The Modern Software Developer

CS146S
Stanford University, Fall 2025
Mihail Eric

Guest Lecture - 10/10/25 (8:30am PT, 420-041)

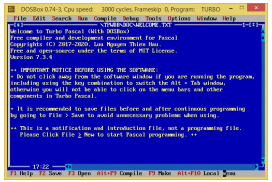
Cognition Head of Research, Silas Alberti

The AI IDE: Fundamentals to Power User

Why

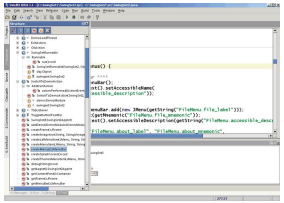
- IDE (Integrated Development Environment)
  - All-in-one workspaces for software development containing editor, compiler, debugger, and more
  - Most development work is done there so it’s a natural form factor for AI enhancement
  - In their evolution, there’s always a see-saw between functionality consolidation and developer customization

A brief history



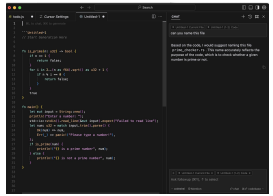
Release of Turbo Pascal, first true IDE

1983



Intellij IDEA released with advanced contextual code navigation, refactoring, code completion

2001



Cursor released, one of the first widely used AI native IDEs

2023

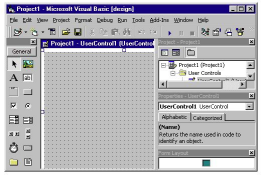
1980

1997

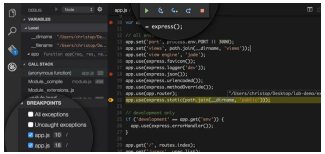
2015

2030

Microsoft Visual Studio released, offering advanced debugging capabilities for the C++/Visual Basic language



Microsoft VSCode released offering lightweight editor with highly extensible ecosystem

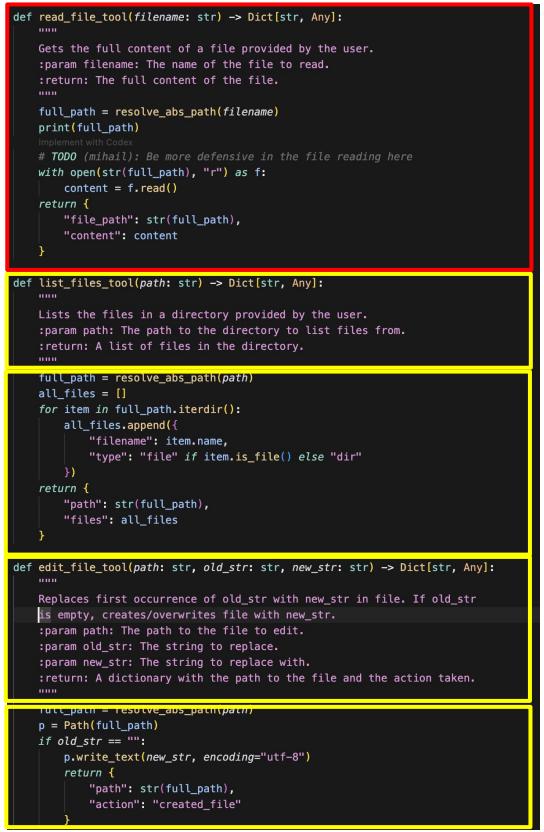


Usage

- Bread-and-butter modes
  - Inline
  - Function
  - Single-file
  - Multi-file
- True AI-native
  - Background agents
  - MCP
  - Learn memories
  - Bugbot (PR review)

Let’s see some of this in action

How an AI IDE works under-the-hood



```python
def read_file_tool(filename: str) -> Dict[str, Any]:
    """
    Gets the full content of a file provided by the user.
    :param filename: The name of the file to read.
    :return: The full content of the file.
    """
    full_path = resolve_abs_path(filename)
    print(full_path)
    Implement with Codex
    # TODO (mihail): Be more defensive in the file reading here
    with open(str(full_path), "r") as f:
        content = f.read()
    return {
        "file_path": str(full_path),
        "content": content
    }
```

```python
def list_files_tool(path: str) -> Dict[str, Any]:
    """
    Lists the files in a directory provided by the user.
    :param path: The path to the directory to list files from.
    :return: A list of files in the directory.
    """
    full_path = resolve_abs_path(path)
    all_files = []
    for item in full_path.iterdir():
        all_files.append({
            "filename": item.name,
            "type": "file" if item.is_file() else "dir"
        })
    return {
        "path": str(full_path),
        "files": all_files
    }
```

```python
def edit_file_tool(path: str, old_str: str, new_str: str) -> Dict[str, Any]:
    """
    Replaces first occurrence of old_str with new_str in file. If old_str
    is empty, creates/overwrites file with new_str.
    :param path: The path to the file to edit.
    :param old_str: The string to replace.
    :param new_str: The string to replace with.
    :return: A dictionary with the path to the file and the action taken.
    """
    full_path = resolve_abs_path(path)
    p = Path(full_path)
    if old_str == "":
        p.write_text(new_str, encoding="utf-8")
        return {
            "path": str(full_path),
            "action": "created_file"
        }
```

Tab-complete

- Small context window around current code is encrypted
- Server receives and runs infilling LLM
- Suggestion sent back and displayed

Chat

- Store code chunks as embeddings in semantic index on server (obfuscated filename + code)
- Any query retrieves most relevant chunks and feeds as context into LLM
- IDE regularly re-index code chunks and syncs embeddings
- Chunk diffs are computed via Merkle trees for efficient updates

Best practices

- For simple changes you don’t have to be too thoughtful about prompting
- For more complex tasks, you’re going to become a product manager
    - Carefully crafted specs doc

Best practices

- **Goal**
  - What is the purpose of the change
- **Definitions**
  - What prereqs does the LLM need to know about the problem
- **Plan**
  - High-level implementation breakdown
- **Source files being changed**
  - What parts of the codebase are relevant and why
- **Test cases**
  - How will testing be done
- **Edge cases**
  - What special cases need to be accounted for
- **Out-of-scope**
  - What should *not* be changed
- **Extensions**
  - What changes will be relevant later so the LLM can future-proof its design and not take shortcuts

Let’s see some of this in action

Best practices

- Optimize your codebase so that a human and an agent could understand what's going on
- Much of LLM confusion comes from trying to finish a task with a messy repo as context
- Provide optimal context for LLMs by describing
  - Repo orientation
  - File structure
  - Setup and environment
  - Best practices
  - Code style
  - Access patterns
  - APIs and contracts
  - *All of this should be thoroughly documented*
  - **Tip:** a monorepo design in your repo is highly encouraged

Best practices

- Help LLM navigate your codebase with agent configurations
  - claude.md
    - *`CLAUDE.md` is a special file that Claude automatically pulls into context when starting a conversation. This makes it an ideal place for documenting: common bash commands, core files and utility functions, code style guidelines, testing instructions.*
  - cursorrules
  - AGENTS.md
    - Open format
  - llms.txt
    - Provide that navigation guidance for LLMs scraping the web
  - **Note:** The agents won't always adhere to these descriptions/directives. They are intended as guidance.

Samples

```
# Sample AGENTS.md file

## Dev environment tips
- Use `pnpm dlx turbo run where <project_name>` to jump to a package instead of scanning with `ls`.
- Run `pnpm install --filter <project_name>` to add the package to your workspace so Vite, ESLint, and TypeScript can see it.
- Use `pnpm create vite@latest <project_name> -- --template react-ts` to spin up a new React + Vite package with TypeScript checks ready.
- Check the name field inside each package's package.json to confirm the right name—skip the top-level one.

## Testing instructions
- Find the CI plan in the .github/workflows folder.
- Run `pnpm turbo run test --filter <project_name>` to run every check defined for that package.
- From the package root you can just call `pnpm test`. The commit should pass all tests before you merge.
- To focus on one step, add the Vitest pattern: `pnpm vitest run -t "<test name>"`.
- Fix any test or type errors until the whole suite is green.
- After moving files or changing imports, run `pnpm lint --filter <project_name>` to be sure ESLint and TypeScript rules still pass.
- Add or update tests for the code you change, even if nobody asked.

## PR instructions
- Title format: [<project_name>] <Title>
- Always run `pnpm lint` and `pnpm test` before committing.
```

Let’s see some of this in action