Welcome to OopisOS v4.0

"Your Friendly Neighborhood LLM"

What is OopisOS?

OopisOS is a sophisticated OS simulation that runs entirely on your local machine. It's a self-contained, persistent world built on a foundation of privacy and exploration, featuring a rich command-line environment, a secure multi-user file system, and a suite of powerful integrated tools. All user data is stored locally; your world remains your own.

The "Friendly Neighborhood LLM Edition" integrates a powerful and flexible AI toolkit directly into the OS core, while enforcing the true portability that allows your entire digital environment to travel with you on any USB drive, leaving no trace behind.

Key Features at a Glance

- **Your AI Copilot, On Your Terms**: The `gemini` command is now a versatile AI gateway. Chat directly with your own local models via Ollama or LM Studio, or use the default provider for powerful, file-system-aware tool use.
- **A Rock-Solid File System:** A persistent VFS powered by IndexedDB with a full suite of management tools: `ls`, `find`, `tree`, `diff`, `cp`, `mv`, `rm`, `zip`, and `unzip`.
- **An Empowered Command-Line:** Full history, tab-completion, piping (`|`), redirection (`>`), backgrounding (`&`), sequencing (`;`), and environment variable support with a customizable prompt via `PS1`.
- **A True Multi-User Security Model:** Create users (`useradd`) and groups (`groupadd`). Manage permissions with `chmod`, `chown`, and `chgrp`. Execute commands with elevated privileges using `sudo` and safely edit the rules with `visudo`.
- **A Full Suite of Applications:**
    - `edit`: A powerful text editor with live Markdown preview.
    - `paint`: A character-based art studio for your inner artist.
    - `explore`: A graphical, two-pane file explorer.
    - `chidi`: An AI-powered document analysis tool.
    - `gemini`: A tool-using AI assistant for your terminal.
    - `adventure`: A powerful, data-driven text adventure engine to play and build interactive fiction.
    - `basic`: An integrated development environment for the classic BASIC programming language.
    - `log`: A journaling system for secure, private logs.

Quick User Guide

This guide covers the essential commands for navigating and interacting with the OopisOS environment. For a quick list of all commands, type `help`. For a detailed manual on a specific command, type `man [command_name]`.

Command Crash Course

Command & Key Flags	What It Does
`help [cmd]`	Displays a list of commands or a command's basic syntax.
`ls [-l -a -R -r -t -S -X -d]`	Lists directory contents. The cornerstone of observation.
`tree [-L level] [-d]`	Like `ls`, but fancier. Displays files and directories in a tree.
`cd [directory]`	Changes your current location in the file system.
`pwd`	Prints the full path of your current working directory.
`mkdir [-p] [dir_name]`	Creates a new, empty directory.
`rmdir`	Removes empty directories.
`cat [-n] [file...]`	Displays the entire content of a file. Use `-n` for line numbers.
`echo [text]`	Repeats text back to you. Primarily used for writing to files.
`touch [-c -d 'date'] [file]`	Creates an empty file or updates its timestamp.
`rm [-r -f -i] [item]`	Removes (deletes) a file or directory. This is permanent.
`cp [-r -f -p -i] [src] [dest]`	Copies a file or directory.
`mv [-f -i] [src] [dest]`	Moves or renames a file or directory.
`grep [-i -v -n -c -R] [pat]`	Searches for a text pattern within files.
`find [path] [expr]`	A powerful tool to find files based on various criteria.

Data & System Utilities

Command & Key Flags	What It Does
`head [-n lines] [-c bytes]`	Outputs the first part of a file.
`tail [-n lines] [-c bytes]`	Outputs the last part of a file.
`more` / `less`	Pagers to display content one screen at a time.
`diff [file1] [file2]`	Compares two files and shows the differences.
`sort [-r -n -u] [file]`	Sorts lines of text alphabetically or numerically.
`uniq [-c -d -u] [file]`	Filters or reports repeated adjacent lines.
`wc [-l -w -c] [file]`	Counts the lines, words, and bytes in a file.
`shuf [-n count] [-i L-H]`	Generates a random permutation of its input lines.
`awk 'program' [file]`	A powerful tool for pattern-based text processing.
`csplit`	Splits a file into sections determined by context lines.
`xargs [command]`	Builds and executes command lines from standard input.
`bc`	An arbitrary-precision calculator language.
`df [-h]`	Reports the file system's overall disk space usage.
`du [-h -s] [path]`	Estimates disk space used by a specific file/directory.
`cksum [file]`	Calculates a checksum and byte count for a file.
`sync`	Commits filesystem caches to persistent storage.
`base64 [-d]`	Encodes or decodes data to standard output.
`ocrypt`	Simple password-based data obscurity.

Application Suite

OopisOS comes with several built-in applications that run in a full-screen, modal interface, providing a richer experience than standard command-line tools.

- **`explore [path]`** Opens a graphical, two-pane file explorer. The left pane shows an expandable directory tree, and the right pane shows the contents of the selected directory. It's a powerful and intuitive way to navigate your file system.

- **`edit [file]`** Opens the OopisOS text editor. It's a surprisingly capable editor with features like live Markdown and HTML preview (`Ctrl+P`), a formatting toolbar, and keyboard shortcuts for saving (`Ctrl+S`) and exiting (`Ctrl+O`).

- **`paint [file.oopic]`** Unleash your creativity with the character-based art editor. It features a full canvas, multiple tools (pencil, eraser, shapes), a color palette, and saves your work to the custom `.oopic` format.

- **`chidi [path]`** The AI Librarian. Point `chidi` at a directory of Markdown files (`.md`) to open a dedicated reading interface. Here, you can ask the AI to summarize documents, generate study questions, or perform a query across all loaded documents to find specific information.

- **`adventure [--create] [file.json]`** Launches the powerful, data-driven text adventure engine. Play the built-in "Architect's Apprentice" or load your own custom adventures from a `.json` file. Use the `--create` flag to enter an interactive editor to build your own worlds. For a full guide on creating your own adventures, see `/docs/adventure.md`.

- **`gemini "<prompt>"`** A tool-using AI assistant that can execute shell commands to answer questions about your file system or provide general knowledge.

- **`basic [file.bas]`** Launches the Oopis Basic Integrated Development Environment. It's a throwback to the classic days of computing, allowing you to write, `LIST`, and `RUN` line-numbered programs. Use `SAVE` and `LOAD` within the IDE to manage your `.bas` files.

- **`log`** Keep a captain's log of your daily adventures.

Advanced Course

Ready to wield true power? This section covers the tools that separate the administrators from the users.

Shell Customization: The `PS1` Variable

You have full control over your command prompt's appearance via the `PS1` environment variable. Use `set` with a string containing special, backslash-escaped characters.

Sequence	Description
`\u`	The current username (e.g., `Guest`).
`\h`	The hostname (e.g., `OopisOs`).
`\w`	The full path of the current working directory.
`\W`	The basename of the current working directory.
`\$`	Displays a `#` if the user is `root`, otherwise a `$`.
`\\`	A literal backslash character.

Privilege Escalation: `sudo` and `visudo`

Some tasks require the power of the `root` user. `sudo` is your key to borrowing that power, safely and temporarily.

- **`sudo [command]`** Executes a single command with `root` privileges. You will be prompted for *your own* password to verify your identity.
- **`visudo`** The only safe way to edit the `/etc/sudoers` file, which controls who can use `sudo`. It locks the file and performs a syntax check on save to prevent you from locking yourself out.

File Archival: `zip` and `unzip`

Manage collections of files by bundling them into a single archive.

- **`zip [archive.zip] [path_to_zip]`** Creates a simulated `.zip` archive containing the file or directory.
- **`unzip [archive.zip] [destination_path]`** Extracts the contents of a `.zip` archive into the specified destination.

Advanced Command Chaining: `&&`, `||`, and Pipes

The shell supports powerful logical operators for more intelligent command-line workflows.

- **The AND Operator (`&&`)**: The command to the right runs **only** if the command to the left succeeds. `mkdir new_project && cd new_project`
- **The OR Operator (`||`)**: The command to the right runs **only** if the command to its left *fails*. `grep "FATAL" system.log || echo "No fatal errors found."`

Developer Documentation

A Developer's Guide to El Código del Taco

1. Design Philosophy: The Tao of Code

This document provides a comprehensive architectural overview of the OopisOS project, intended for developers who will build upon, extend, or maintain the system. To contribute effectively, one must first understand the philosophy that underpins its construction.

OopisOS is an exploration of a fully **self-reliant**, **secure**, and **persistent** client-side application paradigm. Its design is governed by five foundational pillars:

1. **Radical Self-Reliance**: The system is 100% client-side, with no backend dependency for its core logic.
2. **Architected Persistence**: The system state is not ephemeral; the user is the sole custodian of their locally persisted data.
3. **Enforced Modularity**: The system is composed of discrete, specialized components that are orchestrated, never intermingled.
4. **Security by Design**: A strict permission model, sandboxed execution, and secure credential handling are foundational, not features.
5. **Contained & Orchestrated Execution**: All operations flow through controlled channels, ensuring predictability and preventing side effects.

2. The "El Código del Taco" Architectural Model

The most effective way to understand the system is through the "El Código del Taco" model, which deconstructs the application into seven distinct, concentric layers.

Layer	Ingredient	Responsibility	OopisOS Implementation
1	**The Protein**	**Core Business Logic:** The reason the app exists.	`commexec.js`, `lexpar.js`, `fs_manager.js`
2	**The Lettuce**	**Presentation Layer:** The UI/UX.	`terminal_ui.js`, `output_manager.js`, `basic_app.js`, ...
3	**The Cheese**	**Feature & Enhancement Layer:** Valuable but non-essential features.	`BasicInterpreter.js`, `gemini.js`, `chidi.js`, `alias.js`
4	**The Salsa**	**API & Data Layer:** The unifying agent for data access.	`storage.js` (`StorageManager`, `IndexedDBManager`)
5	**The Onions**	**Utility & Environment Layer:** Potent, non-negotiable helpers.	`utils.js`, `config.js`
6	**The Jalapeño**	**Security & Validation Layer:** Controlled heat and protection.	`sudo_manager.js`, `fs_manager.js`'s `hasPermission`
7	**The Fold**	**Build & Deployment:** The final, critical assembly.	`index.html`, `sw.js`

3. Core Component Deep Dive

The Command System: Secure by Process

The `CommandExecutor` is the heart of the system, orchestrating a secure command lifecycle.

1. **Input & Expansion**: User input is received. The `CommandExecutor` expands environment variables (`$VAR`) and aliases.
2. **Tokenization & Parsing**: The `Lexer` and `Parser` in `lexpar.js` convert the command string into a structured, executable representation.
3. **Validation (The Command Contract)**: The `CommandExecutor` validates the parsed command against its registered definition, checking argument counts, path validity, and permissions **before** any logic is run.
4. **Execution**: The command's `coreLogic` is invoked with a secure, validated context object.

The File System: A Bastion of State

- **Structure & Persistence**: The file system exists as a single JavaScript object (`fsData`) that is serialized and stored in IndexedDB via the `FileSystemManager`.
- **Security Gateway**: All file system operations, without exception, must pass through the `hasPermission()` gatekeeper within the `FileSystemManager`, which is the final authority on access control.

User & Credential Management

- **Secure Storage**: Passwords are never stored in plaintext. `UserManager` uses the browser's native **Web Crypto API (SHA-256)** to securely hash all passwords.
- **Centralized Authentication**: All login logic resides in `UserManager`. Secure password input is handled by the `ModalInputManager` to prevent leakage into command history or the screen.

The BASIC Subsystem: A Sandboxed Environment

The Oopis BASIC feature is a two-part system designed for safe, sandboxed code execution:

- **`basic_app.js` (The IDE):** This is a presentation-layer component that provides the full-screen Integrated Development Environment. It manages the UI, program buffer (`LIST`, `SAVE`, `LOAD`), and user commands (`RUN`, `NEW`).
- **`BasicInterpreter.js` (The Engine):** This is a self-contained language parser and executor. Crucially, it has **no direct access** to the file system or command executor.

To interact with the host OS, the interpreter uses a specific set of `SYS_` functions (`SYS_CMD`, `SYS_READ`, `SYS_WRITE`). These functions act as a secure bridge, routing their requests through the main `CommandExecutor` and `FileSystemManager`. This design ensures that any program run via the BASIC interpreter is still subject to the OS's fundamental permission model, maintaining system integrity.

4. Extending the System: The Command Contract

Adding a new command to OopisOS is a simple and secure process that follows a clear, declarative pattern. As of v3.3, you **do not** need to add a `<script>` tag to `index.html`. The `CommandExecutor` will dynamically load your command script on first use.

Step 1: Create the Command File

Create a new file in `/scripts/commands/`. The filename must exactly match the command name (e.g., `mycommand.js`).

Step 2: Define the Command Contract

At the top of your file, create a `const` object for your command's definition. This object declares your command's requirements to the `CommandExecutor`.

```
// scripts/commands/mycommand.js

const myCommandDefinition = {

  commandName: "mycommand",

  // Define accepted flags  flagDefinitions: [    { name: "force", short: "-f", long:
"--force" },    { name: "outputFile", short: "-o", takesValue: true }  ],

  // Define argument count rules  argValidation: {    min: 1,    max: 2,    error:
"Usage: mycommand [-f] [-o file] <source> [destination]"  },

  // Define which arguments are paths and their validation rules  pathValidation: [
{ argIndex: 0, options: { expectedType: 'file' } },    { argIndex: 1, options: {
allowMissing: true } }  ],

  // Define required permissions on those paths  permissionChecks: [    { pathArgIndex:
0, permissions: ["read"] }  ],

  // The core logic function comes next...  coreLogic: async (context) => { /* ... */
}};
```

Step 3: Write the Core Logic

The `coreLogic` function is an `async` function that receives a single `context` object. This object contains everything your command needs, already parsed and validated by the `CommandExecutor`.

```
coreLogic: async (context) => {

  const { args, flags, currentUser, validatedPaths, options } = context;

  // Your logic here. You can trust that:  // - `args` contains the correct number of
non-flag arguments.  // - `flags.force` is a boolean, `flags.outputFile` is a string or
null.  // - `validatedPaths[0].node` is a valid file node.  // - The user has 'read'
permission on validatedPaths[0].

  return { success: true, output: "Execution complete." };}
```

Step 4: Register the Command

Finally, register your command, its one-line description, and its detailed help text with the `CommandRegistry`.

```
// At the end of your file
const myCommandDescription = "A brief, one-line description of the command.";
const myCommandHelpText = `Usage: mycommand [options]...

The detailed help text for the 'man' command.`;

CommandRegistry.register(
    myCommandDefinition.commandName,    myCommandDefinition,    myCommandDescription,
myCommandHelpText);
```

The Command Contract

Adding a new command is a declarative process. You *declare* your command's requirements (flags, arguments, path validation, permissions) to the `CommandExecutor`, which enforces these rules *before* your command's core logic is ever run, ensuring stability and security.

Testing & Showcase Environment

A brand new OS can feel... empty. To combat this, we've included two very special scripts.

The World-Builder: `inflate.sh -` A single command that terraforms your empty home directory into a bustling ecosystem of files, directories, and secrets, all designed for you to test and explore.

The Gauntlet: `diag.sh -` Our quality assurance department in a script. It's a relentless stress test that runs a barrage of commands to ensure all systems are operational.

Application Suite

OopisEdit: The Context-Aware Creative Suite

1. Summary

The `edit` command is the cornerstone of content creation within OopisOS. It is not merely a text editor but a context-aware creative suite that intelligently adapts its interface and features to the type of file being manipulated. By providing a rich, full-screen modal application for text, Markdown, and HTML, `edit` bridges the gap between the raw power of the command line and the nuanced requirements of document authoring and script writing. It is the primary tool for generating the very content that gives the OopisOS ecosystem its depth and purpose.

2. Core Functionality & User Experience

The `edit` command launches a sophisticated modal application designed for a seamless and productive workflow. Its intelligence lies in its ability to understand what it's editing.

1. **Invocation:** A user starts the editor by typing `edit [filename]`. If the file exists, it is loaded; if not, a blank slate is presented, ready to be saved to that new file path.
2. **Context-Aware Modes:** The editor inspects the file's extension and automatically configures itself:
    - **`.txt`, `.sh`, `.js`, etc. (Text Mode):** Provides a clean, straightforward text editing experience focused on code and plain text.
    - **`.md` (Markdown Mode):** Activates a powerful split-screen view with the raw text on one side and a live, rendered HTML preview on the other. A formatting toolbar appears, offering one-click access to common Markdown syntax like bold, italics, lists, and links.
    - **`.html` (HTML Mode):** Also uses the split-screen view, rendering the user's HTML code in a sandboxed `<iframe>` for a live, safe preview.
3. **Productivity-Focused UI: - View Toggling (`Ctrl+P`):** Users can cycle between the split-screen view, an editor-only view for focused writing, and a preview-only view for clean reading.
    - **Status Bar:** A persistent status bar provides crucial information at a glance, including the current filename, dirty status (unsaved changes), line/word/character counts, and the precise cursor position.
    - **Keyboard-First Design:** The editor is built for efficiency, with essential keyboard shortcuts for saving (`Ctrl+S`), exiting (`Ctrl+O`), undo/redo (`Ctrl+Z`/`Ctrl+Y`), and text formatting.

3. Technical & Architectural Deep-Dive

The `edit` command's implementation is a prime example of the OopisOS philosophy of enforced modularity. The application is cleanly separated into two main components: `EditorManager` and `EditorUI`.

- **`EditorManager` (The Brain):** This module is the single source of truth for the editor's state. It knows nothing about the DOM. Its sole responsibilities are to track the `currentFilePath`, the `originalContent` (to determine if the file is `isDirty`), the `currentFileMode` (text, markdown, or html), and to manage the `undoStack` and `redoStack`. All core logic for text manipulation and state changes resides here.

- **`EditorUI` (The Hands):** This module is responsible for all DOM manipulation. It builds the editor's layout, renders the text area and preview pane, updates the status bar, and listens for user input events. It takes its instructions from the `EditorManager` but has no knowledge of the underlying file system or state logic.

- **Secure Rendering:** To prevent cross-site scripting (XSS) vulnerabilities, user-generated content is handled safely. Markdown is rendered using the `marked.js` library with its sanitization feature enabled. Untrusted HTML content is rendered inside a sandboxed `<iframe>`, which isolates it from the main application's DOM and scripts.

This strict separation makes the editor robust, secure, and easy to maintain. A change to the UI in `EditorUI` cannot accidentally break the state logic in `EditorManager`.

4. Synergy with the OopisOS Ecosystem

The `edit` command is the creative engine that fuels the rest of the OS. It is not an isolated tool but a central hub of activity.

- **Scripting (`run`):** It is the primary tool for writing the `.sh` scripts that are executed by the `run` command, enabling users to automate tasks and create complex programs.

- **AI Analysis (`chidi`, `gemini`):** It is used to author the `.md` documents that the `chidi` AI Librarian analyzes. It's also perfect for crafting complex, multi-line prompts to be saved in a file and then piped to the `gemini` command using `cat`.

- **System Configuration:** As a text editor, it is the natural tool for modifying system configuration files, such as `/etc/oopis.conf` to change the terminal prompt or `/etc/sudoers` (via the safe `visudo` command) to manage permissions.

- **Game Development (`adventure`):** Users can write the entire narrative, room descriptions, and item interactions for a custom text adventure in a `.json` file using `edit`, then immediately playtest it with the `adventure` command.

5. Strengths & Opportunities

Strengths:

- **Context-Awareness:** Automatically switching modes based on file extension is its most powerful and user-friendly feature.
- **Live Preview:** The split-screen preview for Markdown and HTML is an essential feature for modern development and writing workflows.
- **Robustness:** The separation of state (`EditorManager`) and UI (`EditorUI`) makes the application stable and less prone to bugs.
- **Security:** The use of sanitization and sandboxed iframes for rendering user content demonstrates a commitment to security by design.

Opportunities for Future Enhancement:

- **Syntax Highlighting:** Implementing syntax highlighting for common file types (`.js`, `.css`, `.sh`) would significantly improve the code editing experience.
- **Search and Replace:** A find/replace feature is a standard expectation for text editors and would be a valuable addition.
- **Theming:** Allowing users to select different color schemes for the editor would enhance personalization.

The `edit` command is the heart of productivity in OopisOS. It successfully elevates a simple command-line utility into a rich, graphical application without sacrificing the keyboard-driven efficiency that power users expect. By intelligently adapting to the user's needs and providing a secure, feature-rich environment for content creation, `edit` perfectly embodies the OopisOS philosophy of building powerful, self-reliant, and user-centric tools.

OopisOS and Paint: The Digital Canvas

1. Summary

The `paint` command launches the OopisOS character-based art studio, a surprisingly robust application for creating ASCII and ANSI art. It serves as a primary creative outlet within the OS, embodying the system's philosophy of providing powerful, self-contained tools that are both functional and fun. `paint` transforms the terminal from a purely textual interface into a visual canvas, offering users a unique way to express themselves and generate assets for other parts of the OopisOS ecosystem, such as the `adventure` game engine.

2. Core Functionality & User Experience

The paint application provides a focused, full-screen, modal experience for digital art creation.

1. **Invocation:** The user launches the editor with `paint [filename.oopic]`. If the file exists, it's loaded onto the canvas; otherwise, a new canvas is created, ready to be saved to that filename.
2. **The Paint Interface:** The UI is composed of three main areas:
    - **Toolbar:** A comprehensive set of tools for creation, including a pencil, eraser, shape tools (line, rectangle, ellipse), character selector, color palette, brush size controls, zoom, and a grid toggle.
    - **Canvas:** A fixed-size (80x24) grid where users draw by placing characters with specific foreground colors.
    - **Status Bar:** Provides real-time feedback on the current tool, character, color, brush size, cursor coordinates, and zoom level.
3. **Core Tools & Features:**
    - **Drawing Tools:** Users can select between a freehand `pencil`, an `eraser` to clear cells, and `shape` tools for drawing lines, rectangles, and ellipses.
    - **Character & Color:** Any printable ASCII character can be selected for drawing. A default palette of colors is available, with an option to select any custom hex color.
    - **Brush Size:** The pencil and eraser tools can be adjusted from a 1x1 to a 5x5 brush size for broader strokes.
    - **Undo/Redo:** A multi-level undo/redo stack (`Ctrl+Z` / `Ctrl+Y`) allows for non-destructive editing.
4. **Saving and File Format:** Artwork is saved to a custom `.oopic` file format. This is a JSON-based format that stores the canvas dimensions and a 2D array representing each cell's character and color, making it human-readable and easy to parse.

5. **Keyboard-Driven Workflow:** The application is designed for power users, with keyboard shortcuts for nearly every action, including tool selection (`P`, `E`), color selection (`1-6`), and saving/exiting (`Ctrl+S`, `Ctrl+O`).

3. Technical & Architectural Deep-Dive

The paint application is a model of the OopisOS modular design philosophy.

- **Separation of Concerns:** The application logic is neatly divided between `PaintManager` and `PaintUI`.
    - `PaintManager`: The "brain" that manages the application state, including the canvas data model (a 2D array of cell objects), tool selection, color, brush size, and the undo/redo stacks. It contains all the core drawing logic.
    - `PaintUI`: The "hands" responsible for all DOM manipulation. It builds the layout, renders the canvas data into a grid of styled `<span>` elements, and forwards user input events to the `PaintManager`.
- **Canvas Rendering:** The canvas is not a single `<canvas>` element but a CSS grid of individual `<span>` elements. This allows each character cell to be styled independently with its own color, making "ANSI" style art possible, and simplifies the logic for updating specific cells without redrawing the entire canvas.
- **State Management:** The `undoStack` is a core component. Before a drawing action begins, the current state of the entire canvas is pushed onto the stack. This allows for simple and reliable undo/redo functionality.
- **File Format (`.oopic`):** The choice of a JSON-based file format over a binary one is deliberate. It aligns with the transparent, text-based nature of OopisOS and allows users to inspect or even manually edit their artwork using standard tools like `cat` and `edit`.

4. Synergy with the OopisOS Ecosystem

The `paint` application is not an isolated feature but a deeply integrated part of the creative toolchain.

- **Asset Creation:** Its primary purpose is to allow users to create visual assets. These can be simple icons, title screens for scripts, or detailed maps and character portraits for custom games running on the `adventure` engine.
- **Standard File Operations:** Art files (`.oopic`) are treated like any other file in the system. They can be listed with `ls`, moved with `mv`, copied with `cp`, and organized into directories with `mkdir`, fully integrating them into the user's workflow.
- **Scripting and Automation:** A user could write a script using `run` that displays different `.oopic` files using the `cat` command (which would show the raw JSON) to create simple, frame-based animations or storyboards.

5. Strengths & Opportunities

Strengths:

- **Creative Freedom:** Provides a unique and powerful creative outlet that perfectly matches the retro-futuristic aesthetic of OopisOS.
- **Intuitive UI:** Despite its complexity, the toolbar and keyboard shortcuts make the application easy to learn and efficient to use.
- **Robust Feature Set:** The inclusion of shape tools, custom colors, brush sizes, and undo/redo elevates it far beyond a simple pixel editor.
- **Ecosystem Integration:** Its role as an asset creator for other applications, especially `adventure`, gives it a clear and compelling purpose within the OS.

Opportunities for Future Enhancement:

- **Advanced Tools:** A "fill bucket" tool for coloring large areas, and a "select" tool for moving or copying sections of the canvas, are logical next steps.
- **Animation Support:** Evolving the `.oopic` format to support multiple frames could turn the application into a powerful tool for creating animated sprites and cutscenes.
- **Character Sets:** Allowing users to select from different character sets (e.g., box-drawing characters, block elements) could enable more sophisticated artwork.

The `paint` application is a testament to the creative potential of OopisOS. It is a feature-rich, well-architected, and deeply enjoyable tool that provides a tangible benefit to users. By providing a canvas for artistic expression, `paint` solidifies the identity of OopisOS not just as a simulated operating system, but as a complete, self-contained sandbox for creation.

OopisOS and Gemini: The Tool-Using AI Assistant

1. Summary

The `gemini` command is the conversational AI powerhouse of OopisOS. It is a versatile assistant that can function as a powerful, system-aware agent capable of executing commands to answer complex questions, or as a direct-line conversationalist to locally-run models. This dual nature makes it one of the most powerful and flexible tools in the entire OS, seamlessly bridging the gap between natural language and shell commands.

Unlike `chidi`, which is a specialist application for analyzing a pre-defined set of documents, `gemini` is a generalist that can reason about the state of the system, plan a course of action, and synthesize information from multiple sources—including the live file system—to fulfill a user's request.

2. Core Concepts: The Two Modes of Operation

The `gemini` command operates in two primary modes, determined by the chosen provider.

A. Tool-Using Agent (Default `gemini` Provider)

When using the default `gemini` provider, the command becomes a sophisticated, multi-step agent. This process is entirely automated:

1. **Planning:** The AI first acts as a **Planner**. It analyzes your prompt and the current directory context, then formulates a step-by-step plan of whitelisted shell commands (`ls`, `cat`, `grep`, `find`, etc.) required to gather the necessary information.

2. **Execution:** The `CommandExecutor` securely runs the commands from the plan. The output of these commands is captured.

3. **Synthesis:** The AI then acts as a **Synthesizer**. It reviews your original question and the collected command outputs, and formulates a final, comprehensive, natural-language answer.

This is the most powerful mode, as it allows the AI to dynamically interact with your file system to answer questions like, "Which of my text files contains the word 'OopisOS'?"

B. Direct Chat (Local Providers like `ollama` or `llm-studio`)

When you specify a local provider using the `-p` flag, `gemini` acts as a direct conduit to your model. The conversational history and your prompt are sent directly, without a planning or tool-use phase. This is ideal for creative tasks, general knowledge questions, or code generation where file system context is not required. The `-f` flag can force the planner model used by the Gemini command to be passed to the local model. If the local model is properly configured, it can also interact with the filesystem just like Gemini.

3. Configuration: Connecting to Local Models

You can configure OopisOS to connect to any OpenAI-compatible local LLM provider, such as Ollama or LM Studio.

All provider configurations are stored in `scripts/config.js` within the `Config.API.LLM_PROVIDERS` object.

You can add or modify entries in this object to point to your preferred local endpoints and models.

4. Synergy with the OopisOS Ecosystem

The `gemini` command is a "meta-command" that leverages the entire OopisOS toolchain.

- **File System Tools:** It directly uses commands like `ls`, `cat`, and `find` as its "eyes and ears" to understand the user's environment. The quality of its answers to file-related questions depends on the richness of the file system it can explore.
- **Scripting:** A user can create complex prompts, save them to a `.txt` file using `edit`, and then pipe them into the `gemini` command: `cat` my_prompt.txt `| gemini`.
- **Complement to `chidi`:** `gemini` and `chidi` form a complete AI toolkit. Use `chidi` for deep analysis of a known set of documents. Use `gemini` to discover information or ask general questions about the system state.

The `gemini` command represents a paradigm shift in user interaction within OopisOS. It transforms the command line from a place where users must know the exact commands to a place where they can simply state their goals in natural language. Its ability to plan, execute tools, and synthesize results makes it an indispensable assistant for both novice and power users, truly fulfilling the promise of a human-AI collaborative operating system.

OopisOS and Chidi: A Sandbox, Tool-Using AI Agent With Memory

1. Summary

The `chidi` command launches the Chidi.md application, a cornerstone of the OopisOS ecosystem that transforms the system from a mere collection of files into a cohesive, analyzable knowledge base. It is a purpose-built platform for deep work on a specific corpus of documents, integrating a clean reading environment with a powerful, multi-faceted AI toolkit.

Unlike the generalist `gemini` command, `chidi` provides a focused, application-level experience. Its ability to recursively gather documents, present them in a dedicated modal interface, and run targeted AI analysis elevates it beyond a simple command into a flagship application for the entire OS.

2. Core Functionality & User Experience

The user's journey with `chidi` is designed for focus and power:

1. **Invocation:** The user invokes `chidi` with a path to a single `.md` file or, more powerfully, to a directory (e.g., `chidi /docs/api`).
2. **Recursive File Discovery:** The command's logic recursively traverses the specified path, gathering all `.md` files while respecting the current user's read and execute permissions. This builds the "corpus" for the application session.
3. **Modal Application Launch:** The `ChidiApp` launches in a full-screen modal view, deliberately removing the user from the terminal to allow for focused reading and analysis.
4. **The Chidi.md Interface:**
    - **Navigation:** Users can navigate between multiple files using `PREV`/`NEXT` buttons or a dropdown file selector.
    - **Clean Reading:** The main pane presents a beautifully rendered HTML view of the selected Markdown file via `marked.js`.
    - **AI Toolkit:** A dedicated control panel offers primary AI-driven actions:
        - **Summarize:** Generates a concise summary of the currently viewed document.
        - **Study:** Generates a list of potential study questions or key topics.
        - **Ask:** The most powerful feature. It allows the user to ask a natural language question across the *entire corpus* of loaded documents.

- **Session & Log Management:**
    - **Save Session:** Users can save the entire state of their analysis—the original document plus all generated AI responses—to a new, self-contained HTML file in the virtual file system.
    - **Verbose Log:** A toggle allows power users to view the AI's step-by-step reasoning, including which files it selected for context.

5. **Exit:** A clear "Exit" button or the `Esc` key closes the application, returning the user to their fully preserved terminal session.

3. Technical & Architectural Deep-Dive

The implementation of `chidi` adheres to our established modular design principles.

- **Separation of Concerns:** A clear distinction exists between the command (`chidi.js`) and the application (`chidi_app.js`), which handles all UI, state, and API interaction.
- **API Key Management:** The application correctly identifies its dependency on the Gemini API key and gracefully prompts the user for it if not found, ensuring a smooth onboarding experience.
- **Intelligent Contextual Scoping (RAG):** The "Ask" feature employs a sophisticated Retrieval-Augmented Generation (RAG) strategy. Instead of naively sending all content to the API, it first performs a local keyword search to identify the most relevant documents. It then constructs a highly-focused prompt containing only the content of these top-scoring files. This approach optimizes for relevance, reduces API costs, and dramatically improves the quality and accuracy of the generated answer.

4. Synergy with the OopisOS Ecosystem

The `chidi` command is deeply integrated with the OopisOS toolchain.

- **Content Creation:** It is the perfect companion to the `edit` command. Users can create and organize documentation, then immediately use `chidi` to analyze that work.
- **File Management:** It works in concert with `find`, `mkdir`, and `cp`, which are essential tools for organizing the document collections that `chidi` will consume.
- **Demonstration & Onboarding:** The `inflate.sh` script creates a perfect, ready-made environment for users to immediately test the full power of `chidi` on the sample `/docs` directory.
- **Complement to `gemini`:** It provides a clear functional distinction: `gemini` is a generalist conversational partner that plans and executes shell commands to answer questions, while `chidi` is a specialist application for deep, context-aware analysis of a pre-defined set of documents.

5. Strengths & Opportunities

Strengths:

- **Purpose-Built UI:** The modal interface provides a focused "application" experience that encourages deep work, free from the distractions of the command line.
- **Multi-File Corpus:** The ability to analyze an entire directory of documents at once is its killer feature, transforming a folder of disparate files into a single, queryable knowledge base.
- **Session Persistence:** The ability to save a complete analysis session (document + AI responses) to a new file transforms `chidi` from a simple reader into a genuine research and note-taking tool. This saved HTML file becomes a permanent, shareable artifact of the user's analytical work.
- **Efficient AI Implementation:** The RAG strategy for the "Ask" feature is technically sophisticated and highly effective, demonstrating a mature approach to AI integration that respects both API limits and the user's need for relevant results.
- **Seamless Integration:** It feels like a natural and powerful extension of the core OS features, providing a clear answer to the question, "What do I do with all these text files I've created?"

Opportunities for Future Enhancement:

- **Expanded File Type Support:** While the renderer can display `.txt` files, the command's file discovery logic is currently limited to `.md`. Expanding this discovery to include `.txt` or even extracting comments from code files (`.js`, `.sh`) remains a clear opportunity for growth. This would allow a developer to ask `chidi` to "explain the purpose of the `_handleAuthFlow` function" by analyzing the comments within `user_manager.js`.
- **Integration with `find`:** A future version could allow for more complex corpus creation, such as `find / -name "*-log.md" -mtime -7 | chidi`, piping a list of files directly into the application. This would enable dynamic, on-the-fly creation of knowledge bases for analysis.
- **Conversational "Ask":** The "Ask" feature is currently a one-shot query. Evolving this into a conversational model, where the user can ask follow-up questions based on the initial answer, would significantly enhance its analytical power.
- **Automated Tagging and Linking:** An advanced feature could allow `chidi` to analyze the entire corpus and suggest new connections, generating a summary document with wiki-style links between related concepts found in different files.

The `chidi` command and its associated application are an unqualified success. It is a robust, well-designed feature that adds immense value and a new dimension of utility to OopisOS. It elevates the platform from a simple shell simulation to a genuine productivity and learning environment. The implementation is sound, the user experience is polished, and its potential for future expansion is significant.

Oopis Adventure Engine

Welcome, Architect. This document is your guide to the OopisOS Text Adventure Engine, a powerful tool for creating and playing interactive fiction directly within the operating system. Whether you want to play the built-in game or build your own world from scratch, this manual will show you how.

1. How to Play

Getting started is simple. Just type `adventure` into the terminal to launch the default story, "The Architect's Apprentice."

```
> adventure
```

This will open the game in a full-screen modal. To exit the game and return to the terminal at any time, type `quit`. For a full list of in-game commands, type `help`.

You can also load custom adventures created by other users (or yourself!) by providing the path to a valid adventure file.

```
> adventure /path/to/my_game.json
```

2. The Anatomy of an Adventure

Every adventure is a world defined by a single `.json` file. This file contains all the rooms, items, characters, and logic that make up the game. The engine is designed to be data-driven, meaning you don't need to write any code to create a complex adventure; you just need to describe your world in the JSON format.

Let's break down the main components using examples from the default adventure.

Rooms

Rooms are the fundamental building blocks of your world. Each room is an object with a unique ID.

- `name`: The title of the room, displayed in the status bar.
- `description`: The main text shown to the player when they enter or `look`.
- `exits`: An object mapping directions (or custom exit names) to the ID of another room.
- `isDark`: (boolean) If `true`, the room's description will be hidden unless the player has a lit `isLightSource` item.
- `onListen` / `onSmell` / `onTouch`: Custom text that is displayed when the player uses a sensory verb (`listen`, `smell`, `touch`) in the room without a specific target.
- `points`: The number of points awarded to the player for visiting the room for the first time.

Items

Items are the objects that populate your world. They can be taken, used, and interacted with in various ways.

- `id`: A unique identifier for the item.
- `name`: The full name of the item (e.g., "brass key").
- `noun`: The primary noun for the parser (e.g., "key").
- `adjectives`: An array of adjectives to help the parser disambiguate (e.g., `take brass key` vs. `take iron key`).
- `description`: The text shown when the player `look`s at the item.
- `location`: The ID of the room, NPC, or container item where the item starts. Can also be `"player"` for starting inventory.
- `canTake`: (boolean) Determines if the player can pick up the item.
- `unlocks`: The `id` of an item this item can unlock. Used with the `unlock [item] with [key]` command.
- `points`: Score awarded for taking the item for the first time.

Stateful & Interactive Items

Items can be much more than static objects. They can have states, contain other items, and react to player actions.

- **Containers:** To make an item a container, add these properties:
    - `isContainer`: `true`
    - `isOpenable`: `true`
    - `isOpen`: `false` (initial state)
    - `isLocked`: `true` (optional)
    - `contains`: An array of item IDs that are inside.

    State-Dependent Descriptions: An item's description can change based on its `state` property. This is perfect for things that can be turned on or off.

    Complex Interactions (`onPush`, `onUse`): You can define what happens when a player `push`es, `pull`s, or `turn`s an item. This can change the item's own state and even affect other items in the world.

    The `use [item] on [target]` command allows for even more specific puzzles.

    - `onUse`: An object where keys are the `id` of the item being used.
    - `conditions`: An array of requirements that must be met for the action to succeed.
    - `failureMessage`: The message shown if conditions are not met.
    - `destroyItem`: If `true`, the item used (e.g., the page) is removed from the player's inventory.

NPCs (Non-Player Characters)

You can add characters to your world for the player to interact with.

- `dialogue`: An object mapping keywords to responses. The player triggers these with `ask [npc] about [keyword]`. A `default` response is used for `talk to [npc]` or if no keyword matches.
- `onShow`: Defines how an NPC reacts when the player uses `show [item] to [npc]`.

Daemons (Timed Events)

Daemons are background processes that can trigger events based on game turns. This is useful for providing hints or creating time-sensitive events.

- `trigger.type`: Can be `every_x_turns` or `on_turn` (for a specific move number).

- `action.type`: Currently supports `message`, which displays text to the player.

Win Conditions

You define how the game is won in the root of the JSON file.

```
"winCondition": {
       "type": "itemUsedOn",
       "itemId": "page",
       "targetId": "terminal"},
"winMessage": "You have won!"
```

- `type`: Can be `itemInRoom` (player must drop a specific item in a specific room), `playerHasItem` (player must simply acquire an item), or `itemUsedOn` (player must use one item on another).

3. Creating Your First Adventure

1. **Create a JSON File:** Create a new file named `my_adventure.json` using `touch` or `edit`.
2. **Define the World:** Start with the basic structure: a title, starting room, and a win condition.
3. **Build the Rooms:** Create at least two room objects inside the `rooms` section. Give them names, descriptions, and link them with `exits`.
4. **Add Items:** Populate your rooms with items. Create a key and a locked object, or a simple item the player needs to find.
5. **Upload the File:** Use the `upload` command to add your `.json` file to the virtual file system.
6. **Play!** Launch your adventure with `adventure my_adventure.json`.

The OopisOS Text Adventure Engine provides a flexible and powerful framework for interactive storytelling. Now go, and build something extraordinary.

Oopis Basic: A Primer on Embedded Scripting

Welcome to Oopis Basic, the integrated programming language of OopisOS. This guide details all supported keywords, commands, and special system functions, providing a robust toolset for everything from simple scripts to interactive programs. It is a throwback to the classic days of computing, allowing users to write, `LIST`, and `RUN` line-numbered programs directly within the OS.

The Oopis Basic IDE

The gateway to BASIC is its own Integrated Development Environment (IDE), launched with the `basic` command. This provides a focused, full-screen experience for writing and executing code.

IDE Commands

These commands are typed directly into the prompt and are not part of a program.

- **RUN**: Executes the program currently in memory.
- **LIST**: Displays the lines of the current program, in order.
- **SAVE "filename.bas"**: Saves the current program to the specified file in the virtual file system.
- **LOAD "filename.bas"**: Clears the current program and loads a new one from the specified file.
- **NEW**: Clears the program currently in memory.
- **EXIT**: Exits the BASIC environment and returns to the OopisOS terminal.

The Language: Keywords & Syntax

Every line in a BASIC program must start with a line number. This structure is fundamental to the language's execution flow, especially for control statements like `GOTO`.

- **PRINT**
  - Prints the value of the expression to the screen. The expression can be a string literal in quotes, a variable, or a combination of them using +.
- **INPUT**
  - Pauses the program, displays an optional prompt, and waits for the user to enter a value, which is then stored in the specified variable. String variables must end with a $.
- **LET**
  - Assigns the value of an expression to a variable. The LET keyword is optional.
- **IF / THEN**
  - If the condition is true, it performs the action. The action can be any valid BASIC statement, such as GOTO or PRINT.
- **GOTO**
  - Unconditionally jumps program execution to the specified line number.

- **GOSUB**
  - Jumps to a subroutine at the specified line number, remembering where it came from.
- **RETURN**
  - Returns from a subroutine to the line immediately following the GOSUB call.
- **REM**
  - A remark or comment. The rest of the line is ignored by the interpreter.
- **END**
  - Terminates the program execution.

OopisOS System Integration Functions

The true power of Oopis Basic lies in its integration with the host operating system. These special functions, used within expressions, allow your BASIC programs to interact directly with the underlying OS, read and write files, and execute shell commands. This is accomplished via a secure bridge, ensuring that any program run via the interpreter is still subject to the OS's fundamental permission model.

- SYS_CMD ("command")
  - Executes an OopisOS shell command and returns its output as a string.
- SYS_READ ("filepath")
  - Reads the content of a file from the virtual file system and returns it as a string.
- SYS_WRITE ("filepath", "content")
  - Writes the provided content to a specified file in the virtual file system, creating or overwriting it as needed. Returns nothing.

Roadmap and Future Enhancements

The Oopis Basic subsystem, while robust, is a foundational implementation with a clear path for future growth. The development team is actively exploring the following enhancements to increase its power and flexibility:

- **Expanded Mathematical Functions:** Integration of standard mathematical functions beyond basic arithmetic, such as `SIN()`, `COS()`, `TAN()`, `SQR()` (square root), and `RND()` (random number generation).
- **Looping Constructs:** Implementation of `FOR...NEXT` loops to allow for more complex and efficient iterative algorithms, reducing the reliance on `IF...GOTO` structures.
- **Data Structures:** Introduction of `DATA` and `READ` statements to allow for embedding static data directly within a program, as well as support for single-dimension arrays (`DIM`) for managing lists of numbers or strings.

- **Enhanced String Manipulation:** Adding functions like `LEFT$()`, `RIGHT$()`, `MID$()`, and `LEN()` to provide finer control over string processing.

- **Graphical "Poke" Commands:** A potential `SYS_POKE(x, y, char, color)` function that would allow BASIC programs to directly draw characters to the terminal screen, enabling the creation of simple animations and graphical applications from within the BASIC environment.

OopisOS and Log: The Personal Journal

1. Summary

The `log` command is the gateway to the OopisOS Personal Journal, a secure and simple application for timestamped entries. It embodies the OS philosophy of radical self-reliance by storing all entries as individual, user-owned Markdown files. It features a full-screen application mode for creating, viewing, searching, and managing the journal archive.

2. Core Functionality & User Experience

The user's interaction with the `log` command is designed for a focused, application-driven workflow.

- **Invocation:** The user types `log` with no arguments to launch the application.
- **The `log` Interface:** The full-screen modal application presents a two-pane interface.
    - **Timeline (Left Pane):** A chronological list of all journal entries, showing the date and the entry's title.
    - **Content (Right Pane):** An editable text area displaying the full Markdown content of the selected entry.
    - **Search:** A search bar allows for instant, client-side filtering of all entries by content.
    - **Creation & Editing:**
        - A "New Entry" button prompts the user for a title, then creates a new, blank entry with that title, ready for editing.
        - The main content view is a fully editable text area. Changes can be saved on demand with a "Save Changes" button or the `Ctrl+S` shortcut.

3. Technical & Architectural Deep-Dive

The `log` application is a model of OopisOS modularity, separating its command, logic, and UI components.

- **File-Based Backend:** By treating each entry as a separate file, the app leverages the existing `FileSystemManager` for all persistence and security. Entries can be backed up, moved, or deleted using standard OS commands.
- **Separation of Concerns:**
    - **`log.js` (Command):** Its primary role is to launch the main application.

    - **`log_app.js` (Application):**
        - `LogManager`: The application's "brain." Manages the state, including reading files from `~/.journal/`, filtering them for search, and handling the logic for creating and saving entries.
        - `LogUI`: The "hands." Responsible for building the two-pane interface, rendering the entry list and editable content view, and forwarding user events to the `LogManager`.

4. Synergy with the OopisOS Ecosystem

The `log` application is not an isolated tool but a natural extension of the OopisOS toolchain.

- **`edit`:** For complex, multi-file operations or direct manipulation of a log file, a power user can still use `edit ~/.journal/....md` to bypass the `log` app's interface.
- **`grep` and `find`:** Power users can perform advanced queries on their journal entries directly from the command line (e.g., `grep "OopisOS" ~/.journal/*.md`).
- **`backup`:** Since all log data is just files in the user's home directory, it is automatically included in any system `backup`.

The `log` application is a robust, secure, and intuitive tool that enhances the productivity and personal utility of OopisOS. It successfully demonstrates the power of the system's modular architecture by integrating seamlessly with core components like the file system, command executor, and modal application layer. It is a testament to the design philosophy of providing simple, powerful, and user-centric tools.

Security

I. Philosophy

OopisOS treats security as a foundational principle, not an afterthought. Our model is built on three pillars: **client-side sandboxing**, **explicit user permissions**, and **architected containment**.

Most importantly, we believe that **your data is none of our business.** OopisOS is designed to be a private, self-contained world that runs entirely in your browser. We have no servers, we collect no telemetry, and we have no access to your files or credentials.

II. The Core Security Model

Our security is not a single feature, but a series of interlocking components that govern every action within the OS.

Authentication (`UserManager`)

- **Secure Hashing:** Passwords are never stored in plaintext. They are securely hashed using the browser's native **Web Crypto API with the SHA-256 algorithm** before being stored locally.
- **Audited Flows:** Login (`login`) and user-switching (`su`) flows are handled through a single, audited authentication manager to prevent timing attacks, bypasses, or credential leakage.

Authorization (`FileSystemManager`)

- **Centralized Gatekeeping:** All file system access is gated by the `FileSystemManager.hasPermission()` function. There are no back doors.
- **Granular Permissions:** This function rigorously checks file ownership (`owner`, `group`) against the file's octal mode (`rwx`) and the current user's identity for every operation.
- **Superuser Exception:** The `root` user is an explicit, carefully managed exception that bypasses standard permission checks, as is standard in Unix-like systems.

Privilege Escalation (`SudoManager`)

- **Controlled Elevation:** The `sudo` command allows for temporary, controlled privilege escalation. Access is governed by the `/etc/sudoers` file, which is only editable by `root` via the `visudo` command.

- **Scoped Privileges:** Escalated privileges are granted for only a single command and are immediately revoked within a `try...finally` block to ensure they do not persist longer than necessary.

III. The User's Security Toolkit: Data Integrity and Transformation

Beyond the system's built-in protections, OopisOS provides you with the tools to create your own "chain of custody" for your data. These utilities allow you to verify, secure, and transport your information with confidence.

Command	Role in Security	Use Case Example
`cksum`	**Verification:** Calculates a checksum (a unique digital fingerprint) for a file. This allows you to verify that a file has not been altered or corrupted.	`cksum my_script.sh`
`base64`	**Transformation:** Encodes binary data into plain text. This is essential for safely sending or storing complex files in text-based systems without data loss.	`cat photo.oopic | base64 > photo.txt`
`ocrypt`	**Obscurity (Educational):** A simple password-based cipher for obscuring data. **This is not secure encryption**, but serves to demonstrate the principles of data transformation.	`ocrypt mypass secret.txt > hidden.dat`
`sync`	**Persistence:** Manually forces all pending filesystem changes to be written to the database, ensuring data integrity before a critical operation.	`sync`

This suite of tools embodies the OopisOS philosophy: security is not just something the system *has*, it is something the user *does*. By combining these commands, you can create a verifiable workflow to ensure your data remains exactly as you intended.

IV. Data Privacy & Persistence

OopisOS is designed to be completely private.

- **Local Storage:** All your data—the file system, user accounts, and session information—is stored exclusively in your browser's `localStorage` and `IndexedDB`. It never leaves your computer.
- **User Control:** You have full control over your data. You can export it with the `backup` command or permanently erase it with the `reset` command.

V. Best Practices for Users

- **Guard the Root Password:** Do not share your `root` password. It provides unrestricted access to the entire virtual file system.
- **Principle of Least Privilege:** Operate as a standard user (`Guest`) for daily tasks. Only use `su` or `sudo` when administrative privileges are required.
- **Audit Permissions:** Regularly review file permissions using `ls -l` to ensure they are set as you expect.
- **Be Wary of Unknown Scripts:** Be cautious when running scripts (`run` command) or viewing files from untrusted sources, just as you would on any other OS.

VI. Reporting a Vulnerability

The security of OopisOS is our top priority. If you believe you have found a security vulnerability, we encourage you to report it to us responsibly.

Please email a detailed description of the issue to **oopismcgoopis@gmail.com**. We are committed to working with you to understand and resolve the issue promptly.

OopisOS v3.7 Command Reference

A Note on the Structure of This Document

1. **Observation & Security:** The foundational tools for seeing the file system and understanding its rules (`ls`, `chmod`, `find`).
2. **User & Group Management:** The commands that manage the actors in our security model (`useradd`, `sudo`, `chown`).
3. **Fundamental File Operations:** The essential tools for daily work (`cat`, `mkdir`, `cp`, `rm`).
4. **Text Processing & Automation:** A powerful suite of utilities for manipulating data and automating tasks (`grep`, `sort`, `awk`, `run`).
5. **Networking & System Integrity:** Tools for connecting to the outside world and managing the system's state (`wget`, `backup`).
6. **High-Level Applications:** The suite of built-in, full-screen applications (`edit`, `paint`, `chidi`).
7. **Shell & Session Control:** Commands for customizing your environment and managing your session (`alias`, `set`, `history`, `logout`).

Core Concepts: Observation & Security

Command	Description
`ls`	Lists directory contents. Use `-l` for details, `-a` for hidden files, `-R` for recursion.
`tree`	Displays the contents of a directory in a visually structured, tree-like format.
`pwd`	Prints the full, absolute path of your current working directory.
`diff`	Compares two files line by line, showing additions, deletions, and common lines.
`df`	Reports the virtual file system's overall disk space usage. Use `-h` for human-readable sizes.
`du`	Estimates and displays the disk space used by a specific file or directory.
`chmod`	Changes the permission mode of a file (e.g., `chmod 755 script.sh`).
`find`	Searches for files based on criteria like name (`-name`), type (`-type`), or permissions (`-perm`).
`cksum`	Print checksum and byte counts of files.

The Social Fabric: User & Group Management

Command	Description
`useradd`	Creates a new user account and their home directory.
`removeuser`	Permanently deletes a user account, with an option to remove their home directory.
`groupadd`	Creates a new user group.
`groupdel`	Deletes an existing user group.
`usermod`	Modifies a user's group memberships (`-aG <group> <user>`).
`passwd`	Changes a user's password interactively.
`chown`	Changes the user ownership of a file or directory.
`chgrp`	Changes the group ownership of a file or directory.
`sudo`	Executes a single command with superuser (root) privileges.
`visudo`	Safely edits the `/etc/sudoers` file to manage who can use `sudo`.
`login`	Logs in as a different user, replacing the current session entirely.
`logout`	Logs out of a stacked session created with `su`, returning to the previous user.
`su`	Switches to another user, stacking the new session on top of the old one.
`whoami`	Prints the username of the currently active user.
`groups`	Displays the group memberships for a specified user.
`listusers`	Lists all registered user accounts on the system.

The Workshop: Fundamental File Operations

Command	Description
`mkdir`	Creates a new directory. Use `-p` to create parent directories as needed.
`rmdir`	Removes empty directories.
`touch`	Creates a new empty file or updates the timestamp of an existing one.
`echo`	Writes arguments to the output. Used with `>` or `>>` to write to files.
`cat`	Concatenates and displays the content of one or more files.
`head`	Outputs the first part (default: 10 lines) of a file.
`tail`	Outputs the last part (default: 10 lines) of a file.
`cp`	Copies files or directories. Use `-r` for recursive, `-p` to preserve metadata.
`mv`	Moves or renames files and directories.
`rm`	Removes (deletes) files or directories. Use `-r` for recursive, `-f` to force.
`zip`	Creates a simulated `.zip` archive containing a specified file or directory.
`unzip`	Extracts files and directories from a simulated `.zip` archive.
`upload`	Opens a dialog to upload files from your local machine into the OopisOS file system.
`export`	Opens a dialog to download a file from the OopisOS file system to your local machine.

The Assembly Line: Text Processing & Automation

Command	Description
`grep`	Searches for a pattern within files or standard input (`-i` for case-insensitivity).
`sort`	Sorts lines of text from a file or standard input (`-n` for numeric, `-r` for reverse).
`uniq`	Reports or filters out adjacent repeated lines (`-c` to count, `-d` for duplicates).
`wc`	Counts lines (`-l`), words (`-w`), and bytes (`-c`) in files or standard input.
`awk`	A powerful pattern-scanning and text-processing language for complex data extraction.
`more`	Displays content one screen at a time.
`less`	An improved pager for displaying content.
`bc`	An arbitrary-precision calculator language.
`csplit`	Splits a file into sections determined by context lines.
`shuf`	Generates a random permutation of its input lines.
`xargs`	Builds and executes command lines from standard input, connecting commands.
`run`	Executes a shell script (`.sh` file), with support for arguments (`$1`, `$@`).
`delay`	Pauses execution for a specified number of milliseconds. Essential for scripts.
`printscreen`	Captures all visible text in the terminal and saves it to a file.
`base64`	Encodes or decodes data to standard output.

The Bridge: Networking & System Integrity

Command	Description
`wget`	A non-interactive network downloader for fetching files from URLs.
`curl`	A versatile tool for transferring data from or to a server, often used for API interaction.
`ps`	Displays a list of currently running background processes started with `&`.
`kill`	Terminates a background process by its Job ID (found via `ps`).
`savestate`	Manually saves a complete snapshot of the current user's session and file system.
`loadstate`	Restores the last manually saved state for the current user.
`backup`	Creates a secure, downloadable backup file of the entire OS state with a checksum.
`restore`	Restores the entire OS from a downloaded backup file, wiping the current state.
`ocrypt`	Simple symmetric encryption/decryption using a password.
`sync`	Commit filesystem caches to persistent storage.
`clearfs`	Permanently erases all contents within the current user's home directory.
`reboot`	Reboots the OopisOS virtual machine by reloading the page, preserving all data.
`reset`	Wipes ALL OopisOS data (users, files, settings) and performs a factory reset.

The Cockpit: High-Level Applications

Command	Description
`edit`	Opens a powerful, full-screen text editor with live Markdown/HTML preview.
`paint`	Launches a graphical, character-based art studio for creating ASCII and ANSI art.
`explore`	Opens the graphical file explorer.
`chidi`	The "AI Librarian." Launches a modal application to read and analyze Markdown files with AI.
`gemini`	Interacts with a tool-using Gemini AI model that can execute commands to answer questions.
`adventure`	Starts the interactive text adventure game engine.
`basic`	Launches the Oopis Basic IDE for creating and running `.bas` files.
`log`	A personal, timestamped journal and log application.

The Environment: Shell & Session Control

Command	Description
`help`	Displays a list of all commands or a specific command's syntax.
`man`	Formats and displays the detailed manual page for a given command.
`history`	Displays or clears the command history for the current session (`-c` to clear).
`clear`	Clears the terminal screen of all previous output.
`alias`	Creates or displays command shortcuts (e.g., `alias ll='ls -l'`).
`unalias`	Removes one or more defined aliases.
`set`	Sets or displays session-specific environment variables (e.g., `set MY_VAR="hello"`).
`unset`	Removes one or more environment variables.
`date`	Displays the current system date and time.
`check_fail`	A diagnostic tool to verify that a given command correctly produces an error.

FAQ

What is OopisOS and what are its core principles?

OopisOS is an operating system that represents a collaborative endeavor between human direction and artificial intelligence, as stated in its MIT License. It is designed with open access, shared innovation, and transparent acknowledgment of its novel creation methods as core principles. The directory structure reveals a comprehensive system with scripts for core functionalities like file system management, user management, and session management, alongside various applications and commands.

How does OopisOS manage file system permissions and security?

OopisOS implements a robust security model with "Centralized Gatekeeping" through the FileSystemManager.hasPermission() function, ensuring all file system access is strictly controlled. It employs "Granular Permissions," checking file ownership (owner, group) against the file's octal mode (read, write, execute) and the current user's identity. The "root" user is a carefully managed "Superuser Exception" that bypasses standard checks. For "Privilege Escalation," the sudo command allows temporary, controlled elevation, governed by the /etc/sudoers file, which only root can edit via visudo. These escalated privileges are scoped to a single command and immediately revoked.

What are some of the key utilities available in OopisOS for data and system management?

OopisOS offers a wide array of data and system utilities, many of which mirror standard Unix-like commands. These include:

- **Text Processing:** head, tail, diff, sort, uniq, wc, awk, grep.
- **File System & Disk Usage:** df (disk free) and du (disk usage) for reporting space usage.
- **Data Manipulation:** base64 for encoding/decoding, cksum for checksums, and xargs for building and executing commands from standard input.
- **Archival:** zip and unzip for creating and extracting archives.
- **Scripting:** The ability to create and execute .sh scripts with argument support ($1, $@, $#) and error handling, though governed by a maximum script steps limit to prevent infinite loops.

How does OopisOS handle command-line input and parsing?

The LexPar module is responsible for processing command-line input. The Lexer component breaks the raw input string into a sequence of Token objects, identifying different TokenTypes such as WORD, STRING_DQ (double-quoted strings), STRING_SQ (single-quoted strings), and various OPERATORs (e.g., >, >>, <, |, &&, ||, ;, &). The Parser then takes these tokens to build a structured, executable representation of the command(s), handling concepts like command arguments, redirection, and piping.

What kind of applications and interactive experiences does OopisOS offer?

Beyond standard command-line utilities, OopisOS includes several applications:

- **chidi_app.js**: A Markdown file reader and analyzer, capable of viewing and interacting with .md files, even processing them from piped input. It supports Markdown previewing and syntax highlighting.
- **editor.js.bak**: A text editor, presumably for creating and modifying files.
- **explorer.js**: A file explorer for navigating the file system.
- **oopis_paint.js**: A painting application, suggesting graphical capabilities beyond a purely text-based terminal, with features like color selection, brush size, and drawing shapes.
- **text_adventure.js**: An interactive text adventure game, which includes sophisticated natural language processing for understanding player commands, disambiguating items and characters, and managing game state (inventory, location, score).

Can users customize their OopisOS environment?

Yes, users can customize their environment. The alias command allows users to create and manage shortcuts for commands, such as alias ll='ls -la' for a long directory listing. These aliases can contain spaces and special characters if quoted. Additionally, the set command enables users to manage environment variables, like set GREETING="Hello World", which can then be expanded and used within commands or scripts, such as echo $GREETING.

How does OopisOS handle paths and file system navigation?

OopisOS supports both absolute paths (starting with /) and relative paths (. for the current directory, .. for the parent directory). The fs_manager.js script includes functions like resolveAbsolutePath to correctly handle these paths, even when they involve special characters or spaces. Permissions are strictly enforced during navigation; for instance, the cd command requires 'execute' permissions on the target directory. The system also defines constants for file system elements like ROOT_PATH, CURRENT_DIR_SYMBOL, PARENT_DIR_SYMBOL, and PATH_SEPARATOR.

How does Portable Mode work?

Portable Mode is enabled by default. All your data is stored in the `./data` folder. This allows you to keep the entire OS and all its data on a portable drive, like a USB stick.

I'm running in Portable Mode from a USB drive. Why are some commands slow?

The performance of OopisOS in Portable Mode is directly tied to the read and write speed of the portable media it's running on. USB flash drives, especially older ones, can be significantly slower than an internal SSD. Operations that involve heavy file system access—like running the `diag.sh` script, unzipping large archives with `unzip`, or performing a recursive search with `find`—may take longer. This is expected behavior and a trade-off for the convenience of portability.

I switched computers and now my API keys for the `gemini` command are gone. What happened?

Your Gemini API key is intentionally not included in a system `backup` and does not travel with the Portable Mode `data` folder. This is a security measure. The API key is stored in your browser's (or the Electron app's) `localStorage`, which is specific to that installation. This prevents your private API key from being accidentally shared if you give a copy of your OopisOS folder to someone else. You will need to re-enter your API key the first time you use an AI-powered command on a new machine.

About & Credits

The Creators

OopisOS is the unlikely result of a caffeine-and-code-fueled fever dream, a collaboration between:

- **Andrew Edmark** (The Human Element, Chief Architect)
- **Gemini** (The AI Assistant, Humble Narrator, and Primary Cause of Bugs-Turned-Features)

The Social Contract (aka The Boring Legal Bit)