Michael's Brief: Agentic AI Programming for Python Developers

1. Can you start by telling us a bit about your background and how you got into working with AI and large language models (LLMs)?

Matt Makai brings a wealth of experience as a developer and community leader. He's currently the VP of Developer Relations at DigitalOcean, and before that he led DevRel and Developer Experience at LaunchDarkly and AssemblyAI. Many Python developers know Matt as the creator of **Full Stack Python**, an open-source book/site for Python learning, and he's also behind a project called Plushcap. Earlier in his career, Matt spent 9+ years at Twilio, where he built up Developer Evangelism and content programs. This background shows he's deeply involved in helping developers succeed, which naturally led him to explore new tools that can boost developer productivity – like AI programming assistants.

Matt's journey with AI really took off around 2020 when GPT-3's private beta came out. Experimenting early with GPT-3 gave him a sense of LLMs' potential. Over the last couple of years, he has worked extensively with both cloud AI APIs (such as OpenAI and Anthropic's offerings) and local LLMs. For example, he's used **Anthropic's Claude** via APIs and also local models with tools like **Ollama**. This hands-on exploration helped him see how AI could move from toy examples to being useful in real-world coding. He mentions starting with simple scripts and using an editor called Cursor to integrate AI into coding. But one observation he had was a **"gap between toy projects and using LLMs in established codebases"** – meaning it was non-trivial to jump from fun demos to leveraging AI in large, production-quality Python projects. This realization set the stage for his current focus on **Agentic AI programming**, where he's figuring out how to effectively use AI agents in day-to-day development.

Fast-forward to today (mid-2025), Matt spends the majority of his coding time using **Claude Code**, Anthropic's AI coding assistant. He's integrated Claude into his natural workflow – using it in a terminal/tmux + Vim environment, which suggests he prefers minimal, text-based tools. We'll definitely dig into how that works. His background shows a pattern: he's always adopting tools that help developers (like Twilio for comms, LaunchDarkly for feature flags, etc.), and now AI is the new frontier he's passionate about. This first question sets the context so listeners know who Matt is and why he's a great person to talk about AI programming.

2. For those new to the term, what exactly is "agentic AI programming"? How would you describe it in simple terms?

Agentic AI programming refers to using AI as an active agent in the software development process, rather than just a passive assistant. In simple terms, you can think of the developer as the architect and the AI as the builder. The human specifies what needs to be done (the high-level design or goals), and the AI agent takes on the task of writing code, making changes, running tests, and even debugging to fulfill those goals. It's like you sketch out a blueprint of a house, and the AI is the construction crew that actually builds it. The AI agent doesn't just suggest one-line code completions; it can perform multi-step tasks autonomously (or semi-autonomously),

hence the term “agentic.” It **plans and executes**actions in the codebase.

An example of agentic coding: on the left, a human’s high-level design (in a markdown file), and on the right, an AI agent (Claude) autonomously writing and editing code based on that design. The developer focuses on the architecture and requirements, while the AI handles the detailed implementation and repetitive edits.

In practical terms, with agentic AI programming you might do something like: tell the AI, “Implement feature X following this design,” or “Refactor this module to improve performance,” and the AI will generate the necessary code across multiple files, maybe run the code or tests to verify, and iterate if needed. This is a step beyond tools like Copilot which autocomplete as you type. **Agentic AI** implies the AI has a degree of agency – it can decide to create new files, modify multiple functions, run build or test commands, etc., under some supervision. As one developer nicely put it, *“you focus on the creative, high-level design while AI handles all the tedious implementation details.”* In summary, agentic AI programming means leveraging AI agents to offload the grunt work of coding (boilerplate, plumbing, minor fixes) so that the human can concentrate on design, architecture, and the tricky logic.

3. How does this agentic approach differ from using something like GitHub Copilot or a simple code assistant in your editor?

This is a great question because many Python developers have experience with Copilot or code suggestions, but **agentic AI** is a different paradigm. Traditional code assistants (like Copilot) are *passive*: they react to what you’re doing in real-time by suggesting the next line or a small snippet. They’re essentially advanced autocomplete. In contrast, an **AI agent**is *active* and goal-driven. With an agent, you could say, “Hey AI, go through my project and create unit tests for all functions,” or “Optimize this query and update the code wherever necessary,” and the agent will attempt to carry out the request, not just suggest code but actually modify the project (often after presenting a plan or asking for confirmation).

For example, Copilot might help you write a single function faster, but an AI agent could analyze your entire codebase to find performance bottlenecks or fix errors. One anecdote from a developer using Claude Code really highlights this difference: he had a file with 38 mypy type errors and literally **“turned it over to the agent”**. The AI went through and fixed all those typing issues across the file while he took a 15-minute break. When he came back, the agent explained the changes it made, and after a brief review (he even debated one change with the AI), the file was clean – mypy passed with no errors. That kind of autonomous multi-step fix is something Copilot alone wouldn’t do; it would require you to prompt and accept each change manually.

Another key difference is **planning and understanding context**. Agentic AI systems often have a “planning” stage: they can summarize their approach before executing. They maintain a larger context of the whole project (especially models like Claude with 100K token windows). Copilot, by design, has a more limited context (usually just the file and a bit around it, or at most some open tabs). Agentic AIs can ingest your entire repository or documentation to make decisions. In essence, with agentic programming the AI behaves more like a junior developer that you delegate tasks to, whereas Copilot is like an ever-present pair programmer whispering suggestions in your ear as you type.

Finally, agentic systems can use **tools** and perform actions like running tests or searching the code. They might follow a chain-of-thought approach: decide what to do, carry it out, verify the result, all within one session. Copilot doesn't run code or check its work – it's up to the developer to do that. So, the agentic approach involves a tighter feedback loop where the AI can validate and correct its output (with things like tests or debug output) before finalizing changes. This often results in a higher level of autonomy. Of course, with more autonomy comes the need for more oversight – you don't want an AI agent blindly refactoring your code without review – but when used carefully, it can massively boost productivity by tackling the tedious or large-scale tasks that would be boring or time-consuming for a human.

4. What are some examples of agentic AI tools or frameworks out there, especially for Python developers? What does the broader ecosystem look like?

The ecosystem for AI agents and agentic frameworks is evolving quickly, and Python developers actually have a lot of choices right now. On a high level, there are a few categories to be aware of:

- **AI Coding Assistants with Agentic Features:** These are tools focused on writing and modifying code. For instance, **Anthropic's Claude Code** (which Matt uses) is one. OpenAI has been integrating some agent-like capabilities in its Codex/ChatGPT (like the *"Code Interpreter"* or advanced plugins that can run code). There are also open source projects like **Cursor** (an editor that integrates AI agents into your coding workflow) and **OpenCode**, **GooseAI**, **Devin.ai**, etc., which aim to provide an environment where an AI can edit code, use tools, and even commit changes.
- **LLM Agent Frameworks:** These are libraries to build AI agents that can do various tasks (not only coding). **LangChain** is a popular one in Python; it provides a framework for connecting LLMs to tools (like web search, calculators, Python REPLs) using a concept called ReAct (Reason+Act). With LangChain, you can, for example, create an agent that reads documentation, executes code, or interacts with APIs as part of answering a query. However, some developers find LangChain overly complex or heavy – it adds many abstractions, which sometimes makes it hard for the AI to work through, ironically. There are leaner frameworks emerging, like **Pocket Flow** (a very minimal agentic coding framework) which was built in response to LangChain's complexity. The idea with Pocket Flow is to keep things simple (just ~100 lines of code) so an AI can easily read its own tooling code and understand how to use it. Simplicity helps agents perform better, since they don't get lost in the framework itself.
- **Autonomous Task Agents:** These were made famous by projects like **AutoGPT** and **BabyAGI** (from the spring of 2023). They allow an AI to autonomously generate goals and sub-tasks to achieve a user-defined objective, using tools along the way. For example, AutoGPT could be told to "research a topic and write a report" and it will spawn multiple tool-using steps (web searches, writing files, etc.) by itself. In coding, an analog might be telling an agent to "build a Flask web app that does X" – it would plan the project, create files, maybe run `pip install` for dependencies, etc. These projects showed what's possible when you let an AI off the leash a bit. However, they can be inefficient and often get stuck without human feedback, so they're more experimental. Still, they introduced the idea of an AI agent that isn't just a single-turn assistant but an ongoing, autonomous worker.
- **DevOps and Debugging Agents:** Some tools focus on specific developer workflows, like debugging. For instance, there are emerging agents that can monitor an application and suggest fixes, or integrate with

CI/CD pipelines to automatically attempt to fix failing tests. An example is **Cursor's background agents** that run in your continuous integration to catch and fix issues proactively. We're also seeing things like **MCP (Multi-Tool Protocol)** servers which give agents controlled access to external tools like browsers or databases. This broader ecosystem of tools is about extending what the agent can do beyond just code generation – letting it interface with the real environment (running code, setting up containers, etc.) under supervision.

For Python developers specifically, it means you can choose an approach based on your needs:

- If you want an AI pair programmer in your IDE – tools like GitHub Copilot, Amazon CodeWhisperer, or the more AI-agentive ones like Replit's Ghostwriter or Cursor might suit you.
- If you want to explicitly build agents that, say, read a bunch of data and make decisions, libraries like LangChain or Haystack (for search-based Q&A) are relevant.
- If you want the AI to actually modify your project code, **Claude Code** is one of the frontrunners right now (especially since it has that large context window and tool usage).
- There are also **browser-based or CLI tools** like Microsoft's recently announced "GitHub Copilot X" CLI tools, which can do things like `gh copilot test` to write tests, etc., functioning somewhat as an agent on your repo. Open source efforts like **Aider** (which uses GPT to edit files via git diffs) allow similar agentic editing.

The ecosystem is really active. It seems every week there's a new framework or tool. The key trend is enabling AI to work with **full projects and external tools** instead of just one-file autocomplete. We'll likely hear from Matt which ones he's tried and prefers (he's mentioned Anthropic's Claude and also tools like Cursor, which suggests he keeps an eye on developer-centric AI tooling). The space is broad, but it's an exciting time because Python's simplicity and huge ecosystem makes it a prime playground for these AI agent frameworks.

5. You were an early user of GPT-3 back in 2020. What did those early AI coding experiments look like, and how have things evolved since then up to now (mid-2025)?

When GPT-3 came out in 2020, it was mind-blowing, but people were mostly using it for natural language tasks at first (chatting, content generation, etc.). Matt got into that private beta, so he had a chance to explore coding uses early on. Back then, the idea of using GPT-3 to write Python code was novel – we saw demos of it generating simple functions or trivial scripts from prompts. Matt likely started by calling the GPT-3 API with prompts like "Write a Python function that does X" and got back code. These were "toy projects" in the sense that you might use GPT-3 to solve Advent of Code problems or create a simple Flask app from scratch with a prompt. It was fun, but often required lots of prompt tweaking and the model would easily make mistakes beyond trivial scenarios.

Since 2020, a few big changes have happened:

- **Model Improvements:** We got specialized code models like OpenAI's Codex in 2021 (basis of Copilot) and more powerful general models (GPT-4 in 2023, Claude from Anthropic, etc.). These models got much better at understanding and producing code. They also got larger context windows – GPT-3 had 4K

tokens, now Claude has up to 100K, which is huge. By mid-2025, models can ingest whole codebases or large docs and reason about them. Matt's notes mention he's used Anthropic's models a lot (Claude). Claude is known for being good at following instructions and might have advantages in conversational coding tasks (plus the large context).

- **Tools & Workflows:** In 2020, people manually called the API or used early integrations. Now, we have purpose-built tools (like the ones we discussed earlier) that streamline the process. For example, Matt mentioned he initially used an editor called **Cursor** to kickstart using LLMs in coding – Cursor was one of the first IDEs that integrated GPT-style suggestions and even agentic behaviors. Over time, new interfaces like VS Code extensions or the Claude Code CLI have emerged, making it easier to loop the AI into your normal dev workflow. We've gone from copy-pasting prompts into a playground to having CLI commands where the AI can fetch files, propose diffs, etc.
- **From Suggestions to Agents:** Earlier usage was mostly single-turn or short interactions (e.g., "write this function for me"). Now there's this shift to an **interactive, iterative agent** model. For instance, by 2023 we saw the rise of ChatGPT where you converse with the AI about code, and by 2024-2025 tools like Claude Code let you have a persistent session with memory of your project. Matt himself notes that a big gap he felt was between those early toy experiments and integrating AI into an **"established codebase."** Today, he's largely closed that gap by developing strategies to actually use AI in a large existing project.

To put it in perspective, early on Matt might have used GPT-3 to prototype a small script or get help on a function. Now, in 2025, he might be using Claude to add a feature to a complex Django application by having it read the existing code, suggest changes, and even generate multiple files in one go. The evolution is not just in the AI's capabilities, but in understanding how to *manage* the AI's output in a real dev workflow (with version control, testing, etc.). He's learned lessons on what works and what doesn't (for example, he discovered some requests like "write a bunch of unit tests for my app" didn't pan out well – we'll get into that). The bottom line is: since GPT-3, the field has progressed from nifty demos to actually boosting productivity in professional projects, and Matt's experience mirrors that journey from skepticism and experimentation to daily practical use.

6. When you first started integrating AI into your coding workflow, what approaches worked well and which ones didn't? (In other words, what were some things you tried that were surprisingly successful, and what were some that failed or fell short?)

Early experimentation with AI in coding is full of hits and misses. Matt has highlighted a few in the show notes. On the **"what worked"** side, he found success by having the AI **analyze the codebase and engage in conversation about it**. For example, simply using the AI to answer questions about the existing code was very useful. Imagine loading your project into the AI's context and then asking, "How is data flow handled in this part of the app?" or "Can you summarize what this module does?" The AI can act like a supercharged search/documentation tool in those cases, helping a developer get up to speed on a large codebase. Matt also noted that a good place to apply AI first was **changing pieces of the UI**. This makes sense – UI code (HTML templates, CSS, simple component rendering) can often be tedious, and an AI can make small changes (like

altering layouts, text, colors, or adding a new field to a form) quite reliably because it's relatively constrained in scope and easy to verify visually. Those kind of tasks worked well, giving the AI something concrete and bounded to do.

Another thing that likely worked is having the AI help with incremental refactors or improvements, rather than outright massive generation. For instance, asking "Can you refactor this function to be more readable?" or "I think this query is slow, how can we optimize it?" leverages the AI's strength in offering suggestions and even writing code, but still keeps the human in charge of validating and integrating those suggestions.

Now to the **"didn't work"** side: One clear example Matt gave is telling the AI to *"create a bunch of unit tests"* for an existing codebase. While it sounds reasonable (who wouldn't want the AI to write all their tests?), in practice this tends to fail for a few reasons. If you just prompt an AI to generate many tests, it might produce tests that **don't actually reflect the intent of your code** or that rely on specifics that aren't true. Without the context of exactly how functions should behave (which often lives in the human's head or in docs), the AI might assert things incorrectly. Also, writing dozens of tests is a large output – LLMs can lose track of details or produce repetitive boilerplate. Developers have reported that AI-generated tests often need heavy fixing, and they sometimes end up being a time sink rather than a time saver. So Matt's experience confirmed that: "make a bunch of tests for my whole app" was too broad and the results weren't great.

Likewise, we can infer other things that probably didn't work well: using the AI without a clear plan or on unbounded tasks. For instance, asking it to implement a complex feature end-to-end in one go might have led to issues (the AI could hallucinate function names or misuse APIs, requiring the developer to do a lot of cleanup). Also, if he tried using AI in areas where he himself wasn't comfortable or didn't understand the domain, reviewing the AI's output could become difficult – so a lesson might have been to apply AI in areas where you can verify it.

Another subtle thing that doesn't work is expecting the AI to magically know your architecture. Early on, if he just prompted "Add feature X", the AI might go down a wrong path if it misunderstood the structure. So what works better (as he learned) is first have the AI analyze or outline the approach (converse with it about how to do it within your specific codebase), then proceed with implementation. In summary, what worked was **using AI as a collaborator** – asking it to explain, suggest, or do targeted edits. What didn't work was treating it as an all-knowing coder and giving it very broad commands like mass test creation or expecting a perfect large feature in one shot. Those often led to disappointment and rework.

7. What unique challenges or considerations come up when applying AI coding assistants to an existing, large codebase versus using them on a brand new project?

Using AI in a big, established codebase is a different ballgame compared to a fresh project. Matt's notes already hint at some challenges. In an existing codebase, you have to deal with **context and consistency**: the AI needs to understand how the parts of the system work together, the coding style, and the conventions used. With a new project, you can kind of let the AI propose a structure from scratch (or you choose a structure and have it fill in code). But in a mature project, the AI's output has to fit in with everything already there.

One challenge is that large codebases often have **hidden assumptions or side effects** that aren't immediately obvious. A human developer learns these over time or through documentation. An AI might read the code, but it might miss subtle things like "oh, this function is called by an external cron job" or "this config value is crucial for these queries." So, using AI on an established project means the developer needs to be vigilant in reviewing AI changes for unintended consequences. A related issue is **technical debt**: older codebases have quirks or outdated patterns (we'll discuss technical debt more later). An AI could inadvertently propagate those quirks because it sees them in the code and thinks that's the way to do things.

Another consideration is **tooling and setup**: In a toy project, you might not care about environment or tests initially. But an established project likely has a build process, tests, linters, CI, etc. If the AI is going to run or test code, that environment needs to be accessible. Matt found value in using tools like the **Django Debug Toolbar** when optimizing a query with the AI's help. In a large app, you have profilers, debuggers, test suites – hooking the AI's suggestions into those feedback loops (like measuring a query time before and after AI's change) becomes important to know if the change is good. So, existing projects demand *measuring and validation* of AI contributions.

Also, an existing project often means multiple people and a version control history. Introducing AI-generated code needs to be done in a way that your team can review and accept. You might have to break down the AI's contributions into smaller commits to get them through code review, rather than one huge automated PR.

By contrast, with a greenfield project, you have more freedom to let the AI generate boilerplate or scaffold out a design it "thinks" is good. You're not constrained by legacy decisions. Matt mentioned he felt a gap between toy (greenfield) and real (brownfield) projects – that gap includes things like *framework conventions, coding standards, dependency management,* and so on that you only encounter in real-world apps. It likely took trial and error for him to effectively prompt the AI within a large Django app context, for example, whereas on a new project you might just say "use FastAPI to build X" and accept what comes.

In summary, with a large codebase, the challenges include: ensuring the AI truly understands the context (sometimes you have to feed it a lot of your code or explicitly point it to relevant files), maintaining consistency with existing code (style and architecture), avoiding the replication of existing bugs or bad patterns, and verifying that new changes don't break anything. It requires a more disciplined approach – often an **incremental, iterative strategy** works best (small changes at a time, lots of tests), whereas on a new project you can be more sweeping. We'll hear from Matt on strategies he used to overcome these challenges, like maybe starting with read-only mode (just asking questions) before trustfully letting it write code, and using version control to manage risk.

8. How do you use AI to help understand or improve an existing codebase? For instance, you mentioned asking it questions like "How can I optimize this database query?" and using tools like Django Debug Toolbar to gauge improvements – can you walk us through that?

This question digs into a practical workflow. Matt mentioned that one of the first useful things an AI can do in a legacy codebase is to answer questions about it or suggest improvements. Let's use the example given: optimizing a database query in a Django app. Here's how that process might look:

First, you'd give the AI the relevant context. With something like Claude or ChatGPT, you might provide the model with the Django model definition, the query code (maybe a Django ORM query or raw SQL), and possibly some info on performance (like "this query is slow, taking 5 seconds to run with 10k records"). Then you ask: **"How can I optimize this query?"** The AI will analyze the code and likely draw on general knowledge of Django optimization. It might suggest adding an index, or using `.select_related` / `.prefetch_related` if it sees foreign key traversals, or perhaps denormalizing something if appropriate. Essentially, it acts like a consultant pointing out inefficiencies in how the query is built.

Now, Matt specifically references using **django-debug-toolbar**. This is a tool that shows SQL queries executed per request, their timing, etc., in a Django app. A smart way to involve the AI is: run your web page with debug toolbar, see that a certain request made, say, 100 database queries or one very slow query. You feed that information to the AI: e.g., "This request triggers 100 queries, mostly similar; here's an example query and result. How can we reduce the number of queries?" The AI can then suggest techniques like query caching, combining queries, or using Django's select_related (to join and fetch related objects in one go). You could then attempt the change it suggests or have the AI write the code for it. After that, you run the debug toolbar again to *measure* the effect. This closes the loop: if the AI suggested something and the number of queries dropped or the time went down, you have concrete evidence the AI's suggestion was good.

Matt's approach indicates **using AI as a partner in performance tuning**. The AI can quickly enumerate potential optimizations which you might also know, but it saves time and might catch something you didn't think of. Importantly, the use of tools like debug-toolbar means you're not blindly trusting the AI – you verify the improvement.

Similarly, for understanding code, you might ask the AI questions like: *"Explain how data flows from this API endpoint down to the database"* or *"Why might the function `x` be returning None in this case?"* If you've given it enough of your code, the AI can often trace through logic and give you an explanation. It's like having a very patient senior engineer who has read all your code and can summarize it on demand. This can save you from manually grepping and reading a bunch of files.

So walking through it:

- Identify a performance issue via tooling.
- Provide context to AI and ask for suggestions.
- AI gives potential improvements (and maybe code diffs).
- Implement/test those changes (possibly the AI writes the code for you).
- Measure again with the tool to confirm improvement.

The fact Matt calls out that workflow suggests it was a **successful pattern** for him. It showcases how AI can assist not just in writing features but in diagnosing and improving code quality and performance, acting almost like an automated pair programmer with a lot of Django knowledge at its disposal. And Django, being a very conventional framework, gives the AI strong hints (for example, if it sees a `.all()` in a loop, it knows "oh, N+1 query problem!" and might suggest a fix).

By leveraging both the AI's knowledge and actual runtime metrics, a developer can iteratively enhance an app. Listeners can take away that they too could use this approach: get a baseline measurement, involve the AI for advice, apply changes, and verify results – combining the best of AI's broad knowledge with concrete data from your specific application.

9. You've found that structured frameworks like Django make it easier for AI to write or modify code. Why do you think a prescriptive framework like Django works better with AI assistance than a more flexible one like Flask?

This is a fascinating insight that Matt shared: **prescriptive frameworks (Django) work better with AI** in his experience, compared to microframeworks like Flask. The core reason likely comes down to **convention and consistency**. Django is a "batteries-included" framework that has a lot of conventions – there's a standard way to do models, views, templates, URL routing, etc. Because it's so prescriptive, an AI has an easier time navigating a Django project structure and following the established patterns. Essentially, **Django reduces the degrees of freedom** in how you solve a problem. For an AI that has been trained on tons of Django code (and documentation), it will recognize, "okay, this is a Django Model class, typically if we want to add a field we also need to make a migration; if we want to optimize a query we use the ORM's tools," and so forth. It's operating within known guardrails.

Flask, on the other hand, is very minimal – it's just one file to start, and you can organize a Flask app any way you want. That flexibility is wonderful for developers who want simplicity, but for an AI, it's less guidance. Two Flask applications could be structured totally differently. The AI might not readily see where the database code is vs. the routing, because Flask doesn't impose a project layout. In Django, if you ask "add a new endpoint to do X," the AI knows it should create or modify a view in `views.py`, add a URL in `urls.py`, maybe a template in the `templates/` folder, and so on. There's a standard operating procedure.

We have some corroborating insights: Armin Ronacher (creator of Flask) in his blog noted that **stable ecosystems** and frameworks with less churn are easier for LLMs. He even said "LLMs… love to use Flask, because those are quite stable ecosystems with little churn" – which is interesting because it highlights stability, but Django is both stable and prescriptive. Matt's angle is prescriptive is better, Armin's angle is stability (Flask is stable in the sense it hasn't changed much). Possibly, Django hits both: it's stable and it's very structured. The **AI probably performs best when it doesn't have to infer too much about the project's architecture** – Django provides that architecture out of the box.

Another aspect: frameworks like Django come with rich documentation and strong community patterns, which likely were part of the AI's training data. So the AI has many examples to imitate. In contrast, if someone makes a custom architecture in Flask, the AI might not have seen that exact pattern before.

From Matt's personal experience, he said "Django works great because it's so prescriptive." This likely meant when he used AI in a Django app, the results were coherent and required fewer corrections. If he tried with a Flask app where maybe he wired things up in a custom way, the AI might have made more mistakes or needed more guidance.

For Python beginners listening, this is a takeaway that choosing a more structured framework might not only help *you* as a developer (by enforcing good practices) but also helps AI assistants help you. It's almost like giving the AI a map versus giving it a blank canvas. With Django, the AI has a map of where things generally should go.

Finally, consider error rates: With Django, if the AI writes code, there's a good chance it fits in the first try (like adding a model field and updating serializer, etc.). With Flask, the AI might write code that doesn't match your app's way of doing things (maybe it assumes usage of SQLAlchemy when you used something else). That means more manual fixes. Matt also mentioned that when he codes by hand less and relies on AI, *he* writes more bugs – imagine if the framework were loose, the AI might also inject more subtle bugs because of mismatched assumptions. So Django's strictness could actually mitigate that by channeling the AI down correct paths.

10. Can you walk us through your current workflow for using agentic AI while coding (here in mid-2025)? You mentioned using Claude Code integrated with your tmux + Vim setup – how do you actually interact with the AI day-to-day as you develop?

Matt's current workflow is really interesting because it sounds like he's deeply integrated an AI agent into a fairly classic development setup (tmux + Vim, which is very keyboard-and-terminal-driven). Here's how we can imagine it based on clues:

He uses **Claude Code**, which is Anthropic's AI coding assistant. Claude Code can be used through a command-line interface. Likely, Matt runs a CLI program (maybe called `claude` or similar) from within his tmux session. This program might allow him to open a chat-like interface where he can send instructions to Claude and it can edit files or run commands.

One key aspect he mentioned is **Plan Mode** (triggered by pressing Shift+Tab in the Claude interface) and that he spends a lot of time in something called **Claude Code – Plan vs. Auto modes**. Plan Mode is a feature where instead of immediately writing code changes, the AI will outline a step-by-step plan for what it intends to do. For example, if Matt says "Add a new field to the User model and expose it via the API," the AI in Plan Mode might respond with a plan like:

1. Modify `models.py` to add the field.
2. Create a database migration.
3. Update `serializers.py` to include the field.
4. Adjust `views.py` to handle the field.
5. Write a unit test for this new field.

This plan is shown to the developer first. Matt can then review and either refine it or tell Claude "go ahead and implement." If he agrees, switching out of Plan Mode (or hitting some confirmation) will make Claude execute those steps – actually editing the files accordingly. This separation is useful: it ensures the AI and developer agree on the approach before code is changed. Matt specifically said using Plan Mode is valuable; it gives a roadmap and prevents the AI from "going rogue" with unwanted edits. It's like the AI saying "Let me double-

check my understanding with you."

He also mentioned **using Opus vs Sonnet models** – these are two variants of Claude. Based on what we know, Claude "Opus" is the larger, more powerful (and expensive) model, and "Sonnet" is a cheaper, faster one with slightly different behavior. Matt said he switches between Opus and Sonnet, treating Opus for design and Sonnet for implementation. This likely means: when discussing or planning at a high level, he might invoke the Opus model (which might be better at creative or deep reasoning due to more capability). But when it's time to churn out code, he uses Sonnet, which he perhaps finds generates code he likes more or is more straightforward. Interestingly, Armin Ronacher also noted he prefers Sonnet's outputs and uses it exclusively for coding. So Matt's workflow might involve toggling models depending on the task – a pretty advanced technique to get the best of both.

The mention of **"ultra think…"** likely refers to a prompt or mode where he can ask Claude to really deeply consider a tough problem. There's a known tip floating around: if you tell Claude "ultrathink about this problem" it might use more of its 100k token context to reason extensively (maybe akin to giving it permission to produce a very long chain-of-thought). This could help with complex debugging or architectural suggestions. It basically tells Claude to be extra thorough (possibly a feature or just a prompt strategy).

He also said he uses **tmux+vim** – so presumably he splits his terminal, running code editor in one pane and the Claude interface in another. Some devs even integrate the AI so it can open Vim buffers with suggestions. Or, he might copy prompts from Vim into Claude CLI.

Another element: **ccusage**. This is a CLI tool to analyze Claude Code usage. He uses it to measure his productivity and visualize usage like a commit history. Practically, this means every time he uses Claude, it logs interactions (in JSONL logs). The `ccusage` tool can read those and show how many tokens or edits per day, maybe a chart of his activity. Matt sees it almost like a Git commit graph – a representation of how much work he got done with the AI's help. This likely motivates or helps him reflect: for example, "Wow, I had a spike of usage on Friday, no wonder I produced so much code, whereas Monday I hardly used the AI." It's a neat self-metric for how integral AI is to his workflow.

So a day-to-day might look like:

- He fires up tmux and Vim to code as usual. When he hits a task he wants AI help on, he invokes Claude (maybe a command or a chat window).
- He might start in Plan Mode: ask a question or issue an instruction; Claude outlines a plan.
- He reviews the plan, maybe modifies it or says "looks good."
- Claude then switches to execute mode (Auto mode) and makes the changes (it might show a diff or mention "Edited 2 files" as seen in some interfaces).
- He then tests or runs the project to see if it works. If something fails, he can ask Claude to help debug (Claude can read the error trace and suggest a fix).
- He commits the changes to git, often small incremental commits.
- Later, he might run `ccusage` to see stats or just have it as a background monitoring.
- He uses different **color schemes or prompts for different agents** as well. The note about color schemes suggests if he ever uses multiple AI instances (maybe one local model and one cloud model

simultaneously, or one window in plan mode vs another in auto mode), he colors their text differently in the terminal so he doesn't confuse which is which. That helps when orchestrating more than one AI or comparing outputs.

It's quite a sophisticated setup, blending classic tools with cutting-edge AI. But the key is it's still **developer-driven**: he's in control via the terminal, using AI as an extension of his environment.

11. What are some of the hard lessons you've learned from incorporating AI agents into development, and what tips would you share with Python developers who want to try this?

Matt has gathered a bunch of lessons and tips, and they're pure gold for anyone attempting this. Let's go through the major ones he's highlighted:

- **Use Prescriptive/Structured Tools & Frameworks:** We touched on this – using a framework like Django (or any tool that has clear patterns) helps. His tip is essentially if you give the AI a well-structured playground, it will perform better. Conversely, if your project is chaotic or highly unique, you might struggle more.
- **You might introduce more bugs if you just trust the AI blindly.** Matt admitted that he sometimes writes more bugs himself when not coding by hand as much. This paradoxical lesson means: when you rely on the AI, you might get a false sense of security or not think through the code as deeply, leading to bugs sneaking in. The tip here is to **stay vigilant and double-check** AI output. Write or run tests, and don't assume the AI's code is correct. Use the AI to generate code, but use your own skills to validate it.
- **Consistency in Design Patterns:** If your project has a certain style or pattern, try to maintain that. It might be helpful to explicitly tell the AI about these patterns (some people write a contributing guide or an "AI conventions" file in the repo to guide it). Matt suggests that consistency is key – the more consistent your codebase, the easier it is for the AI to slot in new code without breaking things. So, one tip is to maybe refactor weird outlier code to match the predominant patterns *before* unleashing the AI on it.
- **Remove Technical Debt First:** This is a big lesson. Technical debt (like hacky code, outdated practices, messy architecture) can confuse a human, and it definitely confuses an AI. Matt noticed that if you have bad patterns in your code, the AI might copy them because it thinks that's what you want (after all, it sees them in the codebase). For instance, if there's a function that's 500 lines long and badly structured, an AI might mimic that style elsewhere instead of using a cleaner approach. So his tip: spend time paying down tech debt and cleaning up code *before* or while using the AI. A cleaner codebase is not just good in general, but it literally makes the AI's outputs better. As an analogy, if you train an apprentice in a workshop full of clutter and broken tools, they'll pick up bad habits; but in a well-organized shop, they'll learn the right way.
- **Be mindful of architecture – possibly lean towards smaller services or modules.** Matt observed that architecture tends to tilt towards splitting into multiple services or microservices in an AI-assisted workflow. Why might that be? One reason: if each piece of the system is smaller and well-defined, the AI

can handle it more easily (less context to load, fewer side effects to consider). Also, agents aren't very good at juggling a ton of complexity in one go. If you have a monolith with hundreds of intertwined parts, the agent might trip more often. But if your system is divided into separate services or very modular components, you can focus the AI on one at a time. His tip might be: consider breaking big tasks into micro-tasks or big apps into smaller apps where appropriate, so that it's easier for the agent to work on them independently. We're not saying everyone should adopt microservices just for AI, but being mindful of component boundaries and making the AI's "cognitive load" smaller can help.

- **Keep an incremental, git-based workflow:** This is crucial. Don't let the AI change too much at once without version control. Ideally, every AI-induced change is a commit you can inspect. Matt stresses an "incremental git-based workflow" – meaning after each chunk of changes, commit them, run tests, ensure things are good. This also implies you should review diffs of what the AI did. If you see something odd in the diff, you can catch it early. Many AI tools actually output diffs or have the AI propose a code diff that you apply. Embrace that. It provides a safety net to undo changes or bisect issues later.
- **Specific to Claude (or any AI tool) – learn its features:** Matt shares tips like using Claude's plan mode (as we discussed), using different model variants (Opus vs Sonnet) for different tasks, and other commands (some people use Markdown checklists or "ultra think" prompts, etc.). His lesson is that to get the most out of these AI agents, you should *learn the tool* just like you'd learn an IDE's shortcuts. There are hidden gems (like Shift-Tab plan mode) that dramatically improve the workflow once you know them.
- **Monitor your AI usage:** He finds tracking usage via `ccusage` helpful, which is a unique tip. It's less about coding and more about self-management. If you see a day where you used 100k tokens with little to show, maybe you were going down a wrong path or debugging. That reflection can help you adjust how you use the AI. It's analogous to tracking time or keystrokes, but here it's AI tokens.

In summary, Matt's hard lessons boil down to: *be structured and clean in your code, supervise the AI's work with good practices (testing, version control), and adapt your development style to guide the AI (not too big tasks at once, maintain consistency).* For Python devs starting out, his tips would likely include: start with small AI tasks, always review and test, improve your code quality to get better AI results, and incrementally increase the agent's autonomy as you gain trust in it.

12. One interesting thing you noted is that you "write more bugs yourself when you're not coding by hand as much" because of using AI. How does using AI lead to more bugs, and how do you catch or prevent those bugs?

This counterintuitive lesson deserves a closer look. You'd think having an AI help would *reduce* bugs, but Matt observed that he personally introduced more bugs when he leaned heavily on AI-generated code. There are a few reasons this can happen:

- **Reduced Mental Immersion:** When a developer writes code by hand, they're mentally stepping through the logic, and often that process catches errors before they even hit run. If the AI writes a chunk of code, the developer might not fully grok every line of it, especially if they're accepting large suggestions. Matt might skim the AI's output and think it looks fine, but because he didn't "feel" the logic in the same way as writing it, a bug might slip through – something he'd have caught if he wrote it himself. Essentially, the AI

can short-circuit the deep understanding a coder usually builds while coding, and that can let bugs propagate.

- **Overtrusting the AI:** Early on, especially, one might assume the AI has done things correctly ("it's a fancy model, it must know what it's doing"). This trust can lead to fewer thorough reviews. If Matt took the AI's code at face value, he might miss subtle issues. Over time, I suspect he learned to always test and review the AI's code as diligently as if a junior developer wrote it. A tip here: treat AI output as if it's written by a human collaborator whose skills you're still gauging – in other words, **trust but verify**.
- **Misalignment or Slight Errors:** AI might produce code that *nearly* works but has off-by-one errors, minor logic mistakes, etc. If you didn't write it, those errors aren't as obvious to you. For example, maybe the AI uses a >= instead of >, which could be a one-character bug that's hard to notice in a diff review. The human coder might have gotten that right instinctively or tested that scenario while writing.

So how to catch/prevent these bugs? Matt's experience likely drove him to adopt several practices:

- **Write tests (or have tests) and run them frequently.** Automated tests will catch many bugs that you might not see on quick read-through. If the AI writes code, run the test suite or at least test that area manually.
- **Use small iterations.** If you let the AI change 10 things at once and something breaks, it's harder to pinpoint. If it changes 1 thing at a time, you know exactly where to look.
- **Force yourself to understand the AI's output.** One approach is to have the AI explain its changes. In fact, in that earlier anecdote, the AI *"summarized the changes it made and why"* after fixing the 38 mypy errors. That summary helps the developer catch any reasoning mistakes. So you can prompt the AI: "Explain this diff" or "Tell me why this solution works." If the explanation doesn't make sense, there might be a bug.
- **Don't skip code reading.** Matt might have had to discipline himself to still read through AI-generated code carefully. It's tempting to not fully read what you didn't write, but you have to. Some developers format the AI's diff nicely and go over it line by line as a code review.

There's also an aspect of psychological or workflow adjustment: As Matt said, when he's "not coding by hand as much," perhaps he's multitasking or not as laser-focused (since the AI is doing the grunt work). That could lead to mistakes in integrating the changes. Imagine the AI writes a function, and you forget to call it, or you integrate it in the wrong place because you assumed something. The remedy is to stay engaged – even if the AI is writing code, you as the developer are responsible for the correctness and integration of that code.

Another preventative measure is using the AI to your advantage in testing. Ironically, while "write a bunch of unit tests" may not work, you can ask the AI to generate a few tests for the specific code it just wrote. For example, "Now that you implemented this function, can you provide a couple of tests for it?" Those tests might catch obvious errors. They might not be perfect or cover everything, but it's something.

In summary, using AI can lead to more bugs if a developer becomes a bit detached or overtrusting of the code generation. The way to combat that is to remain in a **critical review mode**, use your normal debugging and testing skills, and treat AI output with healthy skepticism. When done right, you can still net out ahead (more productivity and no more bugs than usual), but it requires adjusting your process. Matt's candid point about writing more bugs is almost a caution: *don't get lulled into autopilot.* Keep eyes sharp, and use the AI as a helper,

not an infallible coder.

13. How do you ensure consistency in coding style and design patterns when an AI is generating code? Have you run into issues with the AI adopting bad patterns from the existing code or from its training data?

Ensuring consistency and good design is a real concern with AI-generated code. There are a couple of scenarios: (1) The AI might start following *bad patterns that exist in your codebase* (as we discussed regarding technical debt), and (2) it might introduce patterns from its training that conflict with your style or best practices.

Matt's strategy, as gleaned from his notes, includes proactively managing this. One approach is **explicit instruction**: Some developers maintain a guidelines file (e.g., `CONTRIBUTING.md` or an `AI.md`) outlining the style rules and patterns, and they feed that to the AI. For example, "Follow PEP8 style, use f-strings for formatting, prefer list comprehensions over map, etc." If an AI knows those guidelines up front, it's more likely to produce consistent code. In interactive sessions, if Matt sees the AI doing something inconsistent, he can correct it: "Actually, let's do this using a class-based view since our project uses class-based views in Django," and the AI should adjust.

However, not everything can be caught upfront. So a key is **code review of AI's contributions with an eye on style/design**. Matt or the team would need to enforce the same standards on AI code as on human-written code. If the AI generates a singleton pattern where it's not appropriate, or uses a different naming convention, that should be revised. Over time, if you keep correcting the AI in the same way, it will likely adapt (since many AI systems have some persistent memory or at least you can copy-paste previous corrections as reminders).

There's also the idea of **template or example-driven prompting**: If you want the AI to follow a pattern, show it an example pattern. For instance, "Here is how we typically implement a service class [paste example]. Now implement a new service class for X feature following the same pattern." LLMs are great mimics – if they see a clear example, they can emulate the style and structure. This helps maintain consistency.

Matt pointed out that prescriptive frameworks and removing tech debt help a lot – that's indirectly about consistency too. A clean, consistent base means the AI doesn't have multiple conflicting examples to choose from. If half your code does logging one way and the other half another way, the AI might add to the chaos. But if everything is done one way, it'll stick to that. So one practical tip is to do a cleanup pass: normalize your code style and patterns *before* heavily using the AI, so it doesn't get confused or reinforce the wrong approach.

The AI's training data could also have some outdated or non-ideal patterns. For example, maybe it saw a lot of older Django code and might use patterns that are no longer recommended. Matt needs to watch for that. A trivial example: maybe earlier code often used Python's older format `%` strings, while we prefer f-strings now. The AI might output `%` style unless told otherwise. Or in design patterns: maybe it tries to use a Singleton or global state because it saw that, whereas your app might avoid those. As a developer, you have to intercept those and guide the AI back.

One interesting thing from other AI users: they note LLMs sometimes struggle with principles like DRY (Don't Repeat Yourself) or SRP (Single Responsibility Principle). Chris Howard, in his notes on Claude, mentioned he put DRY and SRP in the global instructions but the AI often "ignores them" and over-engineers or duplicates code. This shows that simply stating "don't repeat code" might not be enough – the AI might still produce somewhat clunky solutions that a human would refine. So maintaining design principles requires either iterative prompting ("Can you refactor this to remove duplication?") or manual intervention after the fact.

To deal with the AI adopting *bad* patterns from the codebase, Matt emphasized removing those bad patterns (technical debt). But if it still happens, he could use the AI to fix them. For example, if he sees the AI wrote some convoluted method because it copied an existing one, he can say "Refactor this new method to be simpler or use modern approach X." The AI can then improve it. In a sense, you can bounce the AI against itself: let it generate, then critique that generation (either yourself or even have the AI critique it by asking "is this code well-designed? how to improve?").

In summary, ensuring consistency and good design with an AI co-pilot involves:

- Giving clear style/design guidelines to the AI.
- Keeping the codebase as clean and consistent as possible (so the AI learns the "right" way).
- Using examples to set patterns.
- Reviewing AI code like any code – don't let things slide just because the AI wrote it.
- Iteratively refining the AI's output (sometimes asking it to improve its own code).

Matt's experience likely has him doing all of the above. He's implicitly said "design patterns consistency" and "bad patterns copying" were lessons, so now he's proactive about it. He might even have some automated linters or formatters – running Black or flake8 on AI outputs could catch style issues automatically, for instance, which helps with consistency too.

14. You emphasized the importance of removing technical debt when working with AI agents. Can you share why technical debt is a bigger issue in this context and how you go about cleaning it up?

Technical debt (old hacks, outdated code, incomplete refactors, etc.) is problematic in any scenario, but it's especially troublesome with AI agents for a few reasons:

- **AI as a Mimic:** As we discussed, an AI will often imitate patterns it sees in your codebase. If your codebase has some ugly, debt-ridden code, the AI doesn't inherently know it's "bad" – it might assume that's the norm. For example, if there's a 100-line function with a bunch of global state usage (a classic sign of tech debt), and you ask the AI to add a feature, it might piggyback on that same function because "that's how things are done here," making the function 150 lines and even harder to manage. Essentially, tech debt can multiply when an AI copies it blindly.
- **Compounding Errors:** Technical debt often means things aren't well-tested or well-understood. When an AI agent starts building on shaky foundations, the resulting code can be even more unstable. It might

inadvertently rely on undefined behavior or make assumptions that were only true accidentally in the old code. Humans tread carefully around landmines in legacy code; an AI might run straight into them.

- **Clarity for the AI:** A cleaner codebase is easier for an AI to reason about. If the AI has to read through convoluted, confusing code, it might misinterpret what's going on, leading to incorrect modifications. Removing technical debt – e.g., refactoring a messy piece into clean functions – means if later you ask the AI to modify that part, it can understand it correctly. There's an insight that in agentic coding, *simplicity is key*. Armin Ronacher mentioned that "simple code significantly outperforms complex code in agentic contexts". Simplifying code (which is part of addressing tech debt) makes it more likely the AI will do the right thing.
- **Design Docs and Comments:** Part of tech debt is often lack of documentation. When cleaning debt, one might add comments or docs. If those are present, feeding them to the AI can guide it. For instance, if you have a comment "# FIXME: temporary workaround, will be refactored later," an AI might avoid building on that workaround if it sees the comment. Without it, it might double-down on the workaround.

So how to remove tech debt with AI in mind? Matt likely takes an approach of incremental refactoring *before* adding new features with AI. Possibly he uses the AI itself to help remove debt: "Hey Claude, this function is too long, can you refactor it into smaller functions that each do one thing?" The AI could do that, and then subsequent tasks will be on cleaner ground. Or "We have some global state, can you refactor to use dependency injection?" etc. It's almost like you have to teach the AI the *right way* by making the code reflect it.

Also, he might prioritize cleaning up parts of the codebase where he anticipates using the AI most. For example, if he knows a certain module will be worked on, he might spend time to bring it up to modern standards (like update an outdated library usage, remove deprecated code) so the AI doesn't output deprecated patterns from reading it.

There's also a future-looking perspective: If AI agents are to be a regular part of development, we might start coding differently to accommodate them – writing code that's not just for humans to understand but also AI-friendly. That could mean more straightforward logic, clearer variable names, etc., which coincidentally are the same things that fight technical debt for human readers.

Matt specifically noted removing tech debt is *more important* in AI-assisted work. I suspect he encountered cases where failing to do so caused the AI to produce something wrong or inefficient, costing him time. So now he likely advocates: **don't skip that cleanup**. It will save you headaches because the AI will operate on a clean foundation.

A concrete example: Suppose the project uses an old authentication approach scattered across many views (tech debt). If he doesn't fix that and asks the AI to add a new auth feature, the AI might copy that scattered approach (because it sees that's how it's done), making the problem worse. If instead, he first refactors to a centralized auth utility, then when the AI adds a feature it will likely plug into that clean utility. The result is better code and less mess.

So in practice, cleaning tech debt might involve tasks like:

- Refactoring long functions/classes.
- Updating outdated API usage.

- Writing missing tests for critical parts (so AI changes can be verified).
- Standardizing duplicate code (so AI doesn't inadvertently create yet another variant).
- Documenting unclear parts.

He likely uses a mix of manual effort and AI help for this, but with careful verification. The overarching message: technical debt is "toxic" for AI contributions, so invest the time to address it, and your AI assistant will serve you much better.

15. Does using AI tools influence how you structure your projects? For example, you suggested it might push towards splitting code into smaller services or microservices – why is that, and have you seen it in practice?

This is a very intriguing point Matt raised: the architecture might tilt towards multiple smaller services or components when using AI in development. Let's unpack why that might be:

Think of how an AI agent works – it has a certain context limit and complexity it can handle at once. If you have a huge monolithic application with a million lines of code, even if the AI can technically read a lot of it (with a large context window), it's hard for the model to keep everything in mind and reason correctly across the entire thing. On the other hand, if your system is broken into microservices or just well-separated modules, you can engage the AI with one piece at a time in isolation. The AI's "mental load" is lighter and it's less likely to make mistakes that span across unrelated areas.

In practice, Matt noticing a trend toward multiple services might mean when he uses AI to work on a feature, he finds it easier if that feature is isolated in its own service or library. Over time, he may have started structuring new parts of the system as separate deployable units because that boundary naturally limits what the AI touches. It also aligns with human best practices of separation of concerns, but here it's concern separation to help the AI.

Another angle: When an AI builds or modifies code, if something goes wrong, it's often easier to debug if the component is small. If an AI agent is working on Service A and it breaks, only Service A fails tests, and Service B is unaffected. This compartmentalization could reduce the risk of AI making broad changes that break everything.

We should note that microservices have their own complexity (deployment, coordination, etc.), so it's not that AI forces microservices always. But maybe microservice-like thinking – i.e., **modularity** – is encouraged. Even within a single repo, structuring code into clearly separated modules or packages can help. One could imagine in the future, we have AI agents each specialized or assigned to different microservices, and a higher-level orchestrator AI coordinating them. That's speculative, but hints of it appear in some tools which let you run parallel agents for different tasks.

Matt also might be seeing that some AI-generated architecture decisions lean that way. Possibly the AI sometimes suggests, "This part of the system could be a separate service." Maybe he's seen the AI propose using an external API or splitting functionality.

In his personal workflow, it might manifest as him intentionally breaking tasks: instead of adding a feature into an existing large app, he might start a new Flask or FastAPI microservice and let the AI build that fresh, then integrate via APIs. Starting fresh can sometimes be easier for the AI (no legacy to confuse it). There's a parallel in human teams: sometimes it's easier to write a new microservice than to insert complex code into a monolith. AI might similarly benefit from the clean slate approach.

Additionally, we have Armin's insight where he actually recommended **Go** for new agentic projects partly because of how straightforward it is to make small services in Go, and how the agent handles that environment well. For Python, if one sticks to Python, perhaps splitting into microservices addresses some of the issues (like Python's slower startup and complex runtime as Armin complained). If each service is smaller, the agent's test-run cycles are faster.

Matt said "Architecture tilts towards splitting into multiple services or even microservices." So he's likely seen that *he himself* started doing this after using AI, or he predicts the industry will. In practice, since he's a DevRel at DigitalOcean, he might have talked to many teams and seen them leaning into microservices as they adopt AI coding assistance, because it offers more control and parallel development (imagine multiple AI agents each handling a service simultaneously – that's easier than one AI dealing with a monolith serially).

So yes, in practice: if he had a large Django monolith, maybe now he's carving pieces out – like a separate service for a specific job. Or at least thinking in terms of smaller chunks when giving tasks to the AI.

For the audience, the takeaway is: using AI might encourage you to design software with clear boundaries and smaller pieces, which is generally good practice, but now with the added benefit of being AI-friendly. So it's like architecture influenced by a new kind of team member (the AI agent). We'll see if Matt provides concrete examples, like "we spun out our analytics pipeline as a separate service because it was easier to let the AI build that from scratch than modify the existing one."

16. What does your version control workflow look like with AI making changes? Are there best practices for using git (committing, reviewing diffs, etc.) when an agent is generating a lot of code?

Integrating AI into your workflow absolutely requires discipline with version control. Matt emphasizes a consistent, incremental git-based workflow, which likely means a few things:

- **Small Commits:** When the AI makes changes, try to keep them in logically grouped, small commits. For example, if the AI implements a new feature, instead of one giant commit that spans 20 files, you might break it into a series: commit 1 for model changes, commit 2 for view logic, commit 3 for tests, etc. This way, if something goes wrong, `git bisect` or revert is manageable. Also, code reviews (even self-reviews) are easier when commits are focused. Matt probably instructs the AI or manually ensures that after each step of a plan, he commits the results. Some AI tooling might even suggest commit messages or auto-commit after each task.
- **Review Diffs Carefully:** Before committing the AI's changes, he likely looks at the diff to see exactly what was altered. As a tip, treat diffs from AI as proposals that you need to approve. This is similar to how you'd review a colleague's pull request. If something looks off, you can either fix it yourself or ask the AI

to correct it. Many AI coding tools output diffs (like a patch file or side-by-side), which helps with this process.

- **Use Branches:** It's wise to work on a separate git branch when doing AI-assisted changes, especially larger ones. That way, you can run tests and even share that branch with colleagues for review without affecting the main. If the AI goes on a wrong track, you can simply abandon that branch or roll back.
- **Frequent Commits vs. Token Limits:** One challenge might be that if the AI is keeping the entire diff in context, doing too many commits might lose some context. But usually, you can just feed it the relevant parts again. It's likely that Matt found a rhythm, like commit after each reasonably self-contained set of changes and maybe tell the AI "I've committed up to here, next let's do X."
- **Automated Tools for Git:** Some people integrate AI with git by using commit hooks or CLI tools. For example, there are experiments where you can describe a change in natural language and an AI generates a git patch for it. If Matt uses anything like that, his workflow could involve commands like `git add -p` to stage changes selectively (if he doesn't like some AI changes), or `git diff` to show the AI's modifications (maybe he could even feed the diff back to the AI and say "summarize this commit" for documentation or review purposes).
- **No skipping code review:** A best practice he likely stresses is never auto-merge AI changes without a human glance, at least. Even if tests pass, a quick sanity check via diff is important. Over time, trust can build, but even then, unexpected weirdness can slip through if not checked.
- **Commit Messages:** Possibly he uses the AI to help write commit messages. That's a minor detail but can be useful. The AI that generated the code knows what it did; you can ask it "give me a concise commit message for these changes" and it might output something nice like "Add new field X to User model and expose it in API (includes migration and tests)".

One can imagine a near-future workflow: AI drafts a change, shows diff, developer edits diff or comments, AI updates, once satisfied, developer says "yes, commit this with message '…'." Some of that is already possible with Claude's toolset. In fact, Claude can operate in a mode where it automatically applies changes to files. Those changes could be staged in git directly by the CLI, and then the user just commits.

So Matt's workflow likely involves cycling through:

1. **Plan** (maybe as a git issue or a mental plan).
2. AI generates code.
3. Diff review and testing.
4. Commit.
5. Repeat for next step.

Another best practice: **One change at a time.** If an AI suggests multiple unrelated edits, separate them. It's tempting to ask "fix these 5 bugs" and commit all fixes together. But if one fix fails, it complicates rollback. Better to tackle one bug-fix at a time – commit, test, then next.

Also, using git helps to track how much the AI is doing. Matt's use of `ccusage` indirectly ties into git: he views his AI usage like commit activity. If one sees a flurry of large commits all authored by "AI", that might raise eyebrows. So keeping them granular and understandable is important for collaboration too (if working with others, they need to read those commits).

In short, the motto is: **treat AI contributions like any other code contributions.** Use branches, small commits, code reviews, testing, and clear commit messages. The workflow doesn't fundamentally change – you're still using git as intended – but you need to be disciplined to integrate the AI smoothly into it rather than letting the AI spit out a huge diff you dump into main.

17. I understand Claude Code has a "Plan Mode" that lets it outline steps before coding. How do you use this, and does it improve the results?

Plan Mode is indeed a feature of Claude Code that many users, including Matt, find extremely useful. How it works: instead of immediately giving you code or executing changes, Claude will first present a plan – essentially a to-do list or outline of what it *intends* to do to fulfill your request. You can think of it as the AI taking a moment to **think out loud (or on paper)** before acting.

How Matt likely uses it: Whenever he's about to have Claude do a non-trivial change (something that touches multiple files or has several steps), he toggles Plan Mode (via Shift+Tab as noted). He then issues his instruction. For example, "Add a new optional field `nickname` to the User profile and update all relevant parts of the app." In Plan Mode, Claude might respond with something like:

- Plan:
    1. Modify `models.py` to add `nickname` field to UserProfile model (with default null).
    2. Create and apply a Django migration for this change.
    3. Update `forms.py` to include `nickname` in the UserProfileForm.
    4. Update `views.py` or wherever profiles are created/updated to save the nickname.
    5. Adjust template `profile.html` to display the nickname if present.
    6. Write/adjust unit tests for profile to cover nickname.

It will list these steps and perhaps ask "Continue with coding?" or wait for confirmation. Matt would review this plan. Maybe he notices a missing step (e.g., "Oh, we also need to update the API serializer for UserProfile"). He can then tell the AI to add that or just keep it in mind to do manually. Or he might see a step that's not needed. He can discuss with the AI: "We don't need a form change because nickname won't be edited via that form, skip that." So they iterate until the plan looks solid.

Only then does he say "Looks good, implement." At that point, Claude exits Plan Mode (or goes into an execute mode) and carries out the steps, making code changes one by one, often presenting a summary or diff of each as it goes.

The benefit of Plan Mode is it **reduces miscommunication and surprises**. Without it, the AI might jump into writing code and possibly do something unexpected like rename unrelated things or choose an odd approach. Plan Mode gives the developer a chance to correct the course early. It's akin to a contractor showing you a blueprint before building the house addition.

From what users say, Plan Mode can lead to better results because the AI's actions are more coherent and it's less likely to forget a step. Also, the user gets a high-level view and can spot omissions. It's far easier to fix a plan step than to refactor code after the AI wrote it incorrectly.

Matt specifically highlighted using Plan Mode (Shift-Tab). He probably learned that it's a huge time-saver to confirm plans. There might be times he doesn't need it (for tiny tasks it might be overkill), but for anything complex, it's now a part of his routine.

One thing to note: some have mentioned Plan Mode currently might automatically switch to an "auto-accept" mode after planning which they find dangerous. So best practice: keep an eye when it transitions out of Plan Mode. You want to ensure you still approve each change. Possibly the tools have improved that.

So yes, Plan Mode improves results by making the AI more step-by-step and giving the human oversight at the plan level. It's almost like pseudo-code or a design outline from the AI before actual code. This reduces unnecessary rework and ensures the AI's mental model of the task aligns with the developer's.

From Matt's perspective, using Plan Mode is one of those advanced habits that has allowed him to tackle bigger tasks with the AI safely. For someone new, it's a tip: **use the AI's planning capabilities, don't always rush into code generation.** A well-laid plan can save a lot of headaches.

18. Claude also has two model options, Opus and Sonnet. Could you explain the difference between them and how you decide when to use one versus the other?

Anthropic's Claude has tiered models, and specifically with Claude Code (the coding assistant), they've referred to models like **Claude Code (Opus)** and **Claude Code (Sonnet)**. Essentially, these correspond to different sizes or capabilities: Opus is the more powerful (and expensive) one, while Sonnet is cheaper and somewhat lighter.

From what we've gathered (and Matt's notes):

- **Opus** might have a larger capacity (perhaps akin to "Claude 2" level with maybe 100k context and more horsepower). It might be better at complex reasoning, deep planning, or understanding very large contexts. This could make it ideal for design discussions, architectural decisions, or tricky debugging where you need the model to really comprehend a lot of information and think it through.
- **Sonnet** is described as "perfectly adequate" for needs and even preferred for code output by some. It's likely faster and cheaper, and sometimes giving more direct answers (maybe less verbose). Armin Ronacher mentioned he exclusively uses the cheaper Sonnet model and prefers its outputs to Opus's. Perhaps Sonnet is more deterministic or straightforward in writing code, whereas Opus might be a bit too verbose or creative (sometimes to a fault) in coding tasks.

Matt says he switches between Opus (for design) and Sonnet (for implementation). So a scenario might be:

- When starting a task, he engages Opus to brainstorm how to implement something or to review a design document. Opus might provide a thorough plan or catch nuances.
- Once the plan is set and it comes down to cranking out code, he switches to Sonnet to actually generate the code diffs. Sonnet might make fewer unnecessary changes and stick to the plan with less "imagination," which is good for coding.

It's a bit like having two AI team members: one is the senior architect (Opus) that gives you the big picture and detailed analysis, and the other is the efficient coder (Sonnet) that writes the code cleanly according to spec. This is a sophisticated use of the tools – many users might just stick to one model, but Matt has identified strengths in each.

In practice, how to switch? The Claude CLI likely has a flag to choose model. He might run something like `claude --model=opus` when he wants it, then revert to default Sonnet for normal work. Or possibly the UI has a toggle.

This also touches cost: Opus is $100/month for the max subscription as he noted, whereas Sonnet usage might be included or cheaper. Using Sonnet primarily could be budget-conscious too, while pulling out Opus when needed.

For the audience: he'll probably explain that from a user perspective, **Opus might be better at complex or creative tasks, Sonnet is faster and quite good enough for most coding tasks**. If you have access to both, you can save Opus for when the problem is particularly gnarly (like cross-file refactor that requires deep understanding or generating a big chunk of code where quality matters). If Sonnet ever gives subpar result, maybe he'd try again with Opus to see if it handles it better.

It's somewhat analogous to how some people used GPT-4 for planning and GPT-3.5 for execution to save time/cost: GPT-4 (like Opus) for the hard part, GPT-3.5 (like Sonnet) for the straightforward stuff.

Matt's experience will give insight into how noticeable the difference is. It might also highlight that more powerful isn't always better – sometimes the simpler model is more predictable. And he apparently "prefers Sonnet's outputs" which suggests Opus might sometimes over-complicate or do things not to his liking.

So summarizing: Opus vs Sonnet differ in size/capability; Matt uses Opus for high-level "thinking" tasks and Sonnet for coding tasks. This way, he optimizes for both quality and efficiency. Listeners will learn that if an AI platform offers model choices, it can be worth matching the task to the model's strength, rather than assuming the biggest model should do everything.

19. Beyond your personal workflow, there are many emerging community-driven practices in agentic coding – for example, using Markdown checklists in prompts, tracking AI usage with tools like ccusage, or even orchestrating multiple AI agents with distinct roles (some folks use different terminal color schemes to distinguish them). Have you explored any of these techniques, and what do you think of their value?

Indeed, the community around AI coding has been sharing a lot of clever tips and "hacks" to get the most out of these tools. Let's break down the examples and get Matt's take:

- **Markdown Checklists:** Some users prompt the AI with a checklist like `- [ ] Step 1: do X\n- [ ] Step 2: do Y\n...` and ask the AI to fill it in or tick them off as it completes tasks. This is a way to force the AI (especially GPT-based ones) into a structured approach. People swear that by having the AI maintain a checklist, it doesn't lose track of subtasks and you can easily discuss each item. Matt mentioned some people love Markdown checklists but he personally hasn't found it necessary, although he "keeps his changes" list which might mean he does maintain a kind of changelog or to-do list himself. He might say that because Claude's Plan Mode already structures tasks, an explicit checklist in the prompt is less needed. But it's interesting for those using models that don't have a dedicated plan feature (like vanilla GPT-4) – a checklist can emulate that planning.
- **Tracking AI usage with ccusage:** Matt does use **ccusage**. It's a tool created by a community developer (Ryo) that analyzes Claude Code's logs to show daily token usage, cost, etc., in nice reports. Matt compares it to a git commit visualization – maybe it shows a calendar heatmap of usage or a bar chart of tokens per day. The value he finds is likely in reflecting on productivity and just satisfying curiosity ("how much am I leaning on AI?"). It might also be a bit of gamification – if he sees a slow day, maybe he thinks he could have utilized the AI more. Or if he sees a spike, he might correlate it with output (like "yeah that was the day I finished feature X with Claude's heavy help."). For teams, tools like this could track cost usage too, which matters if using a paid API. So he'd probably say yes, he uses ccusage and finds it useful for self-monitoring, but it's not directly affecting the code quality – it's more about process improvement.
- **Multiple AI agents with distinct roles (and color schemes):** This is on the cutting edge. Some people experiment with having two AI agents collaborate, or one AI to plan and another to execute (beyond just two modes of the same AI). For example, one might use GPT-4 for one part and Claude for another, or two instances of Claude with different system prompts (one as "QA reviewer", one as "coder"). Using different color schemes in the terminal or editor simply helps the human differentiate the source of outputs – e.g., green text is from Agent A and blue text is from Agent B. Matt referenced a tweet by Amit (likely Amit Chaudhary) showing different colors for different agents in a setup, which implies some people find value in multi-agent setups to cross-verify or parallelize tasks.

Has Matt tried multi-agent stuff? Possibly to some extent. Since he mentioned it, he's aware of it, but he might not heavily rely on it yet. He might share an opinion like: *It's an interesting idea, but managing multiple agents can be complicated.* If one agent is writing code and another is reviewing it, that second agent might not be as reliable as a human reviewer, so you still need to double-check. There's research that having AIs critique each other can improve outcomes, but in practice it can also double the cost and complexity.

He might also mention that Claude's own features (Plan mode, etc.) reduce the need to have a separate "planner agent." However, the idea of specialized agents (one for front-end, one for back-end for instance) could be powerful if orchestrated well. In any case, using colors or separate windows to keep track is just an organizational tip for the developer to not get confused who said what.

Another community practice: some use **role prompts** heavily, like telling the AI it is an expert in X before coding, or using **Rubber Duck debugging** (just explaining the problem to the AI to see if you yourself catch the bug while explaining). Matt might have done rubber-ducking with the AI – it's a known beneficial use.

He might say: He's seen those techniques, tried some, but what he settled on is what he described: using Plan Mode, switching models, etc. The Markdown checklist specifically he said he hasn't found necessary because he can keep track of changes mentally or with simpler notes. That suggests he doesn't want to over-engineer the prompt if not needed – which is fair; if the agent is already doing well, adding a formal checklist might slow things down.

As for multiple agents, if he hasn't had a strong need, he might not complicate his setup. But he definitely adopted the color scheme tip – which indicates at least occasionally multiple agents or instances (maybe he sometimes uses GPT-4 in parallel for some tasks, who knows).

He can comment on each:

- Checklists: good for some folks, but Claude's planning covers it for him.
- ccusage: yes, great for awareness of usage.
- Multi-agents: intriguing, maybe not mainstream yet, but could grow. The color scheme trick is basically an anecdote about how seriously some take it – giving each agent a visual identity.

By exploring these, listeners learn there's a whole community optimizing and tinkering with workflows, and that using an AI in coding isn't one-size-fits-all – you can get creative with how you prompt and manage it. Matt's perspective will likely be practical: use what adds value, but don't adopt hacks just for novelty if they don't actually help your case.

20. Looking ahead, what open-source projects or tools in the agentic AI space are you most excited about? And in a broader sense, as AI agents become more capable, how do you see the role of human developers evolving in the future?

To wrap up, we want Matt's forward-looking insights. He's mentioned interest in where this is going, especially with open source tools. Some open-source projects likely on his radar:

- **Anthropic released Claude Code CLI as open source** (I believe parts of it are open, or at least the ecosystem around it like the MCP tool servers, etc., are open source). For instance, he referenced things like the `playwright-mcp` tool or others in Claude's ecosystem.
- **OpenCode** (by Cursor team or community) is open source. It's an attempt at an open Claude Code alternative.
- **goose.ai** and **Devin.ai** which Armin mentioned, possibly open or at least alternatives.
- The **Pocket Flow** framework we saw is open source (just 100 lines, interesting approach).
- **LangChain** and similar frameworks – though some find them too much, they are open and being evolved constantly.
- **Llama 2** and other open models – with tools like Text-Generation-WebUI or LocalAI for running agents locally. Perhaps by 2025, there are open source models approaching GPT-4 level for coding. That could democratize agentic programming without relying on big cloud APIs.
- **AutoGPT and successors** – open communities improving them (AutoGPT was open source).
- There's also **Code Llama** (open AI model for code), which could be part of open source agent stacks.

Matt's excited likely because more of these tools are shifting to the developer's control. DigitalOcean even now offers LLM APIs, indicating he's in tune with making AI accessible as a service but possibly also supporting open initiatives (DigitalOcean might be enabling easy deployment of open models).

He may mention that open source tools will allow customization and deeper integration into developers' own environments and CI pipelines. For example, hooking an open agent into your GitHub Actions to automatically propose bug fixes on PRs.

Now, the **role of human developers** question is quite profound. With agentic AI advancing, do developers become less hands-on coders and more like architects, reviewers, and maintainers of quality? Many suspect that routine coding will be largely automated, and human devs will focus on defining problems, doing high-level design, and handling the ambiguous parts of projects. Matt might say something akin to: *Developers will move up the abstraction ladder – spending more time on conceptual work, communication of requirements, and verifying AI outputs, and less time on writing boilerplate or repetitive code.*

One could argue developers become **"AI orchestration engineers"** to some extent – knowing how to use these AI tools effectively is a skill in itself (like prompt engineering, curating the context, etc.). But also, developers will still need to do the critical thinking to break down problems in a way an AI can understand.

It's possible he'll say developers aren't going away – rather, their role shifts to be more like a **coach or editor** for the AI. The AI can draft the code, but a human ensures it's correct, ethical, secure, and aligned with user needs. Also, developers might focus more on tasks AI is bad at: e.g., figuring out exactly what the user or business needs (requirements engineering) and then translating that into tasks for the AI. Or dealing with novel problems where AI has no training data.

He might also mention that as AI takes over simpler tasks, there's an opportunity for developers to accelerate innovation – you can attempt more ambitious projects with the same team because AI handles the grunt work.

If he's optimistic: we'll see a flourishing of productivity, maybe single developers can create complex systems (like how one person can do more now with cloud services + AI than a whole team could a decade ago). If he's cautious: he might note we have to adapt our education and thinking, because the "coding" skill might be overshadowed by an "AI supervision" skill.

Either way, he'll likely encourage developers to embrace these tools rather than fear them, because those who learn to use AI will excel in this new landscape. Possibly referencing that quote from the HN friend: *"Today is the worst day you will have with this technology for the rest of your life."* – meaning it only gets better from here, so future devs will have even more powerful aides.

In summary: he's excited about open source tooling which will give developers more control and reduce dependency on any single provider. And he envisions that developers' roles will evolve to higher-level thinking, oversight, and leveraging AI to do more, rather than manually doing every little coding task. This sets a hopeful and forward-looking tone to end the discussion, fitting for a Talk Python episode conclusion.