ChatGPT

# GhostLink Full Stack Implementation

## DreamShell GUI

The DreamShell GUI is a minimalist Tkinter-based overlay providing a familiar desktop-style control panel for GhostLink [1]. It allows visual interaction with GhostLink's tools and data without typing commands. Key features include:

- **Tool Launcher:** Launch diagnostics or macros from a drop-down menu and a **Run** button.
- **System Stats:** Display real-time CPU and memory usage, updated continuously in the GUI.
- **Vault Browser:** Open any vault file (like `macros.vault`) in a text window to view or edit its contents, with a save function.

```
#!/usr/bin/env python3
"""
DreamShell GUI: Minimal GUI overlay for GhostLink.
Inspired by Windows XP-style simplicity with a dark neon theme.
Provides buttons to launch tools/macros, view system stats, and browse vault
files.
Connects to GhostLink core functions via tool scripts.
"""
import tkinter as tk
from tkinter import filedialog, messagebox

# Optionally import other GhostLink tool modules for integration
try:
    import vault_manager  # for macro execution
except ImportError:
    vault_manager = None

# Create main window
root = tk.Tk()
root.title("GhostLink DreamShell")
root.configure(bg="#222222")  # dark background for neon aesthetic

# A frame for main controls (like a taskbar panel)
toolbar = tk.Frame(root, bg="#333333")
toolbar.pack(side=tk.TOP, fill=tk.X)

# System stats label (will be updated periodically)
stats_label = tk.Label(root, text="CPU: --%, Mem: --%", fg="#00FF00",
bg="#222222")
stats_label.pack(pady=5)
```

```
# Option menu for available macros (if macros.vault exists)
macro_options = []
selected_macro = tk.StringVar(root)
selected_macro.set("")  # default empty

try:
    with open("macros.vault", "r") as f:
        for line in f:
            line = line.strip()
            if not line or line.startswith("#"):
                continue
            # Assume format "name: actions"
            if ":" in line:
                name, actions = line.split(":", 1)
                name = name.strip()
                if name:
                    macro_options.append(name)
except FileNotFoundError:
    macro_options = []

if macro_options:
    selected_macro.set(macro_options[0])
    macro_menu = tk.OptionMenu(toolbar, selected_macro, *macro_options)
    macro_menu.config(bg="#444444", fg="#00FF00", activebackground="#555555",
activeforeground="#00FF00")
    macro_menu.pack(side=tk.LEFT, padx=5, pady=5)
    # Run macro button
    def run_selected_macro():
        macro = selected_macro.get()
        if vault_manager:
            vault_manager.run_macro(macro)
        else:
            print(f"[DreamShell] Would run macro: {macro}")
    run_btn = tk.Button(toolbar, text="Run Macro", command=run_selected_macro,
bg="#444444", fg="#00FF00")
    run_btn.pack(side=tk.LEFT, padx=5, pady=5)

# Button to update system stats
def update_stats():
    # Optionally, integrate psutil for real stats
    try:
        import psutil
        cpu = psutil.cpu_percent(interval=0.1)
        mem = psutil.virtual_memory().percent
        stats_label.config(text=f"CPU: {cpu:.1f}%  Mem: {mem:.1f}%")
    except ImportError:
        # Simulated stats if psutil not available
        stats_label.config(text="CPU: 42%  Mem: 73%")
    # schedule next update
    root.after(1000, update_stats)

update_stats()  # start updating immediately
update_stats_btn = tk.Button(toolbar, text="System Stats", command=update_stats,
bg="#444444", fg="#00FF00")
update_stats_btn.pack(side=tk.LEFT, padx=5, pady=5)

# Vault browser - open a vault file in a new window for viewing/editing
def open_vault():
    file_path = filedialog.askopenfilename(title="Open Vault File",
filetypes=[("Vault files", "*.vault"), ("All files", "*.*")])
    if file_path:
        try:
            with open(file_path, "r") as vf:
                content = vf.read()
        except Exception as e:
            messagebox.showerror("Error", f"Failed to open vault: {e}")
            return
        # Create a new window to display and edit the vault content
        vault_win = tk.Toplevel(root)
        vault_win.title(f"Vault - {file_path}")
        vault_text = tk.Text(vault_win, bg="#111111", fg="#00FF00",
insertbackground="#00FF00")
        vault_text.insert("1.0", content)
        vault_text.pack(expand=True, fill=tk.BOTH)
        # Save button to write back changes to file
        def save_vault():
            new_content = vault_text.get("1.0", tk.END)
            try:
                with open(file_path, "w") as vf:
                    vf.write(new_content)
                messagebox.showinfo("Vault Saved", f"Saved changes to
{file_path}")
            except Exception as e:
                messagebox.showerror("Save Error", f"Could not save
{file_path}: {e}")
        save_btn = tk.Button(vault_win, text="Save", command=save_vault,
bg="#444444", fg="#00FF00")
        save_btn.pack(side=tk.BOTTOM, pady=2)

open_vault_btn = tk.Button(toolbar, text="Browse Vault", command=open_vault,
bg="#444444", fg="#00FF00")
open_vault_btn.pack(side=tk.LEFT, padx=5, pady=5)

# Launch the GUI (if running this script directly)
if __name__ == "__main__":
    root.geometry("400x200")
    root.mainloop()
```

## ColdForge Builder

ColdForge is the GhostLink build/deployment toolkit for assembling a pristine core bundle with an immutable manifest [2]. The **ColdForge Builder** script automates packaging the GhostLink core for distribution. It performs the following steps:

- Scans the GhostLink core directory and computes a SHA-256 hash for each core file.
- Generates a JSON manifest mapping filenames to hashes (for use by the integrity verifier `verify_and_restore.py` [3]).
- Packages all core files **and** the manifest into a zip archive (the “pristine” bundle) to guarantee clean deployment on each run.

```
#!/usr/bin/env python3
"""
ColdForge Builder: Rebuilds the GhostLink core bundle.
Gathers all core files, computes SHA-256 hashes for a manifest,
and packages them into a pristine zip file for distribution.
Ensures consistency for verify_and_restore integrity checks.
"""
import os, hashlib, json, zipfile

# Configurable version and output names
VERSION = "1.0"
MANIFEST_FILE = f"GhostLink_v{VERSION}_Manifest.json"
BUNDLE_ZIP = f"GhostLink_Core_v{VERSION}.zip"
CORE_DIR = "."  # base directory of GhostLink core files

# File patterns to exclude from packaging/hashing (e.g., logs, outputs)
EXCLUDE_EXT = {".zip", ".log"}
EXCLUDE_NAMES = {MANIFEST_FILE}

file_hashes = {}
# Walk through all files in the core directory
for root, dirs, files in os.walk(CORE_DIR):
    for fname in files:
        # Skip hidden files and excluded types
        if fname.startswith('.'):
            continue
        if any(fname.endswith(ext) for ext in EXCLUDE_EXT):
            continue
        if fname in EXCLUDE_NAMES or "Manifest" in fname:
            continue
        # Compute SHA-256 hash of the file
        fpath = os.path.join(root, fname)
        rel_path = os.path.relpath(fpath, CORE_DIR)
        h = hashlib.sha256()
        with open(fpath, "rb") as f:
            for chunk in iter(lambda: f.read(4096), b""):
                h.update(chunk)
        file_hashes[rel_path] = h.hexdigest()

# Write the manifest JSON file
with open(MANIFEST_FILE, "w") as mf:
    json.dump(file_hashes, mf, indent=4)
print(f"[ColdForge] Generated manifest {MANIFEST_FILE} with {len(file_hashes)} entries.")

# Create the zip bundle with all core files and the manifest
with zipfile.ZipFile(BUNDLE_ZIP, "w", zipfile.ZIP_DEFLATED) as bundle:
    for rel_path in file_hashes.keys():
        bundle.write(os.path.join(CORE_DIR, rel_path), arcname=rel_path)
    bundle.write(MANIFEST_FILE, arcname=MANIFEST_FILE)
print(f"[ColdForge] Packaged core files and manifest into {BUNDLE_ZIP}.")
```

## Mechanical Recursion Kit

The Mechanical Recursion Kit implements GhostLink's structured diagnostic routine – a deterministic sequence of steps to troubleshoot hardware issues [4]. It provides four key functions that break down a diagnosis into clear phases:

- `recall_symptoms()` – Logs or retrieves stored symptoms at the start (e.g. reads error codes or user-reported issues) [5].
- `scan_system()` – Scans sensors and system status to gather current data (for example, reading live values from engine sensors).
- `isolate_fault()` – Analyzes the collected information to pinpoint the likely root cause of the problem.
- `propose_solution()` – Suggests a corrective action or repair for the identified fault (to be confirmed by the user before execution).

```
#!/usr/bin/env python3
"""
Mechanical Recursion Kit: Structured diagnostics functions.
Provides recall_symptoms(), scan_system(), isolate_fault(), propose_solution()
for systematic troubleshooting. Simulates reading logs/sensors and identifying faults.
"""
# Optionally import modules for sensor data or error codes (e.g., OBD interface)
try:
    import obd_tools as obd
```

```
except ImportError:
    obd = None

# Shared diagnostic data (persistent context across steps)
DIAG_DATA = {}

def recall_symptoms():
    """Recall stored symptoms or error codes (e.g., from logs or ECU memory)."""
    print("RECALL: Checking stored symptoms and error codes...")
    codes = []
    # If OBD tool is available, retrieve Diagnostic Trouble Codes (DTCs)
    if obd and hasattr(obd, "read_dtc"):
        codes = obd.read_dtc()
        if codes:
            print(f"RECALL: Found stored DTCs: {', '.join(codes)}")
        else:
            print("RECALL: No DTCs found.")
    else:
        # Simulated fallback if no OBD interface
        codes = ["P0456"]  # example trouble code
        print(f"RECALL: (Simulated) Stored DTCs: {', '.join(codes)}")
    DIAG_DATA['dtc_codes'] = codes
    return codes

def scan_system():
    """Scan system sensors and status to gather current readings."""
    print("SCAN: Scanning system sensors and status...")
    sensor_readings = {}
    if obd and hasattr(obd, "get_engine_rpm"):
        # Use OBD tools to get live data if available
        try:
            rpm = obd.get_engine_rpm()
            temp = obd.get_coolant_temp()
        except Exception as e:
            rpm = None
            temp = None
        if rpm is not None:
            sensor_readings['engine_rpm'] = rpm
            print(f"SCAN: Engine RPM = {rpm} rpm")
        if temp is not None:
            sensor_readings['coolant_temp'] = temp
            print(f"SCAN: Coolant Temperature = {temp} °C")
    else:
        # Simulated sensor data if no direct hardware interface
        sensor_readings['engine_rpm'] = 3000
        sensor_readings['coolant_temp'] = 85
        print("SCAN: (Simulated) Engine RPM = 3000 rpm, Coolant Temp = 85 °C")
    DIAG_DATA['sensors'] = sensor_readings
    return sensor_readings

# Lookup table for known DTC code meanings (for isolate_fault stage)
CODE_MEANINGS = {
    "P0123": "Throttle/Pedal Position Sensor A Circuit High Input",
    "P0456": "Evaporative Emission System Small Leak Detected"
}

def isolate_fault():
    """Analyze symptoms and sensor data to isolate the likely fault cause."""
    print("🔍 ISOLATE: Analyzing data to identify probable cause...")
    codes = DIAG_DATA.get('dtc_codes', [])
    sensors = DIAG_DATA.get('sensors', {})
    fault_cause = None
    if codes:
        # If a DTC is present, use it as primary clue
        code = codes[0]
        cause = CODE_MEANINGS.get(code, "Unknown Issue")
        fault_cause = f"{code}: {cause}"
        print(f"🔍 ISOLATE: Likely fault cause identified - {fault_cause}")
    else:
        # No error codes; use sensor anomalies as hints
        if sensors.get('engine_rpm') == 0:
            fault_cause = "Engine not running - possible ignition or fuel supply issue"
        elif sensors.get('coolant_temp') and sensors['coolant_temp'] > 100:
            fault_cause = "Engine overheating - potential cooling system failure"
        else:
            fault_cause = "No obvious fault from sensor data"
        print(f"🔍 ISOLATE: {fault_cause}")
    DIAG_DATA['fault'] = fault_cause
    return fault_cause

def propose_solution():
    """Propose a solution or next step based on the isolated fault."""
    print("🛠 PROPOSE: Formulating solution for the identified fault...")
    fault = DIAG_DATA.get('fault', "")
    solution = None
    if not fault or "No obvious fault" in fault:
        solution = "Perform further diagnostics or consult detailed logs."
    elif "P0123" in fault:
        solution = "Check and replace the throttle position sensor; inspect wiring for faults."
    elif "P0456" in fault:
        solution = "Inspect fuel cap and EVAP system for leaks; replace faulty components as needed."
    elif "overheating" in fault:
        solution = "Check coolant level, radiator fan operation, and thermostat functionality."
    elif "ignition or fuel supply" in fault:
        solution = "Verify fuel pump and ignition coils; check fuses and relays in those circuits."
    else:
        solution = "Apply known fix or seek specialized analysis for the fault."
    print(f"🛠 PROPOSE: Suggested solution - {solution}")
    DIAG_DATA['solution'] = solution
    return solution
```

## OBD-II and Bus Tools

This module provides GhostLink's vehicle diagnostics interface, allowing it to function as an OBD-II CAN bus scanner [6]. It includes examples of reading trouble codes, querying live sensor data, and sending OBD mode commands:

- `read_dtc()` – Reads Diagnostic Trouble Codes (DTCs) from the vehicle's ECU (simulating an OBD-II mode 03 request to get stored error codes).
- `get_engine_rpm()` & `get_coolant_temp()` – Return live engine data like RPM and coolant temperature by simulating OBD-II sensor PID queries.
- `clear_dtc()` – Sends a command to clear the stored DTCs (OBD-II mode 04), demonstrating how to reset the check-engine light.
- `scan_obd_bus()` – Scans the CAN bus for active ECU addresses (e.g. detecting engine ECU at 0x7E0, transmission at 0x7E1, etc.).
- `scan_serial_ports()` – Example of scanning serial ports for devices (e.g. listing connected microcontrollers or OBD adapters on COM ports).

```
#!/usr/bin/env python3
"""
OBD-II and Bus Tools: Functions to interact with a CAN-based OBD-II interface.
Includes reading diagnostic trouble codes (DTCs), reading live sensor data,
scanning the CAN bus for devices, and sending OBD mode commands.
"""
# Simulated ECU state for demonstration
STORED_DTC = ["P0123", "P0456"]  # example trouble codes stored in ECU memory

def read_dtc():
    """Read Diagnostic Trouble Codes from the vehicle's ECU (OBD-II mode 03)."""
    print("OBD: Reading DTCs from ECU...")
    if STORED_DTC:
        print(f"OBD: Retrieved trouble codes: {', '.join(STORED_DTC)}")
    else:
        print("OBD: No trouble codes stored.")
    return STORED_DTC.copy()
```

```python
def get_engine_rpm():
    """Read the current engine RPM (OBD-II PID 0C)."""
    rpm = 3000  # example fixed value for RPM
    print(f"OBD: Engine RPM = {rpm} rpm")
    return rpm

def get_coolant_temp():
    """Read the engine coolant temperature (OBD-II PID 05)."""
    temp_c = 85  # example fixed coolant temperature in °C
    print(f"OBD: Coolant Temperature = {temp_c} °C")
    return temp_c

def clear_dtc():
    """Send a command to clear DTCs (OBD-II mode 04)."""
    print("OBD: Sending clear DTC command (mode 04)...")
    STORED_DTC.clear()
    print("OBD: Trouble codes cleared.")
    return True

def scan_obd_bus():
    """Scan the CAN bus for OBD-II ECU addresses (devices)."""
    print("OBD: Scanning CAN bus for ECUs...")
    # Simulate finding two common ECU addresses
    devices = ["7E0 (Engine ECU)", "7E1 (Transmission ECU)"]
    print(f"OBD: Found devices: {', '.join(devices)}")
    return devices

def scan_serial_ports():
    """Scan serial ports for connected devices (e.g., OBD adapters or
microcontrollers)."""
    print("Bus: Scanning serial ports for devices...")
    # Simulated result of scanning COM ports
    found_ports = ["COM3: Arduino", "COM4: OBD-II Adapter"]
    for port in found_ports:
        print(f"Bus: Detected device on {port}")
    return found_ports

# Note: In a real implementation, libraries like `python-can` or `pySerial`
would be used for actual bus communication.
```

## Voltage/GPIO Interface

This module handles low-level hardware control, enabling GhostLink to measure analog voltages and toggle digital outputs [7]. It simulates direct interaction with microcontroller ADC inputs and GPIO pins for electrical diagnostics:

- `read_voltage(pin)` – Reads an analog input channel (e.g. "A0") and returns the voltage level. Uses predefined values or simulates a sensor reading.
- `toggle_pin(pin, state=None)` – Toggles or sets a digital GPIO pin output. If `state` is provided (True/False), sets the pin HIGH or LOW; if not, it flips the current state.
- `log_measurement(pin, value)` – Logs a measurement with timestamp to a persistent vault file (e.g. appends to `sensor_trace.vault` for later analysis of sensor trends).

```
#!/usr/bin/env python3
"""
Voltage/GPIO Interface: Tools for reading analog voltages and controlling
digital GPIO pins.
Includes functions to read analog input values, toggle digital outputs, and log
measurements for analysis.
"""
import random
from datetime import datetime

# Simulated current states for digital pins
PIN_STATE = {}

# Predefined analog values for certain pins (for consistent simulation)
ANALOG_VALUES = {
    "A0": 3.30,  # e.g., 3.3 V on analog pin A0
    "A1": 1.20,  # e.g., 1.2 V on analog pin A1
    "A2": 0.00   # e.g., 0 V on analog pin A2
}

def read_voltage(pin):
    """Read an analog voltage from the specified pin (simulate ADC reading)."""
    # Determine pin identifier (accepts e.g. "A0" or numeric channel)
    if isinstance(pin, str):
        pin_id = pin.upper()
    else:
        pin_id = f"A{pin}"
    # Return a predefined value if available, otherwise simulate a new reading
    if pin_id in ANALOG_VALUES:
        voltage = ANALOG_VALUES[pin_id]
    else:
        voltage = round(random.uniform(0, 5), 2)  # simulate 0-5V reading
        ANALOG_VALUES[pin_id] = voltage  # remember this value
    print(f"GPIO: Analog read {pin_id} = {voltage} V")
```

```python
    return voltage

def toggle_pin(pin, state=None):
    """Toggle or set a digital GPIO pin output. If state is None, the pin state is inverted."""
    # Normalize pin identifier (e.g. 17 -> "GPIO17", or "GPIO17" stays same, "PA5" stays "PA5")
    if isinstance(pin, str):
        pin_id = pin.upper()
        if pin_id.startswith("GPIO"):
            # Ensure standard format "GPIO#"
            try:
                num = int(''.join(filter(str.isdigit, pin_id)))
                pin_id = f"GPIO{num}"
            except ValueError:
                pass  # leave as-is if not purely numeric after 'GPIO'
    else:
        pin_id = f"GPIO{pin}"
    # Determine new state
    if state is None:
        current = PIN_STATE.get(pin_id, False)
        new_state = not current
        PIN_STATE[pin_id] = new_state
        state_str = "HIGH" if new_state else "LOW"
        print(f"GPIO: Toggled {pin_id} -> {state_str}")
    else:
        new_state = bool(state)
        PIN_STATE[pin_id] = new_state
        state_str = "HIGH" if new_state else "LOW"
        print(f"GPIO: Set {pin_id} = {state_str}")
    return PIN_STATE[pin_id]

# Log file for sensor measurements (vault for persistent logging)
SENSOR_LOG_FILE = "sensor_trace.vault"

def log_measurement(pin, value):
    """Log a sensor measurement with a timestamp to the sensor trace vault."""
    timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    log_entry = f"{timestamp}: {pin} = {value}\n"
    try:
        with open(SENSOR_LOG_FILE, "a") as logf:
            logf.write(log_entry)
    except Exception as e:
        print(f"GPIO: Warning - could not write to log ({e})")
    else:
        print(f"GPIO: Logged measurement -> {pin} = {value}")
    return log_entry
```

## Vault and Macro Management

This script manages GhostLink's persistent vault files and user-defined macros. Macros (sequences of tool actions) are stored in plaintext in `macros.vault` for transparency [8]. The Vault manager allows you to list and edit macros, execute them, and perform generic vault file operations:

- `list_macros()` – Reads `macros.vault` and returns all macro names with their action sequences.
- `run_macro(name)` – Fetches the macro by name and executes each tool action in sequence (invoking the corresponding functions from diagnostics or hardware modules in order).
- `edit_macro(name, new_actions)` – Updates or adds a macro in the vault with a new sequence of actions (e.g. `"diagnose_engine: recall_symptoms(); scan_system(); isolate_fault(); propose_solution()"`).
- `read_vault(file)` / `write_vault(file, content)` – Utility functions to read the entire contents of a vault file or write content back to it (for managing other vaults like `core.vault`, etc.).

```python
#!/usr/bin/env python3
"""
Vault and Macro Management: Tools to list, edit, and execute macros from
macros.vault.
Also provides generic vault read/write utilities for managing GhostLink's
persistent state files.
"""
# Optionally import tool modules so macros can invoke their functions
try:
    import mechanical_recursion_kit as diag
except ImportError:
    diag = None
try:
    import obd_tools as obd
except ImportError:
    obd = None
try:
    import gpio_interface as gpio
except ImportError:
    gpio = None

MACROS_FILE = "macros.vault"

def read_vault(filename):
    """Read the entire content of a vault file as a string."""
    try:
        with open(filename, "r") as vf:
            return vf.read()
    except Exception as e:
        print(f"Vault: Error reading {filename} - {e}")
        return None

def write_vault(filename, content):
    """Write the given content to a vault file, replacing its contents."""
    try:
        with open(filename, "w") as vf:
            vf.write(content)
        return True
    except Exception as e:
        print(f"Vault: Error writing {filename} - {e}")
        return False

def list_macros():
    """List all macros defined in the macros.vault file (name -> actions)."""
    macros = {}
    content = read_vault(MACROS_FILE)
    if content is None:
        return macros
    for line in content.splitlines():
        line = line.strip()
        if not line or line.startswith("#"):
            continue
        if ":" in line:
            name, actions = line.split(":", 1)
            name = name.strip()
            actions = actions.strip()
            if name:
                macros[name] = actions
    print(f"Vault: Loaded {len(macros)} macros from {MACROS_FILE}.")
    return macros

def run_macro(name):
    """Execute the macro with the given name by running its sequence of actions."""
    macros = list_macros()
    if name not in macros:
        print(f"Vault: Macro '{name}' not found.")
        return False
    actions_str = macros[name]
    actions = [a.strip() for a in actions_str.split(';') if a.strip()]
    print(f"Macro: Executing macro '{name}' with {len(actions)} action(s)...")
    for action in actions:
        # Parse function name and arguments from the action string
        if '(' in action:
            func_name, args_part = action.split('(', 1)
            func_name = func_name.strip()
            args_str = args_part.rstrip(')')
        else:
            func_name = action.strip()
            args_str = ""
        # Parse arguments into Python values
        args = []
        for arg in [arg.strip() for arg in args_str.split(',') if arg.strip()]:
            if arg.lower() in ("true", "false"):
                val = True if arg.lower() == "true" else False
            elif arg.isdigit():
                val = int(arg)
            else:
                try:
                    val = float(arg)
                except ValueError:
                    val = arg.strip("'\"")  # treat as string
            args.append(val)
        # Find and call the corresponding function
        func = None
        if diag and hasattr(diag, func_name):
            func = getattr(diag, func_name)
        elif obd and hasattr(obd, func_name):
            func = getattr(obd, func_name)
        elif gpio and hasattr(gpio, func_name):
            func = getattr(gpio, func_name)
        else:
            print(f"Macro: Unknown action '{func_name}' - skipping.")
            continue
        try:
            func(*args)
        except Exception as e:
            print(f"Macro: Error executing {func_name} - {e}")
    print(f"Macro: Completed macro '{name}'.")
    return True

def edit_macro(name, new_actions):
    """Edit or add a macro in the macros.vault with the given action
sequence."""
    try:
        with open(MACROS_FILE, "r") as vf:
            lines = vf.readlines()
    except FileNotFoundError:
        lines = []
    updated = False
    for i, line in enumerate(lines):
        if line.strip().startswith(f"{name}:"):
            lines[i] = f"{name}: {new_actions}\n"
            updated = True
            break
    if not updated:
        lines.append(f"{name}: {new_actions}\n")
    success = write_vault(MACROS_FILE, "".join(lines))
    if success:
        print(f"Vault: Macro '{name}' {'updated' if updated else 'added'}.")
    return success
```

## Persona/Tone Profile Controls

This component manages GhostLink's persona and tone settings via a dedicated profile vault [9]. It allows toggling the system's verbosity and style to honor user preferences (e.g. switching to a more detailed or more casual response mode). The persona profile is stored in `persona.vault` and uses simple key-value settings:

- `load_persona()` – Loads the current persona profile from `persona.vault`, applying default values for tone and verbosity if not set.
- `save_persona(profile)` – Saves a given profile (dictionary of settings) back to the vault file.
- `set_persona_setting(key, value)` – Updates a specific setting (e.g. set **verbosity** to "high" or **tone** to "friendly") and persists it [10].
- `get_persona_setting(key)` – Retrieves the current value of a persona setting (for other modules to adjust behavior accordingly).

```
#!/usr/bin/env python3
"""
Persona/Tone Profile Controls: Manage verbosity and tone settings via a persona
profile vault.
Allows toggling output verbosity and tone, and updates the profile at runtime
(GhostLink core will adjust its response style accordingly).
"""
PERSONA_VAULT = "persona.vault"

# Default persona profile
DEFAULT_PROFILE = {
    "tone": "technical",
# possible values: "technical", "friendly", "formal", etc.
    "verbosity": "normal"  # possible values: "low", "normal", "high"
}

def load_persona():
    """Load persona profile from persona.vault (create with defaults if not
found)."""
    profile = DEFAULT_PROFILE.copy()
    try:
        with open(PERSONA_VAULT, "r") as pf:
            for line in pf:
                line = line.strip()
                if not line or line.startswith("#"):
                    continue
                if "=" in line:
                    key, val = line.split("=", 1)
                    key = key.strip()
                    val = val.strip().strip('"\'')
                    profile[key] = val
    except FileNotFoundError:
        # If no profile exists, save defaults
        save_persona(profile)
    return profile

def save_persona(profile):
    """Save the persona profile dictionary to persona.vault."""
    try:
        with open(PERSONA_VAULT, "w") as pf:
            for key, val in profile.items():
                pf.write(f"{key} = {val}\n")
        return True
    except Exception as e:
        print(f"Persona: Error saving profile - {e}")
        return False

def set_persona_setting(key, value):
    """Set a specific persona setting and save it (e.g. tone or verbosity
override)."""
    profile = load_persona()
    if key not in profile:
        print(f"Persona: Unknown setting '{key}'. Valid keys:
{list(profile.keys())}")
        return False
    profile[key] = value
    success = save_persona(profile)
    if success:
        print(f"Persona: Set {key} = {value}")
    return success

def get_persona_setting(key):
    """Get the current value of a persona setting."""
    profile = load_persona()
    return profile.get(key)
```

## Symbolic Integrity Loop

This script implements GhostLink's continuous integrity watchdog loop, combining proactive file verification and the **DecayDaemon** runtime checks [3]. It continuously monitors the system for any "symbolic drift" or anomalies and takes automatic corrective action:

- **File Hash Verification:** Computes SHA-256 hashes of all core files at regular intervals and compares them to the manifest. If any file is altered or corrupted, it immediately logs a warning and restores the original file from the pristine bundle [3].
- **Runtime Resource Monitoring:** Monitors system metrics (e.g. CPU usage) to catch runaway processes or stuck loops. If sustained high CPU usage is detected, it logs an anomaly and simulates restarting the offending module to maintain stability.
- **DecayDaemon Logging:** All interventions (file restores, resets, etc.) are logged with timestamps to an anomaly log, so the user can review any automatic corrections. This ensures transparency in how GhostLink self-heals and stays deterministic.

```python
#!/usr/bin/env python3
"""
Symbolic Integrity Loop: Continuous monitor for file integrity and runtime anomalies.
Watches core file hashes against the manifest and system metrics to detect 'symbolic drift'.
Engages the DecayDaemon for corrective action when anomalies are detected.
"""
import os, json, hashlib, time
from datetime import datetime
import threading

# Configuration: manifest and pristine bundle files
MANIFEST_FILE = "GhostLink_v1.0_Manifest.json"
PRISTINE_BUNDLE = "GhostLink_Core_v1.0.zip"

# Load expected hashes from the manifest file
try:
    with open(MANIFEST_FILE, "r") as mf:
        EXPECTED_HASHES = json.load(mf)
except FileNotFoundError:
    EXPECTED_HASHES = {}

ANOMALY_LOG = "anomaly.log"

# Optional: import psutil for CPU monitoring if available
try:
    import psutil
except ImportError:
    psutil = None
```

```
# Counter for consecutive high-CPU readings
_high_cpu_count = 0

def log_anomaly(message):
    """Log an anomaly event with a timestamp to the anomaly log."""
    timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    entry = f"{timestamp} - {message}\n"
    try:
        with open(ANOMALY_LOG, "a") as logf:
            logf.write(entry)
    except Exception as e:
        print(f"Integrity: Warning - could not write to log ({e})")
    print(f"[DecayDaemon] {message}")

def restore_file(filepath):
    """Restore a file from the pristine bundle zip if it was altered."""
    try:
        import zipfile
        with zipfile.ZipFile(PRISTINE_BUNDLE, 'r') as zf:
            zf.extract(filepath, path=".")
        log_anomaly(f"Restored file {filepath} from pristine bundle.")
    except Exception as e:
        log_anomaly(f"Failed to restore {filepath}: {e}")

def check_integrity():
    """Perform one cycle of integrity checks and resource monitoring."""
    global _high_cpu_count
    # Verify each core file's hash against the manifest
    for rel_path, expected_hash in EXPECTED_HASHES.items():
        try:
            with open(rel_path, "rb") as f:
                data = f.read()
        except FileNotFoundError:
            log_anomaly(f"Core file missing: {rel_path}")
            restore_file(rel_path)
            continue
        actual_hash = hashlib.sha256(data).hexdigest()
        if actual_hash != expected_hash:
            log_anomaly(f"File tampering detected: {rel_path} hash mismatch")
            restore_file(rel_path)
            # Reset the hash in memory after restoring
            try:
                with open(rel_path, "rb") as f:
                    data = f.read()
                EXPECTED_HASHES[rel_path] = hashlib.sha256(data).hexdigest()
            except Exception:
                pass
    # Check CPU usage for runaway processes (if psutil is available)
    if psutil:
        cpu = psutil.cpu_percent(interval=1)
        if cpu > 90:
            _high_cpu_count += 1
        else:
            _high_cpu_count = 0
        if _high_cpu_count >= 5:  # e.g., 5 consecutive high-CPU samples
            log_anomaly("Sustained high CPU usage detected - possible runaway
process (triggering restart)")
            _high_cpu_count = 0

# In a real system, we would reset or restart the problematic module here.

def start_monitor(interval=10):
    """Start the continuous integrity monitor loop in a background thread."""
    def monitor_loop():
        while True:
            check_integrity()
            time.sleep(interval)
    t = threading.Thread(target=monitor_loop, daemon=True)
    t.start()
    print(f"Integrity monitor started (interval={interval}s).")
    return t

if __name__ == "__main__":
    print("Starting symbolic integrity monitoring... (press Ctrl+C to stop)")
    try:
        while True:
            check_integrity()
            time.sleep(10)
    except KeyboardInterrupt:
        print("\nIntegrity monitoring stopped by user.")
```

---

1 2 3 4 5 6 7 8 9 10 GhostLink Prompt Pack (Core Memory Artifact).pdf
file://file-X4wzDg2pHbC7oQC9XvJ5KY