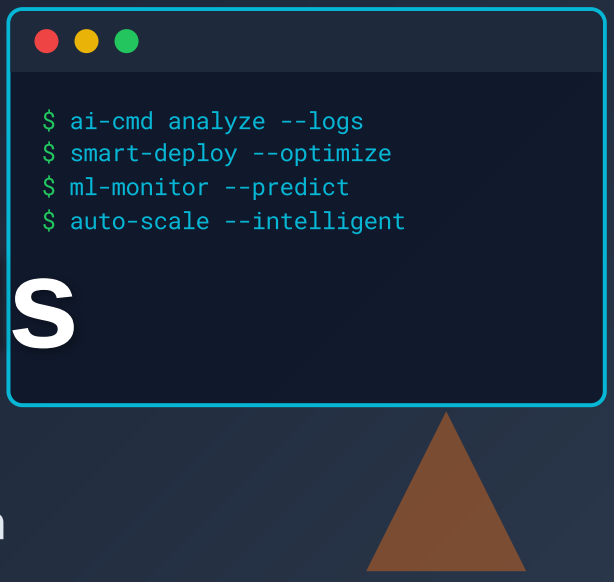
TECHNICAL PLAYBOOK

September 2025

Version 1.0

The Ultimate Playbook for AI-Powered Terminal Commands



Advanced Automation & Intelligence Integration

A comprehensive guide to leveraging artificial intelligence for enhanced command-line productivity, automated workflows, and intelligent system administration.

Advanced Systems Research Division

AI-CMD-PB-2025

Classification: Technical Guide

Table of Contents

The Ultimate Playbook for AI-Powered Terminal Commands

Introduction: The Philosophy of Peak Productivity

How much time do you lose each day to small inefficiencies? The answer is likely more than you think. According to an analysis of common developer workflows, it can be **upwards of 47 minutes daily**. Over a year, that accumulates to more than 100 hours—time that could have been spent solving complex problems, learning new skills, or simply logging off earlier.

> *The difference between an average developer and a great one isn't talent—it's their environment.*

This philosophy is rooted in cognitive science. Nobel laureate Daniel Kahneman's research on the brain's two operating systems—System 1 (fast, automatic) and System 2 (slow, deliberate)—reveals a crucial insight. Every time you manually search for a file, context-switch between your terminal and a browser, or struggle to remember a complex command, you drain your finite System 2 cognitive reserves. These small "paper cuts" add up, pulling you out of the state of deep focus known as "flow."

The chart below visualizes the daily time cost of these common, seemingly minor, workflow frictions.

The solution is not to work harder, but to build an environment that works for you. By automating repetitive tasks and streamlining your workflow, you can offload cognitive burdens to your tools, preserving mental energy for what truly matters: writing great code. This playbook introduces a set of tools designed to do exactly that, turning your terminal into a powerful, intelligent assistant. This setup is based on the principles outlined in the Ultimate Developer Dotfiles project.

Meet the Gemini Slash Commands: Your AI Copilot

At the heart of this optimized environment are the **AI-Powered Gemini Slash Commands**. These are not simple aliases; they are powerful, pre-configured prompts that integrate Google's Gemini AI directly into your command-line workflow. They are designed to act as expert personas, allowing you to generate high-quality code, documentation, summaries, and more with a single, intuitive command.

Key benefits include:

- **Automation & Enhancement:** Turn simple prompts into powerful, structured outputs for complex tasks.
- **Expert Personas:** Get consistent, high-quality results for everything from writing code to summarizing meetings.
- **Streamlined Workflow:** Integrate advanced AI capabilities directly into your terminal, eliminating the need for context switching.

These commands transform your terminal from a simple command executor into a conversational partner, ready to assist with every phase of your development lifecycle.

The Command Reference: A Detailed Breakdown

The following is a comprehensive guide to the custom slash commands available in this setup. They are grouped by category for easy reference. Each entry includes a description, best practices, and a practical example.

System & File Management

Commands for interacting with your operating system and organizing files intelligently.

Command	Description & Usage
`/sys:health-check`	**Description:** Provides a quick summary of system health (disk, memory, CPU). **When to Use:** Before starting a new project or as a daily check-up to monitor system performance. **Example:** `> /sys:health-check`
`/file-organizer`	**Description:** Intelligently organizes files in a specified directory. **When to Use:** To bring order to a chaotic directory like 'Downloads'. **Best Practice:** Always review the suggested commands before executing, as this tool moves files. Consider backing up the directory first. **Example:** `> /file-organizer "test_files"`
`/sys-search`	**Description:** Translates a natural language query into a `ripgrep` command for fast file searching. **When to Use:** When you need to find files or code snippets based on content or name. **Example:** `> /sys-search "find all functions named 'organize_directory' in my python files"`
`/cleanup`	**Description:** Identifies and suggests cleanup opportunities in a codebase or directory. **When to Use:** After a project is complete or when a directory feels cluttered with temporary files or build artifacts. **Example:** `> /cleanup "the current project directory"`

Development & Version Control

Streamline your coding, debugging, and Git workflows.

Command	Description & Usage
`/code:refactor`	**Description:** Refactors a piece of code based on a specific instruction. **Best Practice:** Be specific. Instead of "make it better," say "extract this logic into a separate function." **Example:** `> /code:refactor "Refactor this Python code to be more efficient by creating a reverse mapping of extensions to folder names" @pyclean.py`
`/doc:generate`	**Description:** Generates professional, language-aware documentation (docstrings, comments) for code. **When to Use:** Immediately after writing a function or class, before you forget the details. **Example:** `> /doc:generate @my_script.py`
`/debug-assistant`	**Description:** Analyzes error logs and provides systematic debugging strategies. **Best Practice:** Provide the full error message, the code that caused it, and what you were trying to do for the best analysis. **Example:** `> /debug-assistant "My Python script is failing with a PermissionError. Here is the code: @script.py"`
`/test-create`	**Description:** Generates a complete, runnable test file for the provided code. **When to Use:** Once a function's logic is stable, to quickly generate a comprehensive test suite. **Example:** `> /test:create @pyclean.py`
`/git-commit`	**Description:** Generates a Conventional Commit message from staged changes. **Best Practice:** Pipe the output of `git diff --staged` to the command for an accurate message reflecting only the changes to be committed. **Example:** `> git diff --staged | /git-commit`
`/docker-explain`	**Description:** Explains a `Dockerfile` or `docker-compose.yml` in plain English. **When to Use:** When encountering a new project with a complex Docker setup. **Example:** `> /docker-explain @Dockerfile`

Content & Communication

Draft emails, summarize meetings, and generate reports without leaving the terminal.

Command	Description & Usage
`/content-summarize`	**Description:** Provides a multi-format summary of any text. **When to Use:** To quickly digest long articles, documents, or reports. **Example:** `> /content-summarize "pasted long article text..."`
`/email-draft`	**Description:** Drafts professional emails based on context, tone, and purpose. **Best Practice:** Provide a simple instruction: "who it's for," "what the goal is," and "what the key points are." **Example:** `> /email-draft "Draft a weekly progress report to my manager, Jane. Tell her we've completed the core logic for the PyClean script."`
`/meeting-summary`	**Description:** Summarizes meeting notes into actionable insights. **Best Practice:** Paste the raw, unstructured notes. The AI is excellent at finding key points and action items in messy text. **Example:** `> /meeting-summary "Meeting Notes: John, Sarah. Discussed PyClean. Sarah finished core logic. AI for Sarah: research audio file extensions."`

Productivity & Planning

Organize tasks, plan projects, and enhance your learning.

Command	Description & Usage
`/plan:new`	**Description:** Transforms a vague idea into a structured project plan. **When to Use:** At the very beginning of a project. **Example:** `> /plan:new "a simple Python script that organizes files into subfolders based on extension."`

Command	Description & Usage
`/ticket-create`	**Description:** Formats a simple description into a structured bug report or feature ticket. **When to Use:** To break down a project plan into actionable tasks for systems like Jira or GitHub Issues. **Example:** `> /ticket:create "Feature: Core file organization logic. The script needs to scan a directory and move files."`
`/task-prioritizer`	**Description:** Creates organized, actionable task lists with time estimates from a raw list. **When to Use:** When you have a long to-do list and feel overwhelmed. **Example:** `> /task-prioritizer "My tasks are: deploy pyclean, write the README, email my manager, plan v2."`
`/search-advanced`	**Description:** Activates an advanced search agent for up-to-date information and technical research. **When to Use:** When you need documentation, comparisons, or tutorials for a technical topic. **Example:** `> /search:advanced "Best Python library for modern file system operations, pathlib vs os.path"`

Security

Integrate security checks directly into your development workflow.

Command	Description & Usage
`/security-audit`	**Description:** Performs a comprehensive security audit of code and dependencies. **When to Use:** Before committing code, especially if it handles user input, files, or network requests. **Best Practice:** Use it as a regular step in your workflow, like a linter, to catch potential issues early. **Example:** `> /security-audit "the python script at pyclean.py"`

A Practical Workflow: Building a Project with AI

To see how these commands work in concert, let's walk through the creation of a simple project: "PyClean," a Python script to organize files. This demonstrates how the AI commands can assist at every stage of the development lifecycle.

Phase 1: Project Inception and Planning

Every project begins with an idea. We use the planning commands to transform that idea into an actionable structure.

1. **Check System Readiness:** Ensure your system is ready for work.

```
> /sys:health-check
```

2. **Generate a Project Plan:** Turn a one-sentence idea into a structured plan.

```
> /plan:new "a simple Python script named pyclean.py that organizes files in
```

3. **Create Actionable Tickets:** Break the plan down into tasks for your project management tool.

```
> /ticket:create "Feature: Core file organization logic. The script needs to
```

Phase 2: Development and Research

With a plan in place, we start writing code, using AI to research, document, and refine.

1. **Research Best Practices:** Decide on the best tools for the job.

```
> /search:advanced "Best Python library for modern file system operations, pa
```

2. **Write Code & Generate Documentation:** After writing the initial function, immediately generate its docstring.

```
# In pyclean.py
def organize_directory(source_path_str):
    # ... function logic ...

# In terminal
> /doc:generate @pyclean.py
```

3. **Refactor for Improvement:** Ask the AI to optimize the code you've written.

```
> /code:refactor "Refactor this Python code to be more efficient by creating
```

Phase 3: Testing and Debugging

No code is complete without robust testing. The AI helps generate tests and diagnose failures.

1. **Generate Test Cases:** Create a complete test file for your script.

```
> /test:create @pyclean.py
```

2. **Debug Failures:** If a test fails, ask the debug assistant for help by providing the error message.

```
> /debug-assistant "My Python script is failing with a PermissionError when I
```

Phase 4: Version Control and Communication

Once the code is working, it's time to commit it and communicate progress.

1. **Generate a Commit Message:** After staging your changes, create a conventional commit message automatically.

```
# In your shell
git add pyclean.py test_pyclean.py
```

```
git diff --staged | /git-commit
```

2. **Draft a Progress Report:** Quickly write a professional email to update stakeholders.

```
> /email-draft "Draft a weekly progress report to my manager. Tell her we've
```

Installation and Setup

Transforming your environment is straightforward. You can get started with a single command, which handles backing up existing configurations, installing all necessary tools, and setting up the symbolic links.

One-Line Installation

Run the following command in your terminal to begin the automated setup:

```
curl -fsSL https://raw.githubusercontent.com/yomazini/dotfiles/main/install.sh -o install.
```

The installer is designed to be safe and transparent. It will:

- **Backup** your existing `.vimrc`, `.tmux.conf`, and `.zshrc`.
- **Install** all required tools and dependencies using your system's package manager.
- **Create** symbolic links to the new dotfiles.
- **Set up** plugins and themes for Vim, Tmux, and Zsh.

The Gemini slash commands are automatically installed into the `~/.gemini/commands` directory, ready for immediate use.

For more details, manual installation steps, and a full list of included tools, please visit the official repository: yomazini/dotfiles on GitHub.

Conclusion: An Extension of Your Mind

The goal of a truly optimized development environment is to make your tools disappear. They should not be obstacles to overcome but seamless extensions of your own thought process. By automating the mundane, streamlining the complex, and integrating intelligence directly into your workflow, you free up your most valuable resource: your focus.

This playbook provides a blueprint for achieving that state. The Gemini slash commands, combined with the powerful Vim, Tmux, and Zsh configurations, are more than just a collection of tools—they are a new way of interacting with your machine. Stop fighting your tools. Make them an extension of your mind.

⭐ Star this repo if it helped you! ⭐

Made with ☕ and perseverance by **Youssef Mazini**. Connect with me on LinkedIn.