AI-Based Development

From “AI helps me” to “AI builds for me”

Andrey Meshkov
CTO and Co-Founder of AdGuard
am@adguard.com
@ay_meshkov

SLIDE 1

AI-Based Development

SPEAKER NOTES

Hello everyone, I'm Andrey Meshkov, CTO and Co-Founder of AdGuard. Today I want to talk about something that affects every single developer and tester in this room — how we work with AI.

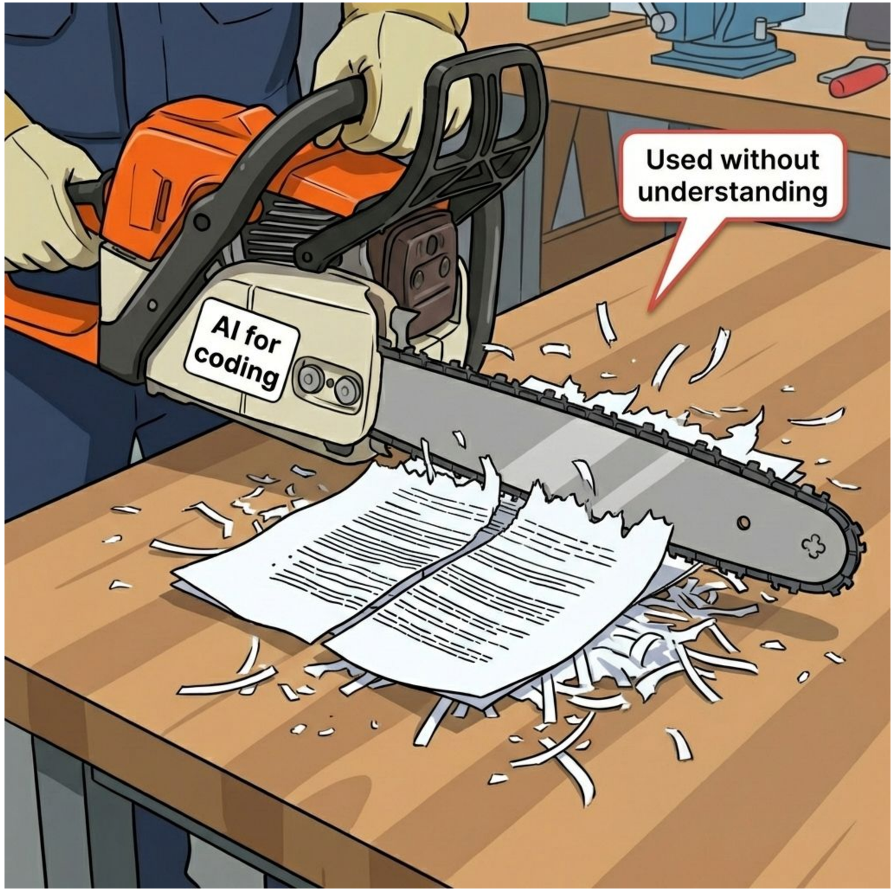
The subtitle of this talk is "From AI helps me to AI builds for me," and that captures the journey we're on. Over the past three years, AI has gone from being a novelty to being an essential part of how software gets built. But here's the thing — most of us are still using it wrong. We're leaving enormous amounts of productivity on the table because we haven't updated our mental model of what AI can do and, more importantly, how we need to change our workflow to take full advantage of it.

By the end of this talk, I want you to walk away with three things: a clear understanding of the correct mental model for working with AI, a practical workflow called Spec-Driven Development that you can start using tomorrow, and knowledge of the specific tools we provide here at AdGuard.

This isn't theoretical — we'll finish with a live workshop where we'll implement a real feature using these techniques. Let's dive in.

Why this talk

- AI already changed how we write code
- But many teams are using it **incorrectly**
- Result: disappointment, distrust, wasted potential



SLIDE 2

Why This Talk

SPEAKER NOTES

Let me start with why this talk is necessary. AI has already fundamentally changed how we write code. That's not a prediction — it's a fact. GitHub reports that Copilot users accept AI-generated suggestions in over 30% of their code.

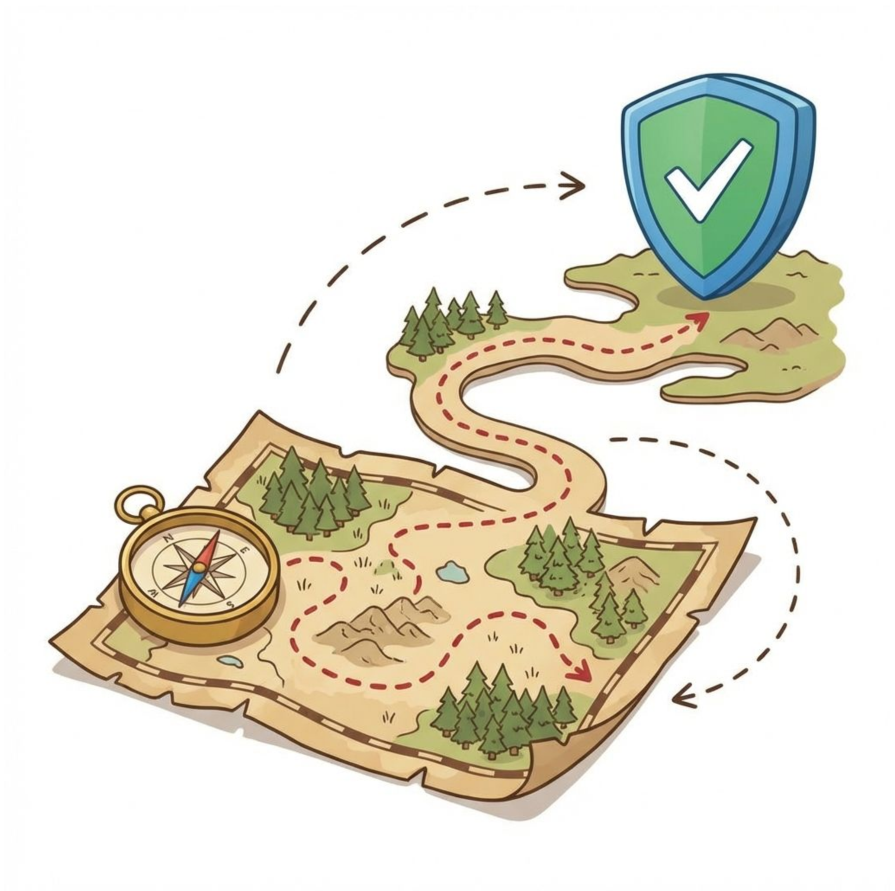
But here's the problem: many teams are using AI incorrectly. They treat it like a fancy autocomplete or a glorified StackOverflow, and when the results don't meet their expectations, they conclude that AI isn't ready yet. The result is disappointment, distrust, and wasted potential.

Look at the image on this slide — it's a chainsaw being used to cut paper. The tool is incredibly powerful, but it's being used without understanding. That's exactly what's happening with AI in many development teams. The chainsaw isn't broken — the person using it just hasn't learned how to use it properly.

This is a pattern I see across the entire industry, and it's especially frustrating because the fix isn't complicated. It doesn't require new technology or expensive tools. It requires a change in how we think about AI and how we structure our work. And that's what we're going to fix today.

What this talk is about

- The **correct mental model**
- The **right workflow**
- How we can use AI at **AdGuard**



SLIDE 3

What This Talk Is About

SPEAKER NOTES

This talk is structured around three main themes.

First, the correct mental model — how should you think about AI as a development tool? What is it actually doing under the hood, and what does that mean for how you interact with it? Getting the mental model right is the foundation for everything else.

Second, the right workflow — once you have the right mental model, how do you translate that into a practical, day-to-day workflow? This is where Spec-Driven Development comes in, and I'll show you exactly how to apply it. This is the most actionable part of the talk.

Third, we'll look at how we can use AI specifically at AdGuard — what tools we provide, what's available today, and what's coming. I'll cover Windsurf, Codex, our Bitbucket AI reviewer, and the SDD workflow templates.

Think of this as a roadmap. We need to understand where we are on this map, where we want to go, and the best route to get there. The compass on this slide represents that orientation — without it, you'll wander aimlessly, trying random AI tools and getting inconsistent results. With it, you'll make steady, measurable progress toward becoming dramatically more productive.

Evolution

How AI-Assisted development evolved

SLIDE 4

Evolution — Section

SPEAKER NOTES

Let's start by looking at how AI-assisted development has evolved over the past few years. I want to walk you through this evolution because understanding where we've been helps us understand where most people get stuck — and why.

3 years ago

- I write code with my hands
- IDE, Google, StackOverflow
- Humans are the bottleneck

This worked fine.
It was slow, but predictable.

SLIDE 5

3 Years Ago

SPEAKER NOTES

Three years ago, our world looked like this. You, the developer, wrote every line of code with your own hands. Your tools were your IDE, Google, and StackOverflow. When you hit a problem, you'd search for a solution, read through forum posts, maybe find a code snippet you could adapt, and then manually type it into your editor.

Humans were the bottleneck in the entire development process — the speed at which software got built was directly limited by how fast a person could think, type, and debug. Everything flowed through your fingers on the keyboard.

And you know what? This worked fine. It was slow, but it was predictable. You knew what you were getting. You knew the quality of the code because you wrote every character yourself. You understood the system intimately because you built it piece by piece.

There was a certain comfort in this approach — a directness and control that many developers still miss. You could point at any line of code and say "I know exactly why this is here." Every decision was deliberate, every line intentional.

But the world was about to change dramatically, and clinging to this model would soon become a competitive disadvantage. The developers and teams that adapted would gain a significant edge over those who didn't.

2 years ago

- ChatGPT appears
- We ask questions
- We copy snippets

AI = StackOverflow++

👉 Many people are **still here**

SLIDE 6

2 Years Ago

SPEAKER NOTES

Then, about two years ago, ChatGPT appeared and everything shifted. Suddenly, we had a tool that could answer coding questions, explain error messages, and generate code snippets on demand.

For most developers, the workflow changed in one specific way: instead of searching Google and StackOverflow, you asked ChatGPT. You'd paste an error message, ask "how do I do X in Python," or request a code snippet, and then you'd copy the response back into your editor.

AI became StackOverflow++. It was faster, more convenient, and often gave better-targeted answers than searching through forum posts. No more wading through five outdated answers to find the one that actually works.

But here's the critical point — and I want you to really hear this — many people are still here. Many developers in this room, and across the industry, are still using AI as nothing more than a better search engine. They ask a question, copy a snippet, and go back to writing code by hand.

That's like buying a Tesla and using it as a storage shed. The tool is capable of so much more, but we're barely scratching the surface because our mental model hasn't evolved with the technology. If this describes your current workflow, don't feel bad — but do recognize that you're leaving enormous productivity gains on the table.

1 year ago

- Agentic coding: Cursor/Windsurf/Claude Code
- “Write this function”
- “Implement this class”
- “Refactor this file”

AI writes **lots of code**

SLIDE 7

1 Year Ago

SPEAKER NOTES

About a year ago, we entered the age of agentic coding. Tools like Cursor, Windsurf, and Claude Code emerged that fundamentally changed the relationship between developer and AI.

Instead of asking questions and copying snippets, you could now give higher-level instructions: "Write this function," "Implement this class," "Refactor this file." The AI would actually produce complete, working code blocks — not just snippets, but substantial implementations.

It could read your existing code, understand context, and generate code that fit into your project. The AI went from answering questions to writing lots of code. This was a paradigm shift. The AI wasn't just a reference tool anymore — it was actively producing deliverables.

This felt like a revolution. You could describe what you wanted in natural language and get working code in seconds. Many developers, myself included, were genuinely excited. It seemed like productivity would skyrocket. Entire functions, classes, even modules could be generated from a brief description.

But then reality set in, and things got complicated. The honeymoon period ended when we started using these tools for real production work, not just demos and toy projects.

What happens next?

- Code doesn't compile
- Logic is wrong
- Edge cases missing
- Tests are weak or absent

Developer conclusion:

"AI can't really code yet"

SLIDE 8

What Happens Next?

SPEAKER NOTES

Here's what happened when people started relying on AI to write substantial amounts of code. The code doesn't compile. The logic is wrong. Edge cases are missing. Tests are either weak or entirely absent.

You'd ask AI to implement a feature, get back something that looked plausible, and then spend hours debugging issues that the AI didn't anticipate. Off-by-one errors, null pointer exceptions, race conditions, incorrect assumptions about your data model — the list goes on.

And it wasn't just small issues. Sometimes the entire approach was wrong — AI chose an architecture that didn't fit your system, or made assumptions about your data that were completely incorrect.

Many developers reached a conclusion at this point: "AI can't really code yet." You can see the frustrated developer on this slide thinking "ChatGPT IS STUPID!" And I understand that frustration. When you ask a tool to do something and it fails repeatedly, the natural conclusion is that the tool doesn't work.

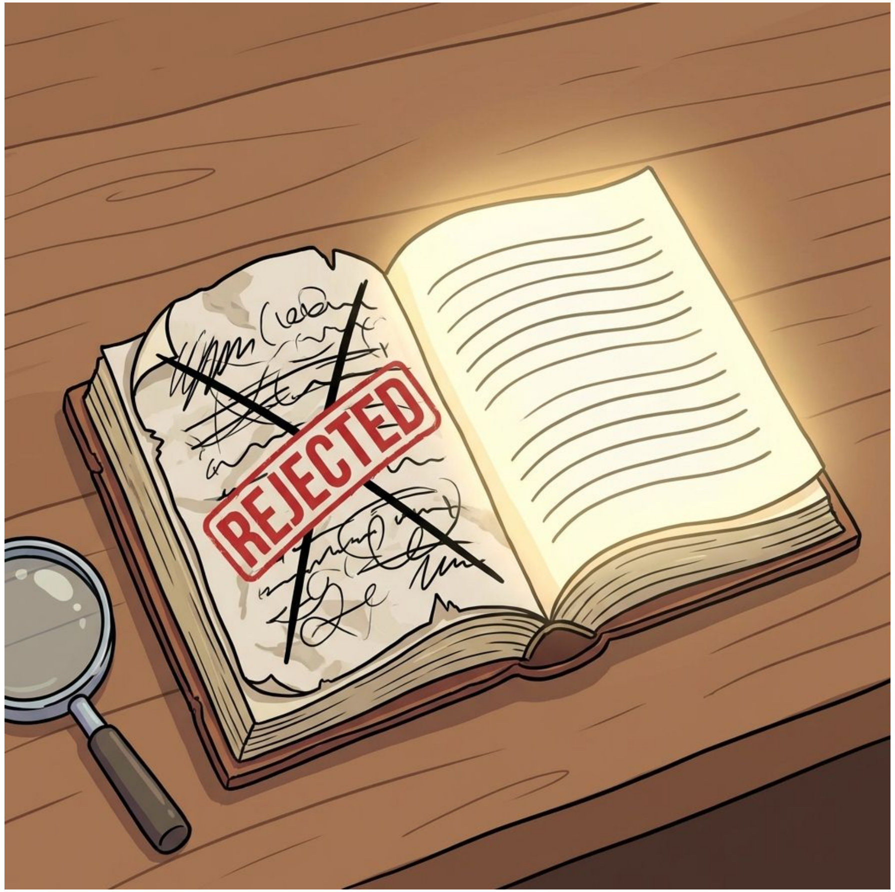
But I want to challenge that conclusion, because it's based on a fundamental misunderstanding of what went wrong. The problem wasn't that AI is incapable — the problem was how we were using it. We were giving it insufficient context, vague instructions, and no way to verify its own output. Then we blamed the tool for the results.

This is a mistake!

You stopped **too early**

- You evaluated AI as a *junior dev*
- But treated it like a *snippet generator*
- You never **changed** your process

AI failing at step 1
≠
AI incapable of finishing the task



SLIDE 9

This Is a Mistake!

SPEAKER NOTES

Dismissing AI because of these failures is a mistake, and here's exactly why. You stopped too early.

Think about what actually happened. You evaluated AI as if it were a junior developer — you expected it to understand your codebase, make good architectural decisions, handle edge cases, and write production-ready code from a one-line instruction. But at the same time, you treated it like a snippet generator — you gave it a brief instruction, expected perfect output on the first try, and threw it away when the output wasn't perfect.

And critically, you never changed your process. You kept working the same way you always had, just with AI generating the code instead of you typing it. Same vague instructions. Same lack of specifications. Same absence of automated verification.

This is like hiring a new team member, giving them zero onboarding, no documentation, no code review process, and then firing them when they make mistakes in their first week. You wouldn't do that to a human, so why do it to AI?

The key insight on this slide is this: AI failing at step one does not mean AI is incapable of finishing the task. It means your process needs to account for iteration, context, and feedback — just like it does with human developers. AI needs guidance, specifications, and feedback loops to produce quality work. Give it those things, and the results are transformative.

AI agents 101

What you need to know about AI agents

SLIDE 10

AI Agents 101 — Section

SPEAKER NOTES

Before we talk about the right workflow, let's make sure everyone understands what AI agents actually are and how they work under the hood. This technical understanding is essential for knowing how to get the best results.

What AI agent is?

- AI agent is a program that:
    - has a reasoning loop powered by an LLM
    - reads context (code, specs, etc.)
    - plans actions
    - decides what to do next
    - asks for or invokes “tools”

SLIDE 11

What AI Agent Is?

SPEAKER NOTES

Let's define what an AI agent actually is, because the term gets thrown around a lot without precision.

An AI agent is a program that has a reasoning loop powered by a Large Language Model, or LLM. It reads context — your code, your specs, your project structure. It plans actions based on that context. It decides what to do next. And it asks for or invokes "tools" to carry out those actions.

Notice that I said "program," not "model." The LLM is the brain, but the agent is the whole system. The agent includes the prompt template, the tool definitions, the memory management, and the orchestration logic that ties it all together.

Cursor is an agent. Windsurf is an agent. Claude Code is an agent. When you type "implement this function" into Cursor, the agent reads your project files, constructs a prompt that includes relevant context, sends it to the LLM, interprets the response, and executes the appropriate actions — writing files, running commands, or asking you for clarification.

Understanding this architecture is crucial because it tells you exactly where the leverage points are. If you want better results from AI, you need to improve the inputs to this loop: better context, better instructions, better tools. The LLM itself is a constant — you can't change how it reasons. But you can dramatically change what it reasons about.

How an AI agent loop works?

1. The program (agent) gives the LLM a prompt:
   “Do X
   Your context: {context}”
2. The context may include:
   a. relevant code, docs
   b. available tools
   c. previous tool outputs and prompts
3. The agent reasons about the task
   based only on that prompt

SLIDE 12

How an AI Agent Loop Works? (Part 1)

SPEAKER NOTES

Let me walk you through the agent loop step by step.

First, the program — the agent — gives the LLM a prompt. This prompt has two parts: the instruction, which is what you asked it to do — "Do X" — and the context, which is everything the agent thinks the LLM needs to know to complete the task.

The context may include relevant code from your project, documentation files, available tools and their descriptions, and the outputs from previous tool calls and prompts. This is important: the LLM sees all of this as one big block of text. It doesn't have separate "memory" for code versus documentation versus your instructions. Everything is flattened into a single prompt.

Step three: the agent reasons about the task based only on that prompt. This is the key constraint — the LLM can only work with what's in the prompt. It doesn't know your unwritten conventions, your team's preferences, or the bugs you fixed last week — unless that information is explicitly included in the context.

This is why context management is so critical, and why tools like Cursor and Windsurf invest heavily in intelligent context selection. The better the context, the better the output. Garbage in, garbage out is more true for AI than for any other tool in your toolbox.

How an AI agent loop works?

1. The LLM responds with:
   a. a **tool call** (run tests, read a file, etc.), or
   b. a **final answer** (task is done)
2. The program (agent) executes the tool call
3. Tool output is appended to the next prompt
4. Repeat until completion

SLIDE 13

How an AI Agent Loop Works? (Part 2)

SPEAKER NOTES

Continuing with the loop. The LLM responds with one of two things — either a tool call or a final answer.

A tool call means the LLM wants to do something: run tests, read a file, search for a pattern in the codebase, write code to a file. A final answer means the LLM believes the task is complete and returns its response.

When the LLM requests a tool call, the program — the agent — executes it. It runs the test, reads the file, or writes the code. Then the output of that tool call is appended to the prompt, and the whole thing goes back to the LLM for the next iteration.

This cycle repeats until the LLM decides it's done and provides a final answer. This is the agentic loop — read, reason, act, observe, repeat.

This loop is why agentic coding is so much more powerful than simple chat. In a chat, you ask one question and get one answer. In an agentic loop, the AI can read your code, try an implementation, run the tests, see they fail, read the error message, fix the code, run the tests again, and keep iterating until everything passes.

It's the difference between asking someone a question and actually delegating a task to them. But here's the catch — the quality of this entire loop depends on the quality of your initial instructions and the context available to the agent. Better inputs lead to fewer iterations and better results.

Capability vs behavior

- AI **is capable** of writing very good code *(often better structured than human-written code)*
- Capability ≠ automatic results
- AI does exactly what **you** set it up to do

Analogy:

- Ferrari can do 300 km/h
- That doesn’t mean you’ll get there without a map

SLIDE 14

Capability vs Behavior

SPEAKER NOTES

This is one of the most important concepts in this entire talk, so please pay close attention.

AI is capable of writing very good code. In fact, it's often better structured than human-written code because it consistently follows patterns, naming conventions, and best practices without getting lazy or taking shortcuts.

But — and this is the crucial distinction — capability does not equal automatic results. Just because AI can write excellent code doesn't mean it will write excellent code every time you ask. AI does exactly what you set it up to do. If you give it vague instructions, it will produce vague results. If you give it precise specifications, it will produce precise implementations.

Let me give you an analogy. A Ferrari can do 300 kilometers per hour. It's an incredibly capable machine. But that doesn't mean you'll get to your destination without a map. Without navigation, without knowing the roads, without a plan for your journey, the Ferrari's speed is meaningless — you'll just get lost faster.

The same is true for AI. The capability is there. The speed is there. What's missing for most developers is the navigation — the clear specifications, the defined constraints, the structured workflow that channels AI's capability into reliable results.

That navigation is what the rest of this talk is about. We're going to build the map that turns AI's raw capability into consistent, predictable, high-quality output.

So what’s your role now?

Old role:

- writing loops
- fighting syntax
- remembering APIs

SLIDE 15

So What's Your Role Now? — Old Role

SPEAKER NOTES

Let's talk about how your role as a developer needs to change.

In the old model, your job was deeply mechanical. You spent your days writing loops, fighting with syntax, and trying to remember API signatures. You'd spend twenty minutes figuring out the exact parameter order for a function you've called a hundred times before but can't quite remember. You'd debug a missing semicolon for an hour. You'd manually write boilerplate code that follows the same pattern you've implemented dozens of times.

Look at the image on this slide — the frustrated programmer, surrounded by syntax errors and endless loops, hands on the keyboard in perpetual battle with the machine. Question marks everywhere. This was the daily reality for most developers.

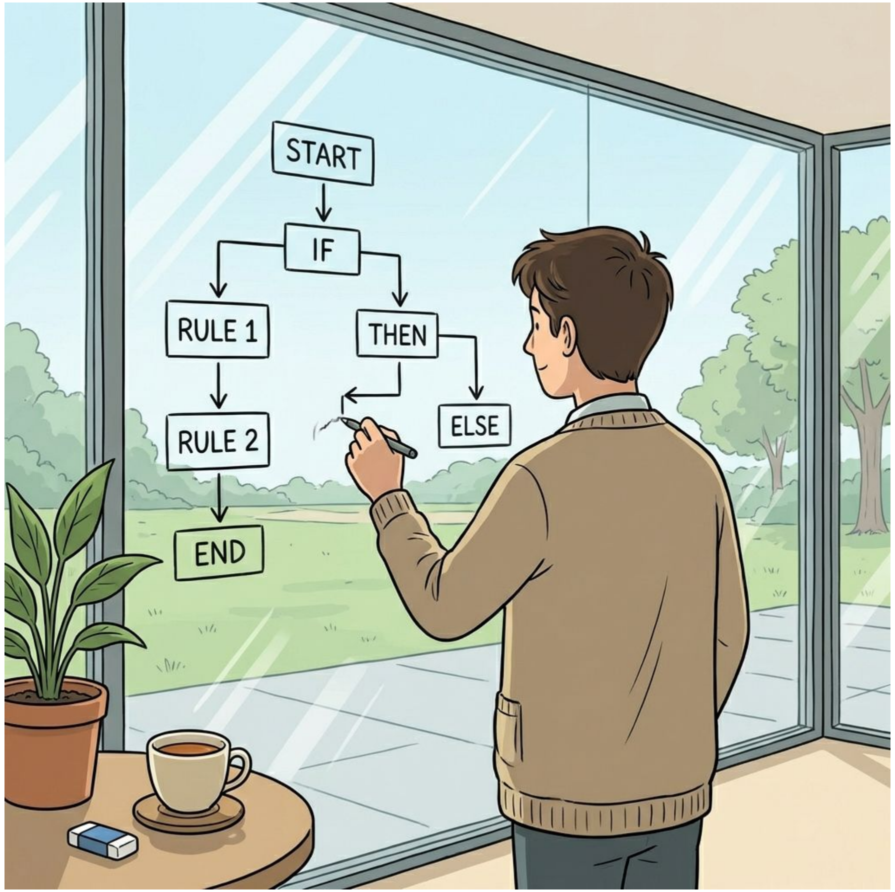
And let's be honest — a lot of this work wasn't intellectually stimulating. It was necessary, but it wasn't the hard part of software engineering. The hard part was always the design decisions: what to build, how to structure it, what patterns to use, how to handle failures. The mechanical typing was just the means of expressing those decisions.

AI has now made the mechanical part almost trivial. Which means the hard part — the design — is all that's left. And that's actually a good thing, because it means we can focus entirely on the work that matters most and requires human judgment.

So what's your role now?

New role:

- defining intent
- setting constraints
- designing behavior
- reviewing results



SLIDE 16

So What's Your Role Now? — New Role

SPEAKER NOTES

Your new role is fundamentally different from what it was before.

Instead of writing code, you're defining intent — clearly articulating what the system should do and why. Instead of fighting syntax, you're setting constraints — specifying the boundaries within which the AI should operate. Instead of remembering APIs, you're designing behavior — defining how components interact, how errors are handled, how data flows through the system. And instead of debugging line by line, you're reviewing results — evaluating whether the AI's output meets your specifications.

Look at the image on this slide. Instead of hunched over a keyboard in a dark room, the developer is standing at a window with natural light, drawing flowcharts on glass, designing systems. There's a cup of tea on the table. This is the new posture of software development. You're the architect, the designer, the reviewer. You're working at a higher level of abstraction, and the AI handles the implementation details.

This shift requires different skills. You need to be better at communication — explaining what you want precisely. Better at specification — writing clear, unambiguous requirements. Better at systems thinking — understanding how components interact and what can go wrong.

The developers who thrive in this new model won't be the fastest typists — they'll be the clearest thinkers. And that's a much more rewarding way to work.

The uncomfortable truth

“I do not write code anymore
And you should not”

- You design systems
- AI writes code
- You verify results

SLIDE 17

The Uncomfortable Truth

SPEAKER NOTES

Here's the uncomfortable truth that many developers resist: "I do not write code anymore. And you should not either."

I know this sounds provocative, and that's intentional. When I say "I don't write code," I don't mean I'm not involved in software development. I mean my role has shifted. I design systems. AI writes code. I verify results.

Look at the meme on this slide — you've probably seen this image before. Everyone is standing around watching one person do the actual work. In the old world, the developer was the person in the hole doing the digging. In the new world, you're the person directing the work — you're the one who decided where to dig, how deep, and what the hole is for.

Some of you might feel uncomfortable with this. Writing code is part of your identity. It's what you trained to do, what you enjoy. But I'd argue that what you actually trained to do is solve problems. Code was just the medium. Now you have a more powerful medium, and your problem-solving skills are more valuable than ever.

The developers who resist this shift — who insist on writing every line by hand — will be left behind. Not because they're bad at coding, but because they're voluntarily handicapping themselves in a world where AI can do the implementation ten times faster. Embrace the shift. Your engineering mind is the valuable part — not your typing speed.

System design is now the main job

- AI changes **how** code is written
  It does **not** change **what** software is.
- Your primary responsibilities:
  - decompose the problem
  - define boundaries
  - choose abstractions
  - define data flow and failure modes

“If the system design is bad,
AI will faithfully implement a bad system.”

SLIDE 18

System Design Is Now the Main Job

SPEAKER NOTES

If AI changes how code is written, what hasn't it changed? It has not changed what software is.

Software is still about solving problems. It's still about handling complexity, managing state, dealing with failures, and serving users. AI changes the implementation layer, but the design layer is entirely yours.

Your primary responsibilities now are: decompose the problem — break large, complex tasks into smaller, well-defined pieces that AI can implement independently. Define boundaries — specify clear interfaces between components so AI knows where one module ends and another begins. Choose abstractions — decide on the right data structures, design patterns, and architectural approaches for your specific situation. Define data flow and failure modes — specify how data moves through the system and what happens when things go wrong.

Here's the quote I want you to remember from this entire talk: "If the system design is bad, AI will faithfully implement a bad system."

AI is an excellent executor. It will do exactly what you ask. If your architecture is wrong, AI will build that wrong architecture beautifully, with clean code, good variable names, and comprehensive tests that all pass — for the wrong system. The quality of the software is determined by the quality of your design, not the quality of the code. AI handles code quality. Design quality is on you.

Rule #1: You must know how to do it by hand

- If you don’t understand:
  - how code is written
  - how systems are built
  - how bugs appear
- Then:
  - you cannot design a system
  - you cannot verify AI output
  - you cannot tell “almost correct” from “dangerous”

“AI does not replace understanding”

SLIDE 19

Rule #1: You Must Know How to Do It by Hand

SPEAKER NOTES

This brings me to the first fundamental rule of working with AI: you must know how to do it by hand.

This might seem contradictory — I just told you to stop writing code, and now I'm telling you that you need to know how. But there's no contradiction.

If you don't understand how code is written, how systems are built, and how bugs appear, then you cannot design a system for AI to implement. You cannot verify whether AI's output is correct. And you cannot tell "almost correct" from "dangerous."

Think about it: if AI writes a function that looks reasonable but has a subtle race condition, how would you know? If AI chooses an O(n-squared) algorithm where an O(n log n) solution exists, how would you catch that? If AI's database query is missing an index hint and will time out in production, how would you spot it?

You need deep technical knowledge — not to write the code yourself, but to be an effective reviewer and designer. Without understanding, you're just blindly accepting whatever AI produces and hoping it works.

The quote at the bottom of this slide sums it up: "AI does not replace understanding." AI replaces typing. It replaces searching for syntax. It replaces boilerplate. But it absolutely does not replace the understanding you need to build good software. Junior developers who think they can skip learning fundamentals because "AI will handle it" are making a dangerous mistake.

Rule #2: Vague intent is useless

- Bad instruction:
  - “Make it fast”
  - AI cannot follow this
- Good instruction:
  - define what “fast” means
  - define constraints and tradeoffs
  - explain how AI can verify it (unit-tests are **very** important)

SLIDE 20

Rule #2: Vague Intent Is Useless

SPEAKER NOTES

Rule number two: vague intent is useless. This is the single most common mistake I see developers make when working with AI agents.

They give instructions like "Make it fast" or "Fix the bug" or "Improve performance" and then wonder why the results are mediocre.

Let me explain why this fails. AI cannot follow vague instructions because vague instructions don't contain enough information to determine the right action. "Make it fast" — fast compared to what? Fast in terms of latency, throughput, memory usage? Fast for which operations? What's the acceptable trade-off?

A good instruction is specific and measurable. Instead of "Make it fast," you should: define what "fast" means — for example, p95 latency under 10 milliseconds. Define constraints and tradeoffs — for example, we can use more memory but not more than 500MB per process. And this is crucial — explain how AI can verify it. Unit tests are very important here. If you give AI a clear performance target and a test that validates that target, the agent can iterate until the test passes. Without a test, the agent has no way to know if its changes actually improved anything.

Think of it this way: you wouldn't tell a human developer "make it fast" and walk away. You'd discuss requirements, agree on metrics, and define acceptance criteria. AI needs the same level of specificity — actually, it needs more, because it can't read between the lines or make assumptions based on your organizational context.

Example: performance task

- ❌ Vague:
    - “Optimize performance”
    - “Make it faster”
- ✅ Actionable:
    - Reduce memory allocations in hot path (define it)
    - No extra heap allocations per request
    - p95 latency < 10ms
    - Start with writing a test that validates the requirements

Example: Performance Task

SPEAKER NOTES

Let me give you a concrete example of the difference between vague and actionable instructions.

The vague version: "Optimize performance" or "Make it faster." These instructions are essentially useless because they give the AI no direction. It might start optimizing the wrong thing, make trade-offs you disagree with, or produce changes that are impossible to verify.

Now here's the actionable version. "Reduce memory allocations in hot path" — and we define exactly what the hot path is. "No extra heap allocations per request" — this is a specific, verifiable constraint. "P95 latency less than 10 milliseconds" — this is a measurable target. And — this is the critical piece — "Start with writing a test that validates the requirements."

By instructing AI to write the test first, you establish an automated verification mechanism. The AI can then iterate on the implementation, running the test after each change to confirm it's making progress toward the goal. The test becomes the acceptance criteria in executable form.

Notice how the actionable version reads almost like a specification document. That's not a coincidence. The more your instructions look like specs, the better AI performs. This brings us directly to the central concept of this talk: Spec-Driven Development.

But this is harder, right?

- Designing systems is:
    - harder than writing code
    - requires more experience
    - requires deeper understanding
    - feels slower at first
- So the obvious question is:
    - “If system design is harder than coding, what can we do about that?”

SLIDE 22

But This Is Harder, Right?

SPEAKER NOTES

At this point, some of you might be thinking: "Andrey, you're telling me to stop writing code and start designing systems instead. But designing systems is harder than writing code."

And you're absolutely right. Designing systems is harder than writing code. It requires more experience — you need to have built enough systems to recognize patterns and anti-patterns. It requires deeper understanding — you need to know not just how things work, but why they work that way and what breaks when assumptions change. And it feels slower at first — writing a specification takes more upfront thought than just starting to code.

So the obvious question is: "If system design is harder than coding, what can we do about that?" How do we make this transition manageable? How do we help developers who are used to jumping straight into code learn to design first and code second?

The answer is not to make design easier — design is inherently complex because software is complex. The answer is to create a structured process that guides you through the design phase, ensures you cover all the important aspects, and produces artifacts that AI can directly use.

That process is what we call Spec-Driven Development, and it's the topic of the next section.

Spec-Driven Development

Spec-Driven Development (SDD) to the rescue!

SLIDE 23

Spec-Driven Development — Section

SPEAKER NOTES

Now let's talk about Spec-Driven Development — SDD — the methodology that ties everything together and makes AI-assisted development actually work. This is the most actionable part of the entire presentation.

What is Spec-Driven Development?

- Instead of:
  - “Write me a function that does X”
- You write:
  - what the system should do
  - constraints, non-goals, edge cases
  - acceptance criteria
- Then AI:
  - implements
  - iterates
  - refactor

SLIDE 24

What Is Spec-Driven Development?

SPEAKER NOTES

Spec-Driven Development, or SDD, is a fundamental shift in how you interact with AI.

Instead of saying "Write me a function that does X" — which is a vague, implementation-level instruction — you write a specification. The specification defines three things.

First: what the system should do — the functional requirements, the behavior, the expected outputs for given inputs. Second: constraints, non-goals, and edge cases — what the system should not do, what's out of scope, and what happens in unusual situations. Third: acceptance criteria — how you know the implementation is correct, the measurable outcomes that define success.

Once you have this specification, AI takes over the implementation. It implements the feature according to the spec. It iterates when things don't work. It refactors when the implementation needs cleanup. All guided by the specification you wrote.

The key insight is that the spec is the interface between human intent and AI execution. Without it, AI guesses. With it, AI has a clear target.

Think of SDD like Test-Driven Development, but at a higher level. In TDD, you write the test first and then implement until the test passes. In SDD, you write the spec first and then AI implements until the spec is satisfied. The spec is more comprehensive than a test — it includes not just verification criteria but also design decisions, constraints, and context that guide the implementation.

“But my AI agent has planning mode”

Most AI agents have a **planning mode** .

What it really is:

- the model writes a short plan for itself
- a lightweight attempt at SDD
- usually implicit and informal

This is:

- ✅ absolutely fine for small tasks
- ❌ not enough for bigger ones

SLIDE 25

"But My AI Agent Has Planning Mode"

SPEAKER NOTES

I know what some of you are thinking: "My AI agent already has a planning mode. Cursor has it, Claude Code has it, Windsurf has it. Doesn't that already do what you're describing?"

Let me explain what planning mode actually is in most AI agents. The model writes a short plan for itself before starting to code. It's a lightweight attempt at Spec-Driven Development. The plan is usually implicit — it's buried in the agent's internal reasoning — and informal — it's not a structured document you can review or refine.

This is absolutely fine for small tasks. If you're asking AI to "add a null check here" or "rename this variable across the project," the built-in planning mode is sufficient. No need for a formal specification.

But for bigger tasks — implementing a new feature, redesigning a module, adding a new API endpoint with business logic — the built-in planning mode is not enough. It doesn't capture the level of detail needed for complex work. It doesn't involve you in the planning process. And it's not a persistent artifact you can reference later or share with your team.

SDD gives you a formal, reviewable, persistent specification that captures the full complexity of the task. It's the difference between a developer thinking "I'll probably add a cache here" and a written design document that specifies the caching strategy, eviction policy, consistency guarantees, and failure modes.

Why SDD is mandatory with AI

- Without specs:
  - AI guesses
  - you debug guesses
  - trust disappears
- With specs:
  - AI has intent
  - AI self-corrects
  - AI refactors safely
  - AI explains decisions

“Specs are how you program the AI.”

SLIDE 26

Why SDD Is Mandatory with AI

SPEAKER NOTES

Let me make the case for why SDD isn't just nice to have — it's mandatory when working with AI.

Without specs, here's what happens. AI guesses — it makes assumptions about what you want based on patterns it's seen in training data. Sometimes those guesses are correct, but often they're not. You debug guesses — you spend your time figuring out why AI made a particular choice and whether it aligns with what you actually needed. Trust disappears — after a few rounds of AI guessing wrong, you stop trusting it and go back to writing code yourself. The whole effort was wasted.

Now, with specs, everything changes. AI has intent — it knows what you want because you've explicitly stated it. AI self-corrects — when something doesn't match the spec, the agent knows it needs to try a different approach. AI refactors safely — it can make changes with confidence because the spec defines what should remain constant. AI explains decisions — it can reference the spec to justify its choices.

Here's the phrase I want you to remember: "Specs are how you program the AI." When you write a specification, you're not writing documentation — you're programming. The spec is your source code, and the AI is the compiler that turns it into an implementation. The better your spec, the better the output. A vague spec produces vague code, just like vague source code produces unpredictable behavior.

SDD is not written alone

- ❌ Spec-Driven Development is **not**:
  - you writing a perfect spec upfront
  - then handing it to AI
- ✅ Correct workflow:
  - You start the spec or ask AI to write it
  - You review and refine the spec together
  - The spec evolves before code exists
- Why this way?
  - It is much easier to notice a design flaw
  - Generally easier than reviewing code

SLIDE 27

SDD Is Not Written Alone

SPEAKER NOTES

One common misconception I want to address: Spec-Driven Development does not mean you sit down alone, write a perfect specification, and hand it to AI. That's waterfall, and it doesn't work — for the same reasons waterfall never worked: you can't anticipate everything upfront.

The correct workflow is collaborative. You start the spec, or even better, you ask AI to write it. You might say: "Based on this JIRA ticket and the existing codebase, draft a specification for implementing feature X." AI generates an initial spec.

Then you review and refine the spec together. You and AI go back and forth — you add constraints AI missed, AI points out edge cases you didn't consider. The spec evolves before any code exists.

Why is this approach better? Because it is much easier to notice a design flaw in a specification than in code. When you review code, you have to mentally simulate its execution, think about state, consider concurrency — it's cognitively expensive. When you review a spec, you're reading plain language descriptions of behavior and checking them against your understanding of the requirements.

It's generally easier than reviewing code. Catching a design flaw at the spec stage saves hours of implementation and debugging time. One paragraph changed in a spec might save a day of refactoring in code. The investment in upfront spec work pays massive dividends downstream.

Example of SDD: Business requirements

- **Prompt 1 (business requirements)**
    - "Please write a feature spec to `specs/.current/spec.md` for adding DNS protection feature to the app. Define user stories, identify edge cases, define functional requirements."
- Review and refine the spec with AI
    - Logical issues
    - Unclear statements
    - You may even start a PR here

SLIDE 28

Example of SDD: Business Requirements

SPEAKER NOTES

Let me walk you through a concrete SDD workflow, step by step.

The first prompt is for business requirements. You might tell the AI agent: "Please write a feature spec to specs/current/spec.md for adding DNS protection feature to the app. Define user stories, identify edge cases, define functional requirements."

Notice how specific this prompt is. You're telling AI exactly where to write the spec — a specific file path in your project. You're telling it what to include — user stories, edge cases, functional requirements. And you're giving it the domain context — DNS protection feature for the app.

The AI will generate a comprehensive specification document, drawing on its knowledge of DNS, security, and your codebase.

Then comes the critical step: you review and refine the spec with AI. You look for logical issues — does the spec contain contradictions or gaps? You look for unclear statements — are there requirements that could be interpreted in multiple ways?

And here's a practical tip: you may even start a Pull Request at this stage. Yes, before any code is written. A PR for the specification allows your team to review the design, suggest changes, and reach consensus before implementation begins. This is incredibly powerful because it shifts review upstream — you're reviewing design decisions when they're cheap to change, rather than implementation details when they're expensive to change.

Example of SDD: Tech spec

- **Prompt 2 (tech spec)**
  - "Please write a technical implementation plan for **`@spec.md`** to `specs/ADG-123/plan.md`"
- Review and refine the plan with AI
  - Research unknowns
  - Research technological choices
  - Verify actual technical details (what code needs to be changed/added)

SLIDE 29

Example of SDD: Tech Spec

SPEAKER NOTES

The second prompt converts business requirements into a technical implementation plan.

You might say: "Please write a technical implementation plan for @spec.md to specs/ADG-123/plan.md." The @ symbol is a way to reference files in many AI agents — it tells the agent to read the spec file and use it as context for creating the plan.

The technical plan differs from the business spec. The business spec says what the system should do from the user's perspective. The technical plan says how to build it — what modules to create, what APIs to define, what data structures to use, what existing code to modify. It's the bridge between requirements and implementation.

Then you review and refine the plan with AI. At this stage, you're looking at different things than before. Research unknowns — are there technical questions that need answers before implementation can start? For example, which DNS library should we use? What's the performance profile?

Research technological choices — is the proposed approach actually the best one? What are the alternatives? What are the trade-offs? Verify actual technical details — the plan should reference specific files, functions, and interfaces in your codebase. AI should know exactly what code needs to be changed or added.

This step is where your engineering expertise is most valuable. You're evaluating technical decisions, identifying risks, and ensuring the plan is feasible and well-structured before any code is written.

Example of SDD: Implement and Validate

- **Prompt 3 (implement)**
  - “Please implement `@plan.md`”
- **Prompt 4 (validate, optional)**
  - “Please verify that `@plan.md` is fully implemented”

Example of SDD: Implement and Validate

SPEAKER NOTES

Steps three and four are where AI does the heavy lifting.

Prompt three is simple: "Please implement @plan.md." That's it. At this point, you've done all the design work. The specification is written and reviewed. The technical plan is detailed and verified. Now AI can execute with confidence because it has all the context it needs.

The agent will read the plan, understand the changes needed, and implement them — writing new files, modifying existing code, running tests, iterating until everything works. Because the plan is detailed and specific, AI doesn't need to make guesses about architecture or approach.

Prompt four is optional but recommended: "Please verify that @plan.md is fully implemented." This asks AI to go back through the plan and check that every item has been addressed. It's a self-review step — the agent compares the implementation against the specification and flags any gaps.

This four-step workflow — spec, plan, implement, validate — might seem like a lot of overhead. But in practice, steps one and two are where you invest your thinking time, and that thinking is what determines the quality of the final product. Steps three and four are mostly automated — AI does the work while you focus on other things.

The total time to completion is often shorter than the old approach of "start coding and figure it out as you go," because you avoid the costly cycles of implementation, discovery of design flaws, and re-implementation.

AGENTS.md

The contract between humans and AI

AGENTS.md — Section

SPEAKER NOTES

Now let's talk about AGENTS.md — the project-level instruction file that serves as the contract between humans and AI agents. If SDD is about individual features, AGENTS.md is about the project as a whole.

What is AGENTS.md?

AGENTS.md is a **project-level instruction file** for AI agents.

It describes (at a minimum):

- how the repository is structured
- how the system is supposed to work
- architectural boundaries and constraints
- development and testing workflows

Understood by all AI agents: https://agents.md/

What Is AGENTS.md?

SPEAKER NOTES

AGENTS.md is a project-level instruction file for AI agents. Think of it as a README, but specifically for AI. While README.md tells human developers how to set up and use the project, AGENTS.md tells AI agents how to work within the project.

At a minimum, it describes: how the repository is structured — what's in each directory, where different types of code live, how the project is organized. How the system is supposed to work — the architecture, the data flow, the key design decisions and the reasoning behind them. Architectural boundaries and constraints — what patterns are required, what patterns are forbidden, what conventions must be followed. Development and testing workflows — how to run tests, how to build the project, how to validate changes.

This file is understood by all major AI agents — Cursor, Windsurf, Claude Code, OpenAI Codex — they all look for and respect AGENTS.md or similar configuration files. You can find the specification and best practices at agents.md.

Having a well-written AGENTS.md is the single most impactful thing you can do to improve AI's performance on your project. Without it, every AI session starts from zero — the agent has to rediscover your conventions, your structure, and your workflow. With it, the agent starts with all the context it needs to be productive immediately.

What is AGENTS.md?

Think of it as:

- onboarding documentation for AI
- shared context for the whole team
- the place where **“how we do things here”** is written down

“If SDD describes **what** to build,
AGENTS.md describes **how we build it**.”

SLIDE 33

What Is AGENTS.md? (Continued)

SPEAKER NOTES

Think of AGENTS.md as serving three purposes.

First, it's onboarding documentation for AI. When a new human developer joins your team, you give them onboarding docs, explain the project structure, walk them through the coding conventions, and show them how to run tests. AGENTS.md does the same thing for AI. Every time an AI agent starts a new session, it's essentially a new team member who needs to be onboarded — it has no memory of previous sessions.

Second, it's shared context for the whole team. Everyone uses the same AGENTS.md, which means everyone's AI agent behaves consistently, follows the same conventions, and produces code in the same style. Without it, two developers might get completely different code from AI for the same task because their agents have different context.

Third, and most importantly, it's the place where "how we do things here" is written down. Every team has unwritten rules — conventions that everyone knows but nobody documented. "We never use global state." "All

API responses must include a timestamp." "Error messages must be user-friendly." These unwritten rules are exactly what AI trips over, because AI cannot read your mind.

The key relationship to remember: "If SDD describes what to build, AGENTS.md describes how we build it." They're complementary tools that together give AI everything it needs to be effective.

What questions should it answer?

AGENTS.md should make these questions trivial:

- where does this code belong?
- what is allowed and what is forbidden?
- ❗ **how do we run tests and verify changes?**
- what are common mistakes to avoid?
- what belongs in the library vs the program?

If a human needs to ask it repeatedly, the agent will get it wrong too.

SLIDE 34

What Questions Should It Answer?

SPEAKER NOTES

What specific questions should AGENTS.md answer? Let me go through the list.

Where does this code belong? When AI creates a new file or function, it needs to know the project's organizational structure. Does this function go in the utils package? The services layer? A new module?

What is allowed and what is forbidden? Are there patterns the team has explicitly decided not to use? For example, "never use global mutable state" or "always use structured logging instead of print statements."

How do we run tests and verify changes? This one gets a red exclamation mark because it's the most critical item on this list. If AI doesn't know how to run your test suite, it cannot verify its own work. It will produce code that looks correct but might fail in ways that only become apparent when a human runs the tests manually. Your AGENTS.md should include the exact commands to run unit tests, integration tests, linting, and the full build process.

What are common mistakes to avoid? Document the traps that even experienced developers fall into. "Don't import module X directly, use the wrapper in utils." "The config format changed in v3, don't use the old schema."

What belongs in the library vs the program? Clear separation of concerns that AI should maintain.

Here's the golden rule: if a human needs to ask a question repeatedly, the agent will get it wrong too. If your team members keep asking "how do I run the tests" — that means it's not documented, and AI will struggle with it just as much as a new hire would.

AGENTS.md is a living document

AGENTS.md is **never finished** .

Correct workflow:

- the agent behaves incorrectly
- you understand **why**
- you add or clarify a guideline
- the agent performs better next time

SLIDE 35

AGENTS.md Is a Living Document

SPEAKER NOTES

AGENTS.md is never finished. Let me repeat that: it is never finished. It's a living document that evolves with your project, your team, and your understanding of what AI needs.

Here's the correct workflow for maintaining it. The agent behaves incorrectly — maybe it puts a file in the wrong directory, uses a deprecated API, or ignores a convention. You understand why — you trace the problem back to a missing or unclear instruction in AGENTS.md. You add or clarify a guideline — you update AGENTS.md with the specific rule that would have prevented the mistake. The agent performs better next time — because now it has the context it was missing.

This is a feedback loop, and it's incredibly powerful. Every mistake AI makes is an opportunity to improve AGENTS.md, which means every future interaction gets better. Over time, your AGENTS.md becomes a comprehensive knowledge base that captures all the subtle conventions, patterns, and constraints that define your project.

Think of it like training a team. You don't hand new team members a perfect manual on day one. You give them initial guidance, they make mistakes, you correct them, and over time the team develops a shared understanding. AGENTS.md is exactly the same process, except the "corrections" are permanent and benefit every future AI session, every developer on the team, simultaneously.

AI needs guardrails

Without guardrails you risk chaos

SLIDE 36

AI Needs Guardrails — Section

SPEAKER NOTES

AI is powerful, but without proper guardrails, that power can cause more harm than good. Let's talk about the minimum safety net every AI-driven project needs.

AI without guardrails is chaos

- AI can:
  - write a lot of code
  - refactor aggressively
  - change things you didn’t ask for
- Without guardrails:
  - regressions slip in
  - quality becomes inconsistent

“The faster the AI moves,
the stronger the guardrails must be.”

AI Without Guardrails Is Chaos

SPEAKER NOTES

AI is incredibly productive — and that's exactly what makes it dangerous without guardrails.

AI can write a lot of code. In a single session, an AI agent can produce hundreds or even thousands of lines of code. It can refactor aggressively — renaming variables, restructuring modules, moving files around. And it can change things you didn't ask for — "improving" code that was working fine, "cleaning up" patterns you intentionally chose, or "fixing" things that weren't broken.

Without guardrails, regressions slip in. AI modifies a function to fix one bug and accidentally breaks another behavior. Quality becomes inconsistent — some parts of the codebase follow one standard, other parts follow different patterns because different AI sessions made different choices.

Here's the quote I want you to internalize: "The faster the AI moves, the stronger the guardrails must be."

This is counterintuitive. You might think that faster development means you should remove barriers and let AI work freely. The opposite is true. When a human writes code slowly, they naturally catch their own mistakes through the mechanical process of typing and reading. AI doesn't have that natural brake. It generates code at machine speed, which means mistakes also happen at machine speed. Without automated checks, those mistakes accumulate faster than any human can review them. Speed without safety is just chaos with good intentions.

The minimum viable guardrails

Every AI-driven project must have:

- Linters, formatters
- Unit tests (non-negotiable)
- markdownlint for specs and rules
- pre-commit hook is recommended

Important:

- AI agent MUST be able to run all that from terminal
- AI agent MUST be able to build your project from terminal

SLIDE 38

The Minimum Viable Guardrails

SPEAKER NOTES

Every AI-driven project must have these minimum guardrails. No exceptions, no negotiation.

Linters and formatters — these catch style violations, common errors, and inconsistencies automatically. AI-generated code should pass the same lint rules as human-written code. This is your first line of defense against sloppy or non-standard output.

Unit tests — this is non-negotiable. I cannot stress this enough. Unit tests are the primary mechanism by which AI verifies its own work. When AI writes code and runs the tests, it gets immediate feedback on whether its changes are correct. Without tests, AI is flying blind — generating code and hoping it works.

Markdownlint for specs and rules — if you're doing SDD, your specifications are important artifacts. They should be clean, well-formatted, and consistent. Markdownlint ensures that your specs maintain a professional standard.

Pre-commit hook is recommended — a pre-commit hook that runs linting, formatting, and tests before every commit ensures that nothing slips through. It's a safety net that catches problems before they enter your codebase.

And here are two absolute requirements. AI agent MUST be able to run all of these checks from the terminal. If your linter requires a GUI, AI can't use it. If your tests require manual setup steps, AI can't run them. AI agent MUST be able to build your project from the terminal. If AI can't build and verify its changes, it's generating code in the dark. Document these commands clearly in your AGENTS.md.

AI at AdGuard

What tools we provide or can provide

AI at AdGuard — Section

SPEAKER NOTES

Now let's look at the specific tools available to us at AdGuard. I'll explain what each tool is best for so you can choose the right one for your situation.

Windsurf

Our go-to AI IDE that we provide to anyone who requests it:
https://windsurf.com/

- Great AI agent, very good at managing context
- Provides add-ons for JetBrains
- Bad at handling LARGE files
- Supports AGENT.md, agent skills, workflows

SLIDE 40

Windsurf

SPEAKER NOTES

Windsurf is our go-to AI IDE. It's the tool we provide to anyone on the team who requests it, available at windsurf.com. Let me explain why we chose it.

First, it's a great AI agent with very good context management. This means Windsurf is smart about which files and code to include in its prompts to the LLM. Good context management is crucial because the quality of AI output depends directly on the quality of the context it receives.

Second, it provides add-ons for JetBrains IDEs. Many of our developers use JetBrains — GoLand, IntelliJ, WebStorm — so having JetBrains integration means you don't have to switch to a completely new editor to benefit from AI assistance.

Third, I want to be transparent about a limitation: Windsurf is bad at handling large files. If you have a file that's several thousand lines long, Windsurf may struggle with context management and produce lower-quality results. The workaround is to break large files into smaller, well-organized modules — which is good practice regardless of AI.

Fourth, it supports AGENTS.md, agent skills, and workflows. This means all the SDD practices we've discussed today are fully supported. Your project's AGENTS.md will be read and followed by Windsurf's agent, ensuring consistent behavior across the team.

If you don't have Windsurf access yet, request it. It's the recommended starting point for AI-assisted development at AdGuard.

OpenAI Codex

OpenAI attempt at AI agent:
https://openai.com/codex/
https://github.com/openai/codex

- Console version (good for automation, API key for CI by request)
- Provides an app
- Supports using ChatGPT account

OpenAI Codex

SPEAKER NOTES

OpenAI Codex is OpenAI's AI coding agent. It's available at openai.com/codex, and the source code is open on GitHub at github.com/openai/codex.

Codex has several characteristics that make it useful for specific scenarios. It has a console version that's particularly good for automation. If you need to integrate AI coding into your CI pipeline — for example, having AI automatically generate tests for new PRs or review code changes as part of your automated workflow — Codex's console version makes this possible. The API key for CI integration is available by request from the platform team.

It also provides a desktop application for interactive use, giving you a more traditional IDE-like experience for day-to-day work.

An important practical benefit: it supports using your regular ChatGPT account. This means you don't need a separate subscription or additional credentials — just your existing OpenAI account.

Codex is particularly interesting for automated workflows. While Windsurf excels at interactive development where you're going back and forth with the AI in real time, Codex shines in automated scenarios where you want to fire off a task, let it run, and get results without constant human interaction. We'll see Codex in action later in the QA section when we discuss automating test case execution.

Gemini / ChatGPT

- Gemini is available to everyone with work account:
  - https://gemini.google.com/
  - Good for asking quick questions, good at image generation
- ChatGPT is available to anyone by request:
  - Same thing as Gemini, good for asking quick questions, image generations, working with requirements
  - Nice desktop app

SLIDE 42

Gemini / ChatGPT

SPEAKER NOTES

Gemini and ChatGPT are our general-purpose AI tools — think of them as your AI assistants for everything that isn't direct code writing.

Gemini is available to everyone with a work Google account at gemini.google.com. It's good for asking quick questions — "how does this API work," "what's the best practice for handling concurrent database writes," that kind of thing. It's also surprisingly good at image generation, which can be useful for creating diagrams, mockups, or presentation graphics for your documentation.

ChatGPT is available to anyone by request. It offers similar capabilities — quick questions, image generation, working with requirements documents — but with a different underlying model and interface. Some people prefer its interaction style. It also has a nice desktop app that makes it convenient for daily use.

The important thing to understand about both of these tools is that they're not AI coding agents. They're conversational AI assistants. You ask a question, you get an answer. They don't have direct access to your codebase, they can't run your tests, and they can't iterate on implementations the way Windsurf or Codex can.

They're best used for research, brainstorming, working with requirements, quick reference, and tasks that don't require direct codebase interaction. Don't try to use them as your primary coding tool — that's what Windsurf and Codex are for.

Cursor / Claude / Claude Code

- Available by request if proven to actually be required
- Very expensive for no obvious reason

SLIDE 43

Cursor / Claude / Claude Code

SPEAKER NOTES

Cursor, Claude, and Claude Code represent the premium tier of AI coding tools. They're available by request, but only if you can demonstrate a specific need that the standard tools don't meet.

Why the restriction? Because they're very expensive — significantly more costly per developer than Windsurf or Codex. And for the vast majority of tasks, Windsurf handles things perfectly well. The additional cost needs to be justified by a concrete benefit.

The cases where you might genuinely need these tools are typically when you're working with very large, complex codebases where the superior reasoning capabilities make a measurable difference. Or when you're doing particularly complex architectural work that benefits from more powerful models — things like designing a distributed system from a high-level spec, or refactoring a large legacy module.

If you find that Windsurf isn't meeting your needs for a specific use case, talk to your team lead about getting access. But be prepared to explain what specific limitation you're hitting and why the premium tools would solve it. "It would be nice to have" isn't sufficient justification for the additional cost — we need to be smart about how we allocate these resources across the organization.

AI reviewer in Bitbucket

- The code is available in BB: PLAT/bit-reviewer
- Can be asked to review any PR:
  - “@bot please review this PR”
- Can be asked questions:
  - “@bot, what do you think about this line”
- Verifies against the linked JIRA issue
- Follows review guidelines from the repo’s AGENTS.md
- Capable of inspecting any file in the repo
- **PRs are welcome (but go through JIRA first)**

SLIDE 44

AI Reviewer in Bitbucket

SPEAKER NOTES

We've built an AI code reviewer that integrates directly into our Bitbucket workflow. The code is available in the PLAT/bit-reviewer repository, and I want to highlight its key capabilities.

It can be asked to review any Pull Request — just mention @bot in a PR comment and say "please review this PR." The reviewer will analyze the changes, check for bugs, evaluate code quality, and leave detailed comments on specific lines.

You can also ask it targeted questions: "@bot, what do you think about this line" or "@bot, is this approach thread-safe?" It's not pattern matching — it can actually reason about your code in context.

What makes our reviewer particularly powerful is that it verifies against the linked JIRA issue. It checks whether the PR actually addresses the requirements specified in the ticket, not just whether the code is technically correct. It follows review guidelines from the repository's AGENTS.md, enforcing your team's specific conventions and standards.

It's also capable of inspecting any file in the repo — not just the files changed in the PR. This means it can check whether your changes are consistent with the rest of the codebase and catch integration issues that a human reviewer might miss.

PRs that improve the reviewer are absolutely welcome — it's an internal tool we're actively developing and iterating on. Just go through JIRA first to coordinate any changes with the platform team.

SDD workflow templates

- The code is available in BB: PLAT/sdd
- **Highly recommended to everyone**
- Two flows available:
  - `/sdd:spec → /sdd:plan → /sdd:implement → /sdd:validate (optional)`
  - `/sdd:quickspec → /sdd:quickimpl → /sdd:validate (optional)`
- **PRs are welcome (but go through JIRA first)**

SDD Workflow Templates

SPEAKER NOTES

We've created a set of SDD workflow templates available in the PLAT/sdd repository. I highly recommend these for everyone on the team.

These templates encode the Spec-Driven Development process we've discussed today into ready-to-use commands that you can invoke directly from your AI agent.

Two flows are available. The full flow for large features: /sdd:spec to generate a specification, then /sdd:plan to create a technical implementation plan, then /sdd:implement to execute the plan, and optionally /sdd:validate to verify the implementation matches the plan. This gives you the full rigor of SDD with structured file organization and workflow orchestration.

The quick flow for minor improvements: /sdd:quickspec to generate a compact specification, then /sdd:quickimpl to implement it directly, and optionally /sdd:validate. This is for smaller changes where the full four-step process would be overkill — bug fixes, minor enhancements, small refactors.

The templates handle file organization, prompt engineering, and workflow orchestration. You just describe what you want to build, and the templates guide you through the SDD process step by step.

Again, PRs to improve these templates are welcome — but please go through JIRA first. Using these templates consistently across the team will dramatically improve the quality and predictability of our AI-assisted development.

Workshop

Let’s use SDD in a real project

Workshop — Section

SPEAKER NOTES

Now let's put all of this theory into practice. We're going to use SDD to implement a real feature in a real project, so you can see exactly how this workflow looks in action.

Workshop

- We will be using PLAT/sdd framework to implement a feature in a real project: https://bit.int.agrd.dev/projects/PLAT/repos/slack-bitbucket/browse
- This project showcases everything discussed above:
  - AGENTS.md
  - SDD approach
  - Guardrails

SLIDE 47

Workshop

SPEAKER NOTES

For our workshop, we'll be using the PLAT/sdd framework to implement a feature in a real project — the slack-bitbucket integration service. You can access the project repository at the link shown on this slide.

This project was specifically chosen because it showcases everything we've discussed today. It has a well-maintained AGENTS.md that defines the project structure, coding conventions, testing requirements, and the complete development workflow. It uses the SDD approach for feature development, with specifications and technical plans stored alongside the code in version control. And it has proper guardrails in place — linting, automated testing, and CI verification.

This isn't a toy example or a demo project. We're going to work on a production codebase with real requirements, real constraints, and real consequences if we get something wrong.

The goal of this workshop is to demonstrate that SDD works in practice, not just in theory. You'll see that the upfront investment in writing specs and plans actually saves time overall, because the implementation phase becomes much smoother when AI has clear, detailed instructions to follow.

What will be showcased

- SDD full flow for a large feature
- SDD quick flow for a minor improvement
- Actualizing project documentation
- Using AI reviewer

What Will Be Showcased

SPEAKER NOTES

In the workshop, we'll demonstrate four distinct workflows that cover the most common development scenarios.

First, the SDD full flow for a large feature. We'll walk through the complete cycle: writing a business specification, creating a technical implementation plan, having AI implement the plan, and validating the results. You'll see how each step builds on the previous one and how the specifications guide AI toward the right implementation without guesswork.

Second, the SDD quick flow for a minor improvement. Not everything needs the full four-step process, and knowing when to use the quick flow versus the full flow is an important skill. We'll show how to handle smaller changes efficiently.

Third, actualizing project documentation. One of the most practical and underappreciated uses of AI is keeping documentation up to date. We'll show how to use AI to update AGENTS.md and other project docs based on recent code changes, ensuring documentation stays in sync with the codebase.

Fourth, using the AI reviewer. We'll create a PR from our workshop changes and ask the AI reviewer to check our work. You'll see how it provides feedback, catches issues, and ensures consistency with the project's established standards.

This hands-on section is where everything from the talk comes together into a complete, practical workflow.

AI for QA

QA automation will never be the same

SLIDE 49

AI for QA — Section

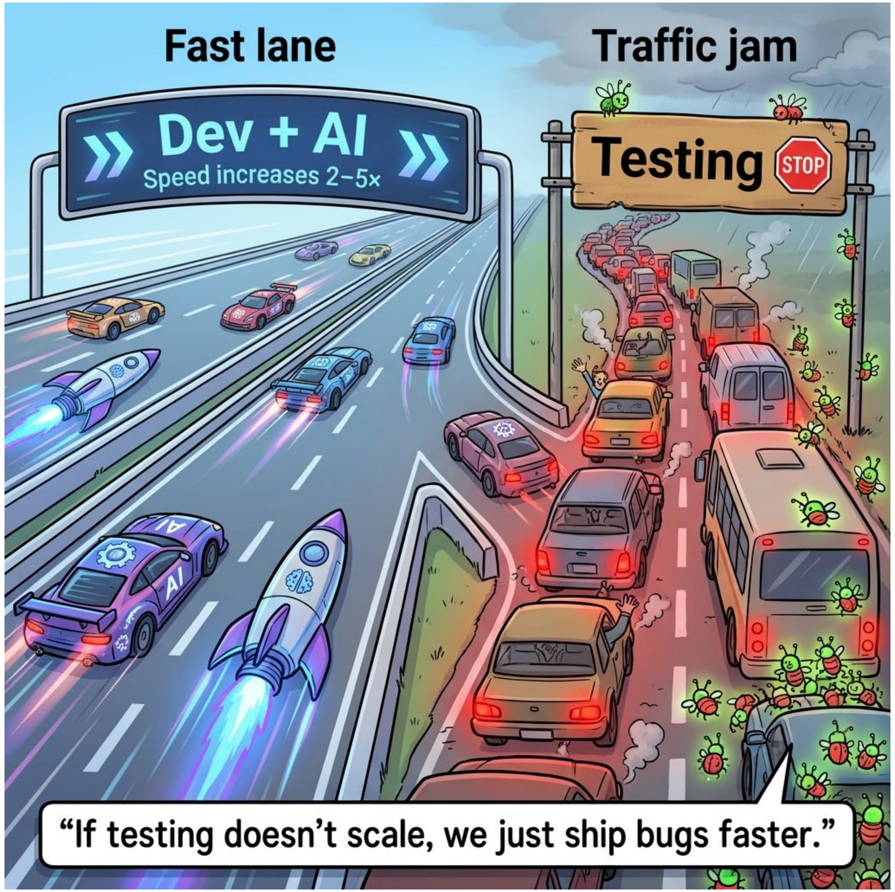
SPEAKER NOTES

The impact of AI extends well beyond development — QA automation is about to be fundamentally transformed. This section is especially important for our testing team, but developers need to understand it too.

Why testers should care

- Dev speed increases with AI
- Testing becomes the bottleneck

“If testing doesn’t scale, we just ship bugs faster.”



SLIDE 50

Why Testers Should Care

SPEAKER NOTES

If you're a tester, this section is especially for you — but developers should listen closely too, because this affects the entire delivery pipeline.

Here's the reality: development speed increases dramatically with AI. Developers using AI coding agents report two to five times faster feature delivery. That's not marketing hype — that's what we're seeing in practice across teams at AdGuard and industry-wide.

But here's the critical problem: testing hasn't sped up at the same rate. If developers are producing features five times faster but the QA cycle takes the same amount of time, you have a bottleneck. Testing becomes the constraint on your entire delivery pipeline.

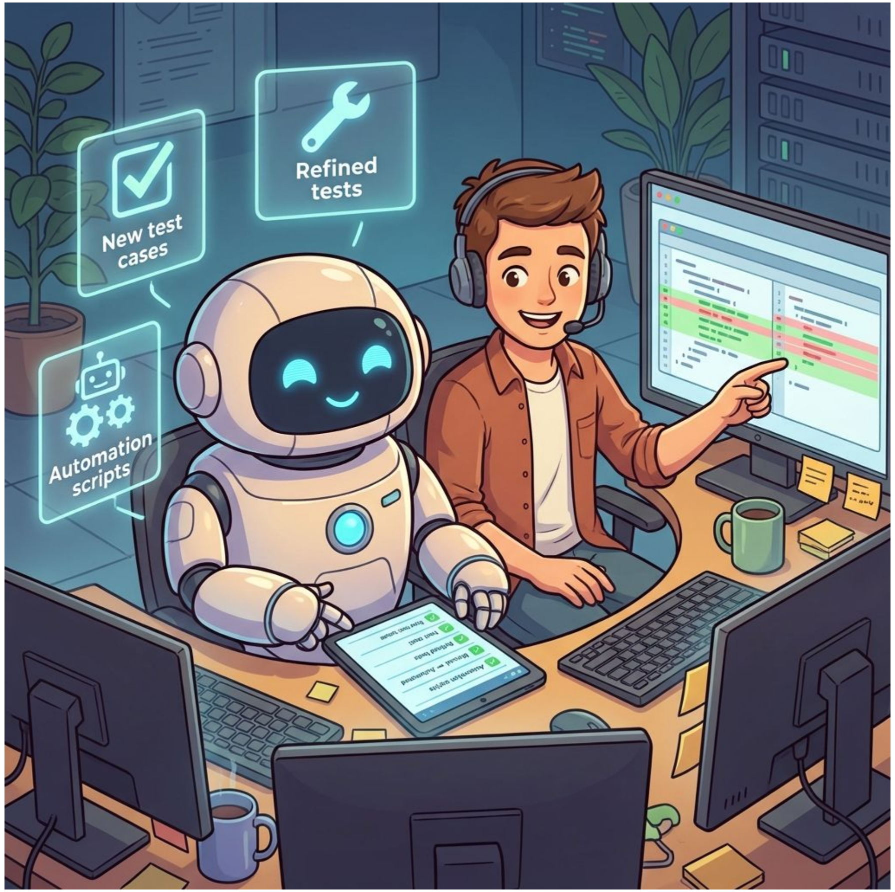
Look at the illustration on this slide. On the left, the "Dev + AI" lane is moving fast — cars zooming along at two to five times normal speed. On the right, the "Testing" lane is a traffic jam. All those features are piling up, waiting to be tested, verified, and signed off.

The quote captures this perfectly: "If testing doesn't scale, we just ship bugs faster." AI makes developers more productive, but if testing can't keep up, all that productivity just means more untested, potentially buggy code reaching production faster.

This is not a hypothetical concern — it's happening right now in teams that embraced AI-assisted development without equally investing in AI-assisted testing. The solution is to apply the same AI techniques to QA that we're applying to development.

What can you use AI for

- Generating test cases (general or for a single change)
- Refining existing test cases
- Automating manual test cases via MCP
- Creating automated test cases



SLIDE 51

What Can You Use AI For

SPEAKER NOTES

So what can AI actually do for QA? Let me walk through the four main categories.

Generating test cases — both general test suites for a component or feature, and targeted test cases for a specific code change. You can give AI your codebase and ask it to identify untested paths, edge cases, and potential failure modes. AI is surprisingly thorough at this because it can analyze code paths systematically without getting fatigued or overlooking patterns.

Refining existing test cases — if you have tests that are flaky, incomplete, or poorly structured, AI can improve them. It can add missing assertions, improve test isolation, reduce duplication, and make tests more robust and readable.

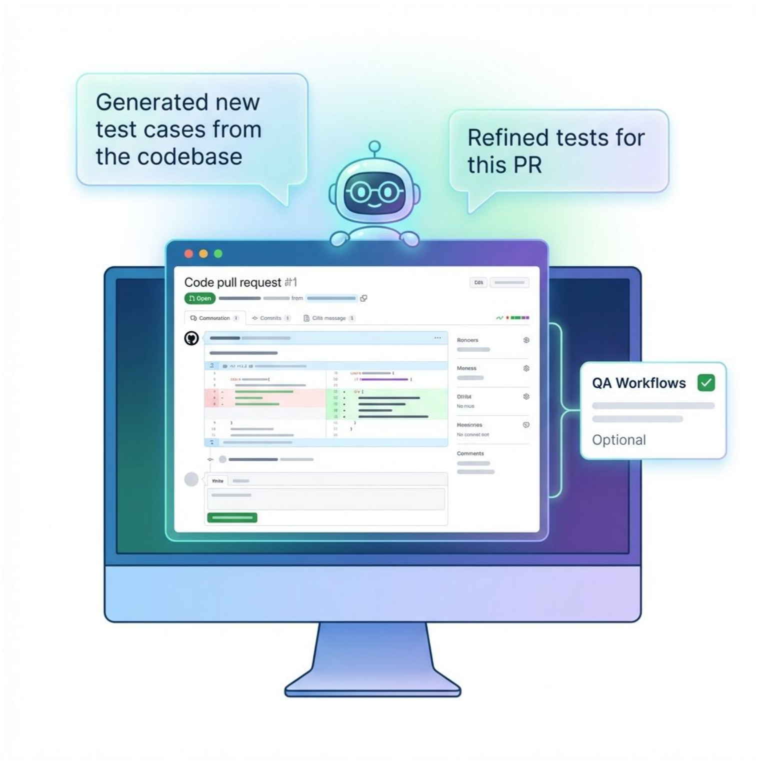
Automating manual test cases via MCP — this is one of the most exciting possibilities. If you have manual test cases that describe step-by-step procedures for verifying features, AI can convert them into automated test scripts that run through those exact same steps programmatically. We'll explore MCP in detail on the next slides.

Creating automated test cases from scratch — given a specification and access to the application, AI can create end-to-end test cases that verify the specified behavior without any human test-writing involved.

The vision is human expertise in test design combined with AI speed in test implementation and execution. You focus on what to test, AI handles how to test it.

Generating and refining test cases

- Use the original code base
- Ask AI to generate test cases
- QA workflows can be added to **PLAT/sdd** as an optional set
- Refining: run it on a specific pull request
    - Idea for AI reviewer: add “generate test cases” functionality


Generated new test cases from the codebase
Refined tests for this PR
Code pull request #1
Open
QA Workflows
Optional

Generating and Refining Test Cases

SPEAKER NOTES

Let's get more specific about generating and refining test cases in practice.

The approach starts with using the original codebase as context. You give AI access to your code, and you ask it to generate test cases that cover the important behaviors. The AI analyzes the code, identifies logical paths, edge cases, error conditions, and potential failure points, then generates tests that cover them systematically.

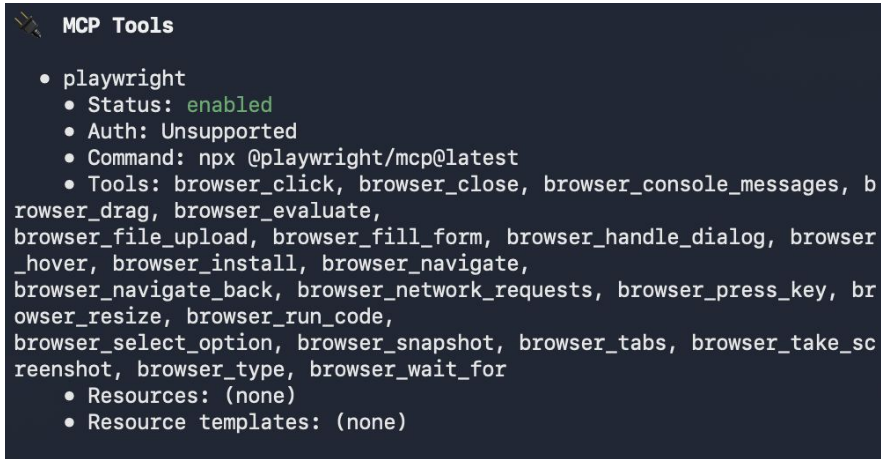
QA workflows for this can be added to PLAT/sdd as an optional set of commands. Imagine running a command like /sdd:generate-tests and getting a comprehensive test suite for a module, or /sdd:refine-tests to improve your existing test coverage based on analysis of what's currently missing.

For refining, the most powerful approach is to run AI on a specific pull request. You point AI at a PR and say "generate test cases for the changes in this PR." The AI reads the diff, understands what changed and why, and creates targeted tests that verify the new behavior specifically. This ensures every code change comes with appropriate test coverage.

Here's an idea we're actively exploring for the future: adding "generate test cases" functionality directly to our AI reviewer in Bitbucket. Imagine every PR automatically receiving suggested test cases alongside the code review comments. The PR author reviews the suggested tests, accepts the good ones, modifies others, and ends up with thorough coverage without spending hours writing tests manually.

Automating manual test cases via MCP

- What is MCP?
  - Open-source standard for connecting external systems to AI agents
  - AI agents know how to read the list of available tools and how to use them using this protocol



```
🔌 MCP Tools

  • playwright
    • Status: enabled
    • Auth: Unsupported
    • Command: npx @playwright/mcp@latest
    • Tools: browser_click, browser_close, browser_console_messages, b
rowser_drag, browser_evaluate,
browser_file_upload, browser_fill_form, browser_handle_dialog, browser
_hover, browser_install, browser_navigate,
browser_navigate_back, browser_network_requests, browser_press_key, br
owser_resize, browser_run_code,
browser_select_option, browser_snapshot, browser_tabs, browser_take_sc
reenshot, browser_type, browser_wait_for
    • Resources: (none)
    • Resource templates: (none)
```

Codex shows the list of tools provided by playwright MCP

SLIDE 53

Automating Manual Test Cases via MCP

SPEAKER NOTES

MCP stands for Model Context Protocol. It's an open-source standard for connecting external systems to AI agents, and it's a game-changer for test automation.

Here's the concept: AI agents need tools to interact with the world — to click buttons in a browser, to make API calls, to interact with databases and file systems. MCP provides a standardized way to give agents these capabilities. Instead of every AI agent implementing its own browser automation or API client, MCP defines a common protocol that tools use to expose their capabilities to any compatible agent.

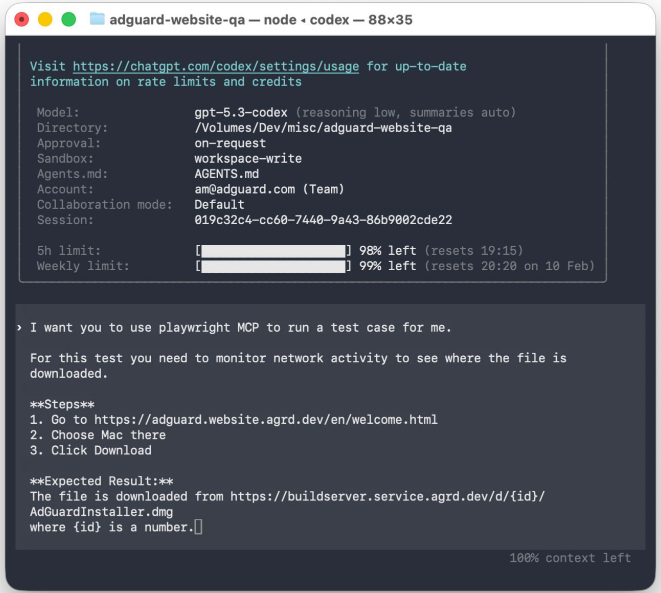
AI agents know how to discover available MCP tools and how to invoke them. On the right side of this slide, you can see a screenshot from Codex showing the MCP tools provided by Playwright — a browser automation library. The tools include things like browser_click, browser_navigate, browser_fill_form, browser_take_screenshot, and many more.

What this means practically: an AI agent with Playwright MCP can control a real web browser, navigate to your application, interact with UI elements — clicking buttons, filling forms, scrolling — verify that elements appear correctly on the page, and report results.

For QA, this is revolutionary. It means AI can execute your manual test cases by actually interacting with your application through a browser, just like a human tester would, but faster, more consistently, and without getting tired or making careless mistakes after running the same test for the hundredth time.

Example: automating unit-tests with MCP

- Use an AI agent that supports MCP
- Recommended is Codex
  - Can be automated via CI
  - Supports regular ChatGPT accounts



```
Visit https://chatgpt.com/codex/settings/usage for up-to-date
information on rate limits and credits

 Model:                  gpt-5.3-codex (reasoning low, summaries auto)
 Directory:              /Volumes/Dev/misc/adguard-website-qa
 Approval:               on-request
 Sandbox:                workspace-write
 Agents.md:              AGENTS.md
 Account:                am@adguard.com (Team)
 Collaboration mode:     Default
 Session:                019c32c4-cc60-7440-9a43-86b9002cde22

 5h limit:               [████████████████████] 98% left (resets 19:15)
 Weekly limit:           [████████████████████] 99% left (resets 20:20 on 10 Feb)

› I want you to use playwright MCP to run a test case for me.

  For this test you need to monitor network activity to see where the file is
  downloaded.

  **Steps**
  1. Go to https://adguard.website.agrd.dev/en/welcome.html
  2. Choose Mac there
  3. Click Download

  **Expected Result:**
  The file is downloaded from https://buildserver.service.agrd.dev/d/{id}/
  AdGuardInstaller.dmg
  where {id} is a number.

                                                                 100% context left
```

SLIDE 54

Example: Automating Tests with MCP

SPEAKER NOTES

Here's a concrete example of how MCP-based test automation works in practice.

We use OpenAI Codex as our AI agent because it supports MCP and can be automated through CI. On the right side of this slide, you can see an actual terminal session. Codex is configured with Playwright MCP and pointed at our adguard-website-qa project.

The prompt is straightforward: "I want you to use Playwright MCP to run a test case for me." Then we provide the test case in a structured, plain-language format. The Steps section says: go to the AdGuard welcome page, choose Mac, click Download. The Expected Results section says: a file is downloaded from a specific build server URL matching a defined pattern.

Codex reads this test case, understands the steps and expected outcomes, and uses Playwright MCP tools to execute it — navigating to the page, clicking the appropriate buttons, monitoring network activity to detect the download, and verifying that the downloaded file comes from the correct URL.

The key benefits of this approach are significant. You can use regular ChatGPT accounts to run Codex, keeping costs manageable. It can be automated via CI — you can run your entire manual test suite as automated checks in your continuous integration pipeline. And because the test cases are written in plain language, not code, any tester can create and maintain them regardless of their programming background.

Be careful with MCP servers

- **Use only official MCP**
- For unofficial third-party MCP, if it’s **REALLY** needed, we’ll need to fork it and use our own version

Be Careful with MCP Servers

SPEAKER NOTES

A word of caution about MCP servers — this is a security topic that everyone needs to take seriously.

The rule is simple: use only official MCP servers. MCP servers have significant access to your systems — they can control browsers, access local files, make network requests, and execute commands. An unofficial or malicious MCP server could potentially exfiltrate data, execute harmful commands, or compromise your development environment.

For unofficial third-party MCP servers: if we determine that a particular server is genuinely needed for our workflow and no official alternative exists, we will fork it and maintain our own version. This means we audit the code line by line, understand exactly what it does, remove any unnecessary capabilities or telemetry, and maintain it ourselves going forward.

Do not install random MCP servers from GitHub or npm without going through the proper review process. This applies even if the server has many stars or appears to be from a reputable source — we need to verify it ourselves.

This is the same security principle we apply to any third-party dependency, but it's actually more important for MCP because these tools have active capabilities — they can take actions on your machine, not just provide data. Treat MCP server selection with the same rigor you'd apply to choosing any piece of infrastructure that has access to your systems and your code.

Creating automated test cases

- Well-written manual test case is an ideal spec
- You can use SDD techniques from above to generate automated cases from manual ones

SLIDE 56

Creating Automated Test Cases

SPEAKER NOTES

Here's the final piece of the QA puzzle that brings everything full circle: creating automated test cases from well-written manual ones.

A well-written manual test case is an ideal specification for automation. Think about what a good manual test case contains: clear preconditions that set up the test environment, numbered steps that describe exactly what to do, expected results that define what success looks like, and edge cases that cover unusual scenarios. That structure is exactly what SDD requires — a clear, detailed specification of behavior.

You can use the same SDD techniques we discussed earlier in this talk to generate automated test cases from manual ones. The process follows the same pattern: take the manual test case as your specification, have AI create a technical plan for how to automate it, have AI implement the automation script using the appropriate framework, and validate that the automated test produces the same results as manual execution.

This creates a virtuous cycle for your QA organization. Your QA team writes manual test cases using their deep expertise in testing methodology and domain knowledge. Those manual test cases get fed into AI, which generates automated versions. The automated tests run in CI, providing faster feedback on every code change. The QA team then focuses their time on writing more manual test cases and on the kinds of exploratory testing that AI truly cannot do — the creative, intuitive testing that finds the bugs nobody anticipated.

Everyone works at what they're best at: humans design tests, AI executes them.

Thank you!

Questions?

SLIDE 57

Thank You! Questions?

SPEAKER NOTES

Thank you for your attention and your time today. Let me quickly summarize the key takeaways.

First, update your mental model: AI is not a search engine, not a snippet generator — it's a capable agent that needs proper guidance through specifications, context, and feedback loops.

Second, adopt Spec-Driven Development: write specs before code, review specs with AI, let AI implement the verified plan. Use the PLAT/sdd templates to get started immediately.

Third, invest in AGENTS.md: document your project conventions, testing workflows, and architectural decisions. Update it continuously based on AI feedback.

Fourth, set up guardrails: linters, unit tests, pre-commit hooks. These are non-negotiable for any AI-driven project.

Fifth, choose the right tools: Windsurf for daily development, Codex for automation, Gemini and ChatGPT for research, and our AI reviewer for code quality.

I'm happy to take any questions now. You can also reach me at am@adguard.com or @ay_meshkov if you think of questions later.

Let's make AI work for us — properly.