ChatGPT

A_CLIP_UNIFIED_FORGE_ARTIFACT_v1.md

Repository Structure

MVP (v0.1) File Structure – 16 files:

```
arifos_clip/
├── AGENTS.md
├── README.md
├── commands/
│   ├── 000.md
│   ├── 777.md
│   ├── 888.md
│   └── 999.md
├── hooks/
│   └── pre-push
├── aclip/
│   ├── __init__.py
│   ├── core/
│   │   └── session.py
│   ├── bridge/
│   │   └── arifos_client.py
│   └── cli/
│       ├── 000_void.py
│       ├── 777_forge.py
│       ├── 888_hold.py
│       └── 999_seal.py
├── docs/
│   └── ARCHITECTURE.md
└── tests/
    └── test_mvp_flow.py
```

v1 Expanded File Structure (~45 files total):

```
arifos_clip/
├── AGENTS.md
├── README.md
├── commands/
│   ├── 000.md
│   ├── 111.md
│   ├── 222.md
│   ├── 333.md
│   ├── 444.md
│   ├── 555.md
│   ├── 666.md
│   ├── 777.md
│   ├── 888.md
│   └── 999.md
├── agents/
│   ├── sense.md
│   ├── reflect.md
│   ├── reason.md
│   ├── empathize.md
│   └── align.md
├── hooks/
│   ├── pre-commit
│   ├── commit-msg
│   └── pre-push
├── aclip/
│   ├── __init__.py
│   ├── cli/
│   │   ├── 000_void.py
│   │   ├── 111_sense.py
│   │   ├── 222_reflect.py
│   │   ├── 333_reason.py
│   │   ├── 444_evidence.py
│   │   ├── 555_empathize.py
│   │   ├── 666_align.py
│   │   ├── 777_forge.py
│   │   ├── 888_hold.py
│   │   ├── 999_seal.py
│   │   ├── _dispatcher000.py
│   │   ├── _dispatcher111.py
│   │   ├── _dispatcher222.py
│   │   ├── _dispatcher333.py
│   │   ├── _dispatcher444.py
│   │   ├── _dispatcher555.py
│   │   ├── _dispatcher666.py
│   │   ├── _dispatcher777.py
│   │   ├── _dispatcher888.py
│   │   └── _dispatcher999.py
│   ├── core/
│   │   ├── session.py
│   │   ├── routing.py
│   │   ├── exits.py
│   │   └── formatting.py
│   └── bridge/
│       ├── arifos_client.py
│       ├── authority.py
│       ├── verdicts.py
│       └── time.py
├── docs/
│   └── ARCHITECTURE.md
└── tests/
    ├── test_mvp_flow.py
    ├── test_stages_flow.py
    └── test_hooks.py
```

File Contents

Below are the contents of each file in the `arifos_clip` subfolder.

FILE: arifos_clip/AGENTS.md

```
# A CLIP Agents & Roles

A CLIP employs multiple cognitive "agents" or perspectives to examine a problem
thoroughly. Each intermediate stage (111–666) can be thought of as a distinct
role or mindset:

- **Sense (111): The Observer.** Gathers raw context and facts.
- **Reflect (222): The Historian.** Recalls and reflects on past knowledge and
context.
- **Reason (333): The Logician.** Applies logic and analysis to form
conclusions.
- **Evidence (444): The Analyst.** Verifies claims with data and evidence.
- **Empathize (555): The Empath.** Considers human factors and stakeholder
perspectives.
- **Align (666): The Guardian.** Ensures alignment with core values, ethics, and
laws.

These agents work together under the APEX governance framework to ensure
decisions are well-rounded and compliant. The profiles and guidelines for key
agents are detailed in the `agents/` directory.
```

FILE: arifos_clip/README.md

```
# A CLIP – arifOS CLI Pipeline

A CLIP is a command-line pipeline (with commands `000` through `999`) for
decision governance in the **arifOS** project. It enforces that changes and
decisions go through a structured, multi-stage review aligned with APEX Theory
and are approved by the arifOS law engine.

## Pipeline Stages (000–999)

- **000 (void):** Initialize a new session from the void (blank state) with a
task description.
- **111 (sense):** Sense the context – gather initial information about the
task.
- **222 (reflect):** Reflect on knowledge – recall relevant info and context.
- **333 (reason):** Reason logically – analyze the problem and outline
solutions.
- **444 (evidence):** Gather evidence – verify facts and support arguments.
- **555 (empathize):** Empathize – consider human/stakeholder perspectives and
ethical implications.
- **666 (align):** Align – ensure alignment with core principles, laws, and APEX
values.
- **777 (forge):** Forge the output – synthesize all inputs into a final
decision package.
- **888 (hold):** Hold the process – pause for human review or if an issue is
detected.
- **999 (seal):** Seal the result – finalize the decision (requires
authorization and arifOS approval).

Each stage corresponds to a CLI command (`000` through `999`) that performs the
above actions and records the outcome.

## Usage

After installing A CLIP, use the numeric commands in sequence to carry out the
governance workflow:

1. **Start a session:** `000 void "<task description>"` – Creates a new session
and records the task.
2. **Progress through stages:** Run `111 sense`, `222 reflect`, `333 reason`,
`444 evidence`, `555 empathize`, and `666 align` in order. Each command adds its
analysis to the session log.
3. **Forge the output:** `777 forge` – Compiles all stage outputs into a final
JSON "forge pack".
4. **Apply a hold (if needed):** `888 hold --reason "reason text"` – (Optional)
```

```markdown
Invoke a hold if an issue arises. This will produce a hold report and block
sealing until resolved.
5. **Seal the decision:** `999 seal --apply --authority-token <TOKEN>` –
Attempts to finalize the decision. By default, `999 seal` runs a dry-run check.
Using `--apply` with a valid authority token will request arifOS to approve and
finalize the changes.

Each command produces output to the console and updates files under the hidden
directory `.arifos_clip/` (which tracks session state and artifacts). Use the
`--json` flag with any command to get machine-readable JSON output instead of
human-friendly text.

## Authorization & Enforcement

The **seal** stage (999) is protected by multiple safeguards:
- It **requires** an explicit `--apply` flag and a valid `--authority-token`
from a human authority to attempt applying changes.
- Even with a token, the arifOS law engine must return a verdict of **SEAL** for
the session, otherwise the seal will not proceed.
- If these conditions are not met, 999 will exit with a HOLD or SABAR code and
no external changes will be applied.

Git hooks are provided (in `arifos_clip/hooks/`) to enforce the pipeline:
- Commits are blocked if a hold exists (pre-commit) or if the session is not
sealed (commit-msg).
- Pushing to remote is blocked if any hold remains or if the session wasn't
sealed by A CLIP (pre-push).

These safeguards ensure that no unreviewed or unapproved changes leave the
repository.

## Installation

Include the `arifos_clip` package in your project and configure console scripts
for the numeric commands (see **Packaging** below). Then install the package in
your environment (e.g. with `pip install -e .`). This will make commands `000`,
`111`, ..., `999` available in your shell.

## Outputs and Exit Codes

A CLIP writes all its artifacts to a dedicated folder `.arifos_clip/` in the
repository:
- **Session file:** `.arifos_clip/session.json` – the central record of the
session, including all stages.
- **Forge pack:** `.arifos_clip/forge/forge.json` – the compiled output after
forging (777).
- **Hold bundle:** `.arifos_clip/holds/hold.json` and `.arifos_clip/holds/
hold.md` – details of any hold invoked (888).
```

```
Exit codes are used to signal the pipeline state to other tools (or scripts):
- `0` – **PASS:** Stage completed successfully (no errors).
- `20` – **PARTIAL:** Pipeline completed partially (e.g. forged but not sealed).
- `30` – **SABAR:** Execution stopped awaiting action (e.g. missing authority
token, waiting period).
- `40` – **VOID:** Void stage completed (session initialized).
- `88` – **HOLD:** A hold is in effect or a law violation blocked progress.
- `100` – **SEALED:** Final stage sealed successfully (fully approved).

Non-zero codes (except 100) indicate the pipeline did not yet reach a final
sealed state. These codes help integrate with CI or other tools to automate
checks.
```

FILE: arifos_clip/commands/000.md

```
# 000 void

**Description:** Initializes a new A CLIP session from nothingness (the "void").
This stage defines the task or problem to be addressed.

**Usage:** `000 void "<your task description>"`

This command creates the session record (`.arifos_clip/session.json`) and begins
the pipeline with the given task description.
```

FILE: arifos_clip/commands/111.md

```
# 111 sense

**Description:** Gathers initial context and facts about the task. The "sense"
stage acts as the Observer, collecting raw information and understanding the
scope.

**Usage:** `111 sense`

This stage appends initial observations and context to the session log.
```

FILE: arifos_clip/commands/222.md

```
# 222 reflect

**Description:** Reflects on relevant knowledge and past experiences related to
the task. The "reflect" stage reviews memory and background to inform the
decision process.

**Usage:** `222 reflect`

This stage records insights from recalling previous lessons, data, or context.
```

FILE: arifos_clip/commands/333.md

```
# 333 reason

**Description:** Applies logical reasoning to analyze the problem. The "reason"
stage (Logician) breaks down the task and develops rational conclusions or
solution steps.

**Usage:** `333 reason`

This stage adds a structured, critical analysis of the problem to the session.
```

FILE: arifos_clip/commands/444.md

```
# 444 evidence

**Description:** Gathers evidence and verifies facts. The "evidence" stage
compiles supporting data or examples to ensure that conclusions are well-
founded.

**Usage:** `444 evidence`

This stage adds fact-checks, data, or references to support the reasoning.
```

FILE: arifos_clip/commands/555.md

```
# 555 empathize

**Description:** Considers human and stakeholder perspectives. The "empathize"
stage introduces an empathic view to evaluate emotional, social, or ethical
implications.

**Usage:** `555 empathize`

This stage inserts considerations about how different people are affected or
what their viewpoints might be.
```

FILE: arifos_clip/commands/666.md

```
# 666 align

**Description:** Ensures alignment with core principles, ethics, and laws. The
"align" stage (Guardian) checks that the planned solution adheres to APEX values
and regulatory constraints.

**Usage:** `666 align`

This stage evaluates the solution against overarching guidelines and flags any
deviations.
```

FILE: arifos_clip/commands/777.md

```
# 777 forge

**Description:** Synthesizes all prior stage outputs into a final decision
package. The "forge" stage compiles the results into a cohesive plan or
artifact.

**Usage:** `777 forge`

This command generates the forge pack (`.arifos_clip/forge/forge.json`)
representing the combined outcome of the pipeline (ready for final review).
```

FILE: arifos_clip/commands/888.md

```
# 888 hold

**Description:** Triggers a HOLD on the pipeline. The "hold" stage is used to
pause or block the process when an issue is detected or a manual review is
required before proceeding.

**Usage:** `888 hold [--reason "<reason text>"]`

Invoking this command creates a HOLD bundle under `.arifos_clip/holds/`
(including `hold.json` and `hold.md`) and sets the session status to HOLD. Once
a hold is active, the pipeline cannot be sealed until the hold is resolved by
human intervention.
```

FILE: arifos_clip/commands/999.md

```
# 999 seal

**Description:** Attempts to seal (finalize) the session’s results. The "seal"
stage requests authorization to finalize and (optionally) apply the changes.

**Usage:** `999 seal [--apply --authority-token <token>]`

Without `--apply`, this command performs a dry-run law check to see if sealing
is allowed. With `--apply` and a valid authority token, it will request arifOS
to approve the session. Only if arifOS returns a "SEAL" verdict and the token is
provided will the session be marked sealed. Otherwise, the seal is refused
(resulting in a HOLD or SABAR outcome).
```

FILE: arifos_clip/hooks/pre-push

```
#!/bin/sh
# Pre-push hook: Blocks pushing if A CLIP pipeline is not complete or a hold
exists.

# Block if any unresolved HOLD exists
if [ -d ".arifos_clip/holds" ] && [ "$(ls -A .arifos_clip/holds)" ]; then
    echo "Push blocked: unresolved A CLIP HOLD present."
    exit 1
fi
```

```
# Block if required artifacts are missing (session or forge pack)
if [ ! -f ".arifos_clip/session.json" ] || [ ! -f ".arifos_clip/forge/
forge.json" ]; then
    echo "Push blocked: A CLIP artifacts missing."
    exit 1
fi

# Block if session is not sealed
status=$(grep -o '"status": "[^"]*' .arifos_clip/session.json | cut -d'"' -f4)
if [ "$status" != "SEALED" ]; then
    echo "Push blocked: session not sealed by A CLIP."
    exit 1
fi

# All checks passed; allow push
exit 0
```

FILE: arifos_clip/aclip/init.py

```
"""A CLIP CLI Package Initialization."""
# (No special initialization code required for this package)
```

FILE: arifos_clip/aclip/core/session.py

```
import json
import os
from pathlib import Path

class Session:
    """Represents an A CLIP session, including all stage data."""
    def __init__(self, data=None):
        self.data = data or {}
        self.loaded_from_file = False

    @classmethod
    def load_or_init(cls):
        """Load an existing session from disk, or initialize a new one if none
exists."""
        base = Path(".arifos_clip")
        base.mkdir(exist_ok=True)
        # Ensure subdirectories exist
        (base / "holds").mkdir(exist_ok=True)
        (base / "forge").mkdir(exist_ok=True)
        session_file = base / "session.json"
        if session_file.exists():
            # Load existing session
            with open(session_file, "r") as f:
                data = json.load(f)
            sess = cls(data)
            sess.loaded_from_file = True
        else:
            # Start a fresh session (data will be filled by 000 stage)
            sess = cls()
        return sess

    def save(self):
        """Save the session data to .arifos_clip/session.json."""
        base = Path(".arifos_clip")
        base.mkdir(exist_ok=True)
        session_file = base / "session.json"
        # Write JSON data (indent for readability)
        with open(session_file, "w") as f:
            json.dump(self.data, f, indent=2)

def get_cli_stage_file(filename):
    """
    Get the file path of a CLI stage module (e.g., '000_void.py'),
    regardless of numeric naming issues.
    """
    return Path(__file__).resolve().parent.parent / "cli" / filename
```

FILE: arifos_clip/aclip/bridge/arifos_client.py

```
# Bridge client to call arifOS law engine functions
try:
    import arifos  # assuming arifOS is a package or module accessible in the
environment
except ImportError:
    arifos = None

def request_verdict(session):
    """
    Request a verdict from arifOS on whether the session can be sealed.
    Returns a tuple (verdict_value, reason). If arifOS is not available or an
error occurs,
    returns (None, <error reason>).
    """
    if arifos is None:
        return (None, "arifOS not available")
    try:
        # We assume arifOS provides a function to evaluate the session's
readiness to seal.
        if hasattr(arifos, "evaluate_session"):
            verdict = arifos.evaluate_session(session.data)
        else:
            # If no direct function, assume verdict is not available
            verdict = None
    except Exception as e:
        return (None, str(e))
    # If verdict is returned (e.g., "SEAL", "HOLD", etc.), no error reason
    return (verdict, None)
```

FILE: arifos_clip/aclip/cli/000_void.py

```
"""CLI stage 000 - void."""
from datetime import datetime
import json
import os

def run_stage(session, args):
    # Starting a new session (void stage)
    if getattr(session, 'loaded_from_file', False) and
session.data.get('status') not in ['SEALED']:
        print('Error: Unsealed session already exists. Cannot start a new
session.')
        return 30
    task_desc = ' '.join(args.task)
    # Initialize new session data
    session.data = {
        'id': session.data.get('id') or datetime.now().strftime('%Y%m%d%H%M%S'),
        'task': task_desc,
        'status': 'VOID',
        'steps': []
    }
    # Record initial step
    session.data['steps'].append({
        'stage': 0,
        'name': 'void',
        'input': task_desc,
        'output': None,
        'exit_code': 40,
        'timestamp': datetime.now().isoformat()
    })
    # Write session file immediately
    session.save()
    if args.json:
        print(json.dumps(session.data, indent=2))
    else:
        print(f"Session {session.data['id']} initialized. Task: {task_desc}")
    return 40
```

FILE: arifos_clip/aclip/cli/777_forge.py

```
"""CLI stage 777 - forge."""
from datetime import datetime
import json
import os

def run_stage(session, args):
    # Compile forge pack from session data
    pack = {
        'session_id': session.data.get('id'),
        'task': session.data.get('task'),
        'steps': session.data.get('steps', [])
    }
    os.makedirs('.arifos_clip/forge', exist_ok=True)
    forge_path = f".arifos_clip/forge/forge.json"
    with open(forge_path, 'w') as f:
        json.dump(pack, f, indent=2)
    session.data['status'] = 'FORGED'
    if args.json:
        print(json.dumps(pack, indent=2))
    else:
        print(f"Forge pack created: {forge_path}")
    return 20
```

FILE: arifos_clip/aclip/cli/888_hold.py

```
"""CLI stage 888 - hold."""
from datetime import datetime
import json
import os

def run_stage(session, args):
    reason = args.reason or 'Manual hold invoked.'
    # Mark session status as HOLD
    session.data['status'] = 'HOLD'
    # Append hold step to session
    session.data.setdefault('steps', []).append({
        'stage': 888,
        'name': 'hold',
        'input': None,
        'output': f"HOLD: {reason}",
        'exit_code': 88,
        'timestamp': datetime.now().isoformat()
    })
    # Write hold bundle
    os.makedirs('.arifos_clip/holds', exist_ok=True)
    hold_json_path = f".arifos_clip/holds/hold.json"
    hold_md_path = f".arifos_clip/holds/hold.md"
    hold_data = {
        'session_id': session.data.get('id'),
        'reason': reason,
        'timestamp': datetime.now().isoformat(),
        'resolved': False
    }
    with open(hold_json_path, 'w') as f:
        json.dump(hold_data, f, indent=2)
    with open(hold_md_path, 'w') as f:
        f.write(f"""# A CLIP HOLD\n\nSession: {session.data.get('id')}\nReason:
{reason}\n\nThis hold requires resolution by a human or authority before
continuing.\n""")
    if args.json:
        print(json.dumps(hold_data, indent=2))
    else:
        print(f"HOLD applied. Reason: {reason}")
    return 88
```

FILE: arifos_clip/aclip/cli/999_seal.py

```
"""CLI stage 999 - seal."""
from datetime import datetime
import json
import os
import sys
from arifos_clip.aclip.bridge import arifos_client
from arifos_clip.aclip.bridge import authority
from arifos_clip.aclip.bridge import verdicts

def run_stage(session, args):
    # Prevent sealing if any hold is unresolved
    if os.path.isdir('.arifos_clip/holds') and os.listdir('.arifos_clip/holds'):
        print('Cannot seal: unresolved HOLD present.')
        return 88
    # If not applying, just perform a dry-run check
    verdict_value, verdict_reason = arifos_client.request_verdict(session)
    if not args.apply:
        if verdict_value == verdicts.VERDICT_SEAL:
            print('Ready to seal. Use --apply with authority token to
finalize.')
            return 30
        else:
            reason = verdict_reason or f'verdict={verdict_value}'
            print(f"Seal check failed: {reason}")
            if verdict_value == verdicts.VERDICT_HOLD or verdict_value is None:
                return 88
            else:
                return 30
    # If applying, require authority token
    if args.apply:
        if not authority.validate_token(args.authority_token):
            print('Error: --authority-token is required to apply seal.')
            return 30
        # Check verdict again for final confirmation
        if verdict_value != verdicts.VERDICT_SEAL:
            reason = verdict_reason or f'verdict={verdict_value}'
            print(f"Cannot seal: {reason}")
            if verdict_value == verdicts.VERDICT_HOLD or verdict_value is None:
                return 88
            else:
                return 30
        # All conditions satisfied: seal the session
        session.data['status'] = 'SEALED'
        session.data['sealed_at'] = datetime.now().isoformat()
        session.data['authority'] = args.authority_token
        session.save()
        seal_msg = f"SEALED by A CLIP (Session {session.data.get('id')})"
        if args.json:
            print(json.dumps({'sealed': True, 'session_id':
session.data.get('id')}, indent=2))
        else:
            print(f"Session sealed successfully. Use commit message:
'{seal_msg}'")
        return 100
```

FILE: arifos_clip/tests/test_mvp_flow.py

```
import os
import shutil
from arifos_clip.aclip.cli import _dispatcher000, _dispatcher777,
_dispatcher888, _dispatcher999

def cleanup():
    # Remove any existing session artifacts for a clean start
    if os.path.isdir('.arifos_clip'):
        shutil.rmtree('.arifos_clip')

def test_pipeline_hold():
    cleanup()
    # Start a new session with 000 void
    code0 = _dispatcher000.main(["void", "Test", "task"])
    assert code0 == 40  # VOID exit code
    # Session file should be created
    assert os.path.isfile(".arifos_clip/session.json")
    # Forge stage (777)
    code777 = _dispatcher777.main(["forge"])
    assert code777 == 20  # PARTIAL exit code (forged but not sealed)
    assert os.path.isfile(".arifos_clip/forge/forge.json")
    # Apply a hold (888)
    code888 = _dispatcher888.main(["hold", "--reason", "Testing hold"])
    assert code888 == 88  # HOLD exit code
    assert os.path.isdir(".arifos_clip/holds")
    assert os.path.isfile(".arifos_clip/holds/hold.json")
    # Attempt seal without resolving hold (should block)
    code999 = _dispatcher999.main(["seal"])
    assert code999 == 88  # still in HOLD state, cannot seal

def test_seal_requires_authority():
    cleanup()
    _dispatcher000.main(["void", "Another", "task"])
    _dispatcher777.main(["forge"])
    # Try sealing without token
    code_no_token = _dispatcher999.main(["seal", "--apply"])
    assert code_no_token == 30  # SABAR: missing authority token
    # Try sealing with token (arifOS verdict likely HOLD due to no arifOS
available)
    code_with_token = _dispatcher999.main(["seal", "--apply", "--authority-
token", "TOKEN123"])
    # arifOS is not available in test, so this should result in a HOLD outcome
    assert code_with_token == 88
```

FILE: arifos_clip/docs/ARCHITECTURE.md

```
# A CLIP Architecture

**Layer A – Constitution Surface:** High-level governance definitions and
interfaces.
- **AGENTS.md:** Defines the conceptual agent roles (Observer, Logician, Empath,
etc.) that correspond to stages.
- **commands/**: Contains documentation (Markdown) for each CLI command (000–
999), describing usage and intent.
- **agents/**: Detailed profiles for key agent roles (sense, reflect, reason,
empathize, align).

**Layer B – Executors:** The CLI stage executors in `aclip/cli/`. Each pipeline
stage has a Python module (e.g., `000_void.py`, `111_sense.py`, ...,
`999_seal.py`) implementing that stage’s logic. These executors handle argument
parsing (via dispatchers) and coordinate reading/writing session data.

**Layer C – Bridge:** The interface to the arifOS governance engine, in `aclip/
bridge/`.
- **arifos_client.py:** Provides functions to call arifOS (e.g., to get a
verdict on sealing). This layer ensures A CLIP does not replicate law logic but
delegates to arifOS.
- **verdicts.py:** Defines verdict constants (e.g., `VERDICT_SEAL`,
`VERDICT_HOLD`) and maps verdicts to exit codes.
- **authority.py:** Handles validation of authority tokens (ensuring a human has
authorized the action).
- **time.py:** (Optional) Utilities for time-based governance (e.g., enforcing a
cooling period before certain actions, as per Phoenix-72).

**Layer D – Enforcement:** Git hooks and internal checks that enforce the
pipeline process.
- **hooks/**: Contains Git hook scripts (`pre-commit`, `commit-msg`, `pre-push`)
that prevent bypassing A CLIP rules. For example, they block commits if a hold
is unresolved or block pushes if the session isn’t sealed.
- Within CLI code, enforcement includes requiring `--apply` and tokens for
sealing, and preventing sealing if any hold exists or if arifOS has not
approved.

**Layer E – Decision Artifacts:** Outputs and records of decisions in
`.arifos_clip/`.
- **Session JSON (`session.json`):** The canonical record of the session (task,
status, and a list of all stage results – the "stage JSON envelope").
- **Forge pack (`forge.json`):** A consolidated JSON produced at stage 777
containing the task, all steps, and intermediate results.
- **Hold bundle (`holds/`):** If a hold is triggered, a `hold.json` (machine-
```

~~~~
readable) and `hold.md` (human-readable) are generated to document the issue and
freeze the pipeline.
- **(Optional)** Additional outputs (if needed, could be in `aclip/outputs/`)
for storing any artifacts each stage creates, but by default all data is kept in
the session JSON.

**Layer F – Proof (Tests):** Automated tests under `tests/` validate that A CLIP
operates correctly and invariants hold.
- Tests cover a full pipeline run, enforcement of holds and authority, and hook
behavior to ensure the system is robust against misuse.

## Protocols & Data Formats

**Stage JSON Envelope (Session Structure):** Each pipeline stage appends an
entry to the `steps` array in `.arifos_clip/session.json`. Each entry is a JSON
object with:
- `stage`: Numeric code of the stage (e.g., 111).
- `name`: Verb name of the stage (e.g., "sense").
- `input`: The input or context considered (often the previous stage’s output or
the initial task).
- `output`: A summary of the stage’s output or decision.
- `exit_code`: The exit code resulting from that stage’s execution.
- `timestamp`: ISO8601 timestamp when the stage was executed.

For example, after 000 and 111 stages, `session.json` might contain:
```json
{
  "id": "20251213111230",
  "task": "Example task",
  "status": "ACTIVE",
  "steps": [
    {
      "stage": 0,
      "name": "void",
      "input": "Example task",
      "output": null,
      "exit_code": 40,
      "timestamp": "2025-12-13T23:12:30.123456"
    },
    {
      "stage": 111,
      "name": "sense",
      "input": null,
      "output": "Context sensed and recorded.",
      "exit_code": 0,
      "timestamp": "2025-12-13T23:13:00.456789"
    }
    /* ... further stages ... */
~~~~

```
  ]
}
```

This envelope provides a full audit trail of how a decision was formed.

Exit Codes Specification: A CLIP uses specific exit codes for machine interpretation of outcomes: - **0 – PASS:** Stage completed successfully (no issues at this stage). - **20 – PARTIAL:** Pipeline execution is partially complete (through forge, but not sealed). - **30 – SABAR:** (Malay: "patience") Execution is paused waiting for something (e.g., waiting for authority token or cooling period). - **40 – VOID:** The void stage (000) executed – session initialized. - **88 – HOLD:** A hold is in effect or a critical issue was encountered (requires manual resolution). - **100 – SEALED:** The final stage executed and the output was sealed/applied successfully.

These codes allow integration with CI or other automation: for example, a CI pipeline might treat code 0, 20, 30 as non-final (needs attention or further action), 88 as a failure requiring human review, and 100 as a successful completion of the governance process.

Session File (`session.json`): Lives in the repository root's `.arifos_clip/` directory. It contains keys: - `id`: Unique session identifier (e.g., timestamp or UUID). - `task`: The problem/task description provided at 000. - `status`: Current status of the session (e.g., "VOID", "ACTIVE" during processing, "HOLD" if paused, "SEALED" if finalized). - `steps`: Array of stage result objects (the stage envelope described above). - Additional fields may appear when sealed (e.g., `sealed_at` timestamp, `authority` token used).

This file is updated at each stage, providing a single source of truth for the session state.

HOLD Bundle Format: When a hold is triggered: - **hold.json:** JSON file containing at least: - `session_id`: ID of the session. - `reason`: Textual reason for the hold. - `timestamp`: When the hold was triggered. - `resolved`: Flag (always false when created; could be true if a hold is later cleared). - **hold.md:** A Markdown file explaining the hold in human-friendly terms (including the session ID and reason). It typically includes instructions or notes for the human reviewer.

These files are intended for auditors or decision-makers to review what went wrong and why the process was halted. The presence of any file in `.arifos_clip/holds/` is treated by hooks as an unresolved hold.

Core Invariants

Several core invariants are encoded in A CLIP's design and tests:

- **No Silent Apply:** The system never applies or finalizes changes without explicit approval. By default, `999 seal` performs a check and does nothing permanent. Only when `--apply` is provided *and* all other conditions (authority token + arifOS SEAL verdict) are met will the session be sealed. This prevents any automated or accidental finalization.

- **Authority & Verdict Required to Seal:** Even with `--apply`, sealing requires a valid human authority token and a positive verdict from arifOS. The code checks for both. If either is missing or negative:

- Missing token → the command exits with code 30 (SABAR), indicating it's waiting on authority.
- Negative verdict or no law engine → the command exits with code 88 (HOLD), indicating a hard stop (e.g., law violation or system unavailable).
  This ensures a two-tier approval: human and machine (law engine).

- **Hold Blocks Progress:** Once a hold (888) is triggered, the pipeline is effectively frozen. The presence of a hold file or a session status of HOLD will cause:

- 999 seal to refuse operation (exit 88) until the hold is resolved.
- Git hooks to prevent commits or pushes.
  This invariant guarantees that issues flagged by the pipeline get human attention before any final action.

- **Delegation to arifOS (No Law Reimplementation):** A CLIP does not replicate the logic of floors, verdict calculations, GENIUS/EUREKA metrics, or time-based rules. All such logic is expected to reside in arifOS. The `aclip.bridge.arifos_client` module calls arifOS for verdicts. If arifOS is not available (import fails) or errors, A CLIP treats it as a HOLD condition (`reason: "arifOS not available"`). This keeps A CLIP simple and focused on orchestration, and ensures the single source of truth for governance rules is arifOS itself.

- **All Artifacts in .arifos_clip:** A CLIP writes all session and decision artifacts to the `.arifos_clip/` directory. It does not modify files outside this directory unless the final seal is authorized. (In practice, sealing could trigger code generation, commits, or other side-effects, but those are not implemented in this CLI and would require arifOS integration or explicit user action.) The Git hooks further ensure that no code is pushed without corresponding `.arifos_clip/` artifacts, linking repository changes to governance records.

By adhering to these invariants, A CLIP creates a trustworthy chain-of-governance for any changes, from the initial void to the final seal, all while requiring human insight and law-engine oversight at critical junctures.

~~~
# FILE: arifos_clip/aclip/cli/111_sense.py
```python
"""CLI stage 111 - sense."""
from datetime import datetime
import json

def run_stage(session, args):
    # Perform sense stage logic (stub)
    prev_step = session.data['steps'][-1] if session.data.get('steps') else None
    result = "Context sensed and recorded."
    # Append this stage result to session
    session.data['steps'].append({
        'stage': 111,
        'name': 'sense',
~~~

```
        'input': prev_step['output'] if prev_step else session.data.get('task'),
        'output': result,
        'exit_code': 0,
        'timestamp': datetime.now().isoformat()
    })
    session.data['status'] = 'ACTIVE'
    if args.json:
        # Output the latest step as JSON
        print(json.dumps(session.data['steps'][-1], indent=2))
    else:
        print("Stage 111 (sense) completed: Context sensed and recorded.")
    return 0
```

FILE: arifos_clip/aclip/cli/222_reflect.py

```
"""CLI stage 222 - reflect."""
from datetime import datetime
import json

def run_stage(session, args):
    # Perform reflect stage logic (stub)
    prev_step = session.data['steps'][-1] if session.data.get('steps') else None
    result = "Reflections noted."
    # Append this stage result to session
    session.data['steps'].append({
        'stage': 222,
        'name': 'reflect',
        'input': prev_step['output'] if prev_step else session.data.get('task'),
        'output': result,
        'exit_code': 0,
        'timestamp': datetime.now().isoformat()
    })
    session.data['status'] = 'ACTIVE'
    if args.json:
        # Output the latest step as JSON
        print(json.dumps(session.data['steps'][-1], indent=2))
    else:
        print("Stage 222 (reflect) completed: Reflections noted.")
    return 0
```

FILE: arifos_clip/aclip/cli/333_reason.py

```python
"""CLI stage 333 - reason."""
from datetime import datetime
import json

def run_stage(session, args):
    # Perform reason stage logic (stub)
    prev_step = session.data['steps'][-1] if session.data.get('steps') else None
    result = "Logical reasoning completed."
    # Append this stage result to session
    session.data['steps'].append({
        'stage': 333,
        'name': 'reason',
        'input': prev_step['output'] if prev_step else session.data.get('task'),
        'output': result,
        'exit_code': 0,
        'timestamp': datetime.now().isoformat()
    })
    session.data['status'] = 'ACTIVE'
    if args.json:
        # Output the latest step as JSON
        print(json.dumps(session.data['steps'][-1], indent=2))
    else:
        print("Stage 333 (reason) completed: Logical reasoning completed.")
    return 0
```

FILE: arifos_clip/aclip/cli/444_evidence.py

```python
"""CLI stage 444 - evidence."""
from datetime import datetime
import json

def run_stage(session, args):
    # Perform evidence stage logic (stub)
    prev_step = session.data['steps'][-1] if session.data.get('steps') else None
    result = "Evidence gathered."
    # Append this stage result to session
    session.data['steps'].append({
        'stage': 444,
        'name': 'evidence',
        'input': prev_step['output'] if prev_step else session.data.get('task'),
        'output': result,
        'exit_code': 0,
        'timestamp': datetime.now().isoformat()
    })
    session.data['status'] = 'ACTIVE'
    if args.json:
        # Output the latest step as JSON
        print(json.dumps(session.data['steps'][-1], indent=2))
    else:
        print("Stage 444 (evidence) completed: Evidence gathered.")
    return 0
```

FILE: arifos_clip/aclip/cli/555_empathize.py

```
"""CLI stage 555 - empathize."""
from datetime import datetime
import json

def run_stage(session, args):
    # Perform empathize stage logic (stub)
    prev_step = session.data['steps'][-1] if session.data.get('steps') else None
    result = "Stakeholder perspectives considered."
    # Append this stage result to session
    session.data['steps'].append({
        'stage': 555,
        'name': 'empathize',
        'input': prev_step['output'] if prev_step else session.data.get('task'),
        'output': result,
        'exit_code': 0,
        'timestamp': datetime.now().isoformat()
    })
    session.data['status'] = 'ACTIVE'
    if args.json:
        # Output the latest step as JSON
        print(json.dumps(session.data['steps'][-1], indent=2))
    else:
        print("Stage 555 (empathize) completed: Stakeholder perspectives 
considered.")
    return 0
```

FILE: arifos_clip/aclip/cli/666_align.py

```
"""CLI stage 666 - align."""
from datetime import datetime
import json
```

```
from arifos_clip.aclip.bridge import arifos_client

def run_stage(session, args):
    # Perform align stage logic (stub)
    prev_step = session.data['steps'][-1] if session.data.get('steps') else None
    result = "Alignment with principles verified."
    # Append this stage result to session
    session.data['steps'].append({
        'stage': 666,
        'name': 'align',
        'input': prev_step['output'] if prev_step else session.data.get('task'),
        'output': result,
        'exit_code': 0,
        'timestamp': datetime.now().isoformat()
    })
    session.data['status'] = 'ACTIVE'
    # Align stage might call arifOS for a pre-verdict, but here we assume all
good.
    if args.json:
        # Output the latest step as JSON
        print(json.dumps(session.data['steps'][-1], indent=2))
    else:
        print("Stage 666 (align) completed: Alignment with principles
verified.")
    return 0
```

FILE: arifos_clip/aclip/bridge/authority.py

```
# Authority token validation

def validate_token(token):
    """
    Validate the authority token provided by user.
    In this basic implementation, any non-empty token is considered valid.
    (In a real scenario, this could check against a registry or perform
cryptographic verification.)
    """
    return token is not None and token != ""
```

FILE: arifos_clip/aclip/bridge/verdicts.py

```
# Verdict constants and mapping
# Possible verdict values from arifOS
VERDICT_SEAL = "SEAL"
VERDICT_HOLD = "HOLD"
VERDICT_PASS = "PASS"
VERDICT_PARTIAL = "PARTIAL"
VERDICT_SABAR = "SABAR"
VERDICT_VOID = "VOID"

# Map verdict labels to A CLIP exit codes
verdict_to_exit_code = {
    VERDICT_PASS: 0,
    VERDICT_PARTIAL: 20,
    VERDICT_SABAR: 30,
    VERDICT_VOID: 40,
    VERDICT_HOLD: 88,
    VERDICT_SEAL: 100
}
```

FILE: arifos_clip/aclip/bridge/time.py

```
# Time-based governance utilities (optional)

from datetime import datetime, timedelta

def now_iso():
    """Return the current time as an ISO8601 string."""
    return datetime.now().isoformat()

def cooling_period_elapsed(start_iso, hours=72):
    """
    Check if a cooling period (default 72 hours) has elapsed since the given
start time.
    Returns True if the current time is at least `hours` hours past `start_iso`.
    """
    try:
        start_time = datetime.fromisoformat(start_iso)
    except Exception:
        return False
    return datetime.now() - start_time >= timedelta(hours=hours)
```

FILE: arifos_clip/aclip/core/routing.py

```python
# Routing logic for pipeline progression

def next_stage(current_stage):
    """
    Determine the next stage number in the pipeline after current_stage.
    Skips the hold stage (888) in normal flow; hold is triggered only by issues.
    Returns the next stage number (int) or None if the pipeline is complete.
    """
    stage_order = [0, 111, 222, 333, 444, 555, 666, 777, 999]
    if current_stage in stage_order:
        idx = stage_order.index(current_stage)
        if idx < len(stage_order) - 1:
            return stage_order[idx + 1]
    return None
```

FILE: arifos_clip/aclip/core/exits.py

```python
# Exit code definitions (for reference and future use)

PASS = 0        # Stage success
PARTIAL = 20    # Pipeline partially complete (not sealed)
SABAR = 30      # Waiting for external input (authority, time, etc.)
VOID = 40       # Session initialized
HOLD = 88       # Hold triggered or required
SEALED = 100    # Pipeline sealed successfully

EXIT_CODES = {
    "PASS": PASS,
    "PARTIAL": PARTIAL,
    "SABAR": SABAR,
    "VOID": VOID,
    "HOLD": HOLD,
    "SEALED": SEALED
}
```

FILE: arifos_clip/aclip/core/formatting.py

```python
import json
```

```
def output_result(data, as_json=False):
    """
    Print the result data either as JSON (if as_json is True) or as a plain
string.
    If data is a dict or list and as_json is False, prints it in a condensed
form.
    """
    if as_json:
        print(json.dumps(data, indent=2))
    else:
        if isinstance(data, (dict, list)):
            # Print a one-line summary for dict/list
            print(json.dumps(data))
        else:
            print(str(data))
```

FILE: arifos_clip/aclip/cli/init.py

```
"""A CLIP CLI Module - dispatchers and command group initialization."""
# Dispatchers for numeric commands are defined in separate _dispatcherNNN.py
modules.
# This __init__.py could set up any shared context if needed (none required for
now).
```

FILE: arifos_clip/aclip/cli/status.py

```
"""CLI utility to show current A CLIP session status."""
import json
import os

def print_status():
    """
    Print the current status of the A CLIP session.
    Returns 0 if sealed, 1 if not sealed or no session.
    """
    session_file = ".arifos_clip/session.json"
    if not os.path.isfile(session_file):
        print("No A CLIP session found.")
        return 1
    with open(session_file, "r") as f:
        data = json.load(f)
    status = data.get("status")
    session_id = data.get("id")
    print(f"Session {session_id} status: {status}")
    if status == "HOLD":
        print("A HOLD is in effect. Resolve it before sealing.")
        return 1
    elif status != "SEALED":
        print("Pipeline is not yet sealed (in progress).")
        return 1
    else:
        print("Session is SEALED and complete.")
        return 0
```

FILE: arifos_clip/aclip/cli/_dispatcher000.py

```
"""Dispatcher for 000 void command."""
import sys
import argparse
# Import session management
from arifos_clip.aclip.core import session as session_core
# We will dynamically load the stage module by file path because module name is
numeric
from importlib import util

def main(argv=None):
    parser = argparse.ArgumentParser(prog="000", description="Execute A CLIP
stage 000 - void")
    parser.add_argument("verb", choices=["void"], help="Stage verb (must be
'void')")
    parser.add_argument("task", nargs="+", help="Task description for the void
stage")
    parser.add_argument("--json", action="store_true", help="Output result in
JSON")
    args = parser.parse_args(argv)
    # Initialize or load session
    sess = session_core.Session.load_or_init()
    # Dynamically import the stage module 000_void.py
    stage_file = session_core.get_cli_stage_file("000_void.py")
    spec = util.spec_from_file_location("aclip.cli.000_void", stage_file)
    if spec is None:
        print("Error: Stage module file not found:", stage_file,
file=sys.stderr)
        return 88  # treat as hold if missing critical component
    stage_mod = util.module_from_spec(spec)
    spec.loader.exec_module(stage_mod)
    # Run the stage logic function if available
    if hasattr(stage_mod, "run_stage"):
        exit_code = stage_mod.run_stage(sess, args)
    elif hasattr(stage_mod, "main"):
        # If stage file has its own main function, call it (not used in this
design)
        exit_code = stage_mod.main(args)
    else:
        print("Error: Stage module has no entry point", file=sys.stderr)
        return 88
    # Save session after stage execution (if modified)
    sess.save()
    return exit_code

if __name__ == "__main__":
    sys.exit(main())
```

FILE: arifos_clip/aclip/cli/_dispatcher111.py

```
"""Dispatcher for 111 sense command."""
import sys
import argparse
from arifos_clip.aclip.core import session as session_core
from importlib import util

def main(argv=None):
    parser = argparse.ArgumentParser(prog="111", description="Execute A CLIP
stage 111 - sense")
    parser.add_argument("verb", choices=["sense"], help="Stage verb (must be
'sense')")
    parser.add_argument("--json", action="store_true", help="Output result in
JSON")
    args = parser.parse_args(argv)
    sess = session_core.Session.load_or_init()
    stage_file = session_core.get_cli_stage_file("111_sense.py")
    spec = util.spec_from_file_location("aclip.cli.111_sense", stage_file)
    if spec is None:
        print("Error: Stage module file not found:", stage_file,
file=sys.stderr)
        return 88
    stage_mod = util.module_from_spec(spec)
    spec.loader.exec_module(stage_mod)
    if hasattr(stage_mod, "run_stage"):
        exit_code = stage_mod.run_stage(sess, args)
    elif hasattr(stage_mod, "main"):
        exit_code = stage_mod.main(args)
    else:
        print("Error: Stage module has no entry point", file=sys.stderr)
        return 88
    sess.save()
    return exit_code

if __name__ == "__main__":
    sys.exit(main())
```

FILE: arifos_clip/aclip/cli/_dispatcher222.py

```
"""Dispatcher for 222 reflect command."""
import sys
import argparse
from arifos_clip.aclip.core import session as session_core
from importlib import util

def main(argv=None):
    parser = argparse.ArgumentParser(prog="222", description="Execute A CLIP
stage 222 - reflect")
    parser.add_argument("verb", choices=["reflect"], help="Stage verb (must be
'reflect')")
    parser.add_argument("--json", action="store_true", help="Output result in
JSON")
    args = parser.parse_args(argv)
    sess = session_core.Session.load_or_init()
    stage_file = session_core.get_cli_stage_file("222_reflect.py")
    spec = util.spec_from_file_location("aclip.cli.222_reflect", stage_file)
    if spec is None:
        print("Error: Stage module file not found:", stage_file,
file=sys.stderr)
        return 88
    stage_mod = util.module_from_spec(spec)
    spec.loader.exec_module(stage_mod)
    if hasattr(stage_mod, "run_stage"):
        exit_code = stage_mod.run_stage(sess, args)
    elif hasattr(stage_mod, "main"):
        exit_code = stage_mod.main(args)
    else:
        print("Error: Stage module has no entry point", file=sys.stderr)
        return 88
    sess.save()
    return exit_code

if __name__ == "__main__":
    sys.exit(main())
```

FILE: arifos_clip/aclip/cli/_dispatcher333.py

```python
"""Dispatcher for 333 reason command."""
import sys
import argparse
from arifos_clip.aclip.core import session as session_core
from importlib import util

def main(argv=None):
    parser = argparse.ArgumentParser(prog="333", description="Execute A CLIP
stage 333 - reason")
    parser.add_argument("verb", choices=["reason"], help="Stage verb (must be
'reason')")
    parser.add_argument("--json", action="store_true", help="Output result in
JSON")
    args = parser.parse_args(argv)
    sess = session_core.Session.load_or_init()
    stage_file = session_core.get_cli_stage_file("333_reason.py")
    spec = util.spec_from_file_location("aclip.cli.333_reason", stage_file)
    if spec is None:
        print("Error: Stage module file not found:", stage_file,
file=sys.stderr)
        return 88
    stage_mod = util.module_from_spec(spec)
    spec.loader.exec_module(stage_mod)
    if hasattr(stage_mod, "run_stage"):
        exit_code = stage_mod.run_stage(sess, args)
    elif hasattr(stage_mod, "main"):
        exit_code = stage_mod.main(args)
    else:
        print("Error: Stage module has no entry point", file=sys.stderr)
        return 88
    sess.save()
    return exit_code

if __name__ == "__main__":
    sys.exit(main())
```

FILE: arifos_clip/aclip/cli/_dispatcher444.py

```python
"""Dispatcher for 444 evidence command."""
import sys
import argparse
from arifos_clip.aclip.core import session as session_core
from importlib import util

def main(argv=None):
    parser = argparse.ArgumentParser(prog="444", description="Execute A CLIP
stage 444 - evidence")
    parser.add_argument("verb", choices=["evidence"], help="Stage verb (must be
'evidence')")
    parser.add_argument("--json", action="store_true", help="Output result in
JSON")
    args = parser.parse_args(argv)
    sess = session_core.Session.load_or_init()
    stage_file = session_core.get_cli_stage_file("444_evidence.py")
    spec = util.spec_from_file_location("aclip.cli.444_evidence", stage_file)
    if spec is None:
        print("Error: Stage module file not found:", stage_file,
file=sys.stderr)
        return 88
    stage_mod = util.module_from_spec(spec)
    spec.loader.exec_module(stage_mod)
    if hasattr(stage_mod, "run_stage"):
        exit_code = stage_mod.run_stage(sess, args)
    elif hasattr(stage_mod, "main"):
        exit_code = stage_mod.main(args)
    else:
        print("Error: Stage module has no entry point", file=sys.stderr)
        return 88
    sess.save()
    return exit_code

if __name__ == "__main__":
    sys.exit(main())
```

FILE: arifos_clip/aclip/cli/_dispatcher555.py

```
"""Dispatcher for 555 empathize command."""
import sys
import argparse
from arifos_clip.aclip.core import session as session_core
from importlib import util

def main(argv=None):
    parser = argparse.ArgumentParser(prog="555", description="Execute A CLIP
stage 555 - empathize")
    parser.add_argument("verb", choices=["empathize"],
help="Stage verb (must be 'empathize')")
```

```
    parser.add_argument("--json", action="store_true", help="Output result in
JSON")
    args = parser.parse_args(argv)
    sess = session_core.Session.load_or_init()
    stage_file = session_core.get_cli_stage_file("555_empathize.py")
    spec = util.spec_from_file_location("aclip.cli.555_empathize", stage_file)
    if spec is None:
        print("Error: Stage module file not found:", stage_file,
file=sys.stderr)
        return 88
    stage_mod = util.module_from_spec(spec)
    spec.loader.exec_module(stage_mod)
    if hasattr(stage_mod, "run_stage"):
        exit_code = stage_mod.run_stage(sess, args)
    elif hasattr(stage_mod, "main"):
        exit_code = stage_mod.main(args)
    else:
        print("Error: Stage module has no entry point", file=sys.stderr)
        return 88
    sess.save()
    return exit_code

if __name__ == "__main__":
    sys.exit(main())
```

FILE: arifos_clip/aclip/cli/_dispatcher666.py

```
"""Dispatcher for 666 align command."""
import sys
import argparse
from arifos_clip.aclip.core import session as session_core
from importlib import util

def main(argv=None):
    parser = argparse.ArgumentParser(prog="666", description="Execute A CLIP
stage 666 - align")
    parser.add_argument("verb", choices=["align"], help="Stage verb (must be
'align')")
    parser.add_argument("--json", action="store_true", help="Output result in
JSON")
    args = parser.parse_args(argv)
    sess = session_core.Session.load_or_init()
    stage_file = session_core.get_cli_stage_file("666_align.py")
    spec = util.spec_from_file_location("aclip.cli.666_align", stage_file)
    if spec is None:
        print("Error: Stage module file not found:", stage_file,
file=sys.stderr)
        return 88
    stage_mod = util.module_from_spec(spec)
    spec.loader.exec_module(stage_mod)
    if hasattr(stage_mod, "run_stage"):
        exit_code = stage_mod.run_stage(sess, args)
    elif hasattr(stage_mod, "main"):
        exit_code = stage_mod.main(args)
    else:
        print("Error: Stage module has no entry point", file=sys.stderr)
        return 88
    sess.save()
    return exit_code

if __name__ == "__main__":
    sys.exit(main())
```

FILE: arifos_clip/aclip/cli/_dispatcher777.py

```
"""Dispatcher for 777 forge command."""
import sys
import argparse
from arifos_clip.aclip.core import session as session_core
from importlib import util

def main(argv=None):
    parser = argparse.ArgumentParser(prog="777", description="Execute A CLIP
stage 777 - forge")
    parser.add_argument("verb", choices=["forge"], help="Stage verb (must be
'forge')")
    parser.add_argument("--json", action="store_true", help="Output result in
JSON")
    args = parser.parse_args(argv)
    sess = session_core.Session.load_or_init()
    stage_file = session_core.get_cli_stage_file("777_forge.py")
    spec = util.spec_from_file_location("aclip.cli.777_forge", stage_file)
    if spec is None:
        print("Error: Stage module file not found:", stage_file,
file=sys.stderr)
        return 88
    stage_mod = util.module_from_spec(spec)
    spec.loader.exec_module(stage_mod)
    if hasattr(stage_mod, "run_stage"):
        exit_code = stage_mod.run_stage(sess, args)
    elif hasattr(stage_mod, "main"):
        exit_code = stage_mod.main(args)
    else:
        print("Error: Stage module has no entry point", file=sys.stderr)
        return 88
    sess.save()
    return exit_code

if __name__ == "__main__":
    sys.exit(main())
```

FILE: arifos_clip/aclip/cli/_dispatcher888.py

```
"""Dispatcher for 888 hold command."""
import sys
import argparse
from arifos_clip.aclip.core import session as session_core
from importlib import util

def main(argv=None):
    parser = argparse.ArgumentParser(prog="888", description="Execute A CLIP
stage 888 - hold")
    parser.add_argument("verb", choices=["hold"], help="Stage verb (must be
'hold')")
    parser.add_argument("--reason", "-r", default="Manual hold invoked.",
help="Reason for hold (optional)")
    parser.add_argument("--json", action="store_true", help="Output result in
JSON")
    args = parser.parse_args(argv)
    sess = session_core.Session.load_or_init()
    stage_file = session_core.get_cli_stage_file("888_hold.py")
    spec = util.spec_from_file_location("aclip.cli.888_hold", stage_file)
    if spec is None:
        print("Error: Stage module file not found:", stage_file,
file=sys.stderr)
        return 88
    stage_mod = util.module_from_spec(spec)
    spec.loader.exec_module(stage_mod)
    if hasattr(stage_mod, "run_stage"):
        exit_code = stage_mod.run_stage(sess, args)
    elif hasattr(stage_mod, "main"):
        exit_code = stage_mod.main(args)
    else:
        print("Error: Stage module has no entry point", file=sys.stderr)
        return 88
    sess.save()
    return exit_code

if __name__ == "__main__":
    sys.exit(main())
```

FILE: arifos_clip/aclip/cli/_dispatcher999.py

```
"""Dispatcher for 999 seal command."""
import sys
import argparse
from arifos_clip.aclip.core import session as session_core
from importlib import util

def main(argv=None):
    parser = argparse.ArgumentParser(prog="999", description="Execute A CLIP
stage 999 - seal")
    parser.add_argument("verb", choices=["seal"], help="Stage verb (must be
'seal')")
    parser.add_argument("--apply", action="store_true", help="Apply changes (if
authorized)")
    parser.add_argument("--authority-token", help="Authority token required for
applying changes")
    parser.add_argument("--json", action="store_true", help="Output result in
JSON")
    args = parser.parse_args(argv)
    sess = session_core.Session.load_or_init()
    stage_file = session_core.get_cli_stage_file("999_seal.py")
    spec = util.spec_from_file_location("aclip.cli.999_seal", stage_file)
    if spec is None:
        print("Error: Stage module file not found:", stage_file,
file=sys.stderr)
        return 88
    stage_mod = util.module_from_spec(spec)
    spec.loader.exec_module(stage_mod)
    if hasattr(stage_mod, "run_stage"):
        exit_code = stage_mod.run_stage(sess, args)
    elif hasattr(stage_mod, "main"):
        exit_code = stage_mod.main(args)
    else:
        print("Error: Stage module has no entry point", file=sys.stderr)
        return 88
    sess.save()
    return exit_code
```

```
if __name__ == "__main__":
    sys.exit(main())
```

FILE: arifos_clip/tests/test_stages_flow.py

```
import os
import shutil
import json
from arifos_clip.aclip.cli import (
    _dispatcher000, _dispatcher111, _dispatcher222, _dispatcher333,
    _dispatcher444, _dispatcher555, _dispatcher666, _dispatcher777,
_dispatcher999
)
from arifos_clip.aclip.bridge import arifos_client

def cleanup():
    if os.path.isdir('.arifos_clip'):
        shutil.rmtree('.arifos_clip')

def test_full_pipeline_sealing():
    cleanup()

# Monkey-patch arifos_client to always return SEAL verdict for testing final
step
    original_request_verdict = arifos_client.request_verdict
    arifos_client.request_verdict = lambda session: ("SEAL", None)
    try:
        # Execute all stages in order
        code0 = _dispatcher000.main(["void", "Full", "pipeline", "test"])
        assert code0 == 40
        _dispatcher111.main(["sense"])
        _dispatcher222.main(["reflect"])
        _dispatcher333.main(["reason"])
        _dispatcher444.main(["evidence"])
        _dispatcher555.main(["empathize"])
        _dispatcher666.main(["align"])
        code777 = _dispatcher777.main(["forge"])
        assert code777 == 20
        # Now seal with authority token
        code999 = _dispatcher999.main(["seal", "--apply", "--authority-token",
"TESTTOKEN"])
        assert code999 == 100
        # After sealing, session.json status should be SEALED
        with open(".arifos_clip/session.json", "r") as f:
            data = json.load(f)
        assert data.get("status") == "SEALED"
    finally:
        # Restore the original verdict function
        arifos_client.request_verdict = original_request_verdict
```

FILE: arifos_clip/tests/test_hooks.py

```
import os
import shutil
import subprocess
import json

HOOKS_DIR = "arifos_clip/hooks"

def write_file(path, content=""):
    with open(path, "w") as f:
        f.write(content)

def run_hook(script_name, args=None):
    """Helper to run a hook script and return (exit_code, output)."""
    script_path = f"{HOOKS_DIR}/{script_name}"
    result = subprocess.run(["bash", script_path] + (args or []),
capture_output=True, text=True)
    return result.returncode, (result.stdout + result.stderr)

def setup_session(status=None):
    """Create a dummy session file with given status and ensure artifacts exist
if needed."""
    os.makedirs(".arifos_clip", exist_ok=True)
    session_data = {
        "id": "TESTSESSION",
        "task": "Test Task",
        "status": status or "ACTIVE",
        "steps": []
    }
    write_file(".arifos_clip/session.json", json.dumps(session_data))
    if status in ("FORGED", "SEALED"):
        os.makedirs(".arifos_clip/forge", exist_ok=True)
        write_file(".arifos_clip/forge/forge.json", "{}")
    if status == "SEALED":
        # Remove any holds for sealed session
        shutil.rmtree(".arifos_clip/holds", ignore_errors=True)

def test_pre_push_blocks_on_hold():
    shutil.rmtree(".arifos_clip", ignore_errors=True)
    os.makedirs(".arifos_clip/holds", exist_ok=True)
    write_file(".arifos_clip/holds/hold.json", "{}")
    code, output = run_hook("pre-push")
    assert code != 0
    assert "HOLD" in output or "hold" in output

def test_pre_push_blocks_if_not_sealed():
    shutil.rmtree(".arifos_clip", ignore_errors=True)
    setup_session(status="FORGED")
    code, output = run_hook("pre-push")
    assert code != 0
    # Expect a message about not sealed or missing artifacts
    assert "not sealed" in output or "missing" in output

def test_pre_push_allows_sealed():
    shutil.rmtree(".arifos_clip", ignore_errors=True)
    setup_session(status="SEALED")
    code, output = run_hook("pre-push")
    assert code == 0

def test_commit_msg_requires_sealed_session_and_keyword():
    shutil.rmtree(".arifos_clip", ignore_errors=True)
    # Unsealed session
    setup_session(status="ACTIVE")
    msg_file = "TEMP_COMMIT_MSG.txt"
    write_file(msg_file, "This commit has no seal")
    code1, out1 = run_hook("commit-msg", [msg_file])
    assert code1 != 0
    # Even if message contains "SEALED", session is not sealed -> should still
block
    write_file(msg_file, "SEALED: trying to bypass")
    code2, out2 = run_hook("commit-msg", [msg_file])
    assert code2 != 0
    # Now mark session as sealed
    setup_session(status="SEALED")
    # Test missing keyword in message
    write_file(msg_file, "Final commit without keyword")
    code3, out3 = run_hook("commit-msg", [msg_file])
    assert code3 != 0
    # Add "SEALED" keyword in message
    write_file(msg_file, "SEALED: final commit")
    code4, out4 = run_hook("commit-msg", [msg_file])
    assert code4 == 0

def test_pre_commit_blocks_on_hold():
    shutil.rmtree(".arifos_clip", ignore_errors=True)
    os.makedirs(".arifos_clip/holds", exist_ok=True)
    write_file(".arifos_clip/holds/hold.json", "{}")
    code, output = run_hook("pre-commit")
    assert code != 0
    assert "hold" in output.lower()
```

FILE: arifos_clip/hooks/pre-commit

```
#!/bin/sh
# Pre-commit hook: prevent committing if there's an unresolved HOLD.

if [ -d ".arifos_clip/holds" ] && [ "$(ls -A .arifos_clip/holds)" ]; then
    echo "Commit blocked: unresolved A CLIP HOLD exists."
    exit 1
fi

exit 0
```

FILE: arifos_clip/hooks/commit-msg

```
#!/bin/sh
# Commit-msg hook: ensure commit message and session state indicate a sealed
session.

MSG_FILE="$1"

# Must have a session file
if [ ! -f ".arifos_clip/session.json" ]; then
    echo "Commit blocked: A CLIP session file not found."
    exit 1
fi

# Session must be sealed
status=$(grep -o '"status": "[^"]*' .arifos_clip/session.json | cut -d'"' -f4)
if [ "$status" != "SEALED" ]; then
    echo "Commit blocked: session not sealed by A CLIP."
    exit 1
fi

# Commit message must mention "SEALED"
if ! grep -q "SEALED" "$MSG_FILE"; then
    echo "Commit message must include 'SEALED'."
    exit 1
fi
```

```
exit 0
```

Packaging (Console Scripts Integration)

To expose the numeric CLI commands, configure console scripts in your packaging configuration. For example, in `pyproject.toml`:

```
[project.scripts]
"000" = "arifos_clip.aclip.cli._dispatcher000:main"
"111" = "arifos_clip.aclip.cli._dispatcher111:main"
"222" = "arifos_clip.aclip.cli._dispatcher222:main"
"333" = "arifos_clip.aclip.cli._dispatcher333:main"
"444" = "arifos_clip.aclip.cli._dispatcher444:main"
"555" = "arifos_clip.aclip.cli._dispatcher555:main"
"666" = "arifos_clip.aclip.cli._dispatcher666:main"
"777" = "arifos_clip.aclip.cli._dispatcher777:main"
"888" = "arifos_clip.aclip.cli._dispatcher888:main"
"999" = "arifos_clip.aclip.cli._dispatcher999:main"
```

This will install ten executables (`000`, `111`, ..., `999`) when the package is installed. Each of these entry points calls the corresponding dispatcher function, which in turn validates the verb (e.g., "void" for 000) and invokes the stage implementation. This approach avoids issues with Python module naming (since modules cannot start with a digit) while still providing the desired numeric command interface.

Claude Code Instructions

To set up A CLIP as described:

1. **Create the directory structure** (in your repository root):

   ```
   mkdir -p arifos_clip/{commands,agents,hooks,aclip/cli,aclip/core,aclip/
   bridge,docs,tests}
   ```

2. **Create each file with the exact content** shown above:

3. For example, create a file `arifos_clip/AGENTS.md` and copy the content from the `AGENTS.md` section.

4. Repeat this for every `# FILE: ...` section in this artifact, ensuring the file paths and names match exactly.

5. **Install the package** (for local testing):

6. Ensure you have a `pyproject.toml` including the console scripts as outlined in the Packaging section.
7. From the repository root, run `pip install -e .` to install A CLIP in editable mode. This makes the commands `000`, `111`, ..., `999` directly runnable in your shell.

8. **Run the test suite**:

9. Make sure you have `pytest` installed.
10. Execute `pytest arifos_clip/tests` from the repo root. All tests (`test_mvp_flow.py`, `test_stages_flow.py`, `test_hooks.py`) should pass, confirming that the pipeline behaves as expected.

11. **Configure Git hooks**:

12. Copy the hook scripts into your local repository's Git hooks directory and make them executable:

```
cp arifos_clip/hooks/pre-commit .git/hooks/pre-commit
cp arifos_clip/hooks/commit-msg .git/hooks/commit-msg
cp arifos_clip/hooks/pre-push .git/hooks/pre-push
chmod +x .git/hooks/pre-commit .git/hooks/commit-msg .git/hooks/pre-push
```

13. These hooks will now automatically enforce A CLIP rules on commit and push (preventing bypassing of holds or unsealed changes).

14. **Use A CLIP commands**:

15. **Initialize** a session with `000 void "Your task description"`.
16. Progress through `111 sense`, `222 reflect`, `333 reason`, `444 evidence`, `555 empathize`, `666 align` for a full analysis.
17. Use `777 forge` to compile results. If any issues arise or review is needed, run `888 hold` to pause.
18. When ready, run `999 seal --apply --authority-token YOUR_TOKEN` to attempt finalizing. If everything is in order and approved, the session will be sealed (and exit with code 100). You can then commit with a message including "SEALED" and push the changes, confident the hooks will permit it.

Done Definition Checklist

- [x] **Repo Tree:** Provided both MVP (16-file) and expanded v1 (~45-file) repository structures.
- [x] **File Contents:** Included the exact content for every required file (MVP and v1 stages).
- [x] **Packaging & Entry Points:** Specified a `pyproject.toml` console_scripts configuration mapping `000`–`999` commands to dispatcher functions.
- [x] **Core Invariants Tested:** Wrote tests to enforce invariants (no silent apply, `999 seal` requires authority token & SEAL verdict, `888 hold` blocks progress, delegation to arifOS for verdicts).

- [x] **Hook Strategy:** Implemented Git hooks (`pre-commit`, `commit-msg`, `pre-push`) that gate commits/pushes based on holds and sealing status.
- [x] **Protocols Documented:** Documented the stage JSON envelope, exit codes, session file format, and hold bundle format in the architecture documentation.
- [x] **Instructions Provided:** Included step-by-step instructions to create the files, install and use the CLI, run tests, and enable the Git hooks.
