



Software Development with Agentic Coding Tools

How LLMs and Agents Work

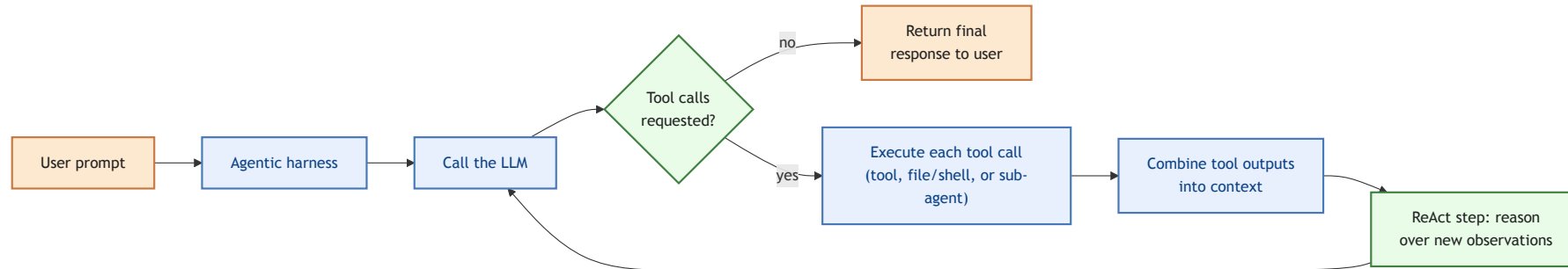
Zachary G. Ives & Harry Smith
University of Pennsylvania
Fall 2026

Today We Want to Talk about Coding Agents and Their Use. Let's Start with Some Simple Questions...

1. What is an LLM?
2. What is an agent?
3. How do LLMs and agents interact?
4. What's a coding agent?

You're all users of these things... Please chime in!

"Agentic Harnesses": Systems that Dispatch Requests to LLMs and Agents



ChatGPT is an agentic harness (with access to the Internet). Claude Code is an agentic harness (with access to your machine).

Is an agentic harness itself an agent?

Software Engineers and Coding Agentic Harnesses

Senior engineers joining a new team have always had to read an unfamiliar codebase quickly.

The skills: *find the entry points, trace a critical path, identify the invariants. Understand why and what is going on*

- Before, IDEs and tooling would help -- e.g., source browsing, block diagrams, "Go to symbol definition"
- Agents accelerate this further — **but only if you verify what they tell you.**

We should understand how they work, to understand how best to use them!

It Starts with (Large) Language Models

Two landmark papers established a way of building neural networks for text.

- In 2017: Vaswani et al at Google: *Attention is All You Need*.
- In 2018: Devlin et al at Google: *BERT: Bidirectional Transformers for Language Understanding*.

But still: in 2021, no one would have thought building a giant predictive model of text would lead to tools that could **interpret instructions** and **solve problems**!

"Stochastic Parrots"

LLMs essentially learn patterns in text, and use these to predict more text. A very influential paper:

On the Dangers of Stochastic Parrots: Can Language Models Be Too Big?

Authors:  [Emily M. Bender](#),  [Timnit Gebru](#),  [Angelina McMillan-Major](#),  [Shmargaret Shmitchell](#)

[Authors Info & Claims](#)

FACCT '21: [Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency](#) • Pages 610 - 623
<https://doi.org/10.1145/3442188.3445922>

- Raised alarm that LLMs look much smarter than they are -- spreading misinformation and biases because they don't understand semantics. They just repeat.
- And, they have big energy costs.
- And: there is a rat race to make them ever bigger.

Let's discuss for a moment: What does "understanding semantics" really mean?

6

It Starts with Language Models

"Stochastic parrots", in the extreme, are fancy autocomplete but probably not that useful. It seems unlikely they could generalize "recipes" and do meaningful things.

But that, in effect, is what has happened.

- We trained large language models over the Internet -- and...
 - They effectively learned to substitute words for similar words, phrases for similar phrases, ...
 - They learned to compose "recipes" from components
 - And they learned to generalize
- Then we used *reinforcement learning* to train them to *follow instructions*

Let's see *a bit* of how this works! You should take courses like **CIS 2210, 2450, 4190, and 6220** to learn this more deeply!

Part 1: How LLMs Are Built

Step 1: Pre-training — Next-Token Prediction

A large language model learns to predict the next token (word) over filtered web-scale text.
Think of your iPhone keyboard autocomplete, trained over everything on the Internet!

Let's do our own version of this.

When the people in the crowd bumped into me, I dropped my

Step 1: Pre-training — Next-Token Prediction

A large language model learns to predict the next token (word) over filtered web-scale text.

Think of your iPhone keyboard autocomplete, trained over everything on the Internet!

Let's do our own version of this.

```
While I was trying to open the front door, I dropped my
```

Step 1: Pre-training — Next-Token Prediction... for Code

A large language model learns to predict the next token (word) over filtered web-scale text. *Think of your iPhone keyboard autocomplete*, trained over everything on the Internet!

So, given this Python code:

```
def factorial(n):  
    if n == 0:
```

The model predicts `return 1` — not because it "knows" mathematics, but because that pattern appears in millions of training files.

- Billions of parameters encode statistical regularities across human-written content
- *No* objectives, *no* plans, *no* truth-seeking drive — it completes text

Reliable for patterns seen many times in training.

Unreliable for novel combinations, recent APIs, or your specific codebase.

11

Step 2: Human Reinforcement of Behavior

Pre-training is almost fully automated. It is then trained for best behavior **by humans**:

- **Supervised fine-tuning (SFT)**: human labelers write examples of ideal assistant behavior. The model is fine-tuned to match — it learns to *play the role* of a helpful assistant.

```
Prompt: Please help me send an email.
```

```
Response 1: I'm a helpful AI assistant.
```

```
Response 2: <call email tool and send>
```

- **RLHF (Reinforcement Learning from Human Feedback)**: a reward model scores completions on *helpfulness* and *harmlessness* ("AI alignment"); the base model is trained.

Another thing that is emphasized is *how to break tasks into step-by-step subtasks* -- what is sometimes called "*chain of thought reasoning*". **Tool use** is one element of that.

Reinforcement Learning Creates *Confidence*, Not *Correctness*

LLMs trained with reinforcement learning produce results that get high ratings from humans.

- Human raters prefer responses that are authoritative, well-structured, and comprehensive-sounding!
- A response that says *"I'm not sure — here are three possibilities"* scores lower than one that states a definite answer, even if the definite answer is wrong.

RLHF produces completions that are fluent and confident... regardless.

When Claude states a function name, a file path, or an API signature with complete confidence, that confidence is a trained behavior — not evidence of accuracy. Always verify.

LLMs "Remember" and "Reason over" Context Windows

An LLM takes a natural language *prompt*. But that is lacking in important *context*.

It usually also gets other information (this is provided by the "agentic harness"):

- Previous prompts and answers, so it can learn
- Suggestions for strategies ("always prefer X to Y")
- Examples to generalize from ("find me names of celebrities like A and B")
- Available resources like agents it can call

All of this shapes the probability distribution for autocompletion of the next word(s) -- thus affecting what the LLM does.

"In-context learning"

Using Claude Code

- You'll generally operate at a high level with Claude Code:
 - Ask it to design a specification
 - Ask it to do an implementation
 - Ask it to define a test suite
 - Ask it to explain things
- But of course, the results may be imperfect, especially as you run the resulting code (or once tests run)

Much of your interaction is in a big loop -- prompt, see and evaluate results, feed back what's wrong or what should be changed.

The Cognitive "ReAct" Loop: Core to the Agentic Harness

Agents running multi-step tasks cycle through a series of stages in a framework called a **harness**:

Reason → **Plan** → **Act (maybe call agents)** → **Observe** → **(repeat)**

Each iteration updates the *context window* with the latest observation.

The loop can:

- **Stall** — infinite fix cycles where each attempt introduces a new error
- **Diverge** — each step takes the agent further from a working state
- **Succeed** — when the task is well-specified and verification is tight

Your job when managing agentic frameworks for coding: detect stalling or divergence early and redirect.

When the Claude Code Cognitive Loop Fails

Signs of a diverging loop where the LLM isn't making progress:

- Error count stays flat or grows across iterations
- Each fix touches different code than the last
- The agent starts suggesting "workarounds" rather than root causes

Example (for Java):

- Agent adds generics → compiler type error → agent removes the type parameter → unchecked cast warning →
- agent adds `@SuppressWarnings("unchecked")` → `ClassCastException` at runtime → agent wraps in `try/catch` → original bug is now hidden.
- Repeat.

Intervention: stop the agent, read the error yourself, identify the root cause, give a targeted instruction.

17

Context Windows and Hallucinations

The model has **no memory outside its context window**.

Claude's context holds roughly 200,000 tokens — about the length of a novel. When a long session fills it, earlier instructions and observations drop off silently.

Hallucinations arise when the model produces a *plausible completion* for which there is no grounded evidence in context.

It pattern-matches rather than retrieves.

Most dangerous for: file paths, function names, API signatures, numeric values, and any claim the model cannot verify from what is currently in its context window.

Stale Knowledge

The model's training data has a cutoff. It does not know about:

- Library versions released after the cutoff
- Breaking changes in recent TypeScript, Node.js, or npm packages
- Security vulnerabilities disclosed after training
- The Spacebar codebase — a course-specific fork with no training signal

When to distrust the model

- Any specific `@version` in a package
- Exact function signatures for libraries
- "The current behavior of X is..."
- Anything marked experimental in the docs

Verification rule

When the model cites a function, option, or configuration value — check it against the current source or documentation.

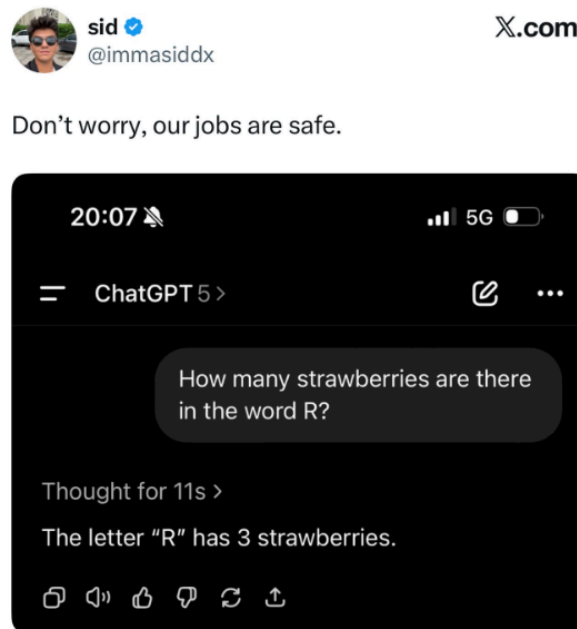
The "Swiss Cheese" Model of LLM Capability

Capability gaps are **unpredictable** and do not correlate with human-perceived task difficulty.

A model that solves Olympiad math problems can confidently mis-compare 9.9 and 9.11.

Every "layer" of the model has gaps; the question is whether the gaps *align* and compound.

- A lot of this comes down to what transformers (the underlying technology) are good at.



The "Swiss Cheese" Model of LLM Capability

Capability gaps are **unpredictable** and do not correlate with human-perceived task difficulty.

A model that solves Olympiad math problems can confidently mis-compare `9.9` and `9.11`.

Every "layer" of the model has gaps; the question is whether the gaps *align* and compound.

- A lot of this comes down to what transformers (the underlying technology) are good at.
- **Or, whether the LLMs have been given an ad hoc rule to call a tool.**

What this means

- Treat LLMs as stochastic tools
- Always verify outputs
- *Especially*: numeric, logical, and API-specific claims

How to probe gaps

- Ask the agent to cite the file and line
- Open the file and verify
- Try a specific counter-example: "What if the user has no roles?"

Before You Accept Any Agent Output

A five-step check — takes under five minutes for most outputs:

1. **Cite check** — does the file / function / line it referenced actually exist?
2. **Read the code** — does the implementation match the explanation?
3. **Edge case** — what input would break this? Can you construct one?
4. **Library check** — is the API call current? Look it up.
5. **Scope check** — does this change anything it was not supposed to touch?

If you cannot complete steps 1–3 because the output is too vague, **ask again with a tighter scope**. Vagueness is a signal, not a time-saver.

Part 2: Using Claude Code to Understand Spacebar

Live demo — ~40 minutes

Spacebar: Discord in Open Source

Most of you have used the Discord chat server (or on the more corporate side, Slack)

On servers (**guilds**), Discord offers chat **channels** (public and private) and direct messages

- Users use clients to connect to the same guild
- Channels have administrators, members, etc.

In open source, **Spacebar** re-implements the public Discord protocols and **Fermi** works as a chat client that works with it.



Spacebar for Java Developers

Spacebar is written in TypeScript + Node.js. It's not Java, but there are commonalities.

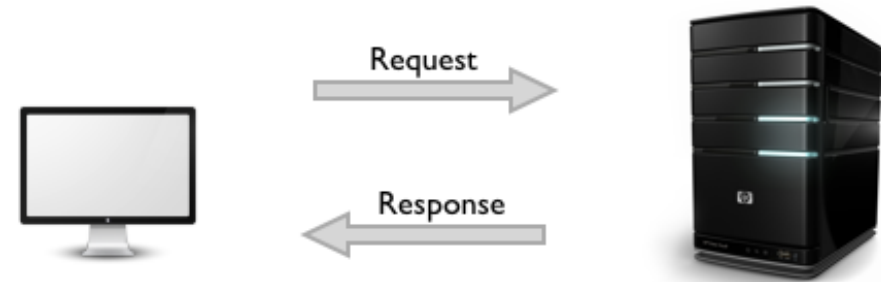
WHAT YOU KNOW (CIS 1200/1210)	SPACEBAR EQUIVALENT
Java <code>interface</code>	TypeScript <code>interface</code> — same idea, nearly identical syntax
Java <code>class</code> with typed fields	<code>@Entity</code> class — also maps to a database table row
<code>javac</code> → <code>.class</code> bytecode	<code>tsc</code> → <code>.js</code> files in <code>dist/</code>
A class that handles one request type	Express route handler in <code>src/api/routes/</code>

You won't need to really do a lot of TypeScript reading today — but you may need to read a bit of what Claude Code gives you.

Spacebar Is Really a Specialized Web Server

A server is a program that **listens on a numbered port** for incoming network TCP connections. The port is a convention, e.g., Spacebar uses **3001**.

- **Socket** — the open connection for sending and receiving data once a client connects. It's like a file, but *read + write*.
- **GET** — an **HTTP** read request: *"give me the resource at this path"*
- **PUT** - an HTTP write request
- **Status code** — 200 OK, 404 Not Found, 403 Forbidden



Wikimedia Commons, CC BY-SA 4.0

Your browser opens a socket, sends a GET, waits for a response, then closes.

So does a Discord client like **Fermi**. For messages, PUT can post, GET can read.

Spacebar Has a Lot of Sub-Modules

Source code is divided into different functional components

MODULE	RESPONSIBILITY
<code>src/api/</code>	Express HTTP server — routes, middleware, request validation
<code>src/gateway/</code>	WebSocket server — real-time event dispatch to clients
<code>src/util/</code>	Shared entities, DTOs, permission logic, config
<code>src/cdn/</code>	File and media upload/serving
<code>src/schemas/</code>	JSON schemas generated from TypeScript types

We'll take a look at some of this today. You'll do more in Lab 1.

Thinking about a Discord-Like Application

Any modern messaging application has at least

- Some notion of channels (which have ~ stable members)
- Some notion of direct messaging

Implicitly, people who aren't on the DM or in the channel **shouldn't be able to see things that they aren't party to**.

- Recall how much of a challenge there was with a Signal chat last year.

So: a big part of a Discord server is managing the interactions between channels and users!

Pete Hegseth

TEAM UPDATE:

TIME NOW (1144et): Weather is FAVORABLE. Just CONFIRMED w/ CENTCOM we are a GO for mission launch.

1215et: F-18s LAUNCH (1st strike package)

1345: "Trigger Based" F-18 1st Strike Window Starts (Target Terrorist is @ his Known Location so SHOULD BE ON TIME) — also, Strike Drones Launch (MQ-9s)

1410: More F-18s LAUNCH (2nd strike package)

1415: Strike Drones on Target (THIS IS WHEN THE FIRST BOMBS WILL DEFINITELY DROP, pending earlier "Trigger Based" targets)

1536: F-18 2nd Strike Starts — also, first sea-based Tomahawks launched.

Discord Concepts and Protection

Core entities

- **Guild** — a "server"; the top-level container
- **Channel** — text, voice, category, DM; belongs to a Guild
- **User** — global identity across guilds
- **Member** — a User's membership in a specific Guild (has roles)
- **Role** — a named set of permission bits within a Guild
- **Permission** — a 64-bit bitfield; each bit is one capability

Resolution order

To compute what a user can do in a channel:

1. Union permission bits across all the user's guild roles
2. Apply channel-level role overwrites (deny first, then allow)
3. Apply channel-level member overwrites last

Guild owner and Admin role bypass all overwrites.

Live Demo: Tracing a Chat Message

Prompt:

Walk me through what happens when a user sends a message to a channel. Trace the path from the HTTP request to the message appearing in another connected client's screen, step by step, citing file and line number at each step.

Expected path: HTTP route handler → permission check → TypeORM DB write → `emitEvent()` → WebSocket dispatch to subscribers.

Verify **two steps** against the actual source.

Watch for: The high-level path (HTTP → permission check → DB write → event → WebSocket) will usually be correct. Specific function names and line numbers are the most likely to be invented — which is exactly what the cite check in the verification checklist catches.

30

Live Demo: The Permission System

Prompt:

"How does the permission system work? What is the resolution order for computing a user's effective permissions in a channel? Cite the function that computes this."

Look for `Permissions.finalPermission()` in `src/util/util/Permissions.ts`.

Verify the resolution order the agent describes matches the domain model slide.

Watch for: The three-step resolution order (roles → channel overwrites → member overwrites) is usually correct at the conceptual level. The exact function name or file path cited may not match what actually exists — or may match perfectly. Either outcome is instructive.

This is the core complexity for **Project 1 and Project 2**.

Good Prompt vs. Bad Prompt

Bad

"Explain the permission code."

- Vague scope → agent summarizes
- Output you cannot verify line by line
- No structure to push back against

Good

"Walk me through `finalPermission()` in `src/util/util/Permissions.ts` step by step in numbered cells. For each step, show the relevant code excerpt (≤ 5 lines) and state what it does to the running permission value."

- Scoped to a specific function
- Structured output — check each cell
- Forces file and line citations

Structured prompts produce *verifiable* output. Vague prompts produce *plausible* summaries.

32

After the Response: Verification in Practice

The five-step check applied to a live agent response:

1. The agent cited `src/util/util/Permissions.ts` — open it in VSCode, find the function (Ctrl/Cmd-P, then filename) and line number (":{number}")
2. It described the resolution order — read the code and confirm each step
3. Edge case: "What if the user has *no* roles?" — trace through the function
4. It mentioned `PermissionOverwrite` — find the entity, check the fields match
5. Scope: did the agent touch any file it was not asked to explain?

If a step fails — ask a follow-up. *"You said X. The code at line 47 says Y. Which is correct?"*

Part 3: Continuous Integration (CI) on Agent P

(Equivalent to GitHub but hosted for our class)

Git Quick Review

We'll be using Git as **version control**. More later, but for now -- you should get comfortable with using Git to `clone` Spacebar and edit it locally -- then `commit` it and `push` back to Agent P.

Four commands cover almost everything you'll do this week:

- `git add <file>` stages a file for the next commit (or `git add -u`).
- `git commit -m "message"` snapshots the staged changes, with a message describing what changed and why — not "fix stuff".
- `git push` sends your local commits to Agent P, where they become visible and the CI check on the next slide can run against them.
- `git pull` fetches and merges any commits already on Agent P that aren't in your local copy yet — run it before you start a new session.

REPOSITORY

[Manage your SSH key](#) and connect to the Penn VPN before cloning or pushing.

Ready

```
git clone ssh://git@ssh.git.agent-
```

Copy

[Open repository / sign in ↗](#)

VERIFIED LAB SUBMISSION

Verified · lab1-submission

The tagged commit contains

lab1/submission.md . Autograding is queued or complete.

Submission window ends 8/31/2026,
10:00:00 PM.

Continuous Integration: Your First Green Check

We want to have your Git server **run an action** every time you commit.

Save this as `.github/workflows/ci.yml` — Lab 1 walks you through it step by step:

```
name: CI
on: [push, pull_request]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with: { node-version: '22' }
      - run: npm install
      - run: npm run build
      - run: npm test
```

Ethics & Societal Impact — Verification as a Moral Obligation

An LLM's output is "plausible" by design, not "correct" by design.

Discussion: At what point does failing to verify AI output become a form of negligence rather than a reasonable time trade-off? Does the answer depend on who is harmed, and how? Does CI make a difference?

Further reading: Andrej Karpathy, "Deep Dive into LLMs like ChatGPT" (video; watch before Week 2) · Anthropic Engineering, "Building Effective Agents"

Recap: How LLMs Work, and How to Use One Well

- Claude code is part of an *agentic harness*, a loop around an LLM that can also trigger calls to other agents / agentic harnesses.
- LLMs are built by **pre-training for text prediction, then SFT/RLHF**: a next-token predictor first, then trained to play the role of a helpful assistant — confident phrasing is a trained behavior, not evidence of accuracy.
- **The agentic harness' loop can stall or diverge** — your job is to notice early and redirect with a precise, tighter-scoped instruction.
- **No memory outside the context window, and a training cutoff** — hallucinations and stale API knowledge both follow directly from this.
- **The five-step verification check** — cite check, read the code, edge case, library check, scope check — took under five minutes against a real Spacebar prompt today, and works the same way every time.

Questions?

`zives@engineering.upenn.edu` · `sharry@engineering.upenn.edu`