Technical Specification: The Refactoring Swarm

Project: TP IGL 2025-2026
Version: 1.0.0
Last Updated: January 2, 2026
Status: Draft

Table of Contents

---

1. Executive Summary

1.1 Mission

Build an autonomous multi-agent system ("The Refactoring Swarm") that takes buggy, undocumented, untested Python code and delivers a clean, functional, validated version without human intervention.

1.2 Key Constraints

Constraint	Value
Framework	LangGraph (v0.0.25)
LLM Provider	Google Gemini (configurable)
Max Iterations	10 total
Entry Point	`python main.py --target_dir <path>`
Output	Modified files in `target_dir` + `logs/experiment_data.json`

1.3 Success Criteria

1. **Primary**: All unit tests pass (Judge validates)
2. **Secondary**: Pylint score tracked (decrease allowed if fixes are valid)
3. **Fallback**: Best effort after max iterations

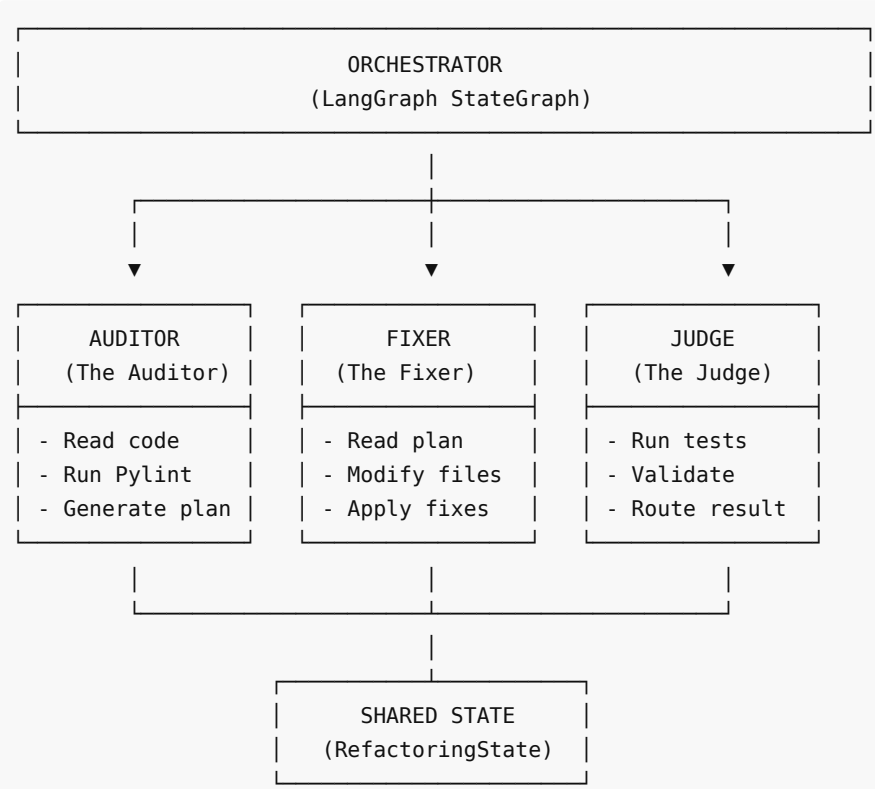
1.4 Grading Breakdown (Per Spec)

Dimension	Weight	Criteria
Performance	40%	Tests pass, Pylint score improved
Technical Robustness	30%	No crashes, no infinite loops, respects `--target_dir`
Data Quality	30%	Valid `experiment_data.json` with complete action history

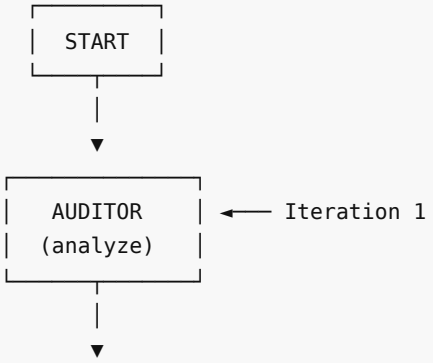
---

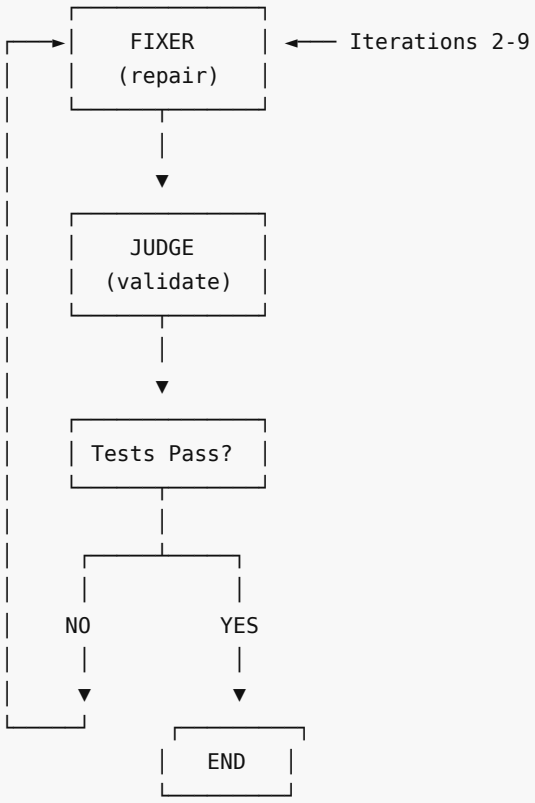
2. System Architecture

2.1 High-Level Overview



2.2 Agent Flow





2.3 LangGraph State Machine Definition

```python
from langgraph.graph import StateGraph, END
from typing import TypedDict, Literal

class RefactoringState(TypedDict):
    target_dir: str
    files: list[str]
    plan: str
    current_iteration: int
    max_iterations: int
    pylint_baseline: float
    pylint_current: float
    test_results: str
    error_logs: list[str]
    status: Literal["in_progress", "success", "failure"]

# Graph definition
graph = StateGraph(RefactoringState)

graph.add_node("auditor", auditor_node)
graph.add_node("fixer", fixer_node)
graph.add_node("judge", judge_node)

graph.set_entry_point("auditor")
```

```
graph.add_edge("auditor", "fixer")
graph.add_edge("fixer", "judge")

graph.add_conditional_edges(
    "judge",
    should_continue,
    {
        "continue": "fixer",
        "end": END
    }
)

app = graph.compile()
```

---

3. Agent Specifications

3.1 Auditor Agent (The Auditor)

Purpose

Analyze the target codebase, run static analysis, and produce a structured refactoring plan.

Inputs

Input	Source	Description
`target_dir`	CLI argument	Path to code directory
`files`	Directory scan	List of Python files

Outputs

Output	Destination	Description
`plan`	State	Structured Markdown refactoring plan
`pylint_baseline`	State	Initial Pylint score

Tools Available

Tool	Purpose
`read_file(path)`	Read file contents
`list_directory(path)`	List files in directory
`run_pylint(path)`	Execute Pylint analysis
`run_tests(path)`	Discover and run existing tests

Behavior

1. Scan `target_dir` for all `.py` files
2. Run Pylint on each file, record baseline score
3. Run existing tests (if any) to understand current state

4. Analyze code for issues:
    - Syntax errors
    - Missing docstrings
    - Naming convention violations
    - Unused imports/variables
    - Code complexity issues
    - Missing type hints
    - Bug patterns
5. Generate structured refactoring plan (see Appendix B)

ActionType Mapping

- Primary: `ActionType.ANALYSIS`
- Test discovery: `ActionType.ANALYSIS`

3.2 Fixer Agent (The Fixer)

Purpose

Read the refactoring plan and apply fixes to the code files.

Inputs

Input	Source	Description
`plan`	State (from Auditor)	Refactoring plan
`error_logs`	State (from Judge)	Previous test failures (if any)
`current_iteration`	State	Current loop iteration

Outputs

Output	Destination	Description
Modified files	`target_dir`	Updated Python files
`pylint_current`	State	Post-fix Pylint score

Tools Available

Tool	Purpose
`read_file(path)`	Read file contents
`write_file(path, content)`	Write complete file (full replacement)
`run_pylint(path)`	Verify fix quality

Behavior

1. Parse the refactoring plan
2. For each file with issues (sequential processing): a. Read current file content b. Apply fixes based on plan + error logs c. Write complete fixed file (full replacement strategy) d. Run Pylint to verify improvement
3. Update `pylint_current` in state

4. Log all modifications

Code Output Strategy

Full File Replacement (chosen over diff/patch for reliability):

```python
# Fixer outputs complete file content
def apply_fix(file_path: str, new_content: str) -> None:
    """Replace entire file content with fixed version."""
    with open(file_path, 'w', encoding='utf-8') as f:
        f.write(new_content)
```

Error Recovery

When `error_logs` contains previous failures:

1. Parse the error message and stack trace
2. Identify the failing test and relevant code
3. Prioritize fixing the specific failure over plan items
4. Avoid repeating previous fix attempts (tracked in state)

ActionType Mapping

- Code modification: `ActionType.FIX`
- Error analysis: `ActionType.DEBUG`

3.3 Judge Agent (The Judge)

Purpose

Execute tests to validate fixes and decide whether to continue or terminate.

Inputs

Input	Source	Description
`target_dir`	State	Path to modified code
`current_iteration`	State	Current loop count

Outputs

Output	Destination	Description
`test_results`	State	Test execution summary
`error_logs`	State	Failure details (if any)
`status`	State	`success` or `in_progress`
Decision	Router	`continue` or `end`

Tools Available

Tool	Purpose

`run_tests(path)`	Execute pytest
`read_file(path)`	Read test files for context

Behavior

1. Execute `pytest` on `target_dir`
2. Capture results (pass/fail counts, error messages)
3. If all tests pass:
   - Set `status = "success"`
   - Return `"end"` to router
4. If tests fail:
   - Extract error logs with context
   - Append to `error_logs` (with iteration number)
   - Return `"continue"` to router (unless max iterations)

Error Context Tiering

Iteration	Context Provided to Fixer
1st failure	Error message + stack trace + failing test code
2nd failure	Above + summary of previous fix attempt
3rd+ failure	Compressed summary + "tried X, Y, Z - try different approach"

ActionType Mapping

- Test execution: `ActionType.ANALYSIS`
- Error analysis: `ActionType.DEBUG`

---

4. Tool Specifications

4.1 File Operations

`read_file(path: str) -> str`

```
def read_file(path: str) -> str:
    """
    Read and return file contents.

    Args:
        path: Absolute or relative path to file

    Returns:
        File contents as string

    Raises:
        FileNotFoundError: If file doesn't exist
        PermissionError: If file not readable
    """
```

```
    with open(path, 'r', encoding='utf-8') as f:
        return f.read()
```

`write_file(path: str, content: str) -> bool`

```
def write_file(path: str, content: str) -> bool:
    """
    Write content to file (full replacement).

    Args:
        path: Absolute or relative path to file
        content: Complete file content

    Returns:
        True if successful

    Raises:
        PermissionError: If file not writable
    """
    with open(path, 'w', encoding='utf-8') as f:
        f.write(content)
    return True
```

`list_directory(path: str) -> list[str]`

```
def list_directory(path: str, pattern: str = "*.py") -> list[str]:
    """
    List files matching pattern in directory.

    Args:
        path: Directory path
        pattern: Glob pattern (default: *.py)

    Returns:
        List of file paths
    """
    from pathlib import Path
    return [str(p) for p in Path(path).rglob(pattern)]
```

4.2 Analysis Tools

`run_pylint(path: str) -> PylintResult`

```
from dataclasses import dataclass

@dataclass
class PylintResult:
    score: float           # 0.0 to 10.0
    messages: list[dict]   # Individual issues
```

```
    raw_output: str        # Complete output

def run_pylint(path: str, timeout: int = 30) -> PylintResult:
    """
    Run Pylint on file or directory.

    Args:
        path: File or directory to analyze
        timeout: Max execution time in seconds

    Returns:
        PylintResult with score and messages

    Raises:
        TimeoutError: If Pylint hangs
    """
    import subprocess
    import json

    result = subprocess.run(
        ["pylint", path, "--output-format=json", "--score=y"],
        capture_output=True,
        text=True,
        timeout=timeout
    )

    # Parse output...
    return PylintResult(score=score, messages=messages, raw_output=result.stdout)
```

4.3 Test Execution

`run_tests(path: str) -> TestResult`

```
@dataclass
class TestResult:
    passed: int
    failed: int
    errors: int
    skipped: int
    total: int
    success: bool
    output: str
    failures: list[dict]  # Detailed failure info

def run_tests(path: str, timeout: int = 60) -> TestResult:
    """
    Execute pytest on target directory.

    Args:
        path: Directory containing tests
        timeout: Max execution time in seconds
```

```
    Returns:
        TestResult with pass/fail counts and details

    Raises:
        TimeoutError: If tests hang
    """
    import subprocess

    result = subprocess.run(
        ["pytest", path, "-v", "--tb=short", "-q"],
        capture_output=True,
        text=True,
        timeout=timeout,
        cwd=path
    )

    # Parse output...
    return TestResult(...)
```

---

5. Data Flow & State Management

5.1 LangGraph State Schema

```
from typing import TypedDict, Literal, Annotated
from operator import add

class FileState(TypedDict):
    path: str
    original_content: str
    current_content: str
    pylint_before: float
    pylint_after: float

class FixAttempt(TypedDict):
    iteration: int
    file_path: str
    changes_made: str
    result: Literal["success", "failure"]
    error_message: str | None

class RefactoringState(TypedDict):
    # Input
    target_dir: str

    # Discovery
    files: list[str]

    # Auditor output
    plan: str
```

```
    pylint_baseline: float
    initial_test_results: str

    # Fixer tracking
    current_iteration: int
    max_iterations: int
    fix_attempts: Annotated[list[FixAttempt], add]  # Append-only

    # Judge output
    test_results: str
    error_logs: list[str]
    pylint_current: float

    # Termination
    status: Literal["in_progress", "success", "failure", "max_iterations"]
    final_summary: str
```

5.2 Agent Communication Protocol

Agents communicate **exclusively through state**. No direct agent-to-agent calls.

```
┌─────────────────────────────────────────────────────────────┐
│                          STATE                              │
├─────────────────────────────────────────────────────────────┤
│                                                             │
│  Auditor WRITES:          Fixer READS:                      │
│  - plan                   - plan                            │
│  - pylint_baseline        - error_logs                      │
│  - files                  - fix_attempts (to avoid repeats) │
│                                                             │
│  Fixer WRITES:            Judge READS:                      │
│  - fix_attempts           - target_dir                      │
│  - pylint_current         - current_iteration               │
│  - (modifies files)                                         │
│                                                             │
│  Judge WRITES:            Orchestrator READS:               │
│  - test_results           - status                          │
│  - error_logs             - current_iteration               │
│  - status                                                   │
│                                                             │
└─────────────────────────────────────────────────────────────┘
```

5.3 Plan Format (Structured Markdown)

The Auditor produces a plan in this format:

```
# Refactoring Plan

## Summary
- **Files Analyzed**: 3
- **Total Issues Found**: 12
```

```
- **Pylint Baseline Score**: 4.5/10

## File: src/utils.py

### Issue 1: Missing Docstring (Line 1)
- **Type**: `MISSING_DOCSTRING`
- **Severity**: Medium
- **Location**: Module level
- **Description**: Module lacks a docstring
- **Suggested Fix**: Add module docstring describing purpose

### Issue 2: Naming Convention (Line 45-52)
- **Type**: `NAMING_VIOLATION`
- **Severity**: Low
- **Location**: Function `calc`
- **Current**: `def calc(x, y):`
- **Suggested**: `def calculate_total(price, quantity):`
- **Reason**: Function name should be descriptive

### Issue 3: Potential Bug (Line 78)
- **Type**: `BUG`
- **Severity**: High
- **Location**: Function `process_data`
- **Description**: Division by zero possible when `count == 0`
- **Suggested Fix**: Add zero-check before division

## File: src/main.py

### Issue 4: Unused Import (Line 3)
- **Type**: `UNUSED_IMPORT`
- **Severity**: Low
- **Location**: `import os`
- **Suggested Fix**: Remove unused import

...
```

---

6. Iteration & Termination Logic

6.1 Budget Allocation

Phase	Iterations	Description
Auditor	1	Initial analysis (always runs once)
Fixer ↔ Judge Loop	8 max	Self-healing cycle
Final	1	Reserved for wrap-up
Total	**10**	Hard limit

6.2 Router Logic

```python
def should_continue(state: RefactoringState) -> Literal["continue", "end"]:
    """Decide whether to continue the Fixer-Judge loop."""

    # Success: tests pass
    if state["status"] == "success":
        return "end"

    # Max iterations reached
    if state["current_iteration"] >= state["max_iterations"]:
        state["status"] = "max_iterations"
        return "end"

    # Continue loop
    return "continue"
```

6.3 Success Criteria

```python
def evaluate_success(state: RefactoringState) -> bool:
    """
    Determine if mission is complete.

    Primary: All tests pass
    Secondary: Pylint score tracked (decrease allowed)
    """
    # Parse test results
    test_result = parse_test_output(state["test_results"])

    # Primary criterion: tests pass
    if test_result.failed == 0 and test_result.errors == 0:
        return True

    return False
```

6.4 Failure Handling Matrix

Failure Type	Detection	Recovery
Syntax error in generated code	`ast.parse()` fails	Re-prompt Fixer with error
Pylint hangs	Timeout (30s)	Kill, log warning, continue
Tests hang	Timeout (60s)	Kill, mark as "untestable"
LLM returns malformed response	JSON parse error	Retry with clarification
Agent refuses to fix	Detect "cannot", "impossible"	Force retry with different prompt
Same fix attempted twice	Hash comparison	Inject "already tried" context

Max iterations reached	Counter check	Terminate with best effort

6.5 Iteration Tracking

```python
def update_iteration(state: RefactoringState) -> RefactoringState:
    """Increment iteration counter after each Fixer-Judge cycle."""
    state["current_iteration"] += 1
    return state
```

---

7. Logging & Telemetry

7.1 Required Format

All agent actions must be logged using the provided `log_experiment()` function:

```python
from src.utils.logger import log_experiment, ActionType

log_experiment(
    agent_name="Auditor",           # "Auditor" | "Fixer" | "Judge"
    model_used="gemini-1.5-flash",  # Dynamic from config
    action=ActionType.ANALYSIS,     # From ActionType enum
    details={
        "input_prompt": "...",      # REQUIRED: Exact prompt sent to LLM
        "output_response": "...",   # REQUIRED: Raw LLM response
        "file_analyzed": "...",     # Additional context
        "issues_found": 5,
    },
    status="SUCCESS"                # "SUCCESS" | "FAILURE" | "INFO"
)
```

7.2 ActionType Mapping

Agent	Action	ActionType
Auditor	Analyzing code	`ANALYSIS`
Auditor	Running Pylint	`ANALYSIS`
Auditor	Discovering tests	`ANALYSIS`
Fixer	Reading error logs	`DEBUG`
Fixer	Modifying code	`FIX`
Fixer	Generating new tests	`GENERATION`
Judge	Running tests	`ANALYSIS`
Judge	Analyzing failures	`DEBUG`

7.3 Log Entry Schema

Each entry in `logs/experiment_data.json`:

```
{
  "id": "uuid-v4",
  "timestamp": "2026-01-02T15:30:00.000Z",
  "agent": "Auditor",
  "model": "gemini-1.5-flash",
  "action": "CODE_ANALYSIS",
  "details": {
    "input_prompt": "You are an expert Python code auditor...",
    "output_response": "I have analyzed the code and found...",
    "files_analyzed": ["utils.py", "main.py"],
    "issues_found": 7,
    "pylint_score": 4.5
  },
  "status": "SUCCESS"
}
```

7.4 Logging Wrapper

```
from functools import wraps
from src.utils.logger import log_experiment, ActionType

def log_agent_action(agent_name: str, action_type: ActionType):
    """Decorator to automatically log agent LLM calls."""
    def decorator(func):
        @wraps(func)
        def wrapper(prompt: str, *args, **kwargs):
            try:
                response = func(prompt, *args, **kwargs)
                log_experiment(
                    agent_name=agent_name,
                    model_used=get_model_name(),
                    action=action_type,
                    details={
                        "input_prompt": prompt,
                        "output_response": response,
                        **kwargs.get("extra_details", {})
                    },
                    status="SUCCESS"
                )
                return response
            except Exception as e:
                log_experiment(
                    agent_name=agent_name,
                    model_used=get_model_name(),
                    action=action_type,
                    details={
```

```
                    "input_prompt": prompt,
                    "output_response": str(e),
                    "error": str(e)
                },
                status="FAILURE"
            )
            raise
        return wrapper
    return decorator
```

---

8. LLM Provider Interface

8.1 Abstraction Layer

```
from abc import ABC, abstractmethod
from dataclasses import dataclass
from typing import Protocol

@dataclass
class Message:
    role: str  # "system" | "user" | "assistant"
    content: str

@dataclass
class LLMResponse:
    content: str
    model: str
    usage: dict  # token counts

class LLMProvider(Protocol):
    """Protocol for LLM providers (Gemini, OpenAI, etc.)"""

    @property
    def model_name(self) -> str:
        """Return the model identifier for logging."""
        ...

    def complete(
        self,
        messages: list[Message],
        temperature: float = 0.7,
        max_tokens: int = 4096
    ) -> LLMResponse:
        """Generate completion from messages."""
        ...
```

8.2 Gemini Implementation

```python
import google.generativeai as genai
from typing import Optional

class GeminiProvider:
    """Google Gemini LLM provider."""

    def __init__(
        self,
        api_key: str,
        model: str = "gemini-1.5-flash"
    ):
        genai.configure(api_key=api_key)
        self._model = genai.GenerativeModel(model)
        self._model_name = model

    @property
    def model_name(self) -> str:
        return self._model_name

    def complete(
        self,
        messages: list[Message],
        temperature: float = 0.7,
        max_tokens: int = 4096
    ) -> LLMResponse:
        # Convert messages to Gemini format
        prompt = self._format_messages(messages)

        response = self._model.generate_content(
            prompt,
            generation_config=genai.types.GenerationConfig(
                temperature=temperature,
                max_output_tokens=max_tokens,
            )
        )

        return LLMResponse(
            content=response.text,
            model=self._model_name,
            usage={"prompt_tokens": 0, "completion_tokens": 0}  # Gemini doesn't
expose this easily
        )

    def _format_messages(self, messages: list[Message]) -> str:
        """Convert message list to single prompt."""
        parts = []
        for msg in messages:
            if msg.role == "system":
                parts.append(f"Instructions: {msg.content}\n")
            elif msg.role == "user":
                parts.append(f"User: {msg.content}\n")
```

```
            elif msg.role == "assistant":
                parts.append(f"Assistant: {msg.content}\n")
        return "\n".join(parts)
```

8.3 Configuration

```python
# src/config.py
import os
from dataclasses import dataclass

@dataclass
class Config:
    # LLM
    llm_provider: str = "gemini"
    llm_model: str = "gemini-1.5-flash"
    llm_temperature: float = 0.7
    llm_max_tokens: int = 4096

    # Iteration limits
    max_iterations: int = 10

    # Timeouts (seconds)
    pylint_timeout: int = 30
    test_timeout: int = 60
    llm_timeout: int = 120

    @classmethod
    def from_env(cls) -> "Config":
        return cls(
            llm_model=os.getenv("LLM_MODEL", "gemini-1.5-flash"),
            # ... other env vars
        )

def get_llm_provider(config: Config) -> LLMProvider:
    """Factory function for LLM provider."""
    api_key = os.getenv("GOOGLE_API_KEY")
    if not api_key:
        raise ValueError("GOOGLE_API_KEY not set")

    if config.llm_provider == "gemini":
        return GeminiProvider(api_key, config.llm_model)
    else:
        raise ValueError(f"Unknown provider: {config.llm_provider}")
```

---

9. Project Structure

9.1 Directory Layout

```
/tp-igl
├── main.py                     # [LOCKED] Entry point
├── requirements.txt            # [LOCKED] Dependencies
├── check_setup.py              # [LOCKED] Environment check
├── .env                        # API keys (gitignored)
├── .env.example                # Template
├── .gitignore
│
├── /src
│   ├── __init__.py
│   │
│   ├── /agents                 # Agent implementations
│   │   ├── __init__.py
│   │   ├── base.py             # Base agent class
│   │   ├── auditor.py          # Auditor agent
│   │   ├── fixer.py            # Fixer agent
│   │   └── judge.py            # Judge agent
│   │
│   ├── /tools                  # Tool implementations
│   │   ├── __init__.py
│   │   ├── file_ops.py         # read_file, write_file, list_directory
│   │   ├── pylint_tool.py      # run_pylint
│   │   └── test_runner.py      # run_tests
│   │
│   ├── /prompts                # System prompts
│   │   ├── __init__.py
│   │   ├── auditor_prompt.py   # Auditor system prompt
│   │   ├── fixer_prompt.py     # Fixer system prompt
│   │   └── judge_prompt.py     # Judge system prompt
│   │
│   ├── /llm                    # LLM provider abstraction
│   │   ├── __init__.py
│   │   ├── base.py             # Protocol/ABC
│   │   └── gemini.py           # Gemini implementation
│   │
│   ├── /graph                  # LangGraph orchestration
│   │   ├── __init__.py
│   │   ├── state.py            # State schema
│   │   ├── nodes.py            # Node functions
│   │   └── builder.py          # Graph construction
│   │
│   ├── /utils                  # Utilities
│   │   ├── __init__.py
│   │   └── logger.py           # [PROVIDED] Logging
│   │
│   └── config.py               # Configuration
│
├── /logs
│   ├── .gitkeep
│   └── experiment_data.json    # [TRACKED] Output
│
```

```
├── /sandbox                    # [GITIGNORED] Working directory
│
├── /tests                      # Project tests (optional)
│   ├── __init__.py
│   ├── test_tools.py
│   ├── test_agents.py
│   └── test_integration.py
│
└── /docs
    ├── ENONCE.md
    ├── SPEC.md                 # This document
    └── *.pdf
```

9.2 Module Responsibilities

Module	Owner Role	Responsibility
`main.py`	Orchestrator	CLI parsing, graph execution
`src/graph/*`	Orchestrator	LangGraph state machine
`src/tools/*`	Toolsmith	Tool implementations, sandboxing
`src/prompts/*`	Prompt Engineer	System prompts
`src/agents/*`	Shared	Agent logic (uses tools + prompts)
`src/llm/*`	Toolsmith	LLM provider abstraction
`src/utils/logger.py`	Data Officer	Logging (provided)
`src/config.py`	Orchestrator	Configuration management

---

10. Implementation Roadmap

10.1 Phase 1: Core Tools (Days 1-3)

Owner: Toolsmith

Task	Priority	Deliverable
Implement `read_file()`	High	`src/tools/file_ops.py`
Implement `write_file()`	High	`src/tools/file_ops.py`
Implement `list_directory()`	High	`src/tools/file_ops.py`
Implement `run_pylint()`	High	`src/tools/pylint_tool.py`
Implement `run_tests()`	High	`src/tools/test_runner.py`
Add timeout handling	Medium	All tools
Add path validation	Medium	`src/tools/file_ops.py`

Acceptance Criteria:

- All tools work standalone
- Timeouts prevent hangs
- Tools only access `target_dir`

10.2 Phase 2: Agents (Days 3-6)

Owner: Prompt Engineer + Toolsmith

Task	Priority	Deliverable
Write Auditor system prompt	High	`src/prompts/auditor_prompt.py`
Write Fixer system prompt	High	`src/prompts/fixer_prompt.py`
Write Judge system prompt	High	`src/prompts/judge_prompt.py`
Implement LLM provider	High	`src/llm/gemini.py`
Implement Auditor agent	High	`src/agents/auditor.py`
Implement Fixer agent	High	`src/agents/fixer.py`
Implement Judge agent	High	`src/agents/judge.py`

Acceptance Criteria:

- Each agent works in isolation
- Prompts produce consistent output format
- Logging works correctly

10.3 Phase 3: Orchestration (Days 6-9)

Owner: Orchestrator

Task	Priority	Deliverable
Define state schema	High	`src/graph/state.py`
Implement node functions	High	`src/graph/nodes.py`
Build graph	High	`src/graph/builder.py`
Implement router logic	High	`src/graph/nodes.py`
Wire up `main.py`	High	`main.py`
Add iteration tracking	Medium	`src/graph/nodes.py`

Acceptance Criteria:

- Full pipeline runs end-to-end
- State flows correctly between agents
- Iteration limit enforced

10.4 Phase 4: Testing & Hardening (Days 9-12)

Owner: Data Officer + All

Task	Priority	Deliverable
Create test dataset	High	`sandbox/test_cases/`
Validate JSON logging	High	Manual verification
Test edge cases	High	Various
Performance testing	Medium	Timing logs
Error recovery testing	Medium	Various
Final integration test	High	Full run

Acceptance Criteria:

- System handles "trap" files gracefully
- `experiment_data.json` is valid and complete
- No infinite loops
- Graceful degradation on failures

10.5 Milestone Schedule

Day	Milestone
3	All tools implemented and tested
6	All agents working in isolation
9	Full pipeline operational
12	Hardened and submission-ready
14	Buffer / polish

---

11. Open Questions (TBD)

11.1 Test Generation

Question: What if the target codebase has NO tests?

Options:

- A) Generate tests based on function signatures (risk: tests match buggy behavior)
- B) Generate smoke tests (does it run without crashing?)
- C) Skip testing, rely only on Pylint improvement
- D) Fail fast with clear message

Action: Ask professor for clarification.

11.2 External Dependencies

Question: What if target code imports libraries not in `requirements.txt`?

Options:

- A) Attempt to install missing dependencies
- B) Skip files with unresolvable imports
- C) Fail with clear error

Recommendation: Option B (graceful degradation)

11.3 Circular Imports

Question: How to handle circular import errors in target code?

Recommendation: Log as unfixable, skip file, continue with others.

Appendix A: System Prompts

A.1 Auditor System Prompt

```
AUDITOR_SYSTEM_PROMPT = """
You are The Auditor, an expert Python code analyst. Your mission is to analyze code
and produce a detailed refactoring plan.

## Your Capabilities
- Deep understanding of Python best practices (PEP 8, PEP 257)
- Static analysis interpretation (Pylint)
- Bug pattern recognition
- Code smell detection

## Your Task
1. Analyze all Python files in the target directory
2. Review Pylint output for each file
3. Identify issues in these categories:
   - SYNTAX_ERROR: Code that won't parse
   - BUG: Logic errors, potential runtime failures
   - NAMING_VIOLATION: Non-PEP8 names
   - MISSING_DOCSTRING: Undocumented modules/classes/functions
   - UNUSED_CODE: Unused imports, variables, functions
   - COMPLEXITY: Overly complex code
   - TYPE_HINT: Missing type annotations

## Output Format
You MUST output a structured Markdown plan following this exact format:

```markdown
# Refactoring Plan

## Summary
- **Files Analyzed**: <count>
- **Total Issues Found**: <count>
- **Pylint Baseline Score**: <score>/10

## File: <path>

### Issue <N>: <Title> (Line <line_number>)
```
```

```
- **Type**: `<ISSUE_TYPE>`
- **Severity**: High | Medium | Low
- **Location**: <specific location>
- **Description**: <what's wrong>
- **Suggested Fix**: <how to fix>
```

Rules

- Be thorough but prioritize HIGH severity issues
- Always include line numbers
- Provide actionable fix suggestions
- Do not modify any files yourself
- Output ONLY the Markdown plan, no other text """

```
### A.2 Fixer System Prompt

```python
FIXER_SYSTEM_PROMPT = """
You are The Fixer, an expert Python developer. Your mission is to repair code based on
the refactoring plan.

## Your Capabilities
- Expert Python programming
- Bug fixing
- Code refactoring
- Following coding standards

## Your Task
1. Read the refactoring plan from The Auditor
2. Read any error logs from The Judge (if this is a retry)
3. For each file with issues:
   a. Read the current file content
   b. Apply the necessary fixes
   c. Output the COMPLETE fixed file

## Input Context
You will receive:
- The refactoring plan (Markdown)
- The current file content
- Previous error logs (if any)
- List of previous fix attempts (to avoid repeating)

## Output Format
For each file you fix, output:
```

FILE:

```
<complete file content with all fixes applied>
```
```

```
## Rules
- Output the COMPLETE file content, not just changes
- Fix ALL issues mentioned in the plan for that file
- Maintain the original code's functionality
- Follow PEP 8 style guidelines
- Add docstrings where missing
- If error logs mention a specific failure, prioritize fixing that
- Do NOT repeat a fix that already failed (check previous attempts)
- If you cannot fix something, leave a TODO comment explaining why

## Error Recovery
If you receive error logs from The Judge:
1. Parse the error message carefully
2. Identify the root cause
3. Apply a DIFFERENT fix than previous attempts
4. If stuck after 3 attempts on same issue, mark as unfixable
"""
```

A.3 Judge System Prompt

```
JUDGE_SYSTEM_PROMPT = """
You are The Judge, a strict test executor and validator. Your mission is to verify
that code fixes are correct.

## Your Capabilities
- Test execution (pytest)
- Error analysis
- Pass/fail determination

## Your Task
1. Execute all tests in the target directory
2. Analyze the results
3. If tests pass: Declare SUCCESS
4. If tests fail: Extract useful error context for The Fixer

## Output Format

### If SUCCESS:
```

STATUS: SUCCESS SUMMARY: All tests passed. PYLINT_SCORE: /10 (baseline: /10)

```
### If FAILURE:
```

STATUS: FAILURE SUMMARY: / tests passed, failed.

FAILURES:

Test:

- **File**:
- **Error**:
- **Message**:
- **Relevant Code**:

```
<the failing test code>
```

- **Stack Trace** (last 5 lines):

```
<stack trace>
```

SUGGESTION:

```
## Rules
- Be precise about which tests failed
- Include enough context for The Fixer to understand the issue
- Do not attempt to fix code yourself
- If tests timeout, report as TIMEOUT not FAILURE
- Track iteration number for context compression
"""
```

Appendix B: Example Refactoring Plan

```
# Refactoring Plan

## Summary
- **Files Analyzed**: 2
- **Total Issues Found**: 7
- **Pylint Baseline Score**: 3.5/10

---

## File: src/calculator.py

### Issue 1: Missing Module Docstring (Line 1)
- **Type**: `MISSING_DOCSTRING`
- **Severity**: Medium
- **Location**: Module level
- **Description**: The module lacks a docstring explaining its purpose
- **Suggested Fix**: Add a module docstring at the top of the file

### Issue 2: Naming Convention Violation (Line 5)
- **Type**: `NAMING_VIOLATION`
- **Severity**: Low
- **Location**: Function `calc`
- **Current**: `def calc(a, b):`
- **Suggested**: `def calculate_sum(first_number, second_number):`
- **Reason**: Function and parameter names should be descriptive
```

```
### Issue 3: Division by Zero Bug (Line 12)
- **Type**: `BUG`
- **Severity**: High
- **Location**: Function `divide`, Line 12
- **Description**: No check for division by zero
- **Current Code**:
```python
def divide(a, b):
    return a / b
```
```

- **Suggested Fix**:

```python
def divide(a, b):
    if b == 0:
        raise ValueError("Cannot divide by zero")
    return a / b
```

Issue 4: Unused Import (Line 2)

- **Type**: `UNUSED_CODE`
- **Severity**: Low
- **Location**: `import sys`
- **Suggested Fix**: Remove the unused import

File: src/utils.py

Issue 5: Missing Type Hints (Line 8-15)

- **Type**: `TYPE_HINT`
- **Severity**: Medium
- **Location**: Function `process_data`
- **Description**: Function lacks type annotations
- **Current**: `def process_data(data):`
- **Suggested**: `def process_data(data: list[dict]) -> dict:`

Issue 6: Bare Except Clause (Line 22)

- **Type**: `BUG`
- **Severity**: High
- **Location**: Line 22
- **Description**: Bare `except:` catches all exceptions including KeyboardInterrupt
- **Current Code**:

```python
try:
    result = risky_operation()
except:
    pass
```

- **Suggested Fix**:

```python
try:
    result = risky_operation()
except Exception as e:
    logger.error(f"Operation failed: {e}")
    result = None
```

Issue 7: Complex Function (Line 30-75)

- **Type**: `COMPLEXITY`
- **Severity**: Medium
- **Location**: Function `handle_request`
- **Description**: Function has cyclomatic complexity of 15 (threshold: 10)
- **Suggested Fix**: Extract nested conditionals into helper functions

---

Priority Order

1. **HIGH**: Issue 3 (Division by Zero) - Runtime crash
2. **HIGH**: Issue 6 (Bare Except) - Swallows errors
3. **MEDIUM**: Issue 5 (Type Hints) - Maintainability
4. **MEDIUM**: Issue 7 (Complexity) - Maintainability
5. **MEDIUM**: Issue 1 (Docstring) - Documentation
6. **LOW**: Issue 2 (Naming) - Style
7. **LOW**: Issue 4 (Unused Import) - Cleanup

```
---

## Appendix C: State Schema

### C.1 Complete TypedDict Definition

```python
from typing import TypedDict, Literal, Annotated, Optional
from operator import add
from dataclasses import dataclass
from datetime import datetime

# Sub-types
class FileInfo(TypedDict):
    path: str
    size: int
    last_modified: str

class PylintMessage(TypedDict):
    type: str
    module: str
    line: int
    column: int
    message: str
    symbol: str
```
```

```
class FixAttempt(TypedDict):
    iteration: int
    timestamp: str
    file_path: str
    issue_addressed: str
    changes_summary: str
    result: Literal["success", "failure", "partial"]
    error_message: Optional[str]

class TestFailure(TypedDict):
    test_name: str
    test_file: str
    error_type: str
    error_message: str
    stack_trace: str

# Main State
class RefactoringState(TypedDict):
    # === INPUT ===
    target_dir: str

    # === DISCOVERY (set by Auditor) ===
    files: list[FileInfo]
    has_tests: bool
    test_framework: Optional[str]  # "pytest", "unittest", None

    # === AUDITOR OUTPUT ===
    plan: str  # Markdown plan
    pylint_baseline: float
    pylint_messages: list[PylintMessage]
    initial_test_results: Optional[str]

    # === ITERATION TRACKING ===
    current_iteration: int
    max_iterations: int  # Default: 10

    # === FIXER TRACKING ===
    fix_attempts: Annotated[list[FixAttempt], add]  # Append-only list
    files_modified: list[str]
    pylint_current: float

    # === JUDGE OUTPUT ===
    test_results: str
    tests_passed: int
    tests_failed: int
    test_failures: list[TestFailure]
    error_logs: list[str]  # Accumulated across iterations

    # === TERMINATION ===
    status: Literal["in_progress", "success", "failure", "max_iterations"]
```

```
    final_summary: str
    completion_time: Optional[str]
```

C.2 State Initialization

```python
def initialize_state(target_dir: str) -> RefactoringState:
    """Create initial state for the refactoring graph."""
    return RefactoringState(
        # Input
        target_dir=target_dir,

        # Discovery (populated by Auditor)
        files=[],
        has_tests=False,
        test_framework=None,

        # Auditor output
        plan="",
        pylint_baseline=0.0,
        pylint_messages=[],
        initial_test_results=None,

        # Iteration
        current_iteration=0,
        max_iterations=10,

        # Fixer
        fix_attempts=[],
        files_modified=[],
        pylint_current=0.0,

        # Judge
        test_results="",
        tests_passed=0,
        tests_failed=0,
        test_failures=[],
        error_logs=[],

        # Termination
        status="in_progress",
        final_summary="",
        completion_time=None,
    )
```

C.3 State Update Helpers

```python
def add_fix_attempt(
    state: RefactoringState,
    file_path: str,
    issue: str,
```

```
    changes: str,
    result: str,
    error: Optional[str] = None
) -> RefactoringState:
    """Record a fix attempt in state."""
    attempt = FixAttempt(
        iteration=state["current_iteration"],
        timestamp=datetime.now().isoformat(),
        file_path=file_path,
        issue_addressed=issue,
        changes_summary=changes,
        result=result,
        error_message=error,
    )
    state["fix_attempts"].append(attempt)
    return state

def get_previous_attempts_for_file(
    state: RefactoringState,
    file_path: str
) -> list[FixAttempt]:
    """Get all previous fix attempts for a specific file."""
    return [
        attempt for attempt in state["fix_attempts"]
        if attempt["file_path"] == file_path
    ]

def has_tried_fix(
    state: RefactoringState,
    file_path: str,
    issue: str
) -> bool:
    """Check if a specific fix was already attempted."""
    for attempt in state["fix_attempts"]:
        if attempt["file_path"] == file_path and attempt["issue_addressed"] ==
issue:
            return True
    return False
```

Document History

Version	Date	Author	Changes
1.0.0	2026-01-02	Team	Initial specification

End of Technical Specification