**Build Apps With Windsurf’s AI Coding Agents**

**Video 1: Introduction**

Windsurf is a collaborative agentic IDE that provides a workspace where you can work with AI agent. In this course, you will learn how to use windsurf to build several cool applications and also to deeply understand how a coding agent is built on top of Large Language Models (LLM).

Many people are using AI in their coding workflow, say for autocomplete as a feature, but many users, even with AI as a coding tool, are not yet using AI anywhere near to the full extent of what is possible. This course will make you highly skilled in today’s cutting-edge best practices, and hopefully will transform the way we code.

There is a wide spectrum of AI coding tools available for developers. On one end they are simple coding assistant tools that generally makes a single call to a LLM. On the other end, there are autonomous agents that aim to fully automate the coding experience.

Windsurf is built as a collaborative agent that bridges the gap between simple coding assistants and fully autonomous agents.

You will learn what goes under the hood of an agentic AI tool like Windsurf, all while using to build games, fix unit tests, update large repositories, and build an entire Wikipedia topic analysis app from scratch. Having a coding agent that is effective in maintaining context of the code base and track what you, the developer wants and call the right tools is what gives it feel of a good peer program working alongside you.

One of the main factors that make Windsurf a successful AI coding agent is its robust search and discovery capabilities, that let it takes multiple steps to scan multiple files and even search online for documentation, then identify the most relevant code or docs for a task, and then come out with a sequence of code editor to execute the program as intended. This is much more complex than the typical RAG pipeline.

**Video 2: Getting Started & First App**

In this lesson, you will learn how to get started with Windsurf and build your first app in just a couple of minutes with an AI agent.

Windsurf is a VSCode for work. So if you have used VS Code, a lot of this will visually look similar. Because it is a VS Code work, you could also, throughout installation, import all of your settings from existing VS Code setup.

The main thing to notice about Windsurf first is this panel on the right called **Cascade.** Cascade is our AI collaborative agent. If it doesn’t show up, use Cmd + L.

Prompt: Create a snake game with java script and html.

**Video 3: AI Code Assistants 101**

Let us go back in time and look at the history of these tools and how we got here, and separate some of the hype from reality.

To set this up, we will start from human. We could represent the actions that a human would take, as they were developing code as a timeline of various actions, such as navigating through code, making edits, doing research, etc.



When other tools like ChatGPT, GitHub Copilot, Codium and Cursor started coming out, we started entering the age of copilot-like assistance, where instead of humans having to complete every single task, individual small pieces of the overall problem, such as making an edit or doing some research could be replaced by getting an autocomplete suggestion, or asking a chat-like experience question.

A diagram of a diagram

Description automatically generated

This was clearly value additive and has propelled the industry forward. Over time, these have gotten better and better by increasing access to private knowledge sources. In the code world, this could be something as a private code base.

A diagram of a diagram

Description automatically generated

The LLM themselves were not trained on these private code bases but by giving access through methods of retrieval, these AI assistants could give more grounded answers and responses and suggestions to the developer. The best way to think about these copilot like tools is like a person just starting to learn how to code. You cannot really just ask it to do a very long running task, because it can only has a single call to an LLM. But by reviewing it constantly, you can definitely get value and speed yourself through. This is the limit of a copilot-like tools.

A diagram of a computer

Description automatically generated

That is why we talking a lot about Agents. The first iteration of agents that were being discussed is the idea of an autonomous agent (such as Devin from Cognition). The idea is quite simple, instead of every single call to an LLM being an individual invocation that has to be reviewed by a developer, we would have the AI system be able to chain multiple calls to LLMs and integrate access to tools that would change the state or give new inputs and information to the AI system. This, in theory, would be able to allow the Ai system to complete larger and more complex tasks. They still have access to knowledge sources, just like it was for the copilot-like systems.

The best way to think about these autonomous agents is like an intern or a junior engineer, where you need to still scope out the task very clearly for them, and after they complete their work, you have to actually go in and review all the work that it completes.

A diagram of a research process

Description automatically generated

The other reality is that a lot of these autonomous agentic systems, especially in software engineering, do take a lot of time. There could be some parallel development work going on between you as a developer and the AI agent.

A diagram of a research process

Description automatically generated

And so after this idea of autonomous agents, the next kind of generation of tools is this idea of collaborative agents, which I might refer to as flows. The basic idea behind flows is to treat the AI instead of an intern or junior engineer that you delegate work to, treat it more like a peer programmer. And the beauty of this is instead of having to perfectly scope out work and review a bunch of work from an agent, it is kind of this back and forth mind meld like experience, where any actions that the human takes in the IDE can be reasoned about by the agent, and vice versa.

A close-up of a graph

Description automatically generated

This combination of knowledge, sources, tools and understanding of what each other is doing creates this kind of new experience that tools such as Windsurf and our agent called Cascade really kind of build upon in terms of paradigms.

But that being said, there is still limitations as to reality of what these tools are capable of doing. The reality is these flows have some of the same kind of issues and downsides, as any of the other kind of agentic or non-agentic systems that came before it. These are all using LLMs under the hood that have data distribution issues.

Not every single language or framework is equally represented in the public corpus of data that these models are trained upon. There might be imperfect tool calling, there might be limited tool sets that don’t fully match all the different things that a human could do or a developer could do. The reality is there is always going to be some incomplete intent that these agents won’t have. Therefore, guiding it to the ability that you can doesn’t improve the performance.

So, why should we talk about AI Code Assistants now?

1. These are available for use right now. There is a lot of theory about agentic systems, but not too many applications that any individual are able to freely use on their own time. We can use this time to understand what goes into building an agentic product that is generally accessible. And some mental models of how to think about these products underneath the hood.
2. This is a tool for software engineers. You as a developer can understand how to best leverage them, which will give you an edge in your development.

A close-up of a card

Description automatically generated

**Video 4: Fixing Tests Automatically**

In this lesson, you will use AI agent to analyze and resolve issues in some JavaScript code, and in the process, you will see a lot more details of how the agent works.

Prompt: Run all jest tests in this repository.

* Cascade is going through the repository and understanding how we even need to run these tests.
* It is going in, analyzing existing knowledge.
* It has tools such as the ability to suggest terminal commands.

It can detect a failing test and ask if I would want to investigate it. Once prompted yes, cascade will take multiple steps and tools similar to a human. It will use the context awareness to analyze both the test code and the code affecting the test. It will reason about what the issue might be, then it will use tools to actually make edits to the files.

In example, it realizes the issue is with the test case itself.

Another feature: If I change something in the code and just prompt Cascade to update all callsites, it will do so, knowing that what updates have I done.

**Cascade:**

* There is an understanding of the existing code base.
* There is a whole different set of tools that they utilize to take action and investigate and verify the work that it is doing.
* And understands a little bit of the intent that I had as a developer when I was interacting with the text editor.

**Video 5: How a Collaborative AI Code Agent Works**

In this lesson, you will learn about a few mental models that we can use to dissect and understand how the agent works under the hood. The main purpose of this lesson is to understand the components of Cascade, and the considerations in building Agentic systems in general.

Classic Agent Loop

A diagram of a software process

Description automatically generated

This is the classic agent loop, where the LLM is used as a reasoning agent for tool calling.

2 important components of agentic system:

1. Tools – What are all the actions that can be taken, how powerful are those tools for each of those steps.
2. Reasoning model – State-of-the-art for deciding what tools to call (and how to call them) given all the relevant information that the LLM has, are these generally available foundation models from model providers, such as openai and entropic.

But these don’t actually capture the full set of axes that you could use to improve an agentic experience like Cascade.

1. Context Awareness

Not just availability of knowledge sources, but also how you preprocess and reason over the knowledge to maximize any search and discovery.

There are a few different ways to think about context awareness of the problem.

A close-up of a sign

Description automatically generated

1. Human Actions

Understanding human actions allow for deep understanding of the current user intent and allows the agent to stay “in the loop”.

A diagram of tools and tools

Description automatically generated

Tools

3 categories:

1. Search and discovery tools.

You first need to get all the relevant information to even make changes in the 1st place.

1. Tools that allow us to change state of the world
2. Verification

To check that any changes to the state actually improve the overall system and call us closer to the task at hand.

An orange sign with black text

Description automatically generated

Takeaways:

* Context awareness brings in explicit knowledge
* Human actions bring in implicit intent
* Tools are what take action across search & discovery, state changes, and verification
* The LLM is used for combining these and making tool calls, but is common across different agents

**Video 6: Search & Discovery for AI Agents**

In this lesson, we will do a mini deep dive into the problem of search & discovery for AI agents, which actually requires new paradigms, questioning of benchmarks and novel approaches to maximize the impact of agent at the end.

Agenda:

* Why is search & discovery interesting for Code
* “State of the Art” Search & Discovery for Code
* New Paradigm for Search & Discovery

Why is search & discovery interesting for Code?

1. Code is a very distributed store of knowledge

Because of frameworks, libraries, abstractions, not all the relevant information that you need lives in the exact file you are making edits to.

1. Code is often incomplete

Code is not the only relevant information for writing new code (docs, tickets, web, etc). You need more than the existing code to write code.

1. Code is inexact

Different people can create the same outcomes with vastly different implementations.

The bottom line: Garbage in, Garbage out

“State of the Art” Search & Discovery for Code

Retrieval Augmented Generation (RAG)

At a basic level, we start with a question from the user (a prompt). There is a retriever that retrieves the relevant context that is necessary, to pass into the LLM along with the question & task to get a response. -> This is all for the copilot-like systems or assistants, we only get a single call to LLM.

The agentic approach actually fundamentally changes this.

Instead of having to iterate a lot on that retrieval to make it more accurate as you only have 1 shot at the LLM & retrieval, the multi-step agentic retrieval approach means that we can actually have multiple shots are retrieval.

A diagram of a flowchart

Description automatically generated

So maybe the retrievers don’t have to be super perfect, but we need to be able to iterate AI. And also importantly, all the retrievers don’t have to be the same. We could have different retrievers for different kind of task that we might need to do in the overall search and discovery problem.

A screenshot of a chat

Description automatically generated

Category 3

If I want to build a new object and just want all the relevant information, what would I actually kind of need?

A screenshot of a computer

Description automatically generated

State-of-the-Art today: Embedding Search

How that works? You have an embedding model that can convert objects (eg: a snippet of code) to a series of numbers (embedding vector). You can do this for all existing snippets of code that already exists in your code base.

A diagram of embedding model

Description automatically generated

Then, at retrieval time, all you do is you take your current work, use the same embedding model to convert it to its own embedding vector. And then compare the embedding vector that was created to all the existing embedding vectors to see what other vectors are close by in this n-dimensional embedding space.

If done correctly, when you do the retrieval, you are pulling in a bunch of snippets of code that are at least similar and ideally relevant to the work that you are currently doing.

This approach is NOT perfect. Because we are operating on embedding vectors rather than in raw text, we are losing a lot of nuances of that original text snippets. Larger models doesn’t mean better retrieval, there seems to be some kind of plateau on how good embedding is at retrieval.

Needle in the haystack kind of benchmark is not the one we are interested in with regards to coding. We are interested in this idea of if I retrieve 50 objects, how many of the ground truth and relevant objects would appear within these 50.