

I Built a Voice-Activated Prompt Enhancer That Reads Your Terminal

How PromptPulse turns "fix the error" into a surgical prompt your AI assistant can actually act on.

You're staring at a wall of red in your terminal. TypeScript is angry. The build is broken. You switch to your AI coding assistant and type:

```
fix the error
```

And the AI responds: *"Could you provide more context? What error are you seeing?"*

So you go back to the terminal. Copy the error. Paste it in. Add the file name. Mention the branch. Describe what you were trying to do. By the time you've written a prompt the AI can work with, you've spent more time explaining the problem than it would have taken to fix it yourself.

I built PromptPulse to eliminate that friction entirely.

What PromptPulse Does

PromptPulse is a lightweight daemon that sits between you and your AI coding assistant. It monitors your terminal in real-time, listens for voice commands, and transforms vague prompts into precise, context-rich instructions.

You say "fix the error." PromptPulse delivers this to your clipboard:

```
Fix the TypeScript compilation error TS2345 in src/auth/middleware.ts:42 -- Argument of type 'string' is not assignable to parameter of type 'AuthToken'. The last command npm run build failed with exit code 1. Working directory: ~/project/backend, git branch: feature/auth-refactor. This is a Node.js project using TypeScript.
```

Same intent. Dramatically better result.

The whole process -- from hotkey press to enhanced prompt on your clipboard -- takes under 5.5 seconds, and most of that is the voice transcription.

The Five-Stage Pipeline

Under the hood, PromptPulse runs a five-stage async pipeline every time you trigger it:

```
Hotkey -> Terminal Capture -> Voice Input -> Context Building -> LLM Enhancement -> Clipboard
```

Each stage is designed to be fast, modular, and fault-tolerant. Let me walk through each one.

Stage 1: Terminal Context Capture

This is where PromptPulse gets its edge. Instead of asking you to paste your terminal output, it reads it directly.

The system supports four terminal backends, each progressively more capable:

tmux is the richest. If you're inside a tmux session, PromptPulse runs `tmux capture-pane` to grab your screen buffer -- the actual text visible in your terminal -- plus your current directory, running process, and session metadata. All three tmux commands run in parallel via `asyncio.gather`, keeping the snapshot under 200ms.

iTerm2 uses the official Python API to pull screen contents, session variables, and even prompt marks for command history. macOS only, but if you're already in iTerm2, it's seamless.

Shell Hook is the universal option. PromptPulse installs a lightweight hook into your shell (zsh, bash, or fish) that writes a JSON state file on every command. No screen buffer, but it captures your working directory, last command, and exit code from any terminal emulator.

Generic is the fallback. It reads your shell history file and detects your CWD via `lsdf` (macOS) or `/proc` (Linux). Always available, no setup required.

Backend detection is automatic. PromptPulse probes in priority order and uses the best available option. You never have to configure it unless you want to.

Stage 2: Voice Capture

PromptPulse records from your microphone using energy-based voice activity detection. When you press the hotkey, it starts by listening to 0.5 seconds of silence to calibrate your ambient noise level, then sets the speech threshold at 3x the noise floor. This means it adapts to your environment automatically -- coffee shop, quiet office, mechanical keyboard, it adjusts.

Once it detects speech, it records until you stop talking (configurable silence timeout, default 1 second). The entire capture happens in 30ms frames streamed through an async queue. No files touch disk until transcription.

Stage 3: Transcription

Three engines, automatic fallback:

- **faster-whisper** runs locally with int8 quantization. No API calls, no data leaves your machine. This is the default.
- **Apple Speech** uses macOS's built-in `SFSpeechRecognizer` via PyObjC. Also fully local.
- **OpenAI Whisper API** for when you want cloud accuracy and don't mind the latency.

If your preferred engine isn't available, PromptPulse falls through to the next one automatically. You don't get an error -- you get a result.

Stage 4: Context Building and Error Detection

This is where the raw terminal snapshot becomes structured intelligence.

The **error detection engine** runs 12 regex patterns against your screen buffer, each with named capture groups that extract file paths, line numbers, error codes, and messages. It covers TypeScript, ESLint, Python tracebacks, Rust compiler errors, Go, Node.js, Jest, pytest, git conflicts, and permission errors. Duplicates are filtered by a seen-set keyed on `(error_type, file, line, code)`.

The **project detector** walks up from your working directory checking for marker files -- `package.json`, `Cargo.toml`, `go.mod`, `pyproject.toml`, `Gemfile`, `pom.xml`, and nine others. It identifies both the project type and root directory without you ever mentioning them.

Everything gets assembled into a frozen, immutable `ContextPayload`: your voice transcript, terminal state, detected errors, project info, git branch, and recent commands.

Stage 5: LLM Enhancement

The context payload feeds into a meta-prompt template -- a carefully crafted instruction set that tells the LLM to act as a "prompt engineer specializing in developer productivity." The template has eight rules, including: always include file paths and error codes, preserve the user's original intent (don't add tasks they didn't ask for), write in second person, and keep it under 200 words.

The LLM client uses LiteLLM under the hood, which means you can point it at Ollama (local, private, free), OpenAI, or Anthropic with a single config change. The default is Ollama with Llama 3.2, meaning **the entire pipeline runs locally with zero API calls**.

If the LLM fails, PromptPulse doesn't give up. It has a fallback template that concatenates your voice input with the detected errors and terminal context into a structured prompt. Not as polished, but still far better than what you would have typed.

The LLM client also classifies errors as transient (rate limits, timeouts, 5xx) versus permanent (auth failures, model not found) and only retries the former, with exponential backoff. Small detail, but it means the tool doesn't hang for 30 seconds when your API key is wrong.

Privacy by Default

This was a non-negotiable design decision. The default configuration uses:

- **Local transcription** via faster-whisper (no audio leaves your machine)
- **Local LLM** via Ollama (no prompts leave your machine)
- **No persistence** of screen buffer contents to disk
- **No telemetry**, no analytics, no phone-home

Cloud providers are available as explicit opt-in. If you want GPT-4o's quality or Whisper API's speed, you can configure that. But out of the box, everything stays on your hardware.

Graceful Degradation as Architecture

The pattern that ties the whole system together is graceful degradation. Every component has a fallback:

Component	Primary	Fallback
Terminal backend	tmux (full screen buffer)	Generic (shell history)
Transcription	faster-whisper (local)	Apple Speech or Whisper API
LLM enhancement	Ollama/OpenAI/Anthropic	Template-based prompt building
Clipboard	Native (pbcopy/xclip)	pyperclip library
Input mode	Voice	Text or clipboard contents

Nothing is a hard dependency. You can run PromptPulse without tmux, without a microphone, without an LLM running, and it will still give you something better than what you started with.

Using It

Install:

```
pip install prompt-pulse
# or
uv pip install prompt-pulse
```

Initialize and set up:

```
prompt-pulse init          # Creates ~/.prompt-pulse/config.yaml
prompt-pulse install-hook  # Installs shell hook (restart shell after)
```

Three ways to use it:

```
# Voice input -- press hotkey, speak, get enhanced prompt on clipboard
prompt-pulse start

# Direct text enhancement
prompt-pulse enhance "fix the build error"

# Enhance whatever's on your clipboard with terminal context
prompt-pulse enhance --clipboard
```

In daemon mode, the default hotkeys are:

Hotkey	Action
Ctrl+Shift+P	Voice capture, enhance, copy to clipboard
Ctrl+Shift+L	Enhance clipboard text with terminal context
Ctrl+Shift+R	Re-enhance the last prompt
Esc	Cancel voice capture

The Numbers

The full source is about 2,500 lines of Python across 12 modules, with 870 lines of tests. The performance budget breaks down like this:

Stage	Target	Notes
Hotkey detection	<50ms	pynput listener in thread
Terminal snapshot	<200ms	Parallel subprocess calls
Voice transcription	<2s	Local Whisper, int8 quantized
Context building	<100ms	Regex + filesystem reads
LLM enhancement	<3s	Local Ollama
Clipboard delivery	<100ms	Native CLI tools

Total	<5.5s	End-to-end after speech ends
-------	-------	------------------------------

Why This Matters

AI coding assistants are remarkably capable, but they're limited by the quality of their input. The gap between what a developer *means* and what they *type* is where productivity dies. Context switching between terminal and chat window, manually copying error messages, remembering to mention the branch and project type -- it's all friction that compounds across a day of work.

PromptPulse eliminates that gap. You express intent; it supplies context. The AI gets a prompt it can act on immediately instead of one it needs to interrogate you about.

It's the difference between a junior developer who says "it's broken" and a senior developer who files a bug report with the stack trace, reproduction steps, and the commit that introduced the regression.

Except now you get the senior developer's prompt by pressing a hotkey and saying three words.

PromptPulse is open source and available on [GitHub](#). It runs on macOS and Linux, requires Python 3.11+, and works entirely offline by default.