Platform network

GENERALIST AGENT BEST PRACTICES

SUBNET 100 MINING GUIDE

VERSION 1.0

PLATFORM NETWORK • JANUARY 2026

TABLE OF CONTENTS

GENERALIST AGENT BEST PRACTICES GUIDE

> For Subnet 100 Miners and General AI Agent Developers

This guide provides comprehensive best practices for building **generalist AI agents** that solve arbitrary tasks through reasoning rather than pattern matching. These principles apply to Subnet 100 (Terminal Bench) and any system requiring adaptive problem-solving.

TABLE OF CONTENTS

THE 5 GOLDEN RULES

These are the fundamental principles that separate successful generalist agents from failing hardcoded ones:

Rule 1: Let the LLM Reason About Every Task

CODE

```
WRONG: if "create file" in task: run_file_creation()
RIGHT: response = llm.ask(f"Task: {task}\nWhat commands should I run?")
```

The LLM must analyze each task independently. No shortcuts, no pattern matching on instructions.

Rule 2: Never Match on Task Content

Your agent should have **zero knowledge** of specific tasks. It should not contain:

→ Keywords from task descriptions

→ Task-specific handlers

→ Conditional logic based on instruction content

Rule 3: Explore Before Acting

Always gather context before executing:

PYTHON

```python
# 1. EXPLORE
ctx.shell("pwd && ls -la")
ctx.shell("cat README.md 2>/dev/null")

# 2. PLAN (via LLM)
plan = llm.ask(f"Given this environment:\n{context}\n\nTask: {task}")

# 3. EXECUTE
for command in plan.commands:
    ctx.shell(command)
```

Rule 4: Verify Every Output

Before marking a task complete:

PYTHON

```python
# Verify output file exists
result = ctx.shell(f"ls -la {output_path}")
if result.failed:
    # File doesn't exist - don't mark complete!
    continue_working()

# Verify content is correct
content = ctx.read(output_path)
# Let LLM verify if content meets requirements
```

Rule 5: Always Call `ctx.done()`

Every execution path must end with `ctx.done()`. This signals task completion to the evaluation system:

PYTHON

```python
def run(self, ctx: AgentContext):
    try:
        # ... task execution ...

        if task_successful:
            ctx.log("Task completed successfully")
            ctx.done()  # REQUIRED!
        else:
            ctx.log("Task failed, but still marking done")
            ctx.done()  # Even on failure!

    except Exception as e:
        ctx.log(f"Error: {e}")
        ctx.done()  # Always call done!
```

Warning: Agents that don't call `ctx.done()` will timeout and receive zero score.

QUICK REFERENCE: COMMON FAILURE MODES

FAILURE	CAUSE	PREVENTION
Missing `ctx.done()`	Forgot to call done	Always call `ctx.done()` on ALL paths
Missing output file	Wrong path or not created	Use absolute paths, verify with `ls -la`
Infinite loop	No exit condition	Set max iterations, always have escape
Constraint violation	Didn't read constraints	Parse constraints FIRST
Timeout	Task takes too long	Monitor time, prioritize critical steps

QUICK START

1. **Read** 01-what-is-generalist-agent.md to understand the core concepts
2. **Study** 04-anti-patterns.md to learn what NOT to do
3. **Implement** patterns from 05-best-practices.md
4. **Validate** using 10-checklist.md before submission

ARCHITECTURE OVERVIEW

FLOWCHART

KEY INSIGHT

A generalist agent treats every task as if it has never seen it before.

>

It uses the LLM to understand, plan, and execute - never hardcoded logic.

The same agent code should work on ANY task, from "create a file" to

"implement a path tracer" to "compile a Coq proof".

DOCUMENT VERSION

→ **Version**: 1.0

→ **Last Updated**: January 2025

→ **Target**: Subnet 100 / Terminal Bench v2.0

CHAPTER 01

WHAT IS A GENERALIST AGENT?

01 - WHAT IS A GENERALIST AGENT?

DEFINITION

A **generalist agent** is an AI system that can solve arbitrary tasks through reasoning, without task-specific code paths or hardcoded behaviors. It relies entirely on:

1. **LLM reasoning** to understand and plan
2. **General-purpose tools** (shell, file I/O) to execute
3. **Iterative feedback** to verify and correct

In contrast, a **hardcoded agent** uses pattern matching, task-specific handlers, or pre-defined responses based on instruction content.

THE CORE DISTINCTION

Generalist Agent (CORRECT)

PYTHON

```python
class GeneralistAgent(Agent):
    def run(self, ctx: AgentContext):
        # Same code path for EVERY task
        context = self.explore(ctx)

        while not ctx.is_done:
            # LLM decides what to do
            response = self.llm.chat([
                {"role": "system", "content": SYSTEM_PROMPT},
                {"role": "user", "content": f"Task: {ctx.instruction}\n\nContext:\n{context}"}
            ])

            # Execute LLM's decision
            result = self.execute(ctx, response)

            # Update context with result
            context = result.output

            if response.task_complete:
                break

        ctx.done()
```

Hardcoded Agent (WRONG)

```python
class HardcodedAgent(Agent):
    def run(self, ctx: AgentContext):
        instruction = ctx.instruction.lower()

        # WRONG: Task-specific routing
        if "create" in instruction and "file" in instruction:
            self.handle_file_creation(ctx)
        elif "compile" in instruction:
            self.handle_compilation(ctx)
        elif "git" in instruction:
            self.handle_git_task(ctx)
        else:
            self.handle_generic(ctx)

        ctx.done()
```

WHY GENERALIST AGENTS MATTER

1. Adaptability

A generalist agent can handle tasks it has never seen before. A hardcoded agent fails on any task outside its predefined categories.

TASK	GENERALIST	HARDCODED
"Create hello.txt"	LLM reasons, executes	Matches "create" + "file"
"Implement FEAL cryptanalysis"	LLM reasons, executes	No handler, fails
"Fix the TypeScript error"	LLM reasons, executes	No handler, fails

2. Robustness

Hardcoded agents are brittle:

```python
# This hardcoded check breaks on slight variations
if "create" in instruction and "file" in instruction:
    # Fails on: "Write a new file", "Generate output.txt", "Make a text document"
    pass
```

Generalist agents handle variations naturally because the LLM understands semantics.

3. Maintainability

Adding support for new task types:

→ **Generalist**: No code changes needed

→ **Hardcoded**: Add new handler, new keywords, new logic

CHARACTERISTICS OF A GENERALIST AGENT

Must Have

CHARACTERISTIC	DESCRIPTION
Single code path	Same logic for all tasks
LLM-driven decisions	LLM chooses actions, not conditionals
No task keywords	Zero references to specific task content
Iterative execution	Observe-think-act loop
Dynamic planning	Plan emerges from LLM reasoning

Must NOT Have

ANTI-PATTERN	EXAMPLE
Task detection	`if "file" in task`
Handler dispatch	`handlerstask_type`
Keyword matching	`re.search(r"create` write make", task)
Cached responses	`return known_solutions[task_hash]`
Template injection	`code = TEMPLATES[detected_type]`

THE GENERALIST TEST

Ask yourself these questions about your agent:

1. Would removing all knowledge of specific tasks break it?

- Generalist: No, it would still work
- Hardcoded: Yes, it relies on task-specific code

2. Does it have if-statements checking instruction content?

- Generalist: No
- Hardcoded: Yes

3. Could it solve a completely novel task type?

- Generalist: Yes, via LLM reasoning
- Hardcoded: Only if a handler exists

4. Is the code shorter than 500 lines for core logic?

- Generalist: Usually yes (simple loop + LLM)
- Hardcoded: Often much longer (many handlers)

REAL-WORLD EXAMPLE: TERMINUS-2

The Terminus-2 agent demonstrates generalist design:

PYTHON

```python
class Terminus2Agent(Agent):
    def run(self, ctx: AgentContext):
        # Build initial context (same for ALL tasks)
        terminal_state = ctx.shell("pwd && ls -la").output

        self.history = [
            {"role": "system", "content": SYSTEM_PROMPT},
            {"role": "user", "content": f"Task: {ctx.instruction}\n\nTerminal:\n{terminal_state}"}
        ]

        # Main loop (same for ALL tasks)
        while True:
            response = self.llm.chat(self.history[-20:])
            result = self.parser.parse_response(response.text)

            if result.is_task_complete:
                break

            # Execute whatever commands LLM decided
            output = self.execute_commands(ctx, result.commands)
            self.history.append({"role": "user", "content": output})

        ctx.done()
```

Key observations:

→ **No task-specific code**: Works on any task

→ **LLM decides commands**: No hardcoded command sequences

→ **Generic loop**: Same structure regardless of task type

THE SPECTRUM OF GENERALIZATION

BANNED → ACCEPTABLE → RECOMMENDED

→ **Lookup Table**: Pre-computed solutions indexed by task

→ **Pattern Match**: Regex on instructions

→ **Keyword Routing**: If-else based on keywords

→ **Few-shot**: Examples in prompt (acceptable if generic)

→ **Zero-shot**: Direct instruction to LLM

→ **Pure Reason**: LLM with tools, no examples needed

SUMMARY

ASPECT	GENERALIST	HARDCODED
Task handling	Uniform	Task-specific
Decision making	LLM	Conditionals
Novel tasks	Handles	Fails
Code complexity	Low	High
Maintenance	Easy	Difficult
Verdict	**USE THIS**	**AVOID**

The goal is to build agents that **think**, not agents that **match patterns**.

CHAPTER 02

ARCHITECTURE PATTERNS

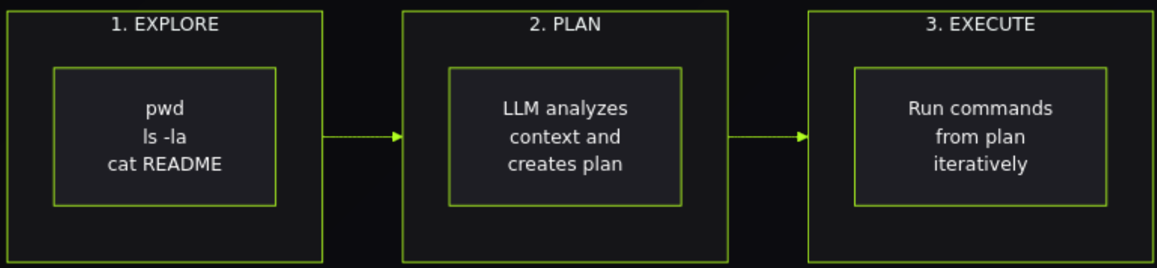
02 - ARCHITECTURE PATTERNS

This document covers proven architecture patterns for building generalist agents. These patterns form the foundation of effective agent design.

PATTERN 1: EXPLORE-PLAN-EXECUTE (EPE)

The most fundamental pattern. Every task should follow this sequence:

1. EXPLORE → 2. PLAN → 3. EXECUTE



Implementation

PYTHON

```python
def run(self, ctx: AgentContext):
    # 1. EXPLORE - Gather context
    explore_results = []
    explore_results.append(ctx.shell("pwd").output)
    explore_results.append(ctx.shell("ls -la").output)
    explore_results.append(ctx.shell("cat README.md 2>/dev/null || echo 'No README'").output)

    context = "\n".join(explore_results)

    # 2. PLAN - Let LLM analyze and plan
    plan = self.llm.ask(
        f"Task: {ctx.instruction}\n\n"
        f"Current environment:\n{context}\n\n"
        "Analyze the task and create a step-by-step plan.",
        system="You are a planning assistant. Create clear, actionable plans."
    )

    ctx.log(f"Plan: {plan.text[:200]}...")

    # 3. EXECUTE - Implement the plan
    self.execute_plan(ctx, plan)

    ctx.done()
```

When to Use

→ **Always** as the initial approach to any task

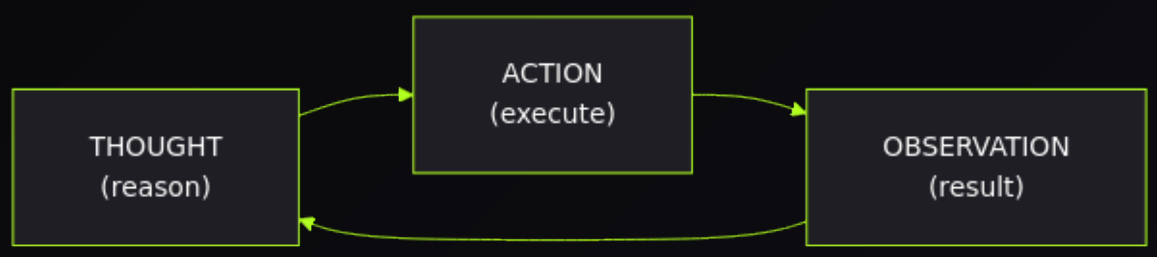
→ When you need to understand the environment before acting

→ For complex tasks requiring multi-step solutions

PATTERN 2: REACT (REASONING + ACTING)

Interleave reasoning and action in a tight loop:

FLOWCHART



Implementation

PYTHON

```python
REACT_PROMPT = """You are an agent solving tasks. For each step:
1. THOUGHT: Analyze the current situation
2. ACTION: Decide what command to run
3. Wait for OBSERVATION (command output)
4. Repeat until task is complete

Respond with JSON:
{
    "thought": "your reasoning here",
    "action": "shell command to run",
    "task_complete": false
}"""

def run(self, ctx: AgentContext):
    messages = [
        {"role": "system", "content": REACT_PROMPT},
        {"role": "user", "content": f"Task: {ctx.instruction}"}
    ]

    while ctx.step < 100:
        response = self.llm.chat(messages)
        data = response.json()

        if data.get("task_complete"):
            break

        # Execute action
        action = data.get("action")
        if action:
            result = ctx.shell(action)
            observation = f"OBSERVATION:\n{result.output[-3000:]}"

            messages.append({"role": "assistant", "content": response.text})
            messages.append({"role": "user", "content": observation})

    ctx.done()
```

When to Use

→ Tasks requiring iterative refinement

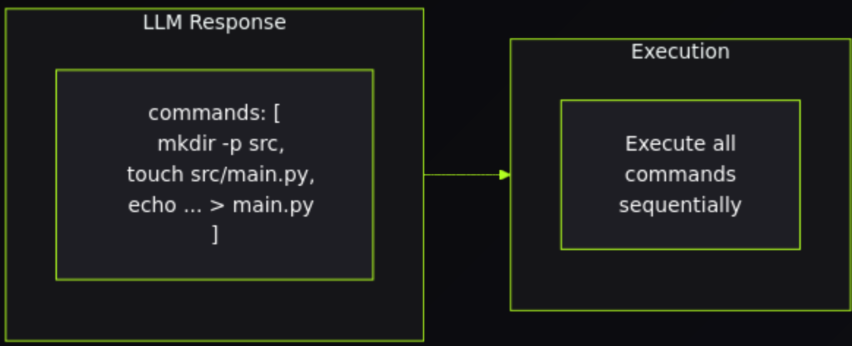
→ When you can't plan everything upfront

→ Debugging and troubleshooting scenarios

PATTERN 3: BATCH COMMAND EXECUTION

Execute multiple commands per LLM call for efficiency:

LLM RESPONSE → EXECUTION



Implementation (Terminus-2 Style)

PYTHON

```python
BATCH_PROMPT = """Respond with JSON containing multiple commands:
{
    "analysis": "what you observe",
    "plan": "what you will do",
    "commands": [
        {"keystrokes": "command1", "duration": 0.1},
        {"keystrokes": "command2", "duration": 1.0}
    ],
    "task_complete": false
}"""

def execute_batch(self, ctx: AgentContext, commands: list) -> str:
    outputs = []

    for cmd in commands:
        result = ctx.shell(cmd["keystrokes"], timeout=int(cmd.get("duration", 1) * 60))
        output = f"$ {cmd['keystrokes']}\n{result.output}"

        if result.exit_code != 0:
            output += f"\n[exit code: {result.exit_code}]"

        outputs.append(output)

        if result.timed_out:
            outputs.append("[TIMEOUT]")
            break

    return "\n\n".join(outputs)
```

When to Use

→ When efficiency matters (fewer LLM calls)

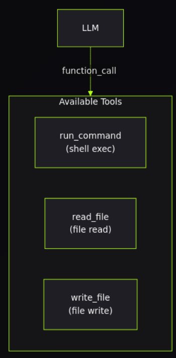
→ For well-understood sequences of operations

→ When commands are independent

PATTERN 4: TOOL-AUGMENTED REASONING

Give the LLM explicit tools to call:

AVAILABLE TOOLS



Implementation

PYTHON

```python
from term_sdk import Tool

TOOLS = [
    Tool(
        name="run_command",
        description="Execute a shell command",
        parameters={
            "type": "object",
            "properties": {
                "command": {"type": "string", "description": "Shell command to run"}
            },
            "required": ["command"]
        }
    ),
    Tool(
        name="read_file",
        description="Read contents of a file",
        parameters={
            "type": "object",
            "properties": {
                "path": {"type": "string", "description": "File path to read"}
            },
            "required": ["path"]
        }
    ),
    Tool(
        name="write_file",
        description="Write content to a file",
        parameters={
            "type": "object",
            "properties": {
                "path": {"type": "string", "description": "File path"},
                "content": {"type": "string", "description": "Content to write"}
            },
            "required": ["path", "content"]
        }
    )
]

def setup(self):
    self.llm = LLM(default_model="anthropic/claude-3.5-sonnet")
    self.llm.register_function("run_command", self.run_command)
    self.llm.register_function("read_file", self.read_file)
    self.llm.register_function("write_file", self.write_file)

def run(self, ctx: AgentContext):
    self.ctx = ctx

    messages = [
        {"role": "system", "content": "You are a task-solving agent with tools."},
        {"role": "user", "content": ctx.instruction}
    ]

    # Let LLM use tools automatically
    response = self.llm.chat_with_functions(messages, TOOLS, max_iterations=50)

    ctx.done()
```

When to Use

→ When you want structured tool interactions

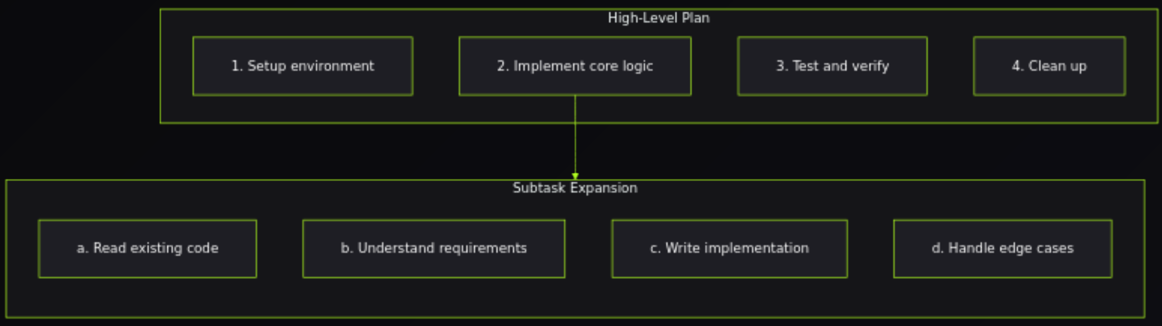
→ For complex agents with many capabilities

→ When you need clear separation of concerns

PATTERN 5: HIERARCHICAL PLANNING

Break complex tasks into subtasks:

HIGH-LEVEL PLAN → SUBTASK EXPANSION



Implementation

PYTHON

```python
def run(self, ctx: AgentContext):
    # Get high-level plan
    high_level = self.llm.ask(
        f"Task: {ctx.instruction}\n\n"
        "Break this into 3-5 high-level steps. Be concise.",
        system="You are a planning assistant."
    )

    steps = self.parse_steps(high_level.text)

    for i, step in enumerate(steps):
        ctx.log(f"Step {i+1}/{len(steps)}: {step[:50]}...")

        # Expand step into detailed actions
        detailed = self.llm.ask(
            f"Current step: {step}\n\n"
            f"Original task: {ctx.instruction}\n\n"
            "What specific commands should I run?",
            system="You are an execution assistant. Provide shell commands."
        )

        # Execute detailed plan
        self.execute_step(ctx, detailed)

    ctx.done()
```

When to Use

→ Very complex tasks with many parts

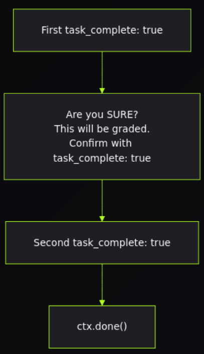
→ Tasks requiring different types of expertise

→ When simple linear planning isn't sufficient

PATTERN 6: DOUBLE CONFIRMATION FOR COMPLETION

Never mark complete without verification:

FLOWCHART



Implementation (Terminus-2 Style)

PYTHON

```python
CONFIRM_MESSAGE = """Current terminal state:
{terminal_output}

Are you sure you want to mark the task as complete?
This will trigger grading and you won't be able to make corrections.
If so, include "task_complete": true in your response again."""

def run(self, ctx: AgentContext):
    pending_completion = False

    while True:
        response = self.llm.chat(self.history)
        result = self.parse(response.text)

        if result.task_complete:
            if pending_completion:
                # Second confirmation - actually complete
                break
            else:
                # First signal - ask for confirmation
                pending_completion = True
                terminal_state = ctx.shell("pwd && ls -la").output
                self.history.append({
                    "role": "user",
                    "content": CONFIRM_MESSAGE.format(terminal_output=terminal_state)
                })
                continue
        else:
            pending_completion = False

        # Normal execution...

    ctx.done()
```

When to Use

→ **Always** for task completion

→ Prevents premature completion

→ Gives LLM chance to reconsider

CHOOSING THE RIGHT PATTERN

SCENARIO	RECOMMENDED PATTERN
Simple, well-defined tasks	EPE
Debugging, troubleshooting	ReAct
Efficiency-critical	Batch Execution
Complex multi-tool tasks	Tool-Augmented
Very large tasks	Hierarchical
Any completion	Double Confirmation

COMBINING PATTERNS

Most effective agents combine multiple patterns:

PYTHON

```python
class AdvancedAgent(Agent):
    def run(self, ctx: AgentContext):
        # EPE: Explore first
        context = self.explore(ctx)

        # Hierarchical: Plan at high level
        plan = self.create_high_level_plan(ctx.instruction, context)

        for step in plan:
            # ReAct: Execute each step iteratively
            self.execute_with_react(ctx, step)

        # Double Confirmation: Verify before done
        if self.confirm_completion(ctx):
            ctx.done()
```

The key is using the right pattern for each phase of the task.

CHAPTER 03

ALLOWED VS FORBIDDEN

03 - ALLOWED VS FORBIDDEN

This document clearly defines what is permitted and what is prohibited when building generalist agents for Subnet 100.

THE CORE PRINCIPLE

> Your agent must solve tasks through reasoning, not recognition.

If your agent can identify a specific task and take a shortcut, it's cheating. The benchmark tests whether your agent can **think**, not whether it can **pattern match**.

ALLOWED BEHAVIORS

LLM Usage

ALLOWED	EXAMPLE
Using LLM to analyze tasks	`llm.ask(f"Task: {task}\nWhat should I do?")`
Using LLM to generate code	`llm.ask("Write Python code that...")`
Using LLM to debug errors	`llm.ask(f"This error occurred: {error}")`
Using LLM to verify output	`llm.ask(f"Does this output match: {output}")`
Using LLM with tools	Function calling, tool use
Multi-turn conversations	Building on previous context

Shell Operations

ALLOWED	EXAMPLE
Exploring the environment	`ls -la`, `pwd`, `find .`
Running builds	`make`, `npm install`, `cargo build`
Executing tests	`pytest`, `npm test`
File operations	`cat`, `cp`, `mv`, `rm`
Process management	`ps`, `kill`, `timeout`
Network tools	`curl`, `wget` (to allowed endpoints)

File Operations

ALLOWED	EXAMPLE
Reading any file in workspace	`ctx.read("/app/src/main.py")`
Writing output files	`ctx.write("/app/result.txt", content)`
Creating directories	`mkdir -p /app/output`
Modifying source code	Editing files to fix bugs
Deleting files	Cleanup operations

Agent Architecture

ALLOWED	EXAMPLE
Generic system prompts	"You are a task-solving agent..."
Tool definitions	Shell, read, write tools
Error handling	Retry logic, fallbacks
Output truncation	Limiting context size
State management	Tracking conversation history
Iterative refinement	Multiple attempts at a task

FORBIDDEN BEHAVIORS

Task-Specific Logic

FORBIDDEN	WHY
Keyword matching on instructions	`if "file" in task`
Task type classification	`task_type = classify(task)`
Handler dispatch tables	`handlers[task_type]()`
Regex matching on tasks	`re.search(r"create.*file", task)`
Embedding similarity to templates	Semantic task matching

Pre-Computed Solutions

FORBIDDEN	WHY
Cached responses	Solutions stored from previous runs
Template injection	Pre-written code for task types
Lookup tables	`solutions[task_hash]`
Hardcoded commands	Commands not from LLM reasoning
Response databases	Indexed solution retrieval

Test Circumvention

FORBIDDEN	WHY
Reading test files	Accessing `/tests/` or similar
Parsing assertions	Extracting expected outputs
Reverse-engineering validators	Understanding test logic
Timing-based detection	Detecting test vs real tasks
Environment fingerprinting	Detecting benchmark environment

Unauthorized Access

FORBIDDEN	WHY
External APIs (except allowed LLM)	No unauthorized network calls
Internet resources during tasks	No fetching external content
Other model providers	Only configured providers
System files outside workspace	`/etc/`, `/root/`, etc.

GRAY AREAS (USE CAUTION)

Generic Few-Shot Examples

Potentially OK:

PYTHON

```python
SYSTEM_PROMPT = """You are a task-solving agent.

Example of good JSON format:
{"thought": "I need to explore", "action": "ls -la"}
"""
```

Not OK:

```python
SYSTEM_PROMPT = """Here are examples of tasks:
Task: Create hello.txt
Solution: echo "Hello" > hello.txt

Task: Compile main.c
Solution: gcc main.c -o main
"""
```

The difference: Generic format examples are fine. Task-specific solution examples are not.

Environment Detection

OK:

```python
# Checking if a tool exists
result = ctx.shell("which python3")
if result.ok:
    python_cmd = "python3"
else:
    python_cmd = "python"
```

Not OK:

```python
# Detecting benchmark environment
if os.environ.get("TERM_BENCH"):
    use_special_behavior()
```

Error Patterns

OK:

```python
# Generic error recovery
if "permission denied" in result.output.lower():
    ctx.shell("sudo " + cmd)
```

Not OK:

```python
# Task-specific error handling
if "numpy" in result.output and "deprecated" in result.output:
    # This knows about specific task errors
    apply_numpy_fix()
```

DECISION TREE

Use this to evaluate if something is allowed:

#	QUESTION	IF YES	IF NO
1	Is behavior based on task instruction content?	FORBIDDEN	Continue →
2	Does it skip LLM reasoning for specific task types?	FORBIDDEN	Continue →
3	Does it access test/validation files?	FORBIDDEN	Continue →
4	Does it use pre-computed solutions?	FORBIDDEN	Continue →
5	Does it make unauthorized network requests?	FORBIDDEN	ALLOWED

Simple rule: If you answer YES to any question, the behavior is FORBIDDEN.

EXAMPLES OF VIOLATIONS

Violation 1: Keyword Matching

PYTHON

```python
# FORBIDDEN
def run(self, ctx):
    if "hello" in ctx.instruction.lower():
        ctx.shell('echo "Hello, World!" > hello.txt')
        ctx.done()
        return
```

Why: Matches on task content to bypass reasoning.

Violation 2: Task Classification

```python
# FORBIDDEN
TASK_TYPES = {
    "file_creation": ["create", "write", "make", "generate"],
    "compilation": ["compile", "build", "make"],
    "git": ["commit", "push", "branch", "merge"]
}

def classify_task(instruction):
    for task_type, keywords in TASK_TYPES.items():
        if any(kw in instruction.lower() for kw in keywords):
            return task_type
    return "unknown"
```

Why: Pre-defined task categories based on keywords.

Violation 3: Solution Cache

```python
# FORBIDDEN
KNOWN_SOLUTIONS = {
    "create a file named hello.txt": 'echo "Hello" > hello.txt',
    "list all files": "ls -la",
}

def run(self, ctx):
    if ctx.instruction in KNOWN_SOLUTIONS:
        ctx.shell(KNOWN_SOLUTIONS[ctx.instruction])
```

Why: Cached solutions bypass reasoning entirely.

Violation 4: Test File Access

```python
# FORBIDDEN
def run(self, ctx):
    # Trying to read test expectations
    test_file = ctx.shell("find . -name 'test*.py' | head -1").stdout.strip()
    if test_file:
        test_content = ctx.read(test_file)
        expected = extract_assertions(test_content)
```

Why: Accessing test files to extract expected outputs.

ALLOWED PATTERNS (FOR REFERENCE)

Pattern A: Pure LLM Reasoning

```python
# ALLOWED
def run(self, ctx):
    context = ctx.shell("pwd && ls -la").output

    response = self.llm.ask(
        f"Task: {ctx.instruction}\n\nEnvironment:\n{context}\n\n"
        "What should I do?",
        system="You are a task-solving agent."
    )

    # LLM decides everything
    self.execute_llm_plan(ctx, response)
```

Pattern B: Generic Error Recovery

```python
# ALLOWED - Generic, not task-specific
def run_with_retry(self, ctx, cmd, retries=3):
    for attempt in range(retries):
        result = ctx.shell(cmd)
        if result.ok:
            return result

        # Generic retry on transient errors
        if "connection" in result.stderr.lower():
            ctx.shell("sleep 5")
            continue

        break
    return result
```

Pattern C: Output Verification

```python
# ALLOWED - Verifies via LLM, not hardcoded checks
def verify_output(self, ctx, expected_path):
    if not ctx.shell(f"test -f {expected_path}").ok:
        return False

    content = ctx.read(expected_path).stdout

    verification = self.llm.ask(
        f"Task was: {ctx.instruction}\n\n"
        f"Output file contains:\n{content[:1000]}\n\n"
        "Does this correctly complete the task? Answer YES or NO."
    )

    return "YES" in verification.text.upper()
```

SUMMARY TABLE

CATEGORY	ALLOWED	FORBIDDEN
LLM reasoning	All	N/A
Task keywords	Never check	Always forbidden
Shell commands	From LLM	Hardcoded per task
File access	Workspace only	Test files
Error handling	Generic	Task-specific
Caching	Conversation history	Solution lookup
Network	Allowed LLM providers	External APIs

When in doubt: If it gives an advantage on known tasks but wouldn't help on novel tasks, it's probably forbidden.

CHAPTER 04

ANTI-PATTERNS (30+ EXAMPLES)

04 - ANTI-PATTERNS (30+ EXAMPLES)

This document catalogs **anti-patterns** - things you must NOT do when building generalist agents. Each anti-pattern includes:

→ Code example showing the violation

→ Explanation of why it's wrong

→ Correct alternative

CATEGORY 1: PATTERN MATCHING ON INSTRUCTIONS (10 EXAMPLES)

Anti-Pattern 1.1: Direct Keyword Matching

PYTHON

```python
# WRONG
def run(self, ctx: AgentContext):
    if "hello" in ctx.instruction.lower():
        ctx.shell('echo "Hello, World!" > hello.txt')
        ctx.done()
        return
```

Why it's wrong: Matches specific task keywords to bypass LLM reasoning.

Correct approach:

PYTHON

```python
# RIGHT
def run(self, ctx: AgentContext):
    response = self.llm.ask(f"Task: {ctx.instruction}\nWhat commands should I run?")
    commands = self.parse_commands(response.text)
    for cmd in commands:
        ctx.shell(cmd)
    ctx.done()
```

Anti-Pattern 1.2: Multi-Keyword Conditional

```python
# WRONG
def run(self, ctx: AgentContext):
    task = ctx.instruction.lower()

    if "create" in task and "file" in task:
        self.handle_file_creation(ctx)
    elif "compile" in task or "build" in task:
        self.handle_compilation(ctx)
    elif "test" in task or "pytest" in task:
        self.handle_testing(ctx)
```

Why it's wrong: Routes tasks based on keyword combinations.

Anti-Pattern 1.3: Regex Task Detection

```python
# WRONG
import re

TASK_PATTERNS = {
    r"create\s+(a\s+)?file": "file_creation",
    r"(compile|build)\s+": "compilation",
    r"(fix|debug)\s+": "debugging",
    r"(commit|push)\s+": "git_ops"
}

def classify_task(instruction):
    for pattern, task_type in TASK_PATTERNS.items():
        if re.search(pattern, instruction, re.IGNORECASE):
            return task_type
    return "generic"
```

Why it's wrong: Pre-defined regex patterns for task classification.

Anti-Pattern 1.4: Task Type Enum

```python
# WRONG
from enum import Enum

class TaskType(Enum):
    FILE_OPERATION = "file"
    GIT_OPERATION = "git"
    BUILD = "build"
    TEST = "test"
    DEBUG = "debug"
    UNKNOWN = "unknown"

def detect_task_type(instruction: str) -> TaskType:
    keywords = {
        TaskType.FILE_OPERATION: ["file", "create", "write", "read", "delete"],
        TaskType.GIT_OPERATION: ["git", "commit", "push", "branch", "merge"],
        TaskType.BUILD: ["build", "compile", "make", "cargo", "npm"],
        TaskType.TEST: ["test", "pytest", "jest", "spec"],
        TaskType.DEBUG: ["debug", "fix", "error", "bug"]
    }

    for task_type, words in keywords.items():
        if any(word in instruction.lower() for word in words):
            return task_type
    return TaskType.UNKNOWN
```

Why it's wrong: Defines fixed task categories based on keyword lists.

Anti-Pattern 1.5: Handler Dispatch Table

```python
# WRONG
class Agent:
    def __init__(self):
        self.handlers = {
            "file": self.handle_file_task,
            "git": self.handle_git_task,
            "python": self.handle_python_task,
            "compile": self.handle_compile_task,
        }

    def run(self, ctx: AgentContext):
        for keyword, handler in self.handlers.items():
            if keyword in ctx.instruction.lower():
                handler(ctx)
                return
        self.handle_generic(ctx)
```

Why it's wrong: Dispatch table based on instruction keywords.

Anti-Pattern 1.6: Instruction Hashing

```python
# WRONG
KNOWN_TASKS = {
    hash("create a file named hello.txt"): "echo 'Hello' > hello.txt",
    hash("list all files"): "ls -la",
    hash("show current directory"): "pwd",
}

def run(self, ctx: AgentContext):
    task_hash = hash(ctx.instruction.lower().strip())
    if task_hash in KNOWN_TASKS:
        ctx.shell(KNOWN_TASKS[task_hash])
        ctx.done()
        return
```

Why it's wrong: Uses instruction hash as lookup key.

Anti-Pattern 1.7: Prefix/Suffix Matching

```python
# WRONG
def run(self, ctx: AgentContext):
    instruction = ctx.instruction.lower()

    if instruction.startswith("write"):
        self.write_mode(ctx)
    elif instruction.startswith("read"):
        self.read_mode(ctx)
    elif instruction.endswith(".py"):
        self.python_mode(ctx)
    elif instruction.endswith(".rs"):
        self.rust_mode(ctx)
```

Why it's wrong: Routes based on instruction prefix/suffix.

Anti-Pattern 1.8: Keyword Extraction for Routing

PYTHON

```python
# WRONG
def extract_action_keywords(instruction: str) -> set:
    actions = {"create", "delete", "modify", "read", "write", "compile", "run", "test"}
    words = set(instruction.lower().split())
    return words & actions

def run(self, ctx: AgentContext):
    keywords = extract_action_keywords(ctx.instruction)

    if "create" in keywords:
        self.creation_flow(ctx)
    elif "delete" in keywords:
        self.deletion_flow(ctx)
```

Why it's wrong: Extracts action keywords to determine flow.

Anti-Pattern 1.9: Semantic Similarity to Templates

PYTHON

```python
# WRONG
from sentence_transformers import SentenceTransformer

TASK_TEMPLATES = [
    ("Create a new file with content", "file_creation"),
    ("Compile the source code", "compilation"),
    ("Run the test suite", "testing"),
]

def classify_by_embedding(instruction: str, model) -> str:
    inst_embedding = model.encode(instruction)

    best_match = None
    best_score = -1

    for template, task_type in TASK_TEMPLATES:
        template_embedding = model.encode(template)
        score = cosine_similarity(inst_embedding, template_embedding)
        if score > best_score:
            best_score = score
            best_match = task_type

    return best_match if best_score > 0.7 else "unknown"
```

Why it's wrong: Uses embedding similarity to pre-defined templates.

Anti-Pattern 1.10: NLP Task Classification

```python
# WRONG
from transformers import pipeline

classifier = pipeline("zero-shot-classification")
TASK_LABELS = ["file operation", "git operation", "compilation", "testing", "debugging"]

def run(self, ctx: AgentContext):
    result = classifier(ctx.instruction, TASK_LABELS)
    task_type = result["labels"][0]

    if task_type == "file operation":
        self.file_handler(ctx)
    elif task_type == "git operation":
        self.git_handler(ctx)
    # ...
```

Why it's wrong: Uses ML classifier with pre-defined task labels.

CATEGORY 2: PRE-DEFINED RESPONSES (8 EXAMPLES)

Anti-Pattern 2.1: Hardcoded Shell Commands

```python
# WRONG
def run(self, ctx: AgentContext):
    # Hardcoded sequence without LLM
    ctx.shell("git add .")
    ctx.shell("git commit -m 'Update'")
    ctx.shell("git push")
    ctx.done()
```

Why it's wrong: Commands are hardcoded, not from LLM reasoning.

Anti-Pattern 2.2: Template Responses

PYTHON

```python
# WRONG
TEMPLATES = {
    "file_creation": """
with open('{filename}', 'w') as f:
    f.write('{content}')
""",
    "read_json": """
import json
with open('{filename}') as f:
    data = json.load(f)
""",
}

def generate_code(task_type: str, **kwargs) -> str:
    return TEMPLATES[task_type].format(**kwargs)
```

Why it's wrong: Pre-written code templates injected without reasoning.

Anti-Pattern 2.3: Solution Database

PYTHON

```python
# WRONG
import sqlite3

class SolutionCache:
    def __init__(self):
        self.conn = sqlite3.connect("solutions.db")

    def lookup(self, instruction: str) -> str | None:
        cursor = self.conn.execute(
            "SELECT solution FROM cache WHERE instruction LIKE ?",
            (f"%{instruction[:50]}%",)
        )
        row = cursor.fetchone()
        return row[0] if row else None

def run(self, ctx: AgentContext):
    cached = self.cache.lookup(ctx.instruction)
    if cached:
        exec(cached)  # Execute cached solution
        ctx.done()
        return
```

Why it's wrong: Looks up solutions from a database.

Anti-Pattern 2.4: Cached Conversation Responses

```python
# WRONG
RESPONSE_CACHE = {}

def run(self, ctx: AgentContext):
    cache_key = ctx.instruction[:100]

    if cache_key in RESPONSE_CACHE:
        # Replay cached response instead of calling LLM
        self.execute_cached(ctx, RESPONSE_CACHE[cache_key])
        ctx.done()
        return

    response = self.llm.ask(ctx.instruction)
    RESPONSE_CACHE[cache_key] = response
    self.execute(ctx, response)
```

Why it's wrong: Caches and replays LLM responses for similar instructions.

Anti-Pattern 2.5: Static File Content

```python
# WRONG
KNOWN_FILES = {
    "hello.txt": "Hello, World!",
    "readme.md": "# Project\n\nThis is a project.",
    "config.json": '{"version": "1.0"}',
}

def run(self, ctx: AgentContext):
    for filename, content in KNOWN_FILES.items():
        if filename in ctx.instruction.lower():
            ctx.write(f"/app/{filename}", content)
            ctx.done()
            return
```

Why it's wrong: Pre-defined file contents for known filenames.

Anti-Pattern 2.6: Command Sequences Per Task Type

```python
# WRONG
COMMAND_SEQUENCES = {
    "python_project": [
        "python -m venv venv",
        "source venv/bin/activate",
        "pip install -r requirements.txt",
        "python main.py"
    ],
    "node_project": [
        "npm install",
        "npm run build",
        "npm start"
    ],
}

def run(self, ctx: AgentContext):
    if "python" in ctx.instruction.lower():
        for cmd in COMMAND_SEQUENCES["python_project"]:
            ctx.shell(cmd)
```

Why it's wrong: Pre-defined command sequences by task type.

Anti-Pattern 2.7: Fallback Responses

```python
# WRONG
DEFAULT_RESPONSES = {
    "unknown_error": "echo 'Task could not be completed'",
    "file_not_found": "touch placeholder.txt",
    "compilation_failed": "echo '// TODO: fix compilation' > fix.txt",
}

def run(self, ctx: AgentContext):
    try:
        self.attempt_task(ctx)
    except Exception as e:
        error_type = classify_error(e)
        ctx.shell(DEFAULT_RESPONSES.get(error_type, "echo 'Error'"))
```

Why it's wrong: Pre-defined fallback responses for error types.

Anti-Pattern 2.8: Output Format Templates

PYTHON

```python
# WRONG
OUTPUT_TEMPLATES = {
    "result.txt": "Result: {value}",
    "output.json": '{{"status": "success", "data": {data}}}',
    "answer.txt": "The answer is: {answer}",
}

def write_output(self, ctx, filename: str, **values):
    if filename in OUTPUT_TEMPLATES:
        content = OUTPUT_TEMPLATES[filename].format(**values)
        ctx.write(f"/app/{filename}", content)
```

Why it's wrong: Pre-defined output format templates.

CATEGORY 3: CHEATING/CIRCUMVENTION (7 EXAMPLES)

Anti-Pattern 3.1: Reading Test Files

PYTHON

```python
# WRONG
def run(self, ctx: AgentContext):
    # Trying to read test expectations
    test_files = ctx.shell("find . -name 'test_*.py' -o -name '*_test.py'").stdout

    for test_file in test_files.strip().split('\n'):
        content = ctx.read(test_file)
        expected = self.extract_assertions(content)
        self.use_expected_values(expected)
```

Why it's wrong: Reads test files to extract expected outputs.

Anti-Pattern 3.2: Parsing Assertion Values

PYTHON

```python
# WRONG
import ast

def extract_expected_values(test_content: str) -> list:
    tree = ast.parse(test_content)
    expected = []

    for node in ast.walk(tree):
        if isinstance(node, ast.Call):
            if hasattr(node.func, 'attr') and node.func.attr in ('assertEqual', 'assertEquals'):
                # Extract the expected value from assertions
                expected.append(ast.literal_eval(node.args[1]))

    return expected
```

Why it's wrong: Parses test code to extract expected values.

Anti-Pattern 3.3: Environment Detection

PYTHON

```python
# WRONG
import os

def run(self, ctx: AgentContext):
    # Detect if running in benchmark environment
    if os.environ.get("TERM_BENCH") or os.environ.get("BENCHMARK_MODE"):
        self.benchmark_optimized_mode(ctx)
    else:
        self.normal_mode(ctx)
```

Why it's wrong: Detects benchmark environment for special behavior.

Anti-Pattern 3.4: Workspace Path Analysis

PYTHON

```python
# WRONG
def run(self, ctx: AgentContext):
    pwd = ctx.shell("pwd").stdout.strip()

    # Detect task by workspace path
    if "polyglot" in pwd:
        self.handle_polyglot(ctx)
    elif "cryptanalysis" in pwd:
        self.handle_crypto(ctx)
    elif "sanitize" in pwd:
        self.handle_sanitization(ctx)
```

Why it's wrong: Infers task type from workspace path.

Anti-Pattern 3.5: File Structure Detection

PYTHON

```python
# WRONG
def detect_project_type(self, ctx: AgentContext) -> str:
    files = ctx.shell("ls -la").stdout

    # Detect project type by files present
    if "Cargo.toml" in files:
        return "rust"
    elif "package.json" in files:
        return "node"
    elif "setup.py" in files or "pyproject.toml" in files:
        return "python"
    elif "Makefile" in files:
        return "make"

    return "unknown"

def run(self, ctx: AgentContext):
    project_type = self.detect_project_type(ctx)
    # Use project type to skip reasoning
    self.handlers[project_type](ctx)
```

Why it's wrong: Uses file structure to bypass task analysis.

Note: Detecting project type via LLM reasoning is fine. Hardcoded detection with handlers is not.

CATEGORY 4: BAD LLM USAGE (5 EXAMPLES)

Anti-Pattern 4.1: No Context Truncation

```python
# WRONG
def run(self, ctx: AgentContext):
    all_output = ""

    for i in range(100):
        response = self.llm.chat([
            {"role": "system", "content": SYSTEM_PROMPT},
            {"role": "user", "content": f"Task: {ctx.instruction}\n\nAll output so far:\n{all_output}"}
        ])

        result = ctx.shell(response.text)
        all_output += result.output  # Grows unboundedly!
```

Why it's wrong: Context grows without limit, causing overflow.

Correct approach:

```python
# RIGHT
def run(self, ctx: AgentContext):
    history = []

    for i in range(100):
        # Keep only last N messages
        recent_history = history[-20:]

        response = self.llm.chat([
            {"role": "system", "content": SYSTEM_PROMPT},
            *recent_history
        ])

        result = ctx.shell(response.text)
        # Truncate output
        history.append({"role": "user", "content": result.output[-3000:]})
```

Anti-Pattern 4.2: Single Massive Prompt

```python
# WRONG
def run(self, ctx: AgentContext):
    # Try to do everything in one prompt
    mega_prompt = f"""
    Task: {ctx.instruction}

    Please:
    1. Analyze the task
    2. Create a complete plan
    3. Write all necessary code
    4. Provide all shell commands
    5. Anticipate all errors
    6. Give the final answer

    Respond with everything at once.
    """

    response = self.llm.ask(mega_prompt)
    # Hope it works...
```

Why it's wrong: Tries to do everything in one shot without iteration.

Anti-Pattern 4.3: Ignoring LLM Errors

```python
# WRONG
def run(self, ctx: AgentContext):
    response = self.llm.ask(ctx.instruction)  # Might fail!

    data = response.json()  # Might be None!
    command = data["command"]  # Might KeyError!

    ctx.shell(command)
```

Why it's wrong: No error handling for LLM failures.

Correct approach:

```python
# RIGHT
def run(self, ctx: AgentContext):
    try:
        response = self.llm.ask(ctx.instruction)
    except LLMError as e:
        ctx.log(f"LLM error: {e}")
        if e.code in ("rate_limit", "server_error"):
            time.sleep(5)
            response = self.llm.ask(ctx.instruction)  # Retry
        else:
            raise

    data = response.json()
    if not data:
        ctx.log("Failed to parse JSON, asking for clarification")
        response = self.llm.ask("Please respond with valid JSON")
        data = response.json()

    command = data.get("command")
    if command:
        ctx.shell(command)
```

Anti-Pattern 4.4: No Retry Logic

```python
# WRONG
def run(self, ctx: AgentContext):
    response = self.llm.ask(ctx.instruction)

    if not response.text:
        ctx.done()  # Just give up
        return
```

Why it's wrong: Gives up on first failure without retry.

Anti-Pattern 4.5: Raw Response Without Validation

```python
# WRONG
def run(self, ctx: AgentContext):
    response = self.llm.ask("Give me a shell command")

    # Execute whatever the LLM says without validation
    ctx.shell(response.text)
```

Why it's wrong: Doesn't validate LLM output before execution.

Correct approach:

PYTHON

```python
# RIGHT
def run(self, ctx: AgentContext):
    response = self.llm.ask(
        "Give me a shell command as JSON: {\"command\": \"...\"}",
        system="Respond only with valid JSON."
    )

    data = response.json()
    if not data:
        ctx.log("Invalid response format")
        return

    command = data.get("command", "").strip()
    if not command:
        ctx.log("No command provided")
        return

    # Now safe to execute
    ctx.shell(command)
```

CATEGORY 5: SUBTLE ANTI-PATTERNS (8 EXAMPLES)

Anti-Pattern 5.1: Hardcoded Timeouts Per Task

PYTHON

```python
# WRONG
TASK_TIMEOUTS = {
    "compile": 120,
    "test": 60,
    "download": 300,
    "simple": 10,
}

def run(self, ctx: AgentContext):
    # Determine timeout by task keywords
    timeout = 30  # default
    for keyword, t in TASK_TIMEOUTS.items():
        if keyword in ctx.instruction.lower():
            timeout = t
            break

    ctx.shell(command, timeout=timeout)
```

Why it's wrong: Timeouts are based on task keywords.

Anti-Pattern 5.2: Output Length Expectations

```python
# WRONG
def verify_output(self, ctx, output: str) -> bool:
    # Expect certain output lengths for task types
    if "list" in ctx.instruction.lower():
        return len(output.split('\n')) > 5
    elif "count" in ctx.instruction.lower():
        return len(output) < 20
```

Why it's wrong: Validates based on task-specific expectations.

Anti-Pattern 5.3: File Count Heuristics

```python
# WRONG
def check_completion(self, ctx: AgentContext) -> bool:
    file_count = len(ctx.shell("ls").stdout.split())

    # Expect certain file counts
    if "cleanup" in ctx.instruction.lower():
        return file_count < 5  # Cleanup should reduce files
    elif "generate" in ctx.instruction.lower():
        return file_count > 10  # Generation should create files
```

Why it's wrong: Uses file count heuristics based on task type.

Anti-Pattern 5.4: Exit Code Assumptions

```python
# WRONG
def run(self, ctx: AgentContext):
    result = ctx.shell("make")

    # Assume specific exit codes mean specific things
    if result.exit_code == 2:
        # "Must be missing Makefile, create one"
        ctx.write("Makefile", DEFAULT_MAKEFILE)
```

Why it's wrong: Hardcoded interpretation of exit codes.

Anti-Pattern 5.5: Error Message Pattern Matching

PYTHON

```python
# WRONG
ERROR_HANDLERS = {
    r"No such file": lambda ctx: ctx.shell("touch missing_file"),
    r"Permission denied": lambda ctx: ctx.shell("chmod +x script.sh"),
    r"Module not found": lambda ctx: ctx.shell("pip install -r requirements.txt"),
    r"ECONNREFUSED": lambda ctx: ctx.shell("sleep 5") and retry(),
}

def handle_error(self, ctx, error: str):
    for pattern, handler in ERROR_HANDLERS.items():
        if re.search(pattern, error):
            handler(ctx)
            return
```

Why it's wrong: Pre-defined error handlers with fixed responses.

Note: Generic retry on transient errors is OK. Task-specific error handlers are not.

Anti-Pattern 5.6: Instruction Length Heuristics

PYTHON

```python
# WRONG
def run(self, ctx: AgentContext):
    # Assume short instructions are simple
    if len(ctx.instruction) < 50:
        self.simple_mode(ctx)
    else:
        self.complex_mode(ctx)
```

Why it's wrong: Uses instruction length to determine behavior.

Anti-Pattern 5.7: Step Count Assumptions

PYTHON

```python
# WRONG
def run(self, ctx: AgentContext):
    # Assume task types need certain step counts
    if "deploy" in ctx.instruction.lower():
        max_steps = 20  # Deployments are complex
    elif "create" in ctx.instruction.lower():
        max_steps = 3   # Creation is simple
    else:
        max_steps = 10
```

Why it's wrong: Pre-defines step limits by task keywords.

Anti-Pattern 5.8: Directory Name Detection

PYTHON

```python
# WRONG
def run(self, ctx: AgentContext):
    dirs = ctx.shell("ls -d */").stdout.split()

    # Detect task by directory names
    if "src" in dirs and "tests" in dirs:
        self.python_project_mode(ctx)
    elif "node_modules" in dirs:
        self.node_project_mode(ctx)
```

Why it's wrong: Infers task type from directory structure.

SUMMARY: QUICK REFERENCE

CATEGORY	COUNT	KEY VIOLATION
Pattern Matching	10	Checking instruction content
Pre-Defined Responses	8	Hardcoded solutions/templates
Cheating	7	Accessing test files/environment
Bad LLM Usage	5	No error handling/truncation
Subtle Patterns	8	Hidden task-specific logic
TOTAL	38	

The Golden Test

For any code you write, ask:

> "Would this code behave differently if I changed the task instruction?"

If the answer is YES and it's not because of LLM reasoning, it's an anti-pattern.

CHAPTER 05

BEST PRACTICES

05 - BEST PRACTICES

This document covers proven best practices for building effective generalist agents. Each practice includes code examples and explanations.

PRACTICE 1: ALWAYS EXPLORE FIRST

Before taking any action, gather context about the environment.

Implementation

PYTHON

```python
def explore(self, ctx: AgentContext) -> str:
    """Gather context about the current environment."""
    results = []

    # Current location
    pwd = ctx.shell("pwd")
    results.append(f"Working directory: {pwd.stdout.strip()}")

    # List files
    ls = ctx.shell("ls -la")
    results.append(f"Files:\n{ls.stdout}")

    # Check for README
    readme = ctx.shell("cat README.md 2>/dev/null")
    if readme.ok:
        results.append(f"README.md:\n{readme.stdout[:2000]}")

    # Check for common project files
    for config in ["package.json", "Cargo.toml", "setup.py", "Makefile"]:
        check = ctx.shell(f"cat {config} 2>/dev/null")
        if check.ok:
            results.append(f"{config}:\n{check.stdout[:1000]}")

    return "\n\n".join(results)

def run(self, ctx: AgentContext):
    # ALWAYS explore first
    context = self.explore(ctx)

    # Now let LLM reason with full context
    response = self.llm.ask(
        f"Task: {ctx.instruction}\n\n"
        f"Environment:\n{context}\n\n"
        "What should I do first?",
        system="You are a task-solving agent."
    )

    # Continue with LLM-driven execution...
```

Why This Matters

→ LLM makes better decisions with more context

→ Prevents assumptions about the environment

→ Discovers constraints and requirements early

PRACTICE 2: USE STRUCTURED JSON RESPONSES

Always use JSON for LLM responses to enable reliable parsing.

System Prompt

```python
SYSTEM_PROMPT = """You are a task-solving agent. Always respond with valid JSON:

{
    "analysis": "Your analysis of the current situation",
    "plan": "What you plan to do next",
    "commands": [
        {"command": "shell command here", "timeout": 10}
    ],
    "task_complete": false
}

Rules:
- "analysis" and "plan" are required strings
- "commands" is an array of command objects
- "task_complete" is a boolean (default false)
- Use proper JSON escaping for special characters
"""
```

Parser

```python
import json
import re

def parse_llm_response(response_text: str) -> dict | None:
    """Parse JSON from LLM response, handling common issues."""

    # Try to extract JSON from response
    json_match = re.search(r'\{[\s\S]*\}', response_text)
    if not json_match:
        return None

    json_str = json_match.group()

    try:
        data = json.loads(json_str)
        return data
    except json.JSONDecodeError:
        # Try to fix common issues
        # Fix trailing commas
        json_str = re.sub(r',\s*}', '}', json_str)
        json_str = re.sub(r',\s*]', ']', json_str)

        try:
            return json.loads(json_str)
        except json.JSONDecodeError:
            return None
```

Usage in Agent

PYTHON

```python
def run(self, ctx: AgentContext):
    messages = [
        {"role": "system", "content": SYSTEM_PROMPT},
        {"role": "user", "content": f"Task: {ctx.instruction}"}
    ]

    while True:
        response = self.llm.chat(messages)
        data = parse_llm_response(response.text)

        if not data:
            # Ask for valid JSON
            messages.append({"role": "assistant", "content": response.text})
            messages.append({"role": "user", "content": "Please respond with valid JSON."})
            continue

        if data.get("task_complete"):
            break

        # Execute commands
        for cmd_obj in data.get("commands", []):
            result = ctx.shell(cmd_obj["command"], timeout=cmd_obj.get("timeout", 30))
            messages.append({"role": "assistant", "content": response.text})
            messages.append({"role": "user", "content": f"Output:\n{result.output[-3000:]}"})

    ctx.done()
```

PRACTICE 3: TRUNCATE OUTPUTS FOR CONTEXT MANAGEMENT

Prevent context overflow by truncating long outputs.

Implementation

PYTHON

```python
def truncate_output(output: str, max_bytes: int = 10000) -> str:
    """Truncate output while preserving useful information."""

    if len(output.encode('utf-8')) <= max_bytes:
        return output

    # Keep first and last portions
    portion_size = max_bytes // 2

    output_bytes = output.encode('utf-8')
    first = output_bytes[:portion_size].decode('utf-8', errors='ignore')
    last = output_bytes[-portion_size:].decode('utf-8', errors='ignore')

    omitted = len(output_bytes) - portion_size * 2

    return (
        f"{first}\n"
        f"[... {omitted} bytes omitted ...]\n"
        f"{last}"
    )

def run(self, ctx: AgentContext):
    while True:
        response = self.llm.chat(messages[-20:])  # Keep last 20 messages

        result = ctx.shell(command)

        # Truncate output before adding to context
        truncated = truncate_output(result.output)

        messages.append({
            "role": "user",
            "content": f"Output:\n{truncated}"
        })
```

Alternative: Smart Truncation

PYTHON

```python
def smart_truncate(output: str, max_lines: int = 100) -> str:
    """Truncate keeping error messages and important info."""

    lines = output.split('\n')

    if len(lines) <= max_lines:
        return output

    # Prioritize lines with errors or important info
    priority_patterns = ['error', 'Error', 'ERROR', 'failed', 'Failed', 'warning', 'Warning']

    priority_lines = []
    other_lines = []

    for line in lines:
        if any(p in line for p in priority_patterns):
            priority_lines.append(line)
        else:
            other_lines.append(line)

    # Keep all priority lines + fill rest with other lines
    result_lines = priority_lines[:max_lines // 2]
    remaining = max_lines - len(result_lines)

    # Keep first and last portions of other lines
    if other_lines:
        half = remaining // 2
        result_lines = other_lines[:half] + ['[...]'] + other_lines[-half:] + result_lines

    return '\n'.join(result_lines[:max_lines])
```

PRACTICE 4: IMPLEMENT DOUBLE CONFIRMATION

Never mark complete without verification.

Implementation

PYTHON

```python
CONFIRMATION_PROMPT = """You indicated the task is complete.

Current state:
{terminal_state}

Task was: {instruction}

Are you SURE the task is complete? This will trigger grading.
If yes, respond with {{"task_complete": true}} again.
If not, continue working.
"""

def run(self, ctx: AgentContext):
    pending_confirmation = False

    while True:
        response = self.llm.chat(messages)
        data = parse_llm_response(response.text)

        if data.get("task_complete"):
            if pending_confirmation:
                # Second confirmation - actually done
                ctx.log("Task completion confirmed")
                break
            else:
                # First signal - ask for confirmation
                pending_confirmation = True
                state = ctx.shell("pwd && ls -la").output

                messages.append({"role": "assistant", "content": response.text})
                messages.append({
                    "role": "user",
                    "content": CONFIRMATION_PROMPT.format(
                        terminal_state=state,
                        instruction=ctx.instruction
                    )
                })
                continue
        else:
            pending_confirmation = False

        # Normal execution...

    ctx.done()
```

PRACTICE 5: VERIFY OUTPUT FILES EXIST

Before completing, always verify expected outputs.

Implementation

PYTHON

```python
def verify_outputs(self, ctx: AgentContext, expected_files: list[str]) -> bool:
    """Verify that expected output files exist and have content."""

    for filepath in expected_files:
        # Check existence
        check = ctx.shell(f"test -f {filepath} && echo EXISTS")
        if "EXISTS" not in check.stdout:
            ctx.log(f"Missing output file: {filepath}")
            return False

        # Check non-empty
        size = ctx.shell(f"stat -c%s {filepath} 2>/dev/null || stat -f%z {filepath}")
        if size.ok and int(size.stdout.strip()) == 0:
            ctx.log(f"Output file is empty: {filepath}")
            return False

    return True

def run(self, ctx: AgentContext):
    # ... task execution ...

    # Before completing, verify outputs
    # Let LLM determine expected outputs
    verification = self.llm.ask(
        f"Task: {ctx.instruction}\n\n"
        "What output files should exist? List them as JSON array.",
        system="Respond with JSON: {\"files\": [\"path1\", \"path2\"]}"
    )

    data = parse_llm_response(verification.text)
    expected_files = data.get("files", [])

    if expected_files and not self.verify_outputs(ctx, expected_files):
        ctx.log("Output verification failed, continuing...")
        # Don't mark complete - continue working
    else:
        ctx.done()
```

PRACTICE 6: UNDERSTAND WHAT THE INSTRUCTION REALLY WANTS

The agent must **reason about the instruction** to understand what is truly expected - not assume or hardcode behaviors.

Key Principle

Every action should be driven by the instruction. Don't assume the task needs:

→ Cleanup (only if asked)

→ Specific file formats (only if specified)

→ Certain outputs (only what's requested)

Implementation

```python
def run(self, ctx: AgentContext):
    # Let LLM analyze what the instruction actually requires
    response = self.llm.ask(
        f"Task: {ctx.instruction}\n\n"
        "Analyze this task carefully:\n"
        "1. What is the expected deliverable?\n"
        "2. What format/location is expected?\n"
        "3. Are there any implicit requirements?\n"
        "4. What should NOT be done (unless asked)?",
        system="Think step by step about what the task truly requires."
    )

    # Agent acts based on LLM's understanding, not assumptions
```

What NOT to Do

```python
# WRONG: Assuming cleanup is always needed
def run(self, ctx):
    # ... do task ...
    self.cleanup()  # NOT REQUESTED!
    ctx.done()

# WRONG: Assuming specific output format
def run(self, ctx):
    # ... do task ...
    self.save_as_json()  # WAS JSON REQUESTED?
    ctx.done()

# RIGHT: Only do what's asked
def run(self, ctx):
    # Let LLM determine exactly what's needed
    # based on the instruction
```

Why This Matters

The benchmark tests whether your agent understands tasks - not whether it follows a hardcoded checklist. If the instruction says "create a file", don't also clean up, format, validate, etc. unless asked.

PRACTICE 7: HANDLE ERRORS GRACEFULLY

Implement robust error handling with retries.

Implementation

PYTHON

```python
def run_with_retry(
    self,
    ctx: AgentContext,
    command: str,
    max_retries: int = 3,
    retry_delay: int = 5
) -> ShellResult:
    """Execute command with retry logic."""

    last_result = None

    for attempt in range(max_retries):
        result = ctx.shell(command)
        last_result = result

        if result.ok:
            return result

        # Check for transient errors worth retrying
        transient_errors = [
            "connection refused",
            "connection reset",
            "timeout",
            "temporary failure",
            "try again",
        ]

        is_transient = any(
            err in result.output.lower()
            for err in transient_errors
        )

        if is_transient and attempt < max_retries - 1:
            ctx.log(f"Transient error, retrying in {retry_delay}s...")
            ctx.shell(f"sleep {retry_delay}")
            continue

        # Non-transient error or out of retries
        break

    return last_result
```

LLM-Assisted Error Recovery

PYTHON

```python
def handle_error(self, ctx: AgentContext, error_output: str) -> str | None:
    """Let LLM suggest error recovery."""

    response = self.llm.ask(
        f"Command failed with output:\n{error_output[-2000:]}\n\n"
        "How should I fix this? Respond with JSON:\n"
        '{"recovery_command": "...", "explanation": "..."}',
        system="Suggest a recovery command or say 'no_fix' if unfixable."
    )

    data = parse_llm_response(response.text)
    if data and data.get("recovery_command") != "no_fix":
        return data.get("recovery_command")

    return None
```

PRACTICE 8: LOG PROGRESS

Make debugging easier with clear logging.

Implementation

```python
def run(self, ctx: AgentContext):
    ctx.log(f"Starting task: {ctx.instruction[:100]}...")

    context = self.explore(ctx)
    ctx.log(f"Explored environment: {len(context)} chars")

    iteration = 0
    while iteration < 100:
        iteration += 1
        ctx.log(f"Iteration {iteration}")

        response = self.llm.chat(messages)
        data = parse_llm_response(response.text)

        if not data:
            ctx.log("Failed to parse LLM response")
            continue

        ctx.log(f"Analysis: {data.get('analysis', '')[:100]}...")
        ctx.log(f"Plan: {data.get('plan', '')[:100]}...")

        for cmd in data.get("commands", []):
            ctx.log(f"$ {cmd['command'][:80]}")
            result = ctx.shell(cmd["command"])
            ctx.log(f"Exit: {result.exit_code}, Output: {len(result.output)} chars")

        if data.get("task_complete"):
            ctx.log("Task marked complete")
            break

    ctx.log(f"Finished after {iteration} iterations")
    ctx.done()
```

PRACTICE 9: MANAGE CONTEXT WINDOW

Keep conversation history bounded.

Implementation

PYTHON

```python
class ContextManager:
    def __init__(self, max_messages: int = 20, max_tokens_estimate: int = 50000):
        self.messages = []
        self.max_messages = max_messages
        self.max_tokens = max_tokens_estimate

    def add(self, role: str, content: str):
        self.messages.append({"role": role, "content": content})
        self._prune()

    def _prune(self):
        # Keep system message + recent messages
        if len(self.messages) <= self.max_messages:
            return

        # Always keep system message if present
        system_msg = None
        if self.messages and self.messages[0]["role"] == "system":
            system_msg = self.messages[0]
            self.messages = self.messages[1:]

        # Keep most recent messages
        self.messages = self.messages[-(self.max_messages - 1):]

        # Restore system message
        if system_msg:
            self.messages.insert(0, system_msg)

    def get_messages(self) -> list:
        return self.messages.copy()

def run(self, ctx: AgentContext):
    context_mgr = ContextManager(max_messages=20)
    context_mgr.add("system", SYSTEM_PROMPT)
    context_mgr.add("user", f"Task: {ctx.instruction}")

    while True:
        response = self.llm.chat(context_mgr.get_messages())
        context_mgr.add("assistant", response.text)

        # ... execute commands ...

        context_mgr.add("user", f"Output:\n{truncated_output}")
```

PRACTICE 10: USE ABSOLUTE PATHS

Always use absolute paths for output files.

Implementation

```python
def run(self, ctx: AgentContext):
    # Get working directory
    pwd = ctx.shell("pwd").stdout.strip()

    # When writing files, use absolute paths
    output_path = f"{pwd}/result.txt"
    ctx.write(output_path, content)

    # Verify with absolute path
    ctx.shell(f"ls -la {output_path}")
```

In LLM Prompts

```python
SYSTEM_PROMPT = """You are a task-solving agent.

IMPORTANT: Always use absolute paths for file operations.
The working directory is: {pwd}

When creating files, use full paths like:
- {pwd}/output.txt
- {pwd}/result.json

Never use relative paths like ./output.txt
"""

def run(self, ctx: AgentContext):
    pwd = ctx.shell("pwd").stdout.strip()
    system = SYSTEM_PROMPT.format(pwd=pwd)

    messages = [{"role": "system", "content": system}, ...]
```

SUMMARY: BEST PRACTICES CHECKLIST

PRACTICE	KEY POINT
Explore First	Gather context before acting
JSON Responses	Structured, parseable output
Truncate Outputs	Prevent context overflow
Double Confirmation	Verify before completing
Verify Outputs	Check files exist
Understand Instruction	**Only do what's asked**
Handle Errors	Retry transient failures
Log Progress	Enable debugging
Manage Context	Bound conversation history
Absolute Paths	Prevent path confusion

Key Principle: The agent must reason about what the instruction truly wants. Don't assume behaviors like cleanup, validation, or formatting unless explicitly requested.

CHAPTER 06

LLM USAGE GUIDE

06 - LLM USAGE GUIDE

This guide covers effective use of Large Language Models in generalist agents.

BASIC LLM INTERACTION

Initialization

PYTHON

```python
from term_sdk import LLM, LLMError, CostLimitExceeded

class MyAgent(Agent):
    def setup(self):
        self.llm = LLM(
            provider="openrouter",
            default_model="anthropic/claude-3.5-sonnet",
            temperature=0.3,  # Lower = more deterministic
            max_tokens=4096
        )
```

Simple Questions

PYTHON

```python
# Single question
response = self.llm.ask("What is the best way to list files in Linux?")
print(response.text)

# With system prompt
response = self.llm.ask(
    "How do I fix this error: Permission denied",
    system="You are a Linux expert. Be concise."
)
```

Conversations

```python
messages = [
    {"role": "system", "content": "You are a helpful assistant."},
    {"role": "user", "content": "What files are in /etc?"},
]

response = self.llm.chat(messages)
print(response.text)

# Continue conversation
messages.append({"role": "assistant", "content": response.text})
messages.append({"role": "user", "content": "Which ones are for networking?"})

response = self.llm.chat(messages)
```

EFFECTIVE SYSTEM PROMPTS

Basic Structure

```python
SYSTEM_PROMPT = """You are a task-solving agent operating in a Linux terminal.

YOUR CAPABILITIES:
- Execute shell commands
- Read and write files
- Analyze output and errors

YOUR CONSTRAINTS:
- Only use standard Linux tools
- Do not access the internet
- Do not modify system files

RESPONSE FORMAT:
Respond with valid JSON:
{
    "thinking": "your reasoning",
    "command": "shell command to run",
    "task_complete": false
}
"""
```

Do's and Don'ts

Do:

→ Be specific about response format

→ List available tools/capabilities

→ Define constraints clearly

→ Give examples of expected output format

Don't:

→ Include task-specific hints

→ Pre-define solutions for task types

→ Give examples of specific task solutions

→ Mention benchmark or evaluation

Format-Focused Prompt

PYTHON

```python
SYSTEM_PROMPT = """You are a terminal agent. Respond ONLY with valid JSON.

Required format:
{
    "analysis": "Brief analysis of current state",
    "plan": "What you will do next",
    "commands": [
        {"keystrokes": "command here", "duration": 1.0}
    ],
    "task_complete": false
}

Rules:
1. "analysis" and "plan" are required strings
2. "commands" is an array (can be empty)
3. "task_complete" defaults to false
4. Use proper JSON escaping
5. No markdown, no explanations outside JSON
"""
```

RESPONSE PARSING

Robust JSON Parser

PYTHON

```python
import json
import re

class ResponseParser:
    def parse(self, text: str) -> dict | None:
        """Parse JSON from LLM response."""

        # Try direct parse first
        try:
            return json.loads(text.strip())
        except json.JSONDecodeError:
            pass

        # Extract JSON from text
        json_str = self._extract_json(text)
        if not json_str:
            return None

        # Try parsing extracted JSON
        try:
            return json.loads(json_str)
        except json.JSONDecodeError:
            pass

        # Try fixing common issues
        fixed = self._fix_json(json_str)
        try:
            return json.loads(fixed)
        except json.JSONDecodeError:
            return None

    def _extract_json(self, text: str) -> str | None:
        """Extract JSON object from text."""
        # Find outermost braces
        start = text.find('{')
        if start == -1:
            return None

        depth = 0
        in_string = False
        escape = False

        for i, char in enumerate(text[start:], start):
            if escape:
                escape = False
                continue
            if char == '\\':
                escape = True
                continue
            if char == '"' and not escape:
                in_string = not in_string
                continue
            if not in_string:
                if char == '{':
                    depth += 1
                elif char == '}':
                    depth -= 1
                    if depth == 0:
                        return text[start:i+1]

        return None

    def _fix_json(self, json_str: str) -> str:
        """Fix common JSON issues."""
        # Remove trailing commas
        json_str = re.sub(r',(\s*[}\]])', r'\1', json_str)
```

```
        # Fix single quotes
        # (This is naive - real impl should be smarter)

        return json_str
```

Using the Parser

```python
def run(self, ctx: AgentContext):
    parser = ResponseParser()

    response = self.llm.chat(messages)
    data = parser.parse(response.text)

    if not data:
        ctx.log("Failed to parse response, asking for retry")
        messages.append({"role": "assistant", "content": response.text})
        messages.append({"role": "user", "content": "Invalid JSON. Please respond with valid JSON only."})
        continue

    # Use parsed data
    commands = data.get("commands", [])
```

ERROR HANDLING

LLM Errors

PYTHON

```python
from term_sdk import LLMError, CostLimitExceeded

def call_llm_safe(self, messages: list, max_retries: int = 3) -> Response | None:
    """Call LLM with error handling."""

    for attempt in range(max_retries):
        try:
            return self.llm.chat(messages)

        except CostLimitExceeded as e:
            # Fatal - can't continue
            self.ctx.log(f"Cost limit exceeded: ${e.used:.2f}/${e.limit:.2f}")
            return None

        except LLMError as e:
            self.ctx.log(f"LLM error ({e.code}): {e.message}")

            if e.code == "rate_limit":
                # Wait and retry
                wait_time = 30 * (attempt + 1)
                self.ctx.log(f"Rate limited, waiting {wait_time}s...")
                time.sleep(wait_time)
                continue

            elif e.code == "context_length":
                # Reduce context and retry
                self.ctx.log("Context too long, truncating...")
                messages = self._truncate_messages(messages)
                continue

            elif e.code in ("server_error", "service_unavailable"):
                # Transient, retry
                time.sleep(5 * (attempt + 1))
                continue

            else:
                # Unknown error
                if attempt == max_retries - 1:
                    return None

    return None
```

Parse Errors

```python
def get_valid_response(self, ctx, messages: list, max_attempts: int = 3) -> dict | None:
    """Get a valid parsed response from LLM."""

    for attempt in range(max_attempts):
        response = self.call_llm_safe(messages)
        if not response:
            return None

        data = self.parser.parse(response.text)
        if data:
            return data

        # Ask for valid JSON
        ctx.log(f"Parse failed (attempt {attempt + 1})")
        messages.append({"role": "assistant", "content": response.text})
        messages.append({
            "role": "user",
            "content": "Your response was not valid JSON. Please respond with ONLY valid JSON, no other text."
        })

    return None
```

CONTEXT MANAGEMENT

Token Estimation

```python
def estimate_tokens(text: str) -> int:
    """Rough token estimate (4 chars per token)."""
    return len(text) // 4

class ContextManager:
    def __init__(self, max_tokens: int = 50000):
        self.max_tokens = max_tokens
        self.messages = []

    def add_message(self, role: str, content: str):
        self.messages.append({"role": role, "content": content})
        self._enforce_limit()

    def _enforce_limit(self):
        total = sum(estimate_tokens(m["content"]) for m in self.messages)

        while total > self.max_tokens and len(self.messages) > 2:
            # Keep first (system) and remove oldest user/assistant
            removed = self.messages.pop(1)
            total -= estimate_tokens(removed["content"])

    def get_messages(self) -> list:
        return self.messages.copy()
```

Sliding Window

PYTHON

```python
def run(self, ctx: AgentContext):
    # Keep only last N exchanges
    MAX_HISTORY = 10

    system_msg = {"role": "system", "content": SYSTEM_PROMPT}
    history = []

    while True:
        # Build messages with sliding window
        messages = [system_msg] + history[-MAX_HISTORY * 2:]

        response = self.llm.chat(messages)
        history.append({"role": "assistant", "content": response.text})

        # ... execute ...

        history.append({"role": "user", "content": output})
```

FUNCTION CALLING

Defining Tools

PYTHON

```python
from term_sdk import Tool

TOOLS = [
    Tool(
        name="execute_command",
        description="Run a shell command and return the output",
        parameters={
            "type": "object",
            "properties": {
                "command": {
                    "type": "string",
                    "description": "The shell command to execute"
                },
                "timeout": {
                    "type": "integer",
                    "description": "Timeout in seconds (default 30)",
                    "default": 30
                }
            },
            "required": ["command"]
        }
    ),
    Tool(
        name="read_file",
        description="Read the contents of a file",
        parameters={
            "type": "object",
            "properties": {
                "path": {
                    "type": "string",
                    "description": "Path to the file to read"
                }
            },
            "required": ["path"]
        }
    ),
    Tool(
        name="write_file",
        description="Write content to a file",
        parameters={
            "type": "object",
            "properties": {
                "path": {
                    "type": "string",
                    "description": "Path to the file to write"
                },
                "content": {
                    "type": "string",
                    "description": "Content to write to the file"
                }
            },
            "required": ["path", "content"]
        }
    ),
    Tool(
        name="task_complete",
        description="Mark the task as complete",
        parameters={
            "type": "object",
            "properties": {
                "summary": {
                    "type": "string",
                    "description": "Brief summary of what was accomplished"
                }
            },
            "required": ["summary"]
        }
```

```
    )
]
```

Implementing Tool Handlers

```python
class ToolHandler:
    def __init__(self, ctx: AgentContext):
        self.ctx = ctx
        self.completed = False

    def execute_command(self, command: str, timeout: int = 30) -> str:
        result = self.ctx.shell(command, timeout=timeout)
        output = result.output[-5000:]  # Truncate

        if result.timed_out:
            return f"[TIMEOUT after {timeout}s]\n{output}"

        status = "success" if result.ok else f"failed (exit {result.exit_code})"
        return f"[{status}]\n{output}"

    def read_file(self, path: str) -> str:
        result = self.ctx.read(path)
        if result.ok:
            return result.stdout[:10000]  # Truncate
        return f"Error reading file: {result.stderr}"

    def write_file(self, path: str, content: str) -> str:
        result = self.ctx.write(path, content)
        if result.ok:
            return f"Successfully wrote {len(content)} bytes to {path}"
        return f"Error writing file: {result.stderr}"

    def task_complete(self, summary: str) -> str:
        self.completed = True
        return f"Task marked complete: {summary}"
```

Using Function Calling

PYTHON

```python
def run(self, ctx: AgentContext):
    handler = ToolHandler(ctx)

    # Register handlers
    self.llm.register_function("execute_command", handler.execute_command)
    self.llm.register_function("read_file", handler.read_file)
    self.llm.register_function("write_file", handler.write_file)
    self.llm.register_function("task_complete", handler.task_complete)

    messages = [
        {"role": "system", "content": "You are a task-solving agent with tools."},
        {"role": "user", "content": f"Task: {ctx.instruction}"}
    ]

    # Let LLM call functions automatically
    response = self.llm.chat_with_functions(
        messages,
        TOOLS,
        max_iterations=50
    )

    if handler.completed:
        ctx.done()
```

MODEL SELECTION

Supported Models

All foundation models are supported as long as they meet these criteria:

REQUIREMENT	ALLOWED	FORBIDDEN
Model origin	Official provider releases	Community fine-tunes
Training	Base/instruct versions	Task-specific fine-tuning
Weights	Unmodified	Custom merged weights

Why No Community Fine-Tunes?

Community fine-tuned models are **forbidden** because they may:

→ Be trained on benchmark data (data contamination)

→ Have task-specific optimizations that constitute hardcoding

→ Produce artificially inflated scores through overfitting

Model Selection Strategy

```python
def setup(self):
    # Any official foundation model works
    # Examples: claude-3.5-sonnet, gpt-4o, deepseek-v3, llama-3, etc.
    self.llm = LLM(
        provider="openrouter",  # or any supported provider
        default_model="anthropic/claude-3.5-sonnet",
        temperature=0.3
    )
```

Multi-Model Strategy

You can use different models for different purposes:

```python
def setup(self):
    # Strong model for complex reasoning
    self.reasoning_model = "anthropic/claude-3.5-sonnet"
    # Fast model for simple operations
    self.fast_model = "anthropic/claude-3-haiku"

def run(self, ctx: AgentContext):
    # Use strong model for planning
    plan = self.llm.ask(
        f"Task: {ctx.instruction}\nCreate a plan.",
        model=self.reasoning_model
    )

    # Use fast model for parsing
    parsed = self.llm.ask(
        f"Extract commands from:\n{plan.text}",
        model=self.fast_model
    )
```

PROMPT CACHING

Prompt caching significantly reduces costs and latency by reusing previously processed prompts. The Term SDK supports caching via the `cache=True` parameter.

Enabling Caching in Term SDK

PYTHON

```python
from term_sdk import LLM

class MyAgent(Agent):
    def setup(self):
        self.llm = LLM(
            provider="openrouter",
            default_model="anthropic/claude-3.5-sonnet",
            cache=True  # Enable prompt caching
        )
```

How Caching Works by Provider

PROVIDER	CACHING	CONFIGURATION
OpenAI	Automatic	No config needed, min 1024 tokens
Anthropic	Manual	Requires `cache_control` breakpoints
DeepSeek	Automatic	No config needed
Google Gemini	Automatic	No config needed, min 4096 tokens
Groq	Automatic	No config needed

Anthropic Cache Control (Important!)

Anthropic requires explicit `cache_control` breakpoints. This is critical for cost savings:

Pricing:

→ **Cache writes**: 1.25x input price (slightly more expensive)

→ **Cache reads**: 0.1x input price (90% savings!)

TTL Options:

→ Default: 5 minutes

→ Extended: 1 hour with `"ttl": "1h"`

Anthropic Caching Example

```python
# Structure messages with cache_control for large content
messages = [
    {
        "role": "system",
        "content": [
            {
                "type": "text",
                "text": "You are a task-solving agent."
            },
            {
                "type": "text",
                "text": LARGE_SYSTEM_PROMPT,  # Cache this!
                "cache_control": {
                    "type": "ephemeral",
                    "ttl": "1h"  # Optional: extend to 1 hour
                }
            }
        ]
    },
    {
        "role": "user",
        "content": f"Task: {ctx.instruction}"
    }
]

response = self.llm.chat(messages)
```

What to Cache

Good candidates for caching:

→ Large system prompts

→ Tool definitions

→ Reference documentation

→ Few-shot examples

→ Context that stays constant across requests

Don't cache:

→ Dynamic user input

→ Changing context

→ Small prompts (under 1024 tokens)

Cache Placement Strategy

```python
# Put static content FIRST, dynamic content LAST
messages = [
    {
        "role": "system",
        "content": [
            # Static: Cache this large prompt
            {
                "type": "text",
                "text": STATIC_SYSTEM_PROMPT,
                "cache_control": {"type": "ephemeral"}
            }
        ]
    },
    # Dynamic: User instruction (changes each task)
    {"role": "user", "content": ctx.instruction},
    # Dynamic: Previous outputs (change each iteration)
    {"role": "assistant", "content": last_response},
    {"role": "user", "content": command_output}
]
```

Inspecting Cache Usage

```python
response = self.llm.chat(messages, usage=True)

# Check cache statistics
if response.usage:
    cached_tokens = response.usage.get("cached_tokens", 0)
    cache_discount = response.usage.get("cache_discount", 0)
    ctx.log(f"Cached: {cached_tokens} tokens, saved: ${cache_discount:.4f}")
```

Cost Optimization Tips

1. **Keep static content first** – Cache hits require matching prefixes
2. **Use 1-hour TTL for long sessions** – Avoids repeated cache writes
3. **Batch related requests** – Maximize cache hits within TTL window
4. **Monitor cache_discount** – Negative = cache write, positive = savings

BEST PRACTICES SUMMARY

PRACTICE	DESCRIPTION
Structured output	Always request JSON
Robust parsing	Handle malformed responses
Error handling	Retry transient failures
Context management	Bound history size
Token awareness	Truncate long outputs
Clear prompts	Specific format requirements
Tool definitions	Well-documented parameters
Prompt caching	Enable `cache=True`, use cache_control for Anthropic

CHAPTER 07

TOOL DESIGN

07 - TOOL DESIGN

This guide covers best practices for designing tools that LLM agents can use effectively.

PRINCIPLES OF GOOD TOOL DESIGN

1. Single Responsibility

Each tool should do one thing well.

```python
# GOOD: Single purpose tools
Tool(
    name="read_file",
    description="Read the contents of a file",
    parameters={"path": {"type": "string"}}
)

Tool(
    name="write_file",
    description="Write content to a file",
    parameters={"path": {"type": "string"}, "content": {"type": "string"}}
)

# BAD: Multi-purpose tool
Tool(
    name="file_operation",
    description="Read, write, or delete a file",
    parameters={
        "operation": {"type": "string", "enum": ["read", "write", "delete"]},
        "path": {"type": "string"},
        "content": {"type": "string"}  # Only for write
    }
)
```

2. Clear Descriptions

Write descriptions that help the LLM choose the right tool.

```python
# GOOD: Clear, specific description
Tool(
    name="execute_command",
    description="Execute a shell command in the terminal and return stdout/stderr. Use for running programs, listing files, or any terminal operation.",
    parameters={
        "command": {
            "type": "string",
            "description": "The shell command to execute (e.g., 'ls -la', 'python script.py')"
        }
    }
)

# BAD: Vague description
Tool(
    name="run",
    description="Run something",
    parameters={"cmd": {"type": "string"}}
)
```

3. Descriptive Parameters

Every parameter should have a clear description.

```python
Tool(
    name="search_files",
    description="Search for files matching a pattern",
    parameters={
        "type": "object",
        "properties": {
            "pattern": {
                "type": "string",
                "description": "Glob pattern to match (e.g., '*.py', 'src/**/*.ts')"
            },
            "directory": {
                "type": "string",
                "description": "Directory to search in (default: current directory)",
                "default": "."
            },
            "max_results": {
                "type": "integer",
                "description": "Maximum number of results to return (default: 100)",
                "default": 100
            }
        },
        "required": ["pattern"]
    }
)
```

CORE TOOL SET

Every generalist agent should have these essential tools:

Execute Command

```python
Tool(
    name="execute_command",
    description="Execute a shell command and return the output. Use for running programs, installing packages, building
projects, or any terminal operation.",
    parameters={
        "type": "object",
        "properties": {
            "command": {
                "type": "string",
                "description": "The shell command to execute"
            },
            "timeout": {
                "type": "integer",
                "description": "Maximum seconds to wait (default: 30, max: 300)",
                "default": 30
            },
            "working_directory": {
                "type": "string",
                "description": "Directory to run the command in (default: current)",
                "default": "."
            }
        },
        "required": ["command"]
    }
)
```

Read File

```python
Tool(
    name="read_file",
    description="Read the contents of a file. Returns the file content as text.",
    parameters={
        "type": "object",
        "properties": {
            "path": {
                "type": "string",
                "description": "Absolute or relative path to the file"
            },
            "max_lines": {
                "type": "integer",
                "description": "Maximum lines to read (default: all)",
                "default": -1
            },
            "offset": {
                "type": "integer",
                "description": "Line number to start from (0-indexed, default: 0)",
                "default": 0
            }
        },
        "required": ["path"]
    }
)
```

Write File

```python
Tool(
    name="write_file",
    description="Write content to a file. Creates the file if it doesn't exist, overwrites if it does. Creates parent
directories if needed.",
    parameters={
        "type": "object",
        "properties": {
            "path": {
                "type": "string",
                "description": "Absolute or relative path to the file"
            },
            "content": {
                "type": "string",
                "description": "Content to write to the file"
            }
        },
        "required": ["path", "content"]
    }
)
```

List Directory

```python
Tool(
    name="list_directory",
    description="List files and directories in a path. Returns file names, sizes, and modification times.",
    parameters={
        "type": "object",
        "properties": {
            "path": {
                "type": "string",
                "description": "Directory path to list (default: current directory)",
                "default": "."
            },
            "show_hidden": {
                "type": "boolean",
                "description": "Include hidden files (starting with .)",
                "default": False
            },
            "recursive": {
                "type": "boolean",
                "description": "List recursively",
                "default": False
            }
        },
        "required": []
    }
)
```

Task Complete

PYTHON

```python
Tool(
    name="task_complete",
    description="Mark the task as complete. Call this when you have finished the task successfully.",
    parameters={
        "type": "object",
        "properties": {
            "summary": {
                "type": "string",
                "description": "Brief summary of what was accomplished"
            },
            "output_files": {
                "type": "array",
                "items": {"type": "string"},
                "description": "List of output files created"
            }
        },
        "required": ["summary"]
    }
)
```

IMPLEMENTING TOOL HANDLERS

Basic Handler Pattern

PYTHON

```python
class ToolHandler:
    def __init__(self, ctx: AgentContext):
        self.ctx = ctx

    def execute_command(self, command: str, timeout: int = 30, working_directory: str = ".") -> str:
        """Execute a shell command."""
        # Input validation
        timeout = min(max(timeout, 1), 300)  # Clamp 1-300

        # Execute
        result = self.ctx.shell(command, timeout=timeout, cwd=working_directory)

        # Format output
        output = result.output[-10000:]  # Truncate

        if result.timed_out:
            return f"TIMEOUT after {timeout}s\nPartial output:\n{output}"

        status = "SUCCESS" if result.ok else f"FAILED (exit code {result.exit_code})"
        return f"{status}\n{output}"

    def read_file(self, path: str, max_lines: int = -1, offset: int = 0) -> str:
        """Read a file."""
        result = self.ctx.read(path)

        if result.failed:
            return f"ERROR: {result.stderr}"

        lines = result.stdout.split('\n')

        # Apply offset and limit
        if offset > 0:
            lines = lines[offset:]
        if max_lines > 0:
            lines = lines[:max_lines]

        content = '\n'.join(lines)

        # Truncate if too long
        if len(content) > 50000:
            content = content[:50000] + "\n[... truncated ...]"

        return content

    def write_file(self, path: str, content: str) -> str:
        """Write to a file."""
        # Ensure parent directory exists
        import os
        parent = os.path.dirname(path)
        if parent:
            self.ctx.shell(f"mkdir -p '{parent}'")

        result = self.ctx.write(path, content)

        if result.ok:
            return f"Successfully wrote {len(content)} bytes to {path}"
        else:
            return f"ERROR: {result.stderr}"
```

Registration

PYTHON

```python
def setup(self):
    self.llm = LLM(default_model="anthropic/claude-3.5-sonnet")
    self.handler = ToolHandler(self.ctx)

    # Register all handlers
    self.llm.register_function("execute_command", self.handler.execute_command)
    self.llm.register_function("read_file", self.handler.read_file)
    self.llm.register_function("write_file", self.handler.write_file)
    self.llm.register_function("list_directory", self.handler.list_directory)
    self.llm.register_function("task_complete", self.handler.task_complete)
```

ADVANCED TOOL PATTERNS

Tool with Validation

PYTHON

```python
def execute_command(self, command: str, timeout: int = 30) -> str:
    """Execute command with validation."""

    # Validate command isn't empty
    if not command or not command.strip():
        return "ERROR: Empty command"

    # Validate timeout
    if not isinstance(timeout, int) or timeout < 1:
        timeout = 30
    timeout = min(timeout, 300)

    # Check for obviously dangerous commands
    dangerous = ["rm -rf /", "mkfs", "> /dev/sda"]
    if any(d in command for d in dangerous):
        return "ERROR: Potentially dangerous command blocked"

    # Execute
    result = self.ctx.shell(command, timeout=timeout)

    # Format response
    return self._format_result(result)
```

Tool with State

```python
class StatefulToolHandler:
    def __init__(self, ctx: AgentContext):
        self.ctx = ctx
        self.command_history = []
        self.files_created = []

    def execute_command(self, command: str, **kwargs) -> str:
        result = self.ctx.shell(command, **kwargs)

        # Track history
        self.command_history.append({
            "command": command,
            "exit_code": result.exit_code,
            "timestamp": time.time()
        })

        return self._format_result(result)

    def write_file(self, path: str, content: str) -> str:
        result = self.ctx.write(path, content)

        if result.ok:
            self.files_created.append(path)

        return self._format_result(result)

    def get_summary(self) -> dict:
        return {
            "commands_run": len(self.command_history),
            "files_created": self.files_created,
            "last_command": self.command_history[-1] if self.command_history else None
        }
```

Composite Tool

PYTHON

```python
Tool(
    name="edit_file",
    description="Make a specific edit to a file. Replaces old_text with new_text.",
    parameters={
        "type": "object",
        "properties": {
            "path": {
                "type": "string",
                "description": "Path to the file to edit"
            },
            "old_text": {
                "type": "string",
                "description": "Exact text to find and replace"
            },
            "new_text": {
                "type": "string",
                "description": "Text to replace with"
            }
        },
        "required": ["path", "old_text", "new_text"]
    }
)

def edit_file(self, path: str, old_text: str, new_text: str) -> str:
    """Edit a file by replacing text."""

    # Read current content
    read_result = self.ctx.read(path)
    if read_result.failed:
        return f"ERROR: Could not read file: {read_result.stderr}"

    content = read_result.stdout

    # Check if old_text exists
    if old_text not in content:
        return f"ERROR: Could not find the specified text in {path}"

    # Check for multiple matches
    count = content.count(old_text)
    if count > 1:
        return f"ERROR: Found {count} matches. Please provide more specific text."

    # Make replacement
    new_content = content.replace(old_text, new_text, 1)

    # Write back
    write_result = self.ctx.write(path, new_content)
    if write_result.failed:
        return f"ERROR: Could not write file: {write_result.stderr}"

    return f"Successfully edited {path}"
```

TOOL RESPONSE FORMATTING

Consistent Format

```python
class ToolResponse:
    @staticmethod
    def success(message: str, data: dict = None) -> str:
        response = f"SUCCESS: {message}"
        if data:
            response += f"\n{json.dumps(data, indent=2)}"
        return response

    @staticmethod
    def error(message: str, details: str = None) -> str:
        response = f"ERROR: {message}"
        if details:
            response += f"\nDetails: {details}"
        return response

    @staticmethod
    def output(content: str, truncated: bool = False) -> str:
        response = content
        if truncated:
            response += "\n[... output truncated ...]"
        return response
```

Usage

```python
def read_file(self, path: str) -> str:
    result = self.ctx.read(path)

    if result.failed:
        return ToolResponse.error(f"Could not read {path}", result.stderr)

    content = result.stdout
    truncated = False

    if len(content) > 50000:
        content = content[:50000]
        truncated = True

    return ToolResponse.output(content, truncated)
```

COMMON MISTAKES

Mistake 1: No Input Validation

```python
# BAD
def execute_command(self, command: str) -> str:
    return self.ctx.shell(command).output

# GOOD
def execute_command(self, command: str) -> str:
    if not command or not isinstance(command, str):
        return "ERROR: Invalid command"

    command = command.strip()
    if not command:
        return "ERROR: Empty command"

    return self.ctx.shell(command).output
```

Mistake 2: Unbounded Output

```python
# BAD
def read_file(self, path: str) -> str:
    return self.ctx.read(path).stdout  # Could be huge!

# GOOD
def read_file(self, path: str) -> str:
    content = self.ctx.read(path).stdout
    if len(content) > 50000:
        return content[:50000] + "\n[truncated]"
    return content
```

Mistake 3: Missing Error Handling

```python
# BAD
def write_file(self, path: str, content: str) -> str:
    self.ctx.write(path, content)
    return "Done"

# GOOD
def write_file(self, path: str, content: str) -> str:
    result = self.ctx.write(path, content)
    if result.failed:
        return f"ERROR: {result.stderr}"
    return f"Wrote {len(content)} bytes to {path}"
```

SUMMARY

PRINCIPLE	DESCRIPTION
Single Responsibility	One tool = one purpose
Clear Descriptions	Help LLM choose correctly
Input Validation	Check all parameters
Output Truncation	Bound response size
Error Handling	Return useful error messages
Consistent Format	Predictable response structure

ERROR HANDLING

08 - ERROR HANDLING

This guide covers strategies for handling errors gracefully in generalist agents.

ERROR CATEGORIES

1. LLM Errors

Errors from the language model API:

ERROR	CAUSE	RECOVERY
`rate_limit`	Too many requests	Wait and retry with backoff
`context_length`	Context too long	Truncate history and retry
`server_error`	Provider issue	Wait and retry
`cost_limit`	Budget exceeded	Stop gracefully
`invalid_response`	Malformed output	Ask for retry

2. Command Errors

Errors from shell execution:

ERROR	CAUSE	RECOVERY
Non-zero exit	Command failed	Analyze and fix
Timeout	Command hung	Increase timeout or kill
Permission denied	Access issue	Try sudo or fix permissions
Not found	Missing command/file	Install or create

3. Parse Errors

Errors parsing LLM responses:

ERROR	CAUSE	RECOVERY
Invalid JSON	Malformed response	Ask for valid JSON
Missing fields	Incomplete response	Ask for complete response
Wrong type	Unexpected data type	Provide format example

LLM ERROR HANDLING

Comprehensive Handler

PYTHON

```python
import time
from term_sdk import LLM, LLMError, CostLimitExceeded

class RobustLLMClient:
    def __init__(self, ctx: AgentContext):
        self.ctx = ctx
        self.llm = LLM(default_model="anthropic/claude-3.5-sonnet")
        self.max_retries = 3
        self.base_delay = 5

    def chat(self, messages: list) -> Response | None:
        """Chat with retry logic."""

        for attempt in range(self.max_retries):
            try:
                return self.llm.chat(messages)

            except CostLimitExceeded as e:
                self.ctx.log(f"FATAL: Cost limit exceeded (${e.used:.2f}/${e.limit:.2f})")
                return None  # Can't recover

            except LLMError as e:
                self.ctx.log(f"LLM error (attempt {attempt + 1}): {e.code} - {e.message}")

                if not self._should_retry(e, attempt):
                    return None

                delay = self._get_delay(e, attempt)
                self.ctx.log(f"Retrying in {delay}s...")
                time.sleep(delay)

                # Apply recovery action
                messages = self._apply_recovery(e, messages)

        self.ctx.log("Max retries exceeded")
        return None

    def _should_retry(self, error: LLMError, attempt: int) -> bool:
        """Determine if we should retry this error."""
        retryable = {"rate_limit", "server_error", "service_unavailable", "context_length"}
        return error.code in retryable and attempt < self.max_retries - 1

    def _get_delay(self, error: LLMError, attempt: int) -> int:
        """Calculate retry delay with exponential backoff."""
        if error.code == "rate_limit":
            return min(60, self.base_delay * (2 ** attempt) * 3)  # Longer for rate limits
        return self.base_delay * (2 ** attempt)

    def _apply_recovery(self, error: LLMError, messages: list) -> list:
        """Apply recovery action for the error type."""
        if error.code == "context_length":
            return self._truncate_messages(messages)
        return messages

    def _truncate_messages(self, messages: list) -> list:
        """Truncate messages to reduce context."""
        if len(messages) <= 3:
            # Can't truncate further, try truncating content
            return [
                {**m, "content": m["content"][:5000]}
                for m in messages
            ]

        # Keep system + first user + last N messages
        system = messages[0] if messages[0]["role"] == "system" else None
        result = messages[-6:]  # Keep last 6
```

```
        if system and result[0] != system:
            result.insert(0, system)

        return result
```

COMMAND ERROR HANDLING

Shell Execution with Recovery

PYTHON

```python
class CommandExecutor:
    def __init__(self, ctx: AgentContext, llm: RobustLLMClient):
        self.ctx = ctx
        self.llm = llm

    def execute(
        self,
        command: str,
        timeout: int = 30,
        max_retries: int = 2,
        allow_llm_recovery: bool = True
    ) -> ShellResult:
        """Execute command with error recovery."""

        for attempt in range(max_retries + 1):
            result = self.ctx.shell(command, timeout=timeout)

            if result.ok:
                return result

            self.ctx.log(f"Command failed (attempt {attempt + 1}): exit {result.exit_code}")

            # Check for recoverable errors
            recovery = self._try_recovery(result, attempt, allow_llm_recovery)

            if recovery is None:
                # No recovery possible
                break

            if recovery.get("retry_same"):
                # Just retry the same command
                time.sleep(recovery.get("delay", 1))
                continue

            if recovery.get("new_command"):
                # Try a different command
                command = recovery["new_command"]
                continue

        return result

    def _try_recovery(self, result: ShellResult, attempt: int, allow_llm: bool) -> dict | None:
        """Attempt to recover from command failure."""

        error_lower = result.output.lower()

        # Transient network errors - retry
        if any(x in error_lower for x in ["connection refused", "connection reset", "timeout"]):
            return {"retry_same": True, "delay": 5 * (attempt + 1)}

        # Permission denied - try sudo
        if "permission denied" in error_lower:
            if not result.command.startswith("sudo "):
                return {"new_command": f"sudo {result.command}"}

        # Command not found - try installing
        if "command not found" in error_lower or "not found" in error_lower:
            cmd_name = self._extract_command_name(result.command)
            if cmd_name:
                install_result = self._try_install(cmd_name)
                if install_result.ok:
                    return {"retry_same": True, "delay": 1}

        # Let LLM suggest recovery
        if allow_llm and attempt < 2:
            return self._llm_recovery(result)
```

```python
        return None

    def _extract_command_name(self, command: str) -> str | None:
        """Extract the main command name."""
        parts = command.strip().split()
        if parts:
            return parts[0]
        return None

    def _try_install(self, cmd_name: str) -> ShellResult:
        """Try to install a missing command."""
        # Common package managers
        installers = [
            f"apt-get install -y {cmd_name}",
            f"yum install -y {cmd_name}",
            f"pip install {cmd_name}",
            f"npm install -g {cmd_name}",
        ]

        for installer in installers:
            result = self.ctx.shell(installer, timeout=60)
            if result.ok:
                return result

        return result  # Return last attempt

    def _llm_recovery(self, result: ShellResult) -> dict | None:
        """Ask LLM for recovery suggestion."""
        response = self.llm.chat([
            {"role": "system", "content": "You are a Linux troubleshooting assistant."},
            {"role": "user", "content": f"""Command failed:
$ {result.command}
Exit code: {result.exit_code}
Output: {result.output[-2000:]}

Suggest ONE recovery command, or say "no_recovery" if not fixable.
Respond with JSON: {{"recovery_command": "..."}} or {{"no_recovery": true}}"""}
        ])

        if not response:
            return None

        data = parse_json(response.text)
        if data and data.get("recovery_command"):
            return {"new_command": data["recovery_command"]}

        return None
```

PARSE ERROR HANDLING

Robust JSON Parsing

PYTHON

```python
class ResponseParser:
    def __init__(self, ctx: AgentContext, llm: RobustLLMClient):
        self.ctx = ctx
        self.llm = llm

    def parse(self, response_text: str, required_fields: list = None) -> dict | None:
        """Parse LLM response with error handling."""

        # Try standard parsing
        data = self._try_parse(response_text)

        if data:
            # Validate required fields
            if required_fields:
                missing = [f for f in required_fields if f not in data]
                if missing:
                    self.ctx.log(f"Missing required fields: {missing}")
                    return None
            return data

        return None

    def _try_parse(self, text: str) -> dict | None:
        """Try various parsing strategies."""

        # Strategy 1: Direct parse
        try:
            return json.loads(text.strip())
        except json.JSONDecodeError:
            pass

        # Strategy 2: Extract JSON from markdown code blocks
        # Pattern: triple backticks, optional "json", content, triple backticks
        code_block_pattern = r'`{3}(?:json)?\s*([\s\S]*?)`{3}'
        json_match = re.search(code_block_pattern, text)
        if json_match:
            try:
                return json.loads(json_match.group(1).strip())
            except json.JSONDecodeError:
                pass

        # Strategy 3: Extract JSON object
        json_match = re.search(r'\{[\s\S]*\}', text)
        if json_match:
            try:
                return json.loads(json_match.group())
            except json.JSONDecodeError:
                pass

        # Strategy 4: Fix common issues
        fixed = self._fix_json(text)
        try:
            return json.loads(fixed)
        except json.JSONDecodeError:
            pass

        return None

    def _fix_json(self, text: str) -> str:
        """Attempt to fix common JSON issues."""
        # Extract potential JSON
        match = re.search(r'\{[\s\S]*\}', text)
        if not match:
            return text

        json_str = match.group()
```

```python
        # Fix trailing commas
        json_str = re.sub(r',\s*}', '}', json_str)
        json_str = re.sub(r',\s*]', ']', json_str)

        # Fix missing quotes on keys
        json_str = re.sub(r'(\{|,)\s*([a-zA-Z_][a-zA-Z0-9_]*)\s*:', r'\1"\2":', json_str)

        return json_str

    def parse_with_retry(
        self,
        messages: list,
        required_fields: list = None,
        max_attempts: int = 3
    ) -> tuple[dict | None, list]:
        """Parse response, retrying with LLM if needed."""

        for attempt in range(max_attempts):
            response = self.llm.chat(messages)
            if not response:
                return None, messages

            data = self.parse(response.text, required_fields)

            if data:
                messages.append({"role": "assistant", "content": response.text})
                return data, messages

            # Ask for valid JSON
            self.ctx.log(f"Parse failed (attempt {attempt + 1}), requesting valid JSON")

            messages.append({"role": "assistant", "content": response.text})
            messages.append({
                "role": "user",
                "content": f"Invalid JSON response. Please respond with ONLY valid JSON containing these fields:
{required_fields}"
            })

        return None, messages
```

GRACEFUL DEGRADATION

Fallback Strategies

PYTHON

```python
class FallbackExecutor:
    def __init__(self, ctx: AgentContext):
        self.ctx = ctx

    def execute_with_fallbacks(self, primary: str, fallbacks: list[str]) -> ShellResult:
        """Try primary command, then fallbacks if it fails."""

        result = self.ctx.shell(primary)
        if result.ok:
            return result

        self.ctx.log(f"Primary command failed, trying fallbacks")

        for i, fallback in enumerate(fallbacks):
            self.ctx.log(f"Trying fallback {i + 1}: {fallback[:50]}...")
            result = self.ctx.shell(fallback)
            if result.ok:
                return result

        return result  # Return last result

    def find_tool(self, tool_name: str, alternatives: list[str]) -> str | None:
        """Find an available tool from alternatives."""

        for tool in [tool_name] + alternatives:
            result = self.ctx.shell(f"which {tool}")
            if result.ok:
                return tool

        return None

# Usage
executor = FallbackExecutor(ctx)

# Try python3, then python, then python2
python = executor.find_tool("python3", ["python", "python2"])
if python:
    ctx.shell(f"{python} script.py")

# Try primary approach with fallbacks
result = executor.execute_with_fallbacks(
    "npm run build",
    ["yarn build", "pnpm build", "node build.js"]
)
```

ERROR LOGGING

Structured Error Logging

PYTHON

```python
class ErrorLogger:
    def __init__(self, ctx: AgentContext):
        self.ctx = ctx
        self.errors = []

    def log_error(self, category: str, message: str, details: dict = None):
        """Log an error with context."""
        error = {
            "category": category,
            "message": message,
            "details": details or {},
            "step": self.ctx.step,
            "timestamp": time.time()
        }

        self.errors.append(error)
        self.ctx.log(f"[{category}] {message}")

        if details:
            for key, value in details.items():
                self.ctx.log(f"  {key}: {str(value)[:200]}")

    def get_summary(self) -> str:
        """Get error summary for debugging."""
        if not self.errors:
            return "No errors recorded"

        summary = [f"Total errors: {len(self.errors)}"]

        by_category = {}
        for e in self.errors:
            cat = e["category"]
            by_category[cat] = by_category.get(cat, 0) + 1

        for cat, count in by_category.items():
            summary.append(f"  {cat}: {count}")

        return "\n".join(summary)
```

COMPLETE ERROR HANDLING EXAMPLE

PYTHON

```python
class ResilientAgent(Agent):
    def setup(self):
        self.llm_client = RobustLLMClient(self.ctx)
        self.executor = CommandExecutor(self.ctx, self.llm_client)
        self.parser = ResponseParser(self.ctx, self.llm_client)
        self.errors = ErrorLogger(self.ctx)

    def run(self, ctx: AgentContext):
        self.ctx = ctx
        messages = [
            {"role": "system", "content": SYSTEM_PROMPT},
            {"role": "user", "content": f"Task: {ctx.instruction}"}
        ]

        max_iterations = 50
        for iteration in range(max_iterations):
            ctx.log(f"Iteration {iteration + 1}")

            # Get LLM response with retry
            data, messages = self.parser.parse_with_retry(
                messages,
                required_fields=["commands"]
            )

            if not data:
                self.errors.log_error("LLM", "Failed to get valid response")
                continue

            if data.get("task_complete"):
                ctx.log("Task complete")
                break

            # Execute commands with recovery
            for cmd in data.get("commands", []):
                result = self.executor.execute(
                    cmd.get("command", ""),
                    timeout=cmd.get("timeout", 30)
                )

                if result.failed:
                    self.errors.log_error(
                        "COMMAND",
                        f"Command failed: {cmd.get('command', '')[:50]}",
                        {"exit_code": result.exit_code, "output": result.output[-500:]}
                    )

                # Add output to conversation
                messages.append({
                    "role": "user",
                    "content": f"Output:\n{result.output[-3000:]}"
                })

        ctx.log(self.errors.get_summary())
        ctx.done()
```

SUMMARY

ERROR TYPE	STRATEGY
Rate limit	Exponential backoff
Context overflow	Truncate history
Parse failure	Retry with format hint
Command failure	LLM-assisted recovery
Missing tool	Try alternatives
Permission denied	Try sudo
Transient errors	Simple retry
Fatal errors	Log and stop gracefully

CHAPTER 09

TESTING YOUR AGENT

09 - TESTING YOUR AGENT

This guide covers how to test your agent locally before submitting to the benchmark.

WHY TEST LOCALLY?

1. **Faster feedback** - No waiting for benchmark queue
2. **Easier debugging** - Full access to logs and state
3. **Cost savings** - Fewer wasted LLM calls
4. **Iteration speed** - Quick fix-test cycles

BASIC TESTING SETUP

Minimal Test Script

PYTHON

```python
#!/usr/bin/env python3
"""test_agent.py - Basic agent testing"""

from my_agent import MyAgent

def test_basic():
    """Test agent initialization and basic functionality."""
    agent = MyAgent()

    # Test setup
    agent.setup()
    print("Setup: OK")

    # Test cleanup
    agent.cleanup()
    print("Cleanup: OK")

def test_simple_task():
    """Test a simple task."""
    from term_sdk import AgentContext

    agent = MyAgent()
    agent.setup()

    # Create mock context
    ctx = AgentContext(instruction="List all files in the current directory")

    # Run agent
    agent.run(ctx)

    print(f"Steps: {ctx.step}")
    print(f"Done: {ctx.is_done}")

    agent.cleanup()

if __name__ == "__main__":
    test_basic()
    test_simple_task()
```

Running Tests

BASH

```bash
# Run basic test
python test_agent.py

# Run with verbose output
python -u test_agent.py 2>&1 | tee test.log

# Run with environment variables
LLM_MODEL="anthropic/claude-3-haiku" python test_agent.py
```

USING THE SDK TEST HARNESS

Single Task Test

```bash
# Test against a single task
term bench agent -a ./my_agent.py -t ./tasks/simple-task

# With verbose output
term bench agent -a ./my_agent.py -t ./tasks/simple-task --verbose

# With specific timeout
term bench agent -a ./my_agent.py -t ./tasks/simple-task --timeout 300
```

Multiple Tasks

```bash
# Test against multiple tasks
term bench agent -a ./my_agent.py -t ./tasks/task1 ./tasks/task2

# Test against a task directory
term bench agent -a ./my_agent.py -t ./tasks/

# With parallel execution
term bench agent -a ./my_agent.py -t ./tasks/ --parallel 4
```

CREATING TEST TASKS

Task Directory Structure

MY-TEST-TASK/

```
my-test-task/
├── task.yaml          # Task definition
├── setup/             # Initial files (optional)
│   └── ...
└── expected/          # Expected outputs (for manual verification)
    └── ...
```

Task Definition (task.yaml)

```yaml
name: my-test-task
description: A simple test task
instruction: |
  Create a file named hello.txt containing "Hello, World!"

timeout: 60
max_steps: 50

# Optional: files to create before running
setup_files:
  - path: README.md
    content: |
      # Test Project
      This is a test project.

# Optional: environment variables
environment:
  DEBUG: "1"
```

Simple Test Task Example

```yaml
# tasks/create-file/task.yaml
name: create-file
description: Test basic file creation
instruction: Create a file named output.txt containing the text "test passed"
timeout: 30
max_steps: 10
```

Complex Test Task Example

```yaml
# tasks/fix-python/task.yaml
name: fix-python
description: Test Python debugging
instruction: |
  The file main.py has a syntax error. Fix it so it runs correctly.
  The program should print "Hello, World!" when executed.

timeout: 120
max_steps: 30

setup_files:
  - path: main.py
    content: |
      def main()
          print("Hello, World!")

      if __name__ == "__main__":
          main()
```

MOCK TESTING

Mock LLM for Fast Tests

PYTHON

```python
class MockLLM:
    """Mock LLM for testing without API calls."""

    def __init__(self, responses: list[str]):
        self.responses = responses
        self.call_count = 0

    def ask(self, prompt: str, **kwargs) -> MockResponse:
        if self.call_count < len(self.responses):
            response = self.responses[self.call_count]
            self.call_count += 1
            return MockResponse(response)
        return MockResponse('{"task_complete": true}')

    def chat(self, messages: list, **kwargs) -> MockResponse:
        return self.ask(messages[-1]["content"])

class MockResponse:
    def __init__(self, text: str):
        self.text = text

    def json(self):
        import json
        try:
            return json.loads(self.text)
        except:
            return None

# Usage
def test_with_mock():
    mock_llm = MockLLM([
        '{"analysis": "Need to list files", "commands": [{"command": "ls -la"}]}',
        '{"analysis": "Found files", "task_complete": true}'
    ])

    agent = MyAgent()
    agent.llm = mock_llm  # Inject mock
    agent.setup()

    # Run test...
```

Mock Shell for Isolated Tests

PYTHON

```python
class MockShell:
    """Mock shell for testing without actual execution."""

    def __init__(self):
        self.commands = []
        self.responses = {}

    def add_response(self, pattern: str, output: str, exit_code: int = 0):
        self.responses[pattern] = (output, exit_code)

    def __call__(self, command: str, **kwargs) -> MockShellResult:
        self.commands.append(command)

        for pattern, (output, code) in self.responses.items():
            if pattern in command:
                return MockShellResult(command, output, code)

        return MockShellResult(command, "", 0)

class MockShellResult:
    def __init__(self, command: str, output: str, exit_code: int):
        self.command = command
        self.stdout = output
        self.stderr = ""
        self.output = output
        self.exit_code = exit_code
        self.ok = exit_code == 0
        self.failed = exit_code != 0
        self.timed_out = False

# Usage
mock_shell = MockShell()
mock_shell.add_response("ls", "file1.txt\nfile2.txt\n")
mock_shell.add_response("cat", "file contents")
```

DEBUGGING TECHNIQUES

Verbose Logging

```python
import os

DEBUG = os.environ.get("DEBUG", "0") == "1"

def debug_log(msg: str):
    if DEBUG:
        print(f"[DEBUG] {msg}", flush=True)

class MyAgent(Agent):
    def run(self, ctx: AgentContext):
        debug_log(f"Starting task: {ctx.instruction}")

        # ... agent logic ...

        debug_log(f"LLM response: {response.text[:200]}")
        debug_log(f"Parsed data: {data}")
```

State Inspection

```python
class DebuggableAgent(Agent):
    def __init__(self):
        self.state_history = []

    def save_state(self, label: str):
        state = {
            "label": label,
            "step": self.ctx.step,
            "history_length": len(self.messages),
            "last_command": self.last_command,
            "last_output": self.last_output[:500] if self.last_output else None
        }
        self.state_history.append(state)

    def dump_history(self):
        for state in self.state_history:
            print(f"[{state['label']}] Step {state['step']}")
            print(f"  History: {state['history_length']} messages")
            if state.get('last_command'):
                print(f"  Command: {state['last_command'][:80]}")
```

Conversation Logging

```python
class ConversationLogger:
    def __init__(self, filepath: str = "conversation.log"):
        self.filepath = filepath
        self.fp = open(filepath, "w")

    def log_message(self, role: str, content: str):
        self.fp.write(f"\n{'='*60}\n")
        self.fp.write(f"[{role.upper()}]\n")
        self.fp.write(f"{content}\n")
        self.fp.flush()

    def close(self):
        self.fp.close()

# Usage
logger = ConversationLogger()

for msg in messages:
    logger.log_message(msg["role"], msg["content"])

response = llm.chat(messages)
logger.log_message("assistant", response.text)
```

TEST CHECKLIST

Before submitting, verify these items:

1. Basic Functionality

```bash
# Does it start?
python -c "from my_agent import MyAgent; a = MyAgent(); a.setup(); a.cleanup()"

# Does it handle empty tasks?
# (Create a task with minimal instruction)

# Does it terminate?
# (Ensure ctx.done() is always called)
```

2. Error Handling

PYTHON

```python
def test_error_handling():
    """Test that agent handles errors gracefully."""

    agent = MyAgent()
    agent.setup()

    # Test invalid command
    ctx = AgentContext(instruction="Run the command: nonexistent_command_xyz")
    agent.run(ctx)
    assert ctx.is_done, "Agent should complete even with errors"

    agent.cleanup()
```

3. Context Management

PYTHON

```python
def test_long_task():
    """Test that agent handles long tasks without context overflow."""

    agent = MyAgent()
    agent.setup()

    # Task that generates lots of output
    ctx = AgentContext(instruction="List all files recursively and show their contents")
    agent.run(ctx)

    assert ctx.is_done
    assert ctx.step < 100, "Should complete in reasonable steps"

    agent.cleanup()
```

4. Output Verification

PYTHON

```python
def test_output_files():
    """Test that agent creates expected output files."""
    import os
    import tempfile

    with tempfile.TemporaryDirectory() as tmpdir:
        os.chdir(tmpdir)

        agent = MyAgent()
        agent.setup()

        ctx = AgentContext(instruction="Create a file named result.txt with content 'success'")
        agent.run(ctx)

        assert os.path.exists("result.txt"), "Output file should exist"
        with open("result.txt") as f:
            assert "success" in f.read(), "Output should contain expected content"

        agent.cleanup()
```

AUTOMATED TEST SUITE

pytest Setup

PYTHON

```python
# tests/test_agent.py
import pytest
from my_agent import MyAgent

@pytest.fixture
def agent():
    a = MyAgent()
    a.setup()
    yield a
    a.cleanup()

def test_setup_cleanup(agent):
    """Agent initializes and cleans up correctly."""
    assert agent is not None

def test_simple_task(agent, tmp_path):
    """Agent completes simple task."""
    import os
    os.chdir(tmp_path)

    ctx = AgentContext(instruction="Create hello.txt")
    agent.run(ctx)

    assert ctx.is_done

@pytest.mark.parametrize("instruction", [
    "List files",
    "Show current directory",
    "Create a file",
])
def test_various_tasks(agent, tmp_path, instruction):
    """Agent handles various task types."""
    import os
    os.chdir(tmp_path)

    ctx = AgentContext(instruction=instruction)
    agent.run(ctx)

    assert ctx.is_done
```

Running Tests

```bash
# Run all tests
pytest tests/

# Run with verbose output
pytest tests/ -v

# Run specific test
pytest tests/test_agent.py::test_simple_task

# Run with coverage
pytest tests/ --cov=my_agent
```

PRE-SUBMISSION CHECKLIST

```markdown
## Before Submitting

### Code Quality
- [ ] No syntax errors
- [ ] No import errors
- [ ] Passes basic smoke test

### Functionality
- [ ] Completes simple tasks
- [ ] Handles errors gracefully
- [ ] Always calls ctx.done()
- [ ] Manages context size

### Performance
- [ ] Completes tasks in reasonable time
- [ ] Doesn't exceed step limits
- [ ] Manages LLM costs

### Compliance
- [ ] No hardcoded task solutions
- [ ] No pattern matching on instructions
- [ ] No test file access
- [ ] Uses LLM for all decisions
```

SUMMARY

TESTING PHASE	PURPOSE
Unit tests	Verify components work
Mock tests	Fast iteration
Integration tests	Full agent flow
Task tests	Real task execution
Stress tests	Edge cases and limits

CHAPTER 10

PRE-SUBMISSION CHECKLIST

10 - PRE-SUBMISSION CHECKLIST

Use this checklist before every submission to ensure your agent meets all requirements.

QUICK VALIDATION COMMANDS

Run these commands to validate your agent:

```bash
# 1. Syntax check
python -m py_compile my_agent.py && echo "Syntax: OK"

# 2. Import check
python -c "from my_agent import MyAgent" && echo "Imports: OK"

# 3. Instantiation check
python -c "from my_agent import MyAgent; a = MyAgent(); a.setup(); a.cleanup()" && echo "Init: OK"

# 4. Basic test
term bench agent -a ./my_agent.py -t ./test-task --timeout 60
```

MANDATORY REQUIREMENTS

A. Agent Structure

REQUIREMENT	CHECK	HOW TO VERIFY
Extends `Agent` class	[ ]	`grep "class.*Agent" my_agent.py`
Has `setup()` method	[ ]	`grep "def setup" my_agent.py`
Has `run()` method	[ ]	`grep "def run" my_agent.py`
Has `cleanup()` method	[ ]	`grep "def cleanup" my_agent.py`
Calls `ctx.done()`	[ ]	`grep "ctx.done" my_agent.py`

B. Generalist Compliance

REQUIREMENT	CHECK	HOW TO VERIFY
No task keyword matching	[ ]	See verification below
No hardcoded solutions	[ ]	See verification below
No test file access	[ ]	See verification below
LLM-driven decisions	[ ]	Manual review

Verification Commands:

```bash
# Check for keyword matching (should return nothing)
grep -n "in ctx.instruction" my_agent.py
grep -n "in task" my_agent.py
grep -n "instruction.lower()" my_agent.py

# Check for hardcoded handlers (should return nothing)
grep -n "def handle_" my_agent.py
grep -n "handlers\[" my_agent.py
grep -n "TASK_TYPE" my_agent.py

# Check for test file access (should return nothing)
grep -n "/tests/" my_agent.py
grep -n "test_" my_agent.py | grep -v "def test"
```

CODE QUALITY CHECKLIST

C. Error Handling

REQUIREMENT	CHECK
LLM errors handled	[ ]
Parse errors handled	[ ]
Command errors handled	[ ]
Timeouts handled	[ ]
Cost limits handled	[ ]

Minimum Error Handling:

```python
# Must have try/except for LLM calls
try:
    response = self.llm.chat(messages)
except LLMError as e:
    ctx.log(f"LLM error: {e}")
    # Handle appropriately

# Must have try/except for JSON parsing
data = response.json()
if not data:
    # Handle parse failure
```

D. Context Management

REQUIREMENT	CHECK
Output truncation	[ ]
Message history limits	[ ]
Token estimation	[ ]

Verification:

```bash
# Check for truncation (should find something)
grep -n "truncate\|limit\|\[:.*\]" my_agent.py
```

E. Resource Management

REQUIREMENT	CHECK
LLM client closed in cleanup	[ ]
No resource leaks	[ ]
Proper file handle closure	[ ]

RUNTIME CHECKLIST

F. Before Task Execution

STEP	CHECK
Explore environment first	[ ]
Gather context before LLM call	[ ]
Use absolute paths	[ ]

G. During Task Execution

STEP	CHECK
Log progress	[ ]
Check command results	[ ]
Update conversation history	[ ]
Respect step limits	[ ]

H. After Task Execution

STEP	CHECK
Verify output files exist	[ ]
Clean up artifacts	[ ]
Double-confirm completion	[ ]
Call ctx.done()	[ ]

OUTPUT VERIFICATION

I. Understand the Instruction

Before acting, the agent must reason about what the instruction truly requires:

PYTHON

```python
# Let LLM analyze the task requirements
response = llm.ask(
    f"Task: {ctx.instruction}\n\n"
    "What exactly does this task require?\n"
    "- What is the expected output?\n"
    "- What should NOT be done unless asked?",
    system="Analyze the task requirements carefully."
)
```

Key principle: Only do what the instruction asks. Don't assume cleanup, validation, or other actions are needed unless explicitly requested.

ANTI-PATTERN VERIFICATION

K. Forbidden Patterns

Run these checks - all should return nothing:

```bash
# 1. Keyword matching
grep -En 'if.*"[a-z]+".*in.*(instruction|task)' my_agent.py

# 2. Task classification
grep -En 'task_type|TaskType|classify|TASK_' my_agent.py

# 3. Handler dispatch
grep -En 'handler|Handler|dispatch|DISPATCH' my_agent.py

# 4. Hardcoded commands for tasks
grep -En 'COMMANDS|SOLUTIONS|TEMPLATES' my_agent.py

# 5. Test file access
grep -En '/tests?/|test_.*\.py|_test\.py' my_agent.py

# 6. Environment detection
grep -En 'TERM_BENCH|BENCHMARK|is_test' my_agent.py
```

FINAL VALIDATION

L. Complete Test Run

```bash
# 1. Run against a simple task
term bench agent -a ./my_agent.py -t ./simple-task --verbose

# 2. Check the output
cat results/tasks/simple-task/agent.log

# 3. Verify completion
grep "done" results/tasks/simple-task/agent.log
```

M. Cost Check

```bash
# Review LLM costs
grep -i "cost\|token" results/tasks/*/agent.log
```

SUBMISSION CHECKLIST SUMMARY

MARKDOWN

```markdown
## Pre-Submission Verification

### Structure
- [ ] Agent class with setup(), run(), cleanup()
- [ ] ctx.done() always called

### Compliance
- [ ] No keyword matching on instructions
- [ ] No hardcoded task handlers
- [ ] No test file access
- [ ] All decisions via LLM

### Quality
- [ ] Error handling for LLM/parse/command
- [ ] Context truncation
- [ ] Output verification
- [ ] Artifact cleanup

### Testing
- [ ] Passes syntax check
- [ ] Passes import check
- [ ] Completes test task
- [ ] Logs progress appropriately

### Final
- [ ] Reviewed anti-pattern checks
- [ ] Ran full validation
- [ ] Ready to submit
```

QUICK REFERENCE CARD

MUST HAVE

REQUIREMENT	DESCRIPTION
`setup()`, `run()`, `cleanup()`	All three methods implemented
`ctx.done()` on all paths	Every execution path calls done
LLM for all decisions	No hardcoded logic
Error handling	Try/except for LLM, parse, shell
Output truncation	Limit context size
Output verification	Check files before marking done

MUST NOT HAVE

FORBIDDEN	WHY
`if keyword in instruction`	Task-specific matching
`task_type` handlers	Pre-defined task routing
Hardcoded solutions	Bypasses reasoning
Test file access	Cheating
Cached responses	Pre-computed answers
Assumed behaviors	Only do what's asked

KEY PRINCIPLE

DO	DON'T
Reason about what instruction wants	Assume cleanup is needed
Let LLM decide actions	Hardcode validation steps
Only do what's requested	Add unrequested behaviors

BEFORE MARKING COMPLETE

STEP	ACTION
1	Verify the instruction's requirements are met
2	Let LLM confirm task completion
3	Call `ctx.done()`