Capability	Gemini Pro in VS Code (Free)	Requires Code Assist Subscription?	Notes (CLI/MCP requirement)
Code Chat (Conversational Assistant) – Ask questions about code, get explanations, help with APIs, etc.	**Yes** – Available via VS Code chat panel [1]. Uses Gemini 2.5 model with large context.	No (free for individuals).	No CLI needed. Chat aware of open file content by default.
Inline Code Completions – Auto-complete code while typing.	**Yes** – Provided by extension in editor [1].	No.	N/A – works out-of-the-box.
On-Demand Code Generation – Generate code for a given prompt (function, class, etc.).	**Yes** – Via chat or `/generate` command in VS Code [10].	No.	Use VS Code *Quick Pick* commands (Ctrl+I) for inline generation.
Code Transformation – Automated refactor, fix, doc generation on existing code.	**Yes** – Via chat or smart commands (`/fix`, `/doc`, `/simplify`, etc.) [10].	No.	N/A – integrated in extension. Produces diff that you can accept.

Feature or Capability	Gemini Pro in VS Code (Free)	Requires Code Assist Subscription?	Notes (CLI/MCP requirement)
Context Window Size – Amount of code it can consider at once.	**Large** – Gemini 2.5 Pro supports up to ~1M tokens context [2] (free version likely uses a large chunk of this).	*Larger* with Enterprise (Gemini 3 with 1M+ tokens guaranteed [11]).	No difference in how to use; paid plans mainly ensure newest model with full context. Free model can already handle entire repo via tools.
Multi-File Support – Ability to read and modify multiple files in one session.	**Yes** – Enabled in **Agent Mode** (Preview) [12]. Agent can open, create, and edit multiple files as part of a plan.	No (available to free users in preview) [3].	Works in VS Code Agent Mode. No separate CLI needed (the extension uses CLI internally).
Plan & Approve Workflow – AI generates a multi-step plan for complex tasks, user approves before execution.	**Yes** – Available in Agent Mode for free [4]. Gemini will propose a plan and await your approval [13].	No.	Part of Agent Mode (no subscription needed). Ensure "Agent" toggle is on in chat.
Built-in Tool Use – AI can use tools like file read/write, search, or run terminal commands.	**Yes** – All **Gemini CLI built-in tools** are accessible in VS Code's agent mode [14]. E.g. `read_file`, `write_file`, `grep`, `shell` (terminal), etc.	No (free).	Some tools (e.g. running shell) will ask for permission [15]. No separate CLI required; extension handles it.
Internet Access (Web Search) – AI can perform Google searches or open URLs for info.	**Yes** – Built-in support in Agent Mode for web search and fetching URLs [16] [17].	No.	Enabled by default in agent mode. Might prompt for permission if needed.
IDE Context Awareness – AI automatically knows about open files and project structure.	**Yes** – Reads open file content, plus can index project symbols in IntelliJ; in VS Code can search project with tools [18] [19].	No (free has it).	"Local codebase awareness" can be configured; enterprise can index private repos deeply, but free uses tools like grep to find content.

Feature or Capability	Gemini Pro in VS Code (Free)	Requires Code Assist Subscription?	Notes (CLI/MCP requirement)
Source Citation – AI cites documentation or code sources for its answers.	**Yes (in normal chat)** – Gemini will cite sources for generated code or answers [20].	No subscription needed.	**Note:** Citations are **disabled in Agent Mode** (agent won't output reference snippets while acting [21]). Use normal chat for cited answers.
Automated GitHub PR Reviews – AI review comments on PRs via `/gemini`.	**Yes** – Free GitHub App available [8].	No (the app is free for individuals).	No CLI/IDE needed; works through GitHub web UI. (Paid plans target enterprise scale, not logic).
Direct GitHub Integration (Issues/PRs) – AI can fetch issues, comment, create PRs, etc. via API.	**Yes, with setup** – Requires configuring an **MCP server** for GitHub or using `git` in shell [6].	No extra subscription, but *manual setup* of tool.	*MCP* (open protocol) integration is supported in free agent mode [22]. You'll need to provide a GitHub token and add the server config.
Running Tests / Build Scripts – AI can execute `pytest`, build commands, etc.	**Yes** – Via the `shell` tool in agent mode (runs in VS Code terminal) [23] [19].	No.	Ensure environment is activated so commands succeed. Agent will ask permission to run commands [15].
Multi-step Task Automation – Agent maintains state across several actions (e.g. edit code, then run code, then edit again).	**Yes** – Agent Mode keeps context and tool outputs in a continuous session [24] [25].	No.	Free tier allows ~1000 agent actions/day [26]. Paid plans raise the limit (1500/2000).
Persistent Agent Sessions – Chat history with agent retained, agent toggle state persists across restarts.	**Yes** – Recent updates preserve agent mode state in VS Code [27] [28].	No.	Included in free (part of extension update).

Feature or Capability	Gemini Pro in VS Code (Free)	Requires Code Assist Subscription?	Notes (CLI/MCP requirement)
Advanced Model Access (Gemini 3) – Access to larger/newer models.	*Partial* – Gemini 3 not immediately in free (as of '25, join waitlist) [29]. Google One **Ultra** subscribers get early access.	**Yes** – Standard/ Enterprise includes Gemini 3 access once available [29].	With free + AI Pro, you use Gemini 2.5 Pro by default. (Still state-of-art in coding [30]).
Context from Private Repos (Customization) – AI is fine-tuned on your internal codebase.	**No** – Not in free.	**Yes** – Enterprise feature (Code Customization) [31] [32].	Free users can still open relevant code or docs manually for context.
Usage Quotas – Daily limits on requests.	**Yes** – Free Individual: *~6000 code completions* + *240 chat messages/day* [1]; *1000 agent/CLI actions/day* [33] [34].	Higher with paid: (Standard ~1500, Enterprise ~2000 agent actions/day) [35] [36].	Google One AI Pro may boost some limits (and ensure "Pro" model usage). Typically free limits are generous for normal dev work.
Gemini CLI (Standalone) – Use AI assistant in terminal outside VS Code.	**Yes** – Free to install/use (open-source CLI) [37].	No subscription needed (license improves quotas).	Not required if using VS Code extension (which leverages CLI). CLI use is optional for non-IDE workflows.
MCP (Model Context Protocol) Tools – Extend agent with community connectors (APIs, DBs, etc.).	**Yes** – Supported in free agent mode (manual config) [38] [39].	No paywall (most MCP servers are community/free).	Some MCP servers are local Node packages or remote endpoints. Requires Node installed to run `npx` commands as configured [40] [41].

Table Key: "Yes" means fully available in specified scenario; "Partial" indicates some limitations (with details in notes). As shown, **almost all features are available with Gemini Pro alone** – a paid Code Assist plan mainly lifts quotas or provides enterprise integrations (e.g. private repo tuning). In the following sections, we explore each feature in detail and provide practical guidance for using them in your Windows 10 + Conda + VSCode environment.

Feature-by-Feature Breakdown

Let's break down Gemini's capabilities one by one, highlighting how they work in our specific context (Windows 10, VSCode, Conda) and noting any differences when **no Code Assist subscription** is involved:

Conversational Coding Assistance (Chat Interface)

Gemini Code Assist provides a chat-based assistant within VS Code that is aware of your code context. In free/pro mode, you can ask questions about your code or get help writing new code in natural language. For example, you might ask: *"How do I implement a singleton pattern in Python?"* or *"What does this error mean?"* The assistant will consider any open file content and project context when formulating answers [18], and often provide **cited references** to official docs or StackOverflow for explanations [20]. This is great for learning APIs or understanding unfamiliar code – e.g., "Explain what the function in `utils.py` does," and it will summarize it (with citations if applicable).

- **Context Awareness:** In VS Code, Gemini sees the content of open files and can even search your workspace if needed. The agent mode documentation notes that in VSCode, context comes from "information in your IDE workspace" and tool outputs like grep or file reads [18]. So, if you have a file open or recently viewed, it prioritizes that. You don't need to paste the whole file; you can refer to it by name in your prompt, and Gemini can invoke a `read_file` tool internally to fetch it.

- **Source Citations:** One unique feature (compared to other code assistants) is that standard Gemini chat will cite sources for code suggestions or explanations [20]. For instance, if you ask "How to use `pandas.DataFrame.apply`?", it may quote from pandas docs with a link. This helps with trust and verification. However, **note:** In Agent Mode (the more autonomous mode), source citation is turned **off** by design [21], because the focus there is on actions, not just Q&A. So, if you want an answer with references, use the normal chat (not agent mode) for that query.

- **Coding Languages and Scope:** Ensure the file you're working on is in a supported language (Gemini supports all popular languages, including Python, JS, Java, etc., and general text). According to Google's docs, you should have language mode set properly in VS Code so Gemini knows how to respond [42]. In our case, Python is fully supported.

How to use: Open the *Gemini Code Assist* panel in VS Code (from the Activity Bar, click the Gemini icon). Start a new chat (ensure the **Agent toggle is off** for normal Q&A mode). You can type questions or requests here. The assistant will respond with explanations or code suggestions. It can also take an active look at your code – e.g., highlight a code block and ask "What's wrong with this?" to get a quick review.

Inline Code Completions and Suggestions

Gemini Code Assist integrates into the editor to provide **autocomplete** and code suggestion as you type. This works similarly to tools like GitHub Copilot:

- As you write code, you'll notice ghost-text suggestions or entire line completions appearing. These are powered by Gemini and do not require any subscription (it's included in the free extension) [1]. For example, typing `def calculate_total(` might prompt a suggestion like `def calculate_total(items: list) -> float:` if Gemini infers what you intend.

- It uses the **open file context** and maybe some project context to make relevant suggestions. It won't be as robust as the chat for complex logic, but it helps with boilerplate and repetitive code.

- There is no special action needed to use this; just start coding and accept suggestions with `Tab` or `Enter` when they appear. If multiple suggestions are available, VS Code might show a drop-down you can cycle through.

Limitations: The free vs paid doesn't change how completions work, except free users are limited to 6k "code-related requests" per day [1]. A single completion counts as one request, so that's plenty. Paid plans would raise this limit if you somehow hit it (e.g., extremely heavy usage). In practice, 6k code completions/day is more than enough for full-day coding.

On-Demand Code Generation (Single-file or Snippet Generation)

When you want Gemini to generate code on demand (beyond inline completions), you have a couple of powerful options:

- **Quick Pick Commands (Smart Actions):** In VS Code, pressing **Ctrl+I** (Windows) opens the Gemini *Quick Pick* menu with special commands like `/generate`, `/fix`, `/doc`, `/simplify`, etc. [10]. These let you ask for specific transformations:
- `/generate …` – generate code based on a description.
- `/fix …` – fix an issue in the selected code.
- `/doc …` – add documentation to the code.
- `/simplify …` – simplify/refactor the code.

For example, you could write a comment like `# TODO: function to parse config file` in your code, place the cursor there, press Ctrl+I, and type `/generate function to parse a JSON config file`. Gemini will then produce the function code in a **diff view** for you to review and accept [43] [44]. The diff view shows the new code to be inserted, and you can apply it if it looks good. This is **free** for all users; it's part of the extension's features.

- **Chat-Based Generation:** Alternatively, you can use the chat interface. For instance, you can prompt: *"Generate a Python class for handling user authentication (with methods login, logout, etc.)"*. Gemini will output the code in the chat reply. In VS Code, you'll typically get a **"Insert"** button or you can copy-paste from the response. The advantage of using chat is you can discuss requirements: e.g., "Make the login method verify password with bcrypt".

Both methods work without a subscription. The difference is that the Quick Pick commands give you a convenient in-place edit (with diff and accept/reject), whereas chat is more freeform and good for iterative asks.

Multi-file generation caveat (free mode): If your request inherently requires creating multiple files, the single-file generation methods won't directly do that. For example, "generate a Flask app with two modules" – normal chat will likely paste both files' code in one response, which you'd have to manually separate. This is where **Agent Mode** (discussed below) shines, because it can create multiple files in one go. In free mode you *do have* agent mode, so you can choose to use that for multi-file tasks. But if you prefer not to invoke the agent, you can always generate code file by file in separate requests.

Agent Mode (Plan-Execute Workflow for Multi-Step Tasks)

Agent Mode is Gemini's "AI pair programmer" that can autonomously perform a sequence of coding tasks. This is extremely useful for complex changes that touch many parts of a project (refactoring multiple modules, adding a feature that requires new files plus tests, etc.). Importantly, **Agent Mode is available in the VS Code extension even without a paid plan** – currently as a Preview feature that you can opt into [3].

How to activate Agent Mode: In the Gemini chat panel, there's a toggle or a tab for "Agent". On VS Code: 1. Open the Gemini chat view (the panel where you normally type questions). 2. Click the **Agent toggle** (it appears as a switch or button). When active, it's highlighted (and labeled, depending on version) [45] [46]. 3. Now whatever prompt you enter will be handled in Agent Mode.

You'll know it's agent mode because the responses will be more action-oriented. If you're on a recent extension version, agent mode might already be enabled by default (or you might have needed to enable "pre-release features" to get it – check extension settings if you don't see the toggle).

What Agent Mode does: Instead of just answering, the agent can **propose a plan and then execute it step by step**. For example, suppose you prompt: *"Add support for discount codes in the shopping cart feature. This will involve updating the Cart model, modifying the checkout function to apply discounts, and adjusting the UI to display the discounted price."* This is a high-level request that touches multiple files.

- **Plan Generation:** Gemini will likely respond with something like: "**Plan:** 1. Update `cart_model.py` to include a field or method for discount code; 2. Update `checkout.py` to apply discount calculation using that code; 3. Modify `template.html` to show the discounted total; 4. Adjust unit tests or add a new test for discount logic." (It may even break steps down further). This plan is presented for your review before any code changes happen [13]. You can ask for clarification or modifications: e.g., "Actually, apply the discount at the service layer, not in the model," and the agent will tweak the plan.

- **Approval:** Once the plan looks good, you click **"Approve"** (or a similar prompt in chat) to let the agent proceed [13] [47]. If you do nothing, it won't execute, giving you full control. This feature ensures you're comfortable with what the AI is going to do – **you're always in the loop** (HiTL oversight).

- **Tool Use & Execution:** After approval, the agent begins executing each step:

- It may read or modify files (you'll see it opening files or showing diffs in VS Code as it works).
- If it needs to create a new file (say, a new module or test file), it will do so via the file write tool.

- It can run shell commands if needed. For instance, it might run `pytest` after making code changes to verify tests pass. When it tries to run a command or perform any action that changes your system (file writes, etc.), **Gemini will ask permission** unless you've pre-approved that tool [48] [24]. You'll get a prompt like: *"Allow running* `pytest` *in /path/to/your/project?"* with options to Allow or Deny. You can allow once or always for the session [15]. If it's a read-only action (like just reading a file or doing a grep search), it might not prompt.

- The VS Code integration shows outputs in real-time. The Google blog notes *"real-time shell command output"* is now supported [49], so if the agent runs tests, you can see the test results streaming in the chat or an output panel. For example, you might see in the chat: *ShellTool*: running `pytest -q` … followed by any test failure output.

- After each step, Gemini uses the result (file changes or command output) as new context and decides the next move [50] [24]. This loop continues until all plan steps are done or something unexpected happens (error, or it needs clarification).

- **Reviewing Changes:** Code edits are usually presented as inline diffs or directly applied in the editor with a highlight. Recent improvements allow "inline diff in chat" for clarity [51]. This means the agent might show a snippet:

  ```
  - total = sum(item.price for item in items)
  + total = sum(item.price for item in items)
  + if cart.discount_code:
  +     total *= get_discount_multiplier(cart.discount_code)
  ```

  You can confirm the edits in your files. If something looks wrong, you can intervene (e.g., stop the agent or edit manually and tell the agent).

- **Persistent Session:** Throughout one agent session, it keeps the context of what it's done. If it finishes and you then give another related instruction in the same chat, it remembers the previous steps (within the context window). Notably, the blog indicates *"persistent agent mode in chat history"* so you can scroll back approved steps [27]. If you close VS Code, the agent toggle state is preserved [52], but you'd start a fresh session next time (it doesn't resume mid-plan after a restart).

What's possible in Agent Mode: A lot! Some examples: - **Refactoring Across Files:** The example above (adding discount code) demonstrates changing multiple files consistently. Agent mode ensures the changes are synchronized (e.g., the model and the part that uses it agree on field names). - **Feature Implementation:** You can task the agent with adding a whole feature, from scaffolding new modules to updating config and docs. It will create new files as needed. For instance: "Set up a logging utility module and use it in all controllers" – plan might be: *Create* `logging_util.py`*, add import and usage in files A, B, C* – and it will do all that. - **Bug Fix + Test Workflow:** If you describe a bug, the agent could locate the cause and fix it, then run tests to verify. For example: "We have a bug where user sessions aren't cleared on logout. Find the cause and fix it, and run the test suite." The agent could plan: find usages of logout, see missing session clear, add that, then execute tests. If a test fails, it might iterate (with your permission) to fix further until tests pass. This is essentially an automated debug cycle. - **Codebase Q&A with Actions:** You can mix natural language with actions. E.g., "List all functions that are not covered by tests." The agent could use the `grep` tool to find `def` lines and correlate with test files, then report, or even generate skeleton tests for them. This goes beyond a static answer – it actually *uses tools* to dig into your repo structure. This is enabled by built-in tools like `grep`, `find_files`, etc., which are available in VS Code's agent mode [53] [54]. Essentially, Gemini CLI's entire toolset is at its disposal within VS Code agent mode.

Limitations and Differences Without Subscription: The agent mode features described are **fully available** to free users in preview. The paid plan does not unlock any new *capability* in agent mode; it

mainly increases the allowance of daily uses and promises enterprise support. For instance, an enterprise user might integrate custom MCP tools or have bigger context (with Gemini 3's 1M tokens fully utilized), but practically, even free Gemini 2.5 Pro has a huge context window (enough for most projects) [2]. So, you as a Pro user should not feel hindered in functionality.

One thing to note: since Agent Mode is relatively new and *preview* for individuals, you might encounter occasional hiccups (the "experience is continuously improved" per Google [55]). E.g., sometimes the plan might miss a step, or a tool might not behave as expected on Windows. Treat it as a powerful assistant, but keep an eye on it (which you are forced to do anyway by approving steps).

Built-in Tools and Execution Abilities

We've touched on tools, but let's clarify what tools Gemini can use in your environment, especially without any additional setup:

Gemini's agent can use **Built-in Tools** which come from the open-source Gemini CLI. According to documentation, in VS Code agent mode, *"All of the Gemini CLI built-in tools are available"* [14]. Key tools include: - **Filesystem tools:** `read_file` (to read file content), `write_file` (to create/modify files), `list_files` (list directory contents), `find_files` (find by name), `grep` (search within files by text) [56] [57]. These enable the agent to scan your project. For example, it might use `grep` to find where a function is used before refactoring it. - **Terminal/Shell tool:** This allows execution of shell commands. In IntelliJ it's listed as `git` and others [58], but in VS Code, typically a generic **ShellTool** is used to run commands in the project directory. On Windows, the agent will use the default shell (likely PowerShell or CMD) to run commands like `dir` or `pytest` [59] [60]. The Real Python guide confirms that on Windows, it smartly chooses `dir` or `ls` as appropriate [60]. So it is aware of the OS.

- **Version control:** There is a `git` tool (in IntelliJ at least) that can run git CLI commands [58]. In VS Code, the agent could also just call `git` via shell. This means it can do things like check `git status`, create branches, or diff changes if you allow. Out of the box, it won't automatically push or create PRs (unless you explicitly instruct it and grant permission). So don't worry – it's not going to publish code without you confirming.

- **Web access:** Gemini can perform a web search and open URLs. This is implemented under the hood via tools (often called something like `search` or `open_url`). The agent mode context mention includes "Google Search responses" and "content from a given URL provided in a prompt" [16]. For example, if you ask, *"Has CVE-2023-1234 been fixed in Django?,"* the agent might do a web search, find the relevant Django release notes, and summarize the answer. This is extremely useful for getting up-to-date info or troubleshooting errors by searching the web – and it's available in free mode. (It will ask for your permission the first time it tries a web search, typically.)

- **Analysis tools:** There's `analyze_current_file` in IntelliJ listing (which likely flags errors/warnings) [61]. In VS Code, similar analysis might be done implicitly or via language server; but the agent can also just read the file and point out issues. For example, "analyze this file for potential bugs" would essentially prompt the model itself to scrutinize the code.

Tool Permissions: When a tool is invoked that could change things or access sensitive data, you'll get a prompt. The Real Python tutorial shows an example: when Gemini CLI wanted to run `dir` or `ls`, it asked: `1. Yes, allow once` `2. Yes, allow always (for this session)` `3. No, suggest changes (or cancel)` [15]. In VS Code, the UI might present a similar choice (perhaps buttons). Always read what command it's asking to run – 99% of the time it will be something benign *you asked for* (like running tests, installing a package, etc.). You have granular control: you could allow all `pytest` commands without asking again, for instance, but still have it prompt for any `rm` command. In fact, you can configure a settings JSON (`~/.gemini/settings.json`) to whitelist or blacklist specific tools or commands [62] [63] – e.g., explicitly block any destructive commands like `rm -rf` just to be safe [64]. By default, though, everything is available but gated by ask-first.

Conda environment and shell: One important detail for our scenario – when the agent runs shell commands, we want it to use our Conda Python and environment. By default, the VS Code extension will spawn the shell in your workspace directory, but it may not auto-activate your conda environment. If you open VS Code normally, the integrated terminal might be plain Powershell or CMD. This means if the agent runs `python` or `pytest`, it could invoke the wrong Python (or fail if no system Python). We will address how to ensure Conda activation in the "Workflows & Setup" section. In short, you can: - Manually activate the env in the terminal **before** approving any agent shell command. - Or instruct the agent as a first step: "Run the environment activation script," so it sets up the environment for subsequent steps. (E.g., have a step 0 in plan: execute `conda activate myenv` or call your `.bat` script.)

Gemini won't automatically know about Conda environments unless told, but once the env is active, any tool or command runs in that context.

GitHub Integration and Using Development Artifacts as Context

Even without using any paid APIs, you can integrate **GitHub artifacts** (issues, PRs, commit history) into your Gemini workflow in a few ways:

- **Manual context provision:** The simplest method is to copy relevant text from GitHub into your prompt or a file. For example, if you have a GitHub Issue describing a feature request or a bug, you can paste the issue description into the chat and then say "Implement the above feature" or "Fix the bug described above." Gemini will take that as part of the input context and act on it. Since Gemini Pro has a very large context window, it can handle even a long issue or discussion thread (many thousands of tokens) without trouble [2].

- Tip: Use a formatting to delineate the context, e.g., paste the issue text in a code block or quote block, so it's clear what is user prompt vs reference. You might say: *"Given the following GitHub issue description, implement the requested feature."* Then quote the issue. The model can then plan the implementation.

- **"Context files" in Markdown:** The agent mode supports context files – if you have a design doc or requirements written in a markdown file, you can load it as context. The docs mention you can create such files and Gemini will consider them [16] [65]. Practically, if you have `DESIGN.md` open in VS Code, the agent likely will treat it as part of the workspace context (especially if you reference it).

Another trick is to use the `read_file` tool explicitly: e.g., you can prompt in agent chat: *"Read the file DESIGN.md for details"*, and it will use the tool to ingest it, then you can ask it to proceed with coding based on that.

- **MCP GitHub connector:** For a more automated integration, Google provides an **MCP server for GitHub** (as part of the Model Context Protocol ecosystem). This is essentially a connector that the agent can use to interface with the GitHub API. By configuring it, you unlock commands like `list_issues`, `get_issue`, `create_pr`, etc., via natural language. For example, you could ask: "Gemini, fetch issue #123 and see what needs to be done," and the agent will call the MCP tool to get the issue data (via GitHub API, using your token) and incorporate that. The MCP registry listing describes it as *"Connect AI assistants to GitHub – manage repos, issues, PRs, and workflows through natural language."* [66].

Using MCP GitHub (if desired): Since you prefer not using CLI extensively, note that enabling this still requires some configuration: - You need Node.js installed (because the MCP servers are Node packages). - Edit your `~/.gemini/settings.json` to add an entry under `"mcpServers"` for GitHub. The docs give an example JSON snippet for a local GitHub server [67] [41]. It looks like: `json`

```
"mcpServers": {
  "github": {
    "command": "npx",
    "args": ["-y", "@modelcontextprotocol/server-github"],
    "env": { "GITHUB_PERSONAL_ACCESS_TOKEN": "ghp_yourTokenHere" }
  }
}
```

This will instruct the agent to run the GitHub MCP server via `npx` when needed. The token should have appropriate scopes (repo access, etc., depending on what you want it to do).

- Once configured and VS Code is restarted, the agent will have new tools (the functions provided by that server) it can use. You can then reference issues by number or ask it to open PRs. For instance: "Use the GitHub tool to create a PR with the changes" – the agent might then prepare a commit and call the MCP function to create a PR (though this is advanced; you might want to do commits yourself for now).

However, setting up MCP is **optional**. For many users, manually providing the relevant GitHub info to the AI is simpler and safer. You might not want the AI auto-managing your repo until you're confident in it. A middle ground is:

- **Using Git (CLI) via agent:** Without MCP, the agent can still use `git` commands on your local repo. For example, after making changes it might suggest: "I can run `git diff` to show you what changed, or `git commit` the changes." If you allow it, it could commit to your local repo (you would then push manually). It could even run `git log` or `git show` to retrieve commit history info if relevant to a task (like analyzing when a bug was introduced). These are standard shell operations that require your okay. In free mode, all of this is available (no subscription needed), since it's just using the local Git installation through shell.

- **GitHub Pull Request Review (Gemini GitHub App):** Separate from the IDE agent, Google offers a GitHub App that can be installed on your repos to perform AI reviews. Since you mentioned GitHub *issues, PRs, discussions*, it's worth noting: with the app installed, you can comment on a PR with `/`

`gemini` and it will generate a review with suggestions automatically [8]. This is a free service for individuals (at least as of now). It's not exactly part of your VS Code environment, but it complements it – e.g., after you push a PR, you can get an AI check on it. The app uses Gemini models under the hood. It might catch things or refactor suggestions. If you want to try this: go to the GitHub Marketplace and install *Gemini Code Assist* on your repo [68]. Then in a PR comment type `/gemini` (or specific commands like `/gemini summarize` or `/gemini fix`) as documented. It will reply in the PR. This doesn't need any of your local configuration and does not require Code Assist subscription either (the Code Assist Standard plan is more for enterprise with multiple users, etc.).

Using other structured artifacts: Beyond GitHub, you might incorporate: - **Jira tickets or other issue trackers:** You can paste ticket descriptions just like GitHub issues. Or write an MCP server for Jira if one exists. But simplest is copy-paste. - **Design diagrams:** If you have an image (like a UML diagram or architecture diagram in an image/PDF), Gemini can't directly "see" images unless using a vision-capable model. As of now, Gemini 2.5 is multimodal in the sense it can handle text and code, and reportedly images/audio [69], but in Code Assist context, I believe image understanding isn't exposed (Gemini 3 might add more vision). One trick: if it's an image with text (like a screenshot of JSON or code), you could use an OCR tool to extract text and feed it. There is an MCP called **Markitdown** that "Convert various file formats (PDF, Word, Excel, images, audio) to Markdown" [70]. That is by Microsoft and could be integrated. But that's probably overkill unless you frequently need to feed non-text info. In most cases, having a textual representation (like a written description of architecture) is easier.

- **Commit history / logs as context:** You can ask the agent to read commit messages to understand evolution. For example: "Open the commit history for file X and summarize changes." It can use `git log -p fileX` via shell to get that info. This is a way to use your VCS history to inform the AI about why something was done, etc. Use with care (lots of output), but it's possible.

Overall, **free Gemini Pro is fully capable of integrating your development artifacts into its prompts** – either through direct inclusion or via the powerful mechanism of MCP tools if you invest some setup time.

Tasks Requiring CLI or Not

The **Gemini VS Code extension** is designed such that you **do not need to separately run the CLI** for almost any interactive development task. The extension itself, when Agent Mode is toggled on, is essentially running an instance of the CLI under the hood [9]. Therefore: - You *do not* have to open a terminal and run `gemini` commands manually to use agent features – you get the chat interface and UI integration which is more convenient. - In fact, some features like the inline diff insertion are unique to the IDE integration (the CLI in a raw terminal would show a diff in text form, but you'd have to apply it yourself; VS Code can directly apply it).

However, there are scenarios where the standalone **Gemini CLI** might be useful: - **Developing outside VS Code:** If sometimes you edit with another editor or are SSH'd into a server (though you said WSL not available, maybe a remote Windows server or something), you can use the CLI to get the same AI assistance in the terminal. - **Scripting and Automation:** You might integrate AI assistance into scripts or CI. For example, a script could call `gemini` CLI to analyze code quality nightly or generate reports. (Be mindful of non-interactive usage and rate limits though.) - **Preference for command-line:** Some developers just prefer text interface. The CLI provides an interactive REPL where you type requests and it performs them, similar to agent mode. - **No GUI environments:** If on a minimal system with just terminal, CLI is the fallback.

Installing and using Gemini CLI on Windows 10: If you choose to install it: 1. **Install Node.js 20+** (if not already). This is required as the CLI is distributed via npm. 2. **Global install:** Run `npm install -g @google/gemini-cli`. (This is the official package name) [71]. Alternatively, use a package manager: there's a Chocolatey and Winget package reportedly [72] [73], or via MSYS2's pacman (`mingw-w64-gemini-cli` exists [74]). But npm is straightforward. 3. **Login:** After install, run `gemini login` (or the first time you run any command it will prompt web auth). It will open a browser for Google auth. Log in with your Google account that has Google One AI Pro. That authenticates the CLI to use your free quota. 4. **Usage:** You can then run `gemini chat` to enter an interactive chat, or use one-shot commands like `gemini run "Your prompt here"`. The CLI supports agentic operations too – by default it starts in agent mode for code projects. You would `cd` into your project folder and run `gemini` – it will detect the git repo and some context, then you can type tasks. - Example (from RealPython): in a project directory, run `gemini`. Then type: *"Summarize what this project does."* It might ask to run `dir` or `ls` to list files (you allow it) and then it will output a summary [25] [75]. This is exactly what VS Code agent would do, but in a text form. - You can also do specific actions, like `gemini fix file.py "Description of fix"` to directly apply a fix – but you'd likely just interactively chat.

1. **MSYS2 consideration:** Since you have MSYS2 (from Ruby), you could run the CLI inside the MSYS2 shell as well. If you prefer a Unix-like environment for the CLI, installing it via pacman (`pacman -S mingw-w64-x86_64-gemini-cli`) might allow running in the MSYS terminal. But this isn't necessary; the Node version works in Windows CMD/Powershell and will call native Windows commands or use Git Bash if it finds it. The CLI is cross-platform.

CLI vs Extension features: The functionality is nearly identical. The VS Code extension **is actually slightly ahead** in user-friendliness (with diff apply, UI buttons, etc.). The CLI might give you more raw control in some cases – e.g., you could pipe in a prompt from a file or use it in CI pipelines. Also, if you want to easily integrate custom MCP servers, sometimes doing it in CLI's config is simpler to debug, then replicate in VS Code. But for day-to-day dev on your machine, the extension covers you.

Conclusion: You likely *do not need* to use the CLI separately unless you have specific use cases for it. All the agentic coding tasks (multi-file edits, running tests, etc.) work inside VS Code's UI because the extension leverages the CLI core [76]. Therefore, you can treat "CLI vs no CLI" as mostly a matter of preference/advanced use. This guide will focus on using VS Code directly, and note if any workflow would require dropping to CLI (few do).

Model Context Protocol (MCP) and External Tools

MCP (Model Context Protocol) is an open standard (spearheaded by Anthropic and embraced by others) for connecting LLMs to external tools and data in a standardized way [77] [78]. In simpler terms, MCP allows you to plug "servers" that expose certain functions (like an API) into your AI assistant. Gemini Code Assist supports MCP to integrate with ecosystem tools [5].

For your environment, what does this mean? - Out-of-the-box, Gemini has a set of built-in tools (which we covered). Those cover many local development needs. - MCP extends this by letting the community (or you) add new tools. Examples from the MCP registry (which is accessible via GitHub) include connectors to cloud services, databases, documentation indexes, etc. Some highlights: - **GitHub Server:** we discussed – manage repo issues/PRs via AI [6]. - **Context7:** a tool by Upstash that provides "up-to-date code docs for any prompt" [79]. This likely means it can fetch library documentation or code examples dynamically. - **Serena:**

"Semantic code retrieval & editing tools" [80] – possibly allows semantic search in your code or advanced refactoring across a codebase (could be useful for large projects). - **Markitdown:** convert PDFs/images to text [70], useful if you have binary docs. - There are others for monitoring (Netdata), browser automation (Playwright), etc., which might be out of scope for local dev but show the range (even Chrome DevTools automation [81]).

Do MCP servers matter for Python-only local development? - **Not strictly necessary** for most tasks. You can do a lot with built-in tools. For instance, if you want to search documentation, the agent can use Google Search itself to find answers (without a specialized doc tool). If you want code retrieval, the agent's `grep` works, though a semantic search like Serena might find related code even if words don't match – that's a bonus but not a requirement. - **When they help:** If you often need to interface with external systems from your code context. For example, if part of your dev workflow involves querying a database schema or a REST API for info while coding, having an MCP tool for that could let the agent do it automatically. Or if you want the agent to manage cloud resources (Google Cloud, AWS) as part of coding (like spin up a test instance), there might be MCP tools for those (Google Cloud integration is mentioned). - **GitHub MCP specifically:** If you frequently reference issues or want the AI to handle branches/PRs, installing the GH server might be worth it. It can save you from copy-pasting issue text and potentially let the AI create PRs or update issues for you (with your approval). This can integrate project management with coding tasks, e.g., "Gemini, according to issue #45, implement that feature and then mark the issue as resolved" – it could do the code, commit, push (if configured), and call GitHub API to comment on the issue. However, that level of autonomy should be tried carefully.

Setup caveats on Windows: - Many MCP servers are distributed via npm (like `@modelcontextprotocol/server-github`). They run as separate Node processes when invoked. On Windows, this should work as long as Node is installed. - If a server requires a local binary (e.g., some might need a database CLI), you must install those dependencies. The configuration instructions for each MCP (usually on their npm or GitHub page) will list what's needed [82]. - The **MSYS2 environment** you have might not be directly needed unless a tool expects a Unix environment. If a particular MCP tool uses Linux-specific commands, you could run it within MSYS2. But since these are Node packages, they should run on Windows as well. For instance, the GitHub MCP will use Node's ability to call GitHub's REST API – that's OS-agnostic. If a tool like "playwright" requires a headless browser, it will download a browser – which works on Windows too. - In case you did want to run an MCP server that only runs on Linux, you could potentially spin it up in a VM or container and mark it as a remote server in settings (notice the example in docs for GitLab uses an `mcp-remote` with a URL [83] – implying you can connect to remote MCP endpoints over HTTP). But again, for local dev, likely not needed.

Security and performance: Running MCP servers means more processes and possibly external data. Only add those you trust, as they can execute code (e.g., the GitHub server will have your token). The doc wisely says "Make sure you trust the source of any MCP servers you use" [39]. Also be aware that each tool call consumes tokens and time. A misbehaving MCP tool could slow responses (e.g., if it tries a long database query). So add what you find genuinely beneficial to your workflow.

Bottom line: You can absolutely use Gemini Pro effectively **without touching MCP servers** beyond the built-in capabilities. MCP is an advanced extension to integrate additional context or actions. Given your described needs (professional Python dev, multiple envs, etc.), the most relevant one is the GitHub integration for issues/PRs, and perhaps a documentation retrieval tool if you want AI to always have the latest library docs. But if that's not a pain point, you can skip MCP setup and just feed needed info manually.

We will include some guidance in the setup section if you decide to enable the GitHub MCP, but it's optional. The free tier doesn't restrict MCP usage – it's available to you with the manual JSON config changes (the VS Code GUI doesn't yet have a one-click install for these servers as of writing, so it's a bit manual but doable) [84] [85].

Recommended Workflows for Python Development (Windows 10 + Conda)

Now let's tie everything together into concrete workflows and best practices for using Gemini Pro in your specific development setup:

Environment Setup and Activation in VS Code

Project Isolation with Conda: You have multiple projects, each with its own Conda env (no global Python). This is great for isolation, but means VS Code/Gemini need to be pointed to the right interpreter and environment for each project. Here's how to ensure that: - **VS Code Python Extension:** Although not explicitly about Gemini, it's highly recommended to install Microsoft's Python extension in VS Code. This will help you switch interpreters per workspace. Once installed, you can select the interpreter (Command Palette → "Python: Select Interpreter"). Choose the python.exe from your project's Conda env. This ensures that when you run code or tests from VS Code (outside Gemini), it uses the right Python. It also affects IntelliSense and debugging. - **Terminal Activation:** By default, the Python extension can auto-activate the env in the integrated terminal. Ensure that setting *"Terminal > Integrated > Activate Environment"* is on. Then, when you open a new terminal in VS Code (Ctrl+`), it should run the activate.bat for your environment. If this doesn't happen (sometimes with Conda it might not, depending on how VS Code is launched), you have options:

- Manually run your env's activate script in the terminal when you start work. Since you have custom .bat scripts for activation, just execute them (`project_env_activate.bat`) in the terminal. Once done, any shell commands from Gemini will inherit that environment.
- Or configure a VS Code **task** or **shell profile**. For example, in your workspace settings, you can add a custom shell profile:

```json
"terminal.integrated.profiles.windows": {
    "CondaEnv": {
        "path": "C:\\Windows\\System32\\cmd.exe",
        "args": ["/K", "C:\\path\\to\\your\\env_activate.bat"]
    }
}
```

Then `set "terminal.integrated.defaultProfile.windows": "CondaEnv"`. This will make every new terminal use your .bat on startup (the `/K` flag means run the command then stay open). You might set this per project.

- Alternatively, as you suggested, just call the .bat via Gemini's agent: e.g., when you start an agent session, tell it as first step: *"Activate the environment

by `running` path\to\activate.bat`”*. It will ask to run that command; you allow it, then the subsequent steps are in that env. This is a viable approach if you often forget to do it manually – you can incorporate env activation into the plan (“Step 1: activate env; Step 2: run tests; Step 3: …”).

Folder Structure and VS Code Workspace: Organize your VS Code workspace such that the project folder (with your code and an environment file like `environment.yml` or `requirements.txt`) is open. Gemini will treat the opened folder as the project context (for searching files, etc.). Keep only one major project open at a time to avoid confusion – the agent’s grep or list files could span whatever is open as the “workspace”. If you need multiple folders open, be specific in prompts about where to act.

Project metadata: Include a `README.md` or design docs in the repo. Gemini can utilize these for context. Also, if you have `requirements.txt` or `pyproject.toml`, Gemini can read them (on request) to understand your dependencies or frameworks used. This can help it generate code consistent with your stack (e.g., it sees Flask in requirements, it will tailor web examples to Flask).

Typical Development Tasks & Gemini Workflows

Let’s go through common Python development tasks and how to leverage Gemini for each:

1. Scaffolding a New Module or Feature

Scenario: You want to add a new feature to an existing project, which involves creating a couple of new modules and perhaps modifying some existing ones. You have a general spec (maybe in an issue or your head).

Approach: Use Agent Mode to handle the multi-file creation, but start by feeding in any specifications: - Open any relevant design notes or the issue description. If it’s an issue on GitHub, copy its content into a new scratch file or directly into the chat. - Start an Agent Mode session: *“Implement Feature X as described above. Plan the steps.”* The agent will propose a plan: e.g., create `feature_x.py`, update `main.py` to call it, add unit tests, etc. - Review the plan. Make sure it didn’t misunderstand (if the spec was in natural language, clarify any misinterpretation now). Perhaps it missed a step to update docs – you can add “Also update the README.” You can edit the plan in your message or instruct the agent to adjust. - Approve the plan. Now watch as the agent creates the files: - It might say “Creating `project/feature_x.py`…” and show the code. VS Code will either open the file or show a diff of a new file. Verify the content. If something is off, you can stop or intervene (you could even type “modify that to do Y” and it will revise). - Then it might modify `main.py` (showing the diff of the changes). - Then create `test_feature_x.py` with some tests (if it was part of plan). - If an external step is needed (e.g., run `pip install` for a new dependency), it might use the terminal. (Ensure env is on; if it tries without, you can cancel, activate env, then ask it to retry the step). - After execution, you have the code across files in place. At this point, you might switch out of agent mode and manually inspect/run things, or continue in agent mode to test. - **Testing**: You can prompt in agent mode, “Run the test suite.” Agent will execute `pytest` (or `python -m unittest`, depending on what it finds). You allow it to run. Suppose a test fails – the agent can capture the output and then likely propose a fix. It might go: “Test failed with XYZ error. I will fix the function accordingly.” You’ll get a diff for

the fix. Approve that. It can run tests again… and so on until green. This is essentially an AI-assisted TDD cycle. (It's good practice to let it write tests too, not just code, to ensure it understands the goal.)

- Once satisfied, you exit agent mode (toggle off) or just stop the agent session. Now you have your feature implemented.

This workflow shows how you can offload a chunk of the grunt work to the agent while you supervise the architecture and decisions.

Tip: If the feature is complex, consider doing it in parts. You could have the agent do step 1-3, then stop and review manually, then start a new agent session for steps 4-5. You can even incorporate manual edits in between and the next agent run will pick those up.

2. Refactoring and Code Improvements

Scenario: You have legacy code that needs refactoring (e.g., break a monolithic script into modules, or renaming a function across many files, or optimizing some logic).

Approach: Depending on scope, use agent mode or targeted commands: - For a broad refactor (many changes): Agent Mode with a plan: *"Refactor the Foo component: move database functions from foo.py into a new db.py, update all imports, and simplify foo.py's logic."* - The plan might detail the moves and changes. Approve, then it will perform multi-file edits (rename symbols, etc.). It can use `find_usages` and `grep` to ensure all references are updated [57]. For example, it might find all places `from foo import db_connect` was used and change them to `from db import db_connect`. - This is where the semantic understanding + project tools help a lot – it's doing what an IDE refactor or your own grep-sed would do, but more intelligently (it might also adjust documentation or comments mentioning the old structure). - For small localized refactor: Use the `/simplify` or `/fix` quick action on a selection. E.g., highlight a convoluted function, press Ctrl+I then choose `/simplify … this function`. It will propose a cleaner version in a diff. You accept it if it looks good. - For adding documentation: You can highlight a function or class and do `/doc this function` [86]. It will insert a docstring (likely following PEP257 style or whatever style it learned, often good enough). - For code style issues: If you see repetitive patterns or non-idiomatic code, you can ask in chat: *"Refactor the code in X.py to use list comprehensions instead of for loops where appropriate."* The agent can do that (either directly in chat output or via a diff with `/fix`). Another example: *"This function is too long, refactor into smaller functions."* The agent might create helper functions. Always review the result; but this can save time.

Testing after refactors: run your tests (possibly via agent). It might catch something missed. If you have continuous integration configured, use that as well.

One limitation in free mode: *Recitation (long explanations) is disabled in agent mode* [21]. If you want the agent to *explain* every change it's doing, that won't happen automatically – it will just do them. But you can ask it in normal chat, "Explain why these changes were made." Or in plan, it often comments each step anyway. This is fine for understanding.

3. Debugging and Bug Fixing

Scenario: A bug is reported (failing test or error at runtime). You need to locate and fix it.

Approach: - If you have an error message or stack trace, feed that to Gemini. E.g., *"We get KeyError 'user_id' on login. Here's the traceback: ... Why is this happening?"* In normal chat mode, it might analyze the traceback and guess the cause. It may cite known issues (if it's a known bug in a library, etc., it might even reference it). - In agent mode, you can be more action-oriented: *"Debug and fix the error when logging in as described in issue #22."* If you provided the issue or error output, the agent might plan: 1. Reproduce or locate the error in code; 2. Propose a code fix; 3. Run tests to verify. - It can search the code for the error string or likely problematic function using `grep` (for KeyError 'user_id', find where user_id is assumed to exist). - Once found, it presents the cause (perhaps "the code assumes user_id in session, but sometimes it's missing"). Then it will attempt a fix (like adding a check or default). - You approve changes and it applies them. - Then agent might run tests to confirm the bug is resolved (especially if you have a test case now for it). If you didn't have a test, it might even suggest adding one to prevent regressions (Gemini often encourages writing a test for the bug [87] – which is good practice). - Example from Real Python: They had a mutable default argument bug; the agent suggested a fix (use `None` default and assign empty list inside) and then guided the user to run tests to verify [87] [88].

- **Iterative fixing:** If the first fix is wrong or incomplete, you can continue the chat: "It still fails in this scenario" – the agent will refine. This interactive loop is where AI shines: it's like having a second pair of eyes to brainstorm possible causes and solutions.

- If the bug is complex (spanning logic across modules), agent mode plan is helpful to coordinate changes. If it's a one-liner fix, a quick action `/fix bug described below` could suffice.

- Always rerun the application or tests yourself after to ensure all is good. AI can miss edge cases, so you as the engineer make the final call.

4. Writing Tests and Running Them

AI can also help with testing: - **Generate tests:** "Write unit tests for the `calculate_discount` function." In chat, it will produce some test code (likely using `unittest` or `pytest`). You might need to wrap it properly into your test files. Or you can directly have agent mode create a new `test_*.py` file with the tests (if it's part of plan or by instructing it to write to that file). - If you have a continuous testing setup (pytest), you can integrate that with agent actions. As described, agent can run `pytest` and interpret results. It might not catch *logical* issues unless a test fails, so rely on tests to signal problems.

- **Property-based or fuzz tests:** You can even ask, "Generate more test cases for edge conditions of this function." It might come up with cases you didn't think of.

- **Test environment with Conda:** Ensure any needed dev dependencies (pytest, etc.) are in the env. If agent tries to run pytest and it's not installed, it will error. The agent might then say "pytest not found, should I install it?" – it could run `pip install pytest` if allowed. It's better you have it set up in advance or permit that step.

5. Automating Repetitive Tasks (scripts, CI, etc.)

Gemini can help create automation scripts: - **Build/Deployment Scripts:** e.g., *"Create a PowerShell script to set environment variables and launch the app."* It can write a `.ps1` or `.bat` file if you specify Windows context. (Be sure to mention if it should target PowerShell vs cmd, etc.) - **GitHub Actions workflow:** *"Write a*

GitHub Actions CI workflow for this project that runs tests on pushes and builds a Docker image." It can produce a YAML file (in chat or even directly create `.github/workflows/ci.yml`). You'll need to supply some specifics (like secrets or build steps), but it often knows typical Python workflow structure (set up Python, install deps, run pytest). This saves time reading syntax docs.

- **Documentation generation:** *"Generate Sphinx docs for the project."* It could create a conf.py and initial index.rst by analyzing your code. (Might need refinement, but a start.)

- **Project Setup Scripts:** *"Create a setup.py for packaging this project."* or *"Create a pyproject.toml for packaging."* It can draft those based on project metadata.

- **Refactoring tasks:** *"Find all TODO comments in the code and make a task list."* It could use grep for "TODO" and present a list. Or *"Remove all debug print statements"* – it can search and either remove or flag them for removal.

In these cases, agent mode can chain find-and-edit operations across the codebase systematically, which is like automating what you'd do with find/replace or multiple manual edits.

6. Architecture Understanding and Design

When joining a project or working on a large codebase, you can use Gemini to understand it: - Ask for a **summary of the project**: *"Summarize this repository's purpose and structure."* The agent might do a quick scan (list files, maybe open important ones) and give you a synopsis [89] [90]. RealPython's example did exactly this, summarizing a todo list CLI project with its key features in bullet points [89] [90]. - You can drill down: *"What are the responsibilities of module X?"*, *"How does the user authentication flow work? (you might need to read auth.py and routes.py)"*. The agent will gather context from those files and explain. - If you have an architecture diagram (textual or in description), feed that to ensure the assistant doesn't propose something against the intended design.

Gemini's large context means it can hold a lot of info, but practically, providing explicit context (like relevant files) in each query yields the best results, rather than expecting it to "already know" the whole repo at once.

7. Human Oversight and Best Practices

No matter the task, remember: - **Stay in charge of critical decisions**: The AI might suggest an approach that you know isn't ideal for your system (maybe not thread-safe, or doesn't handle an edge case). Use your expertise – adjust the plan or code. You can correct the AI: *"Don't use global variables, instead pass a config object."* It will adapt. - **Review code thoroughly:** Especially in security-sensitive or complex logic, read through what Gemini writes. It's good at syntactically correct code, but you ensure it semantically matches requirements. - **Use small iterations:** If unsure, do one step at a time with the agent. You can break a big ask into smaller ones. This also helps pinpoint if something goes wrong. - **Testing is your safety net:** Always run your test suite after major AI-generated changes. If you don't have tests for a section, consider writing some (you can even have Gemini help create those tests before it refactors, as a way to lock in expected behavior). - **Leverage documentation**: If Gemini cites a source or uses an API you're not familiar with, follow the link (they are given in that 【†】 format in the UI) to verify. This is especially relevant in normal chat mode responses (since agent mode won't cite). - **Escaping limits:** If you ask for something and Gemini says *"I cannot do that"* (maybe if it's something disallowed by policy, but coding tasks should be fine),

you might need to rephrase. For example, if it refuses to use certain libraries, clarify that you have the right to use them. Generally with Pro, this is rare unless you venture into disallowed content.

Now, let's delve into specifics of setting up and using these workflows on Windows 10 with your tools:

Scenarios: With vs Without CLI Tools

As touched on, you can accomplish most of the above entirely within VS Code. But for clarity, consider: - **Using only VS Code Extension (no CLI invocation):** This is the default scenario. You get a nice GUI, inline diffs, etc. This should cover you for interactive development. In this mode, the "agentic" features are available by toggling agent mode in the chat – you do not manually start any external processes aside from VS Code itself.

Example: You open VS Code, open the project folder, ensure conda env is active, and open Gemini chat. Whether you're doing Q&A, code generation, or a multi-step agent task, it all happens in that panel. If something requires running code, the extension will handle spawning a terminal instance in the background (you just see the output in chat or an output panel). When code is created or modified, VS Code shows those changes in the editor immediately.

- **Using Gemini CLI (standalone or via terminal) for tasks:** There might be times where jumping into the terminal is useful. For instance:
- If VS Code is running into memory issues with a huge context, maybe running the CLI with a specific flag or command could be a workaround (though unlikely, given VS Code uses the same model).
- If you want to integrate an AI step in an automated script: e.g., run `gemini` as part of a build process to auto-fix code formatting or do static analysis. That could be an advanced use.
- In case the VS Code extension crashes or agent mode is temporarily not working (since it's preview), the CLI is a fallback. You could continue your work in CLI until the IDE is restarted.
- Some connectors (MCP tools) might require a persistent process that could just as well be run via CLI outside VS Code, but that's more for tool developers.

Comparing experiences: - The **extension** provides a richer experience with minimal setup (just login with Google). It's integrated and contextually aware (e.g., it automatically knows which file you're editing). - The **CLI** requires you to navigate in your project and context manually a bit more (though it does detect git repos and might load some context files). - The **CLI** prints diffs and code to the console, which you then apply (it can auto-apply changes to files too, but you'll open them in an editor to see the changes anyway). - The CLI might be slightly more **scriptable**. For example, RealPython shows one could run `gemini cli` commands in a headless fashion to e.g. generate documentation and pipe it to a file.

Conclusion for daily usage: Stick to the VS Code extension for interactivity and convenience. Keep the CLI as a power tool for when you need to operate outside the VS Code UI or perhaps to integrate with other tools (for example, using Gemini in a Vim workflow or in a DevOps pipeline). Given that you prefer not to use CLI unless compelling, you probably won't need it in normal operation on Windows 10.

If you ever do need to run the CLI (e.g., testing it out), you won't break anything – it uses the same quotas as the agent mode (they share the allowance) [33] [34], and they both authenticate via your Google account. So you can mix and match in a day and it's fine (just be mindful both count toward the same daily limits).

Now, let's outline how to actually get everything set up and running step by step, specifically tuned to Windows 10 + Conda + VS Code:

Step-by-Step Setup Instructions (Windows 10, VS Code, Conda)

Follow these steps to configure your environment and Gemini for maximum productivity:

1. Install/Update VS Code – Ensure you have the latest VS Code on Windows 10. LTSC Windows 10 is fine; VS Code runs independently. Having the latest ensures compatibility with the Gemini extension.

2. Install Gemini Code Assist Extension – In VS Code, go to Extensions (Ctrl+Shift+X) and search "Gemini Code Assist". Install the official extension by Google [91]. Alternatively, download from VS Code marketplace. It's listed as free and has Google as publisher.

3. Sign in to Google – After installation, the extension will likely prompt you to sign in to a Google account to use Gemini. Sign in with the account that has Google One AI Premium (Gemini Pro). This will grant the extension access to the Gemini service using your included quota. If it doesn't prompt automatically, there might be a "Sign in to Gemini" command in the command palette – run that, which opens a browser for Google auth.

Troubleshooting: If after sign-in, the extension still acts like you're on a basic model, check the Google One subscription status. The support forum noted some initial bugs in recognizing AI Pro for Code Assist [92]. Typically, being signed in with the Pro account should unlock Gemini 2.5 Pro model (with large context). You can verify by asking in chat "What model are you?" – it might say Gemini 2.5. If there's an issue, updating the extension or logging out/in can help. (As of late 2025, these issues are mostly resolved, but just in case.)

4. (Optional) Enable Preview Features – If Agent Mode isn't readily visible: - Go to VS Code settings (`Ctrl+,`), search "Gemini release channel" or find **Gemini Code Assist: Use Pre-release features** [93]. Enable it if available. Or set the extension to use "Insiders" channel if that was how preview was delivered (the documentation [8] suggests an option to configure release channels). - After enabling, reload VS Code. You should see an "Agent" toggle in the Gemini chat UI or an "Agent" tab. - In latest versions, Agent Mode might already be enabled by default for everyone in preview, so this step might not be needed.

5. Set Up Conda in VS Code: - Install the **Python** extension (ms-python) if not already – it helps with env and runs. - Open your project folder in VS Code. When you open a Python file, you might get a prompt to choose interpreter. Select the Conda env's interpreter (you can browse to `<env>\python.exe`). - Open a new Terminal in VS Code. Check the prompt; if the Python extension auto-activated the env, you should see the env name in the prompt (or `(env)` prefix). If not, manually run your env's activate script (e.g., `conda activate <env>` or `.\env_activate.bat`). You should see environment variables like `PATH` updated. - Confirm by running `python --version` or `which python` (or `Get-Command python` in PowerShell). It should point to your env's Python and show the correct version. - This step ensures that if Gemini's agent calls any Python tool (like running the code or tests), it uses the correct interpreter and has access to installed packages.

6. Configure Gemini Settings (optional) – The extension likely created a folder `~/.gemini` (on Windows, that might be `C:\Users\YourName\.gemini`). In there is `settings.json` for Gemini. You can tweak: - **Tool permissions defaults:** If you want to pre-allow certain safe commands. For example, if you trust it to run tests anytime, you can add: `json "coreTools": ["ShellTool(pytest)", "ShellTool(pip install)"]` This pre-allows `pytest` and pip in shell (just examples). And maybe exclude any dangerous ones: `json "excludeTools": ["ShellTool(del)", "ShellTool(rm -rf)"]` The syntax was shown in docs [62] [94]. You can always just manually allow each time, which is safer if you're not sure.

- **MCP servers:** If you decided to integrate e.g. the GitHub MCP, this is where you add it. Insert the JSON as shown earlier under `"mcpServers"`. Make sure Node is installed so that `npx` works.
- Save the file. When you next start an agent session, those settings take effect. (You might not need to touch this file at all if you're fine clicking allow in the UI for each tool usage as prompted.)

7. Using Gemini in VS Code: - Click the Gemini icon on the Activity Bar to open the interface. Start a chat to verify it responds. - **Normal Q&A mode:** Ensure Agent toggle is off and ask something simple about your code or a coding question to see it works. You'll see a streaming response likely. If it cites sources, those appear as footnotes [20] – that's a sign you're in normal mode, since agent mode wouldn't cite. - **Agent Mode test:** Toggle Agent mode on (the interface might show "Agent" highlighted). Now try a simple command, like *"List all python files in this project."* The agent might attempt to call a tool (grep or list_files). It should ask permission ("Allow file search in directory X?"). Allow it. Then it will return the file list. This sanity check shows that tools are working. - Familiarize yourself with the Stop button (to halt the agent if it's going astray or too slow) and the ability to start a New chat (which resets context if needed).

8. Work on a task – Now pick a real task and go through the workflows we described. For example: - Open a file with a function and try the `/doc` command to generate a docstring. - Or ask in agent mode: *"Explain what this project does and suggest any improvements."* It might run some analysis and give a summary and maybe recommendations (like "there's duplicate code in X and Y"). - Try a moderate refactor via agent: e.g., *"Rename the function* `foo()` *to* `foo_bar()` *across the project and update its references."* The agent plan may use a combination of search and replace operations. Approve and watch it do multiple file edits. This verifies multi-file capability in a controlled way. Check that all references were updated (maybe have it run `pytest` if you have tests, to catch any missed rename).

If all goes well, you now have confidence in using Gemini for bigger features.

9. (Optional) CLI Installation – If you think you'll want to use the CLI: - Install Node.js (skip if you have it). - Run `npm install -g @google/gemini-cli` in an Administrator PowerShell/Command Prompt (so it can install globally). This pulls the CLI. - After, run `gemini --version` to confirm it's installed. Then `gemini login` to auth (it'll open browser, etc.). Once logged in, you can use it. - You can also integrate the CLI with VS Code's terminal. For instance, you could create a VS Code task or debug profile that launches a Gemini CLI session if you ever need that. But that's extra; you can always just Alt+Tab to a normal terminal.

10. Ensure MCP Tools (if configured) Work – If you set up any MCP server (like GitHub): - Test it out: e.g. in agent mode, try a command that uses it. For GitHub, you might say: *"List the titles of open issues in this repo."* The agent should then invoke the GitHub MCP tool (which will run `npx @modelcontextprotocol/server-github` in background if not already running, so there might be a slight delay initially). It will use your token to fetch issues via GitHub API. Then you should see the list. If it errors (maybe token mis-scoped

or server not found), debug the settings or ensure packages are installed. Once confirmed, you have extended capabilities.

Remember, if something isn't working (like agent mode not showing up, or the model seems limited), it could be due to needing an extension update or login issue. Check the VS Code Output panel for "Gemini" logs to see if any error (like auth failure or reaching quota). The extension also has a usage/quota display in the UI for how many requests left (they mentioned "user-friendly quota updates" added [27], so you might see counts somewhere).

By following the above steps, you set a robust foundation: VS Code is integrated with your Conda env and Gemini is ready to assist with high-level AI power.

Prompting Patterns and Best Practices

To get the most out of Gemini, consider these prompting strategies tailored for coding tasks:

1. Be Specific with Instructions: Clearly state what you want and any constraints. Instead of *"Add a feature for reports,"* say *"Add a feature to generate PDF reports of sales. Use the existing* `ReportGenerator` *class if possible, and ensure to include a unit test. The design is described in the attached spec."* This gives the model a concrete goal, relevant class to consider, and the requirement to include a test, increasing the chance it does all parts.

2. Use Step-by-Step for Complex Queries: Even outside full agent mode, you can phrase queries in steps. For example, in chat: *"First, outline the changes needed to implement X. Then we'll implement each."* Gemini will often follow this by listing steps. This is essentially coaxing it to produce a plan that you can manually verify (if you weren't using the built-in plan feature). After that, you could say *"Great, please execute step 1,"* though in normal chat it won't execute – so better to take that plan and either do it yourself or use agent mode for execution. But the key is, structured thinking helps both you and the AI.

3. Leverage Context Effectively: - If discussing code, *show the code* or at least the relevant portion. You can copy a function into the prompt if it's short, or instruct "open file X" to the agent. The assistant can only operate on what it sees. The VS Code context sharing covers open files up to a limit, but in a big project, it won't know everything simultaneously unless you direct it. - Keep only needed files open to avoid confusion (the extension might only send a subset of open file content due to token limits). - For multi-turn conversations, remember it has memory of prior turns (within context window). So you can reference "the function we discussed earlier" and it knows that, as long as it was in this chat.

4. Iterative Refinement: It's often productive to get an initial answer or code, then refine: - Ask Gemini to produce something without worrying too much about perfection (e.g., "draft an implementation of X"). - Then examine that output and follow up: *"Now optimize this part"* or *"This is good, but handle the error case Y as well."* The model will adjust the code accordingly. - This back-and-forth can converge to a solution. It's often faster than trying to stuff every requirement into one single prompt initially.

5. Controlling Tone and Depth: For documentation or explanation, you can specify style: *"Explain this code in a concise manner,"* or *"Add comments to this code explaining non-obvious logic."* It will then do so. If it's too verbose, you can say "shorter." If it's too high-level, ask for more detail.

6. When Agent Mode, when Chat Mode: - **Use Agent Mode** when you want *actions to be taken* (file edits, running code) or a plan for a multi-step change. It's best for development tasks that actually change the code or require looking things up in the project. - **Use normal Chat** when you want *information or analysis* without immediate action, especially if you'd like citations (e.g., asking about library usage, general debugging advice, etc.). Also use chat for brainstorming design approaches (since it can freely discuss and cite known patterns). - You can mix them in a session: maybe use chat mode to gather ideas or have it write a code snippet, then manually integrate that or use agent mode to apply a variation of it.

7. Provide Examples for Clarity: If you want code in a certain style or format, give a quick example. For instance, *"Write the output to a file (e.g., see how we do it in* `utils.py` *)."* The agent can then mimic that pattern from utils if you either provide it or it's able to open that file via context.

8. Keep Prompts Focused, but Context-Rich: A good prompt is one that **frames the problem, gives context, and states the task**. For example:

```
The project is a web app using Flask. We have a module auth.py for authentication.
    We currently store passwords in plain text (bad). Task: implement password hashing.
   Use werkzeug.security to hash passwords when storing, and verify on login.
   Modify auth.py accordingly, and any place that creates or checks users.
   Provide a brief description of changes.
```

This prompt: - Reminds the AI about the tech stack (Flask, which implies certain conventions). - States the problem (plain text passwords) and desired solution (hash them). - Mentions the preferred library (werkzeug.security). - Tells what file to focus on (auth.py) and indicates other places might need changes (user creation, login). - Asks for a description of changes (so I get a summary in the response alongside code).

In agent mode, this might produce a plan like: 1) Import werkzeug.security in auth.py, 2) Update register function to hash password, 3) Update login to verify hash, 4) Ensure any tests updated, 5) Explain changes. Then do it. That's exactly what we want.

9. Handling Large Outputs: If expecting a very large output (like a long code file or many changes), Gemini might summarize or stop. In normal chat, it might cut off if the code is too long. In agent mode, it handles it by doing actual file writes in chunks. If using normal mode for multi-file, prefer to do it file by file, or ask for a high-level outline instead of full code in one go. - If a response is cut off, you can usually type "Continue" and it will resume from where it left off. But in coding, a cut-off can break syntax. So agent mode is safer for multi-file or long code generation because it uses diff insertions rather than dumping everything in text.

10. Multi-step Confirmation: If you are unsure about the agent's plan, don't hesitate to say, *"Before coding, let's discuss the plan:"* and the agent will break out of execution to chat with you about it. Agent Mode specifically allows you to fine-tune the plan pre-approval [13]. Use that to your advantage – it's easier to correct course at the plan stage than after code is written everywhere.

11. Utilize Checkpoints/Revert if needed: The Code Assist tool has a concept of checkpoints (the blog mentions "revert to checkpoints" [95] now GA). This means after agent makes a bunch of changes, you can rollback easily if something went wrong. Familiarize yourself where that is (likely in the VS Code UI, maybe in the timeline or history of the chat). In any case, keep version control snapshots (commit before major AI changes) so you can diff or revert via Git if needed. The AI might not always use best judgment, and undoing is part of a safe workflow.

12. Don't be afraid to correct the AI: If it does something wrong, you can say *"That's not correct because..., please fix it by doing...."* The model will incorporate your feedback and adjust. It doesn't get stuck or offended – it will treat that as additional instruction. This iterative correction is often faster than if you manually fixed and didn't tell it (though if it's a one-line fix, manual is fine; but if the AI misunderstood a requirement, clarify and let it handle the adjustments broadly).

By following these practices, you guide Gemini to be a helpful collaborator rather than a tool that you fight with. The idea is to **communicate with it almost like a junior developer**: give clear requirements, review their "pull request" (the AI's changes), ask for improvements or explain decisions, and gradually arrive at the desired outcome.

Examples of Gemini Pro in Action

Let's walk through a few **practical examples** demonstrating agentic workflows and prompt patterns. These examples will illustrate end-to-end how you might interact with Gemini for different tasks, given the features and environment we've discussed.

Example 1: Multi-Module Feature Addition with Plan & Approve

Task: Add a "discount code" feature to an e-commerce app (as mentioned earlier). This requires changes in multiple files (Cart model, Checkout logic, UI template, plus a new function to validate codes). We have a GitHub issue #101 with details and acceptance criteria.

Steps:

1. **Gather context:** Open the `Cart.py`, `Checkout.py` in VS Code. Also open the issue #101 content (copy it into a file or the prompt). The issue says, for instance: *"Allow users to enter a discount code at checkout to get a percentage off. Codes stored in DB table* `discounts`*. Apply in Cart total. Provide feedback if code invalid/expired."*

2. **Prompt (Agent Mode):** ``` [Issue #101] Feature: Discount Codes for Cart Checkout.

3. Users can apply a discount code for a percentage off their total.
4. Discount codes are stored in database (see discounts table) with fields: code, percent_off, expires.
5. Validate code on checkout: if valid and not expired, apply percent off to Cart total.
6. Show discounted total in the UI and indicate savings.

7. If invalid, show error message. Plan the required code changes across the system to implement this. ``` *(This prompt includes the issue details and explicitly asks for a plan.)*

8. **Gemini Responds with Plan:** For example: ``` Plan:

9. **Model Update** - Update `Cart.py` (Cart model) to hold an applied_discount (code or percent). Add methods to apply a discount to total.

10. **Discount Validation** - Create a new module or function, e.g., `discounts.py`, with `validate_code(code)` that checks the DB (or a stub) for code validity and returns percent_off.
11. **Checkout Logic** - Modify `Checkout.py` (checkout process) to accept a discount code input, use `validate_code`, and if valid, apply to cart total via Cart's method. Handle invalid code (maybe raise exception or flag).
12. **UI Template** - Modify `checkout.html` to add an input field for discount code and display the discounted total if a code is applied, or an error if invalid.
13. **Database Access** - If there's an existing database utility, use it to fetch discount info. If not, simulate with a dictionary (since full DB integration may be out of scope here).
14. **Testing** - Update or add unit tests:
    - Test `validate_code` logic (valid vs invalid vs expired).
    - Test Cart total calculation with a discount.
    - Test checkout flow with valid and invalid code. After implementing, run tests to ensure everything passes. ``` *(The plan is detailed and touches all relevant parts.)*
15. **Review Plan:** You notice it suggests a new `discounts.py` for validation – which is fine. It didn't mention where the error message logic lives (probably in template or passed from backend). You might accept this as is, or tweak:
16. You: "Add: in plan step 3, also handle showing error message to user if invalid (perhaps via flash message or template context)."
17. Gemini: "Acknowledged – will include error message handling via Flask flash in the checkout logic." (if this is a Flask app).
18. **Approve Plan:** Click *Approve.* Now agent executes:
19. **Step 1:** Opens `Cart.py`. It adds something like: `python`
    ```
    class Cart:
        # ...
        applied_discount = None
        def apply_discount(self, percent):
            self.applied_discount = percent
        def total_with_discount(self):
            total = sum(item.price for item in self.items)
            if self.applied_discount:
                total *= (100 - self.applied_discount) / 100
            return total
    ```
    It shows this diff. You allow it to write the file.
20. **Step 2:** Creates `discounts.py`. Code might be: `python`
    ```
    from datetime import datetime
    DISCOUNTS = {"SAVE10": {"percent": 10, "expires": "2025-12-31"}}
    def validate_code(code):
        data = DISCOUNTS.get(code)
        if not data:
            return None
    ```

```
        if datetime.now() > datetime.fromisoformat(data["expires"]):
            return None
        return data["percent"]
```
(This is a stub using a dict for example; in a real app, it might do a DB query. The agent assumed something simple.) It adds this file to project.

21. **Step 3:** Modifies `Checkout.py`. Possibly it's a Flask route handling POST from checkout form:
```
python
    import discounts
    # ...
    code = request.form.get('discount_code')
    percent = None
    if code:
        percent = discounts.validate_code(code)
        if percent:
            cart.apply_discount(percent)
        else:
            flash("Invalid or expired discount code.", "error")
    # ... proceed to calculate totals
    total = cart.total_with_discount() if percent else cart.total()
```
It shows a diff with these additions. You inspect the logic, looks okay. Approve changes.
22. **Step 4:** Modifies `checkout.html`. It adds:
```
html
    <form method="POST">
      <!-- existing fields -->
      <div>
        <label for="code">Discount Code:</label>
        <input type="text" name="discount_code" id="code">
        {% if flash('error') %}<p class="error">{{ flash('error') }}</p>{%
endif %}
      </div>
      <p>Total: {{ cart.total_with_discount() if cart.applied_discount else
cart.total() }}</p>
      <!-- ... -->
    </form>
```
(Assuming the template knows `cart` from context). If the syntax is slightly off, that's okay, you or tests can adjust. Approve it.
23. **Step 5:** It might skip if not needed to explicitly show (the validate_code already kind of covers DB access in stub form).

24. **Step 6:** Creates/updates tests. Possibly it creates `test_discounts.py`:
```
python
    import discounts
    def test_validate_code_valid():
        assert discounts.validate_code("SAVE10") == 10
    def test_validate_code_invalid():
        assert discounts.validate_code("NOSUCH") is None
    def test_validate_code_expired():
        # set an expired code in DISCOUNTS and test
        discounts.DISCOUNTS["OLD"] = {"percent":5, "expires":"2020-01-01"}
        assert discounts.validate_code("OLD") is None
```
Updates `test_cart.py` to test discount application:
```
python
```

```
    def test_cart_discount():
        cart = Cart([Item("A", price=100)])
        cart.apply_discount(10)
        assert abs(cart.total_with_discount() - 90.0) < 1e-6
```
And maybe a test for checkout logic if structured (though that might be integration test which is harder). It at least hints at tests. Approve new test files.

25. After making changes, Gemini might automatically run tests (since it was in plan, or it decides to verify). Let's say it runs `pytest`. *Terminal output:* 1 test failed – maybe `test_checkout` failed because the template context wasn't set properly or flash logic needed adjusting. The agent reads that output.

26. Gemini: *"Test X failed:* `cart` *has no attribute applied_discount in template (meaning context issue). Fix: pass cart to template context properly or adjust template logic."* It then fixes `Checkout.py` to always pass the cart (with discount applied) into `render_template("checkout.html", cart=cart)`. It asks to approve that change. Approve.

27. Re-runs tests. Now all pass.

28. **Finish:** The agent says "Feature implemented and all tests passing." It might provide a summary of changes:

29. *"Added discount application in Cart and Checkout, created new discounts module for code validation, updated UI and tests* [4] *. Users can now enter a code at checkout to receive a discount, or see an error if invalid. All tests (new and existing) pass, confirming the functionality."* (This final summary might not cite sources since agent mode doesn't recite docs, but it gives you a nice recap.)

30. **Post-checks:** You manually run the app to do a quick integration test. It works, or you do minor tweaks if needed (maybe styling of error message).

31. **Commit changes:** Use Git to commit. Optionally, use the PR review app or ask Gemini (in normal chat) to review the changes for any issues. For example, in normal chat mode, you could paste the diff or a summary and ask "Is there any potential problem with this implementation?" It might point out something like "Be careful storing codes client-side" or such, if any. But assuming it's fine, proceed.

This example showed how agent mode seamlessly handled a multi-file coordinated update, complete with tests – all within **free Gemini Pro** capabilities. We leveraged: - Structured prompt (with issue context). - Plan/approve workflow [13] . - Built-in tools (file edits, etc.). - Our Conda environment (ensured tests ran with correct env). - We didn't have to use CLI or any subscription-only feature.

Example 2: Automated Bug Fix with Testing

Task: There's a known bug: whenever a new user registers without specifying an optional profile field, an exception is thrown. We have a failing test (`test_register_no_profile` fails). We want to fix it.

Steps:

1. **Context:** We open `user.py` (which has User model or registration logic) and see where it might be assuming profile data. The test failure says KeyError 'profile'.
2. **Prompt (Chat mode or Agent mode):** For demonstration, try an interactive approach:
3. In **chat mode** first: "We have a bug when registering users without a profile. Here's the traceback: `KeyError: 'profile'` in `user.py line 45`. What could be the cause?" (We paste a snippet around line 45 if possible).
4. *Gemini (chat)* might answer: "Likely the code assumes a 'profile' key in data. If profile is optional, the code should check before using it. Perhaps a missing default value. The fix is to only access 'profile' if it exists or provide a default." (This is conceptual, maybe citing a Python doc about dict.get).
5. Now, we switch to **Agent Mode** to implement: "Fix the KeyError 'profile' issue in user registration. Ensure that if profile info is missing, the code handles it gracefully (e.g., use default or skip). Then run tests to confirm fix."
6. *Gemini (Agent)*: Will plan perhaps one step (since it's a small fix): "Modify `user.py` at the location of profile usage to safely handle missing 'profile' key." You approve (or it might not even need a separate plan due to brevity).
7. It shows diff for `user.py`, e.g. changing: `python profile_data = data['profile']  # old` to: `python profile_data = data.get('profile', None)` and later ensures code doesn't break if profile_data is None (maybe adds an if).
8. Approve change.
9. It runs `pytest` on user tests. They pass now.
10. Gemini: "All tests passed. The KeyError is resolved by using dict.get for 'profile' [90] [96]. No profile defaults to None, avoiding the exception."
11. **Verification:** You double-check that skipping profile doesn't introduce other issues (maybe ensure that None is handled properly downstream). All good.

This was a quick cycle: used chat to diagnose, agent to fix & test. The example illustrates how even for small tasks, the agent mode can directly apply the patch and test it, saving you time.

Example 3: Incorporating GitHub Issue Context

Task: Implement a feature per a GitHub issue that has a detailed spec – e.g., adding an **export to CSV** function for data, described in issue #50 with exact requirements.

Steps:

1. **Grab the issue content:** Copy the text from GH issue #50 (it has, say, acceptance criteria and an example of CSV format required).
2. **Use Agent Mode with context file:** Create a temp Markdown file `issue50.md` in VS Code, paste the issue description there. Perhaps also note which module might be affected (or the issue might say "add new command to export").

3. **Prompt:** “Refer to the requirements in **issue50.md** (opened in editor). Implement the CSV export feature accordingly. Provide any new code in a new module if needed, and update existing code to add an ‘export’ option. Outline steps first.”
4. We explicitly mention the issue file, trusting that agent mode context will include it [16] [65] . If not confident, we could paste it directly in prompt, but let’s assume context works.
5. **Gemini reads issue50.md** (via context or it might use read_file tool implicitly) and generates a plan: ``` Plan:
6. Create `exporter.py` with a function `export_to_csv(data: List[Record], filepath)` that writes data to CSV in the format specified (columns X, Y, Z).
7. Modify `cli.py` (the command-line interface module) to add a new command or option `--export` that calls `export_to_csv`.
8. Ensure that when export is invoked, it retrieves the current data (from database or in-memory) and passes to exporter.
9. Write unit tests for `export_to_csv` (e.g., given sample data, it produces correct CSV output).
10. Update documentation (README or help text) to mention the new export feature. ```
11. **Execute Plan:** Approve. It creates `exporter.py` with the CSV writing code (using Python’s csv module, presumably). Modifies `cli.py` to parse `--export` argument and call the function. Creates `test_exporter.py` with some test records and expected CSV content to assert correctness.
12. **Run tests:** If the project has a test suite, run it. If not, at least run `test_exporter.py` to verify output matches expectation.
13. **Finish:** Possibly in chat mode ask, “Summarize how this implementation meets the issue requirements.” Gemini might bullet-point: *“Exports all fields including X, uses ';' delimiter as specified, handles newline in field by quoting, etc.”* You confirm it matches issue details.

Here we relied on issue context. If we had the GitHub MCP set up, an alternate approach: - We could prompt: “Gemini, fetch issue #50 from GitHub and implement it.” With MCP, it would do something like call `github.get_issue(repo="myrepo", number=50)` behind scenes to get text, then proceed. This removes manual copy-paste. Since we assumed possibly no MCP, we did it manually via a markdown.

Point: You can treat design docs, issue text, etc., as guiding specs in your prompt. The clearer and more structured they are, the better output. Gemini will effectively follow that spec step by step (as seen in plan).

Example 4: CI Pipeline Creation (DevOps Automation)

Task: Set up a GitHub Actions workflow to run tests and linters on each push.

Steps: 1. **Prompt (Chat mode):** “Generate a GitHub Actions YAML workflow for CI that: on every push, sets up Python 3.11, installs dependencies from requirements.txt, runs pytest, and runs flake8 linter. Use Ubuntu latest runners.” 2. **Gemini (chat)** outputs a full YAML: `yaml`

```
name: CI
on: [push]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
```

```
        - uses: actions/setup-python@v4
          with:
            python-version: '3.11'
        - name: Install dependencies
          run: pip install -r requirements.txt
        - name: Run tests
          run: pytest
        - name: Run linter
          run: pip install flake8 && flake8 .
```

It might include flake8 installation or assume it's in requirements. We can adjust. 3. Copy that content into `.github/workflows/ci.yml` in our repo. (Or we could ask agent mode to create the file directly, but typing it ourselves is fine too, or we could have given a file path in prompt to instruct an edit.)

1. Perhaps ask in chat: "Explain each step briefly." It might add comments or just list out what each does (for our understanding/documentation).

This example shows using Gemini for something adjacent to code – writing config/CI script. It's straightforward and doesn't require agent mode because we just need the content. We could have agent mode create the file but that's a single file creation which is easy to just copy.

Example 5: Code Review and Improvement

Task: You wrote some code manually and want an AI second-opinion or improvements.

Scenario: You have a module `data_cleaner.py` you suspect could be refactored or might have edge-case bugs.

Approach: Use normal chat for review: - Open `data_cleaner.py`, copy its content into the chat prompt (if it's not too large, or at least the parts of interest). - Ask: "Review the above code for any potential bugs or improvements. Ensure it handles empty inputs and unusual characters properly." - *Gemini (chat)* will analyze and might point out, for example, "Function X does not handle empty list (would return None). Also, Y could be optimized by using a set instead of list for membership checks [89] [90]. Consider adding input validation for Z." It might even cite a best practice or two. - Then you can either implement those changes yourself or engage agent mode: "Apply those improvements to data_cleaner.py." The agent could then do them. - Or you ask specifically: "Refactor the `clean_names` function as suggested (use set for lookup and strip whitespace properly)." - The agent returns a diff for that function, you approve.

Using chat for review is like having a code linter + senior dev suggesting things. It's particularly valuable if you're unsure about a section or want to see if AI catches something you missed. Keep in mind, it's not guaranteed to find all issues, but it often finds common mistakes or inefficiencies.

These examples illustrate a range of uses – from implementing new features with multi-step plans to quick one-off fixes, leveraging context from external specs, and even non-coding tasks like CI config or code reviews. In each case, **Gemini Pro's free capabilities were sufficient**. We did not need any subscription-only feature: multi-file editing, test execution, and tool use all worked, citing sources was available in

explanatory mode, and the massive context window allowed reading and following spec documents easily [2].

Actionable Recommendations (Next Steps)

To conclude, here are some **concrete next steps and tips** to apply this knowledge in your development workflow:

- **Configure Your Environment:** Apply the setup instructions to get VS Code, the Gemini extension, and your Conda environments working together. Verify that you can run a simple agent task and that your Python env is recognized (this might involve adjusting VS Code settings for integrated terminal as noted). This is foundational – do this before using Gemini on a critical project.

- **Start with a Pilot Project:** Choose a small or medium non-mission-critical project or a new feature as a pilot to get comfortable. Perhaps an open-source project or an internal tool where you can afford some experimentation. Use Gemini extensively on this – generate some code, run agent mode for tasks, etc., to build intuition about its responses and quirks.

- **Leverage Documentation & Learning Resources:** Read through Google's official docs for Gemini Code Assist (especially any sections on VS Code usage, keyboard shortcuts, etc. that we referenced) to uncover features like keyboard shortcuts (`Ctrl+I`) and any new updates (the tool is evolving fast). For instance, see *"Code features overview"* and *"Chat features overview"* in the Cloud documentation to ensure you're aware of all commands [97] [98].

- **Incorporate into Daily Workflow:** Identify repetitive or time-consuming tasks in your daily work that Gemini could help with. Some suggestions:

- Use it as a **glorified Stack Overflow**: ask design and API questions instead of manual web searches.
- Use it for **test generation** when writing new code (to quickly scaffold tests).
- Try it for **code review** of your commits before pushing – catch things early.

- When planning a new feature, maybe have Gemini draft a design or outline based on your description, which you then refine (it can provide a starting point for your thought process).

- **Implement MCP Tools if Needed:** If you find yourself repeatedly copying info from external sources (issues, docs, etc.), consider setting up the relevant MCP connector (e.g., GitHub). This will streamline context gathering. For now, maybe just set up the GitHub one on a trial repository to see how it works (since it requires a token, perhaps use a test token or a dummy repo first to get the hang of commands like listing issues or creating an issue via the agent). If it proves valuable and stable, integrate it into your main workflow with caution and proper tokens.

- **Monitor Usage and Limits:** Keep an eye on how many requests you use per day. The extension might show this, or you can view it via Google's quota page [33]. If you find yourself hitting limits (1000+ agent actions or 240+ chat messages) regularly, you might consider upgrading to a Developer Program premium or similar [99] [100] – but for most, the free quotas suffice. If you do upgrade (Google AI Pro/Ultra, etc.), you might get immediate access to Gemini 3 which could further

improve quality, so it's worth knowing what higher tiers offer, even though you can operate fine without.

- **Stay Updated on Features:** Gemini Code Assist is evolving. Features like Agent Mode were preview and became stable quickly [3] [55]. New tools or integrations (like deeper IDE integration or more MCP servers) are likely to come. Keep an eye on release notes [101] [102] or the Google Developers blog for announcements (e.g., the article we cited [103] [104] was an Aug 2025 update). This ensures you benefit from improvements (e.g., if they allow even larger context or new modalities in the future).

- **Security & Privacy Considerations:** Since you're dealing with code (possibly proprietary), ensure you're comfortable with Google's data usage policies [105]. By default, code you send might be used to improve the model (anonymously) unless you're on enterprise with opt-out. If this is a concern, check if there is a setting (some tools allow opting out of data collection). Also avoid sharing truly sensitive credentials or data with the AI. It's fine with code, but treat it as you would treat an engineer who just joined – trust it with code, but maybe not with production secrets. Use the `excludeTools` setting to prevent any file from being read if it contains secrets.

- **Feedback and Control:** As you use Gemini, note situations where it doesn't perform as expected – ambiguous prompts, certain refactorings it struggles with, etc. Use those as learning: maybe your prompt can be refined. Also consider providing feedback through the tool (they often have a feedback mechanism in the extension) – this can help improve it. On your side, adapt your strategies from lessons learned; you'll get a feel for what tasks it excels at vs. which to do manually.

- **Team Integration (if relevant):** If you work with a team, share your findings and maybe onboard others to use Gemini (since it's free for individuals, it's easy to adopt). Having a team use it means you can collaborate on how to phrase prompts or even pair-program with the AI in the loop. If you're in a code review with a colleague, you can collectively ask Gemini for a second opinion on a tricky piece of code.

- **Consider the CLI for Scripting:** After you're comfortable in VS Code, think about whether any of your project maintenance tasks could be automated with the CLI. For example, a nightly script that uses `gemini` to analyze the codebase for any code smells or to auto-generate a report of TODOs. This is optional, but could be a neat use of the technology beyond the editor. RealPython's tutorial [106] [107] can give ideas – they showcase understanding legacy code and catching bugs with CLI usage.

- **MCP Extensions Possibilities:** If you have unique tools or data sources, you could even write a custom MCP server (the standard is open [108]). For instance, if you have an internal API or data source, integrating it means the AI can fetch data from it during a session. This is an advanced step and not needed for initial success, but keep it in mind if a use-case arises (like querying a custom knowledge base on demand).

In summary, you are well-positioned to use **Gemini Pro in VS Code on Windows 10** as a powerful AI coding assistant without any additional paid features. Start integrating it into your workflow gradually, using the guides above as reference. With practice, you'll find it can significantly accelerate tasks like writing boilerplate, refactoring large swathes of code, and maintaining high code quality through automated

checks and suggestions [109] [12] . Treat it as a collaborator – one that can handle tedious parts quickly – and use your engineering judgment to steer it. By doing so, you'll **boost your productivity** while retaining full control over your projects, fulfilling the goal of *"AI-first coding in your natural language"* with Gemini Code Assist [110] .

---

[1] [5] [8] [11] [26] [29] [33] [34] [35] [36] [37] [68] [99] [100] [110] Gemini Code Assist | AI coding assistant
https://codeassist.google/

[2] [30] [69] Gemini 2.5: Our newest Gemini model with thinking
https://blog.google/technology/google-deepmind/gemini-model-thinking-updates-march-2025/

[3] [4] [12] [27] [28] [49] [51] [52] [55] [95] [103] [104] [109] What's new in Gemini Code Assist - Google Developers Blog
https://developers.googleblog.com/new-in-gemini-code-assist/

[6] [66] [70] [79] [80] [81] MCP Registry · GitHub
https://github.com/mcp

[7] [13] [14] [16] [17] [18] [19] [21] [22] [23] [24] [47] [48] [50] [53] [54] [56] [57] [58] [61] [65] [101] [102] Agent mode overview | Gemini for Google Cloud | Google Cloud Documentation
https://docs.cloud.google.com/gemini/docs/codeassist/agent-mode

[9] [38] [39] [40] [41] [45] [46] [62] [63] [64] [67] [76] [82] [83] [84] [85] [94] Use the Gemini Code Assist agent mode | Gemini for Google Cloud | Google Cloud Documentation
https://docs.cloud.google.com/gemini/docs/codeassist/use-agentic-chat-pair-programmer

[10] [31] [32] [42] [43] [44] [86] [93] [97] [98] Code with Gemini Code Assist Standard and Enterprise | Gemini for Google Cloud | Google Cloud Documentation
https://docs.cloud.google.com/gemini/docs/codeassist/write-code-gemini

[15] [25] [59] [60] [71] [75] [87] [88] [89] [90] [96] [106] [107] How to Use Google's Gemini CLI for AI Code Assistance – Real Python
https://realpython.com/how-to-use-gemini-cli/

[20] [91] [105] Gemini Code Assist - Visual Studio Marketplace
https://marketplace.visualstudio.com/items?itemName=Google.geminicodeassist

[72] How to Install & Use Gemini CLI + MCP: A Step-by-Step Tutorial
https://www.youtube.com/watch?v=we2HwLyKYEg

[73] How to Install Gemini CLI on Windows | Complete Guide 2025
https://www.gemini-cli.blog/docs/install/how-to-install-gemini-cli-on-windows

[74] mingw-w64-x86_64-gemini-cli - MSYS2 Packages
https://packages.msys2.org/packages/mingw-w64-x86_64-gemini-cli

[77] Model Context Protocol - Wikipedia
https://en.wikipedia.org/wiki/Model_Context_Protocol

[78] [108] Model Context Protocol - GitHub
https://github.com/modelcontextprotocol

[92] Will AI Pro Subscription give Gemini 3 in VSCode? : r/GeminiAI
https://www.reddit.com/r/GeminiAI/comments/1p3pvx5/will_ai_pro_subscription_give_gemini_3_in_vscode/