**Comprehensive End-to-End Engineering Specification for CodeCritic Experimentation**

**Overview**

The CodeCritic system is an autonomous multi-agent framework designed to facilitate software generation, evaluation, maintenance, and continuous improvement. This comprehensive manual provides detailed guidance on structuring, executing, and evaluating experiments. These experiments systematically identify the optimal combinations of system components for creating software systems maintainable by artificial intelligence (AI).

**Purpose of Experiments**

The experiments aim to explore different configurations of CodeCritic’s integrated tools and methodologies, including structured logging, multi-agent interactions, symbolic reasoning via symbol graphs, and iterative self-improvement. Each experimental iteration seeks to enhance agent interactions, logging accuracy, and system adaptability to achieve AI-driven software maintenance.

**Folder-File Structure**

The following folder and file structure outlines how the CodeCritic framework organizes its components and outputs:

app

├── abstract\_classes

│ ├── system\_manager\_base.py

│ ├── agent\_base.py

│ ├── state\_manager\_base.py

│ ├── prompt\_generator\_base.py

│ ├── context\_provider\_base.py

│ ├── tool\_provider\_base.py

│ └── scoring\_provider\_base.py

├── extensions

│ ├── system\_managers

│ ├── state\_managers

│ ├── agents

│ ├── tools

│ ├── prompt\_generators

│ ├── context\_providers

│ ├── tool\_providers

│ ├── agent\_prompts

│ ├── system\_prompts

│ └── scoring\_models

├── enums

│ ├── system\_enums.py

│ └── agent\_enums.py

├── utilities

│ ├── file\_management

│ ├── metadata

│ │ ├── logging

│ │ └── footer

│ └── snapshots

├── factories

│ ├── system\_manager.py

│ ├── state\_manager.py

│ ├── agent.py

│ ├── prompt\_manager.py

│ ├── system\_config\_provider.py

│ ├── experiment\_config\_provider.py

│ ├── tool\_provider.py

│ └── scoring\_provider.py

├── registries

│ ├── system\_managers

│ ├── state\_managers

│ ├── agents

│ ├── agent\_engines

│ ├── prompt\_generators

│ ├── agent\_prompts

│ ├── system\_prompts

│ ├── context\_providers

│ ├── tool\_providers

│ ├── tools

│ └── scoring\_models

└── tools

├── black\_runner.py

├── doc\_formatter.py

├── mypy\_runner.py

├── radon\_runner.py

├── ruff\_runner.py

└── sonarcloud\_runner.py

experiments

├── logs

│ ├── structured logging output

│ └── detailed execution logs

└── artifacts

├── intermediate and final outputs

└── snapshots of code states

**Database Schemas**

SQLite is the database backend for structured logging and metadata storage, supporting experiment reproducibility and analysis.

CREATE TABLE agent\_engine (

id INTEGER PRIMARY KEY,

name TEXT,

description TEXT,

model TEXT,

engine\_config TEXT,

artifact\_path TEXT

);

CREATE TABLE agent\_prompt (

id INTEGER PRIMARY KEY,

name TEXT,

description TEXT,

agent\_role TEXT,

system\_type TEXT,

artifact\_path TEXT

);

CREATE TABLE system\_prompt (

id INTEGER PRIMARY KEY,

name TEXT,

description TEXT,

system\_type TEXT,

artifact\_path TEXT

);

CREATE TABLE context\_provider (

id INTEGER PRIMARY KEY,

name TEXT,

description TEXT,

system\_type TEXT,

tooling\_provider\_id INTEGER

);

CREATE TABLE file\_path (

artifact\_path TEXT PRIMARY KEY

);

CREATE TABLE agent\_config (

id INTEGER PRIMARY KEY,

name TEXT,

description TEXT,

agent\_role TEXT,

system\_type TEXT,

agent\_engine\_id INTEGER,

prompt\_generator\_id INTEGER,

artifact\_path TEXT

);

CREATE TABLE tooling\_provider (

id INTEGER PRIMARY KEY,

name TEXT,

description TEXT,

artifact\_path TEXT

);

CREATE TABLE scoring\_provider (

id INTEGER PRIMARY KEY,

name TEXT,

description TEXT,

artifact\_path TEXT

);

CREATE TABLE prompt\_generator (

id INTEGER PRIMARY KEY,

name TEXT,

description TEXT,

agent\_prompt\_id INTEGER,

system\_prompt\_id INTEGER,

content\_provider\_id INTEGER,

artifact\_path TEXT

);

CREATE TABLE state\_manager (

id INTEGER PRIMARY KEY,

name TEXT,

description TEXT,

system\_state TEXT,

system\_type TEXT,

agent\_id INTEGER,

artifact\_path TEXT

);

CREATE TABLE system\_config (

id INTEGER PRIMARY KEY,

name TEXT,

description TEXT,

system\_type TEXT,

state\_manager\_id INTEGER,

scoring\_model\_id INTEGER

);

CREATE TABLE experiment\_config (

id INTEGER PRIMARY KEY,

name TEXT,

description TEXT,

system\_manager\_id INTEGER,

scoring\_model\_id INTEGER

);

CREATE TABLE series (

id INTEGER PRIMARY KEY,

experiment\_config\_id INTEGER

);

**Comprehensive End-to-End Engineering Specification for CodeCritic Experimentation (Part 2)**

**Components and Responsibilities**

**Experiment Lifecycle Management**

**Experiment**

* Initiates and manages the execution of experiments.
* **Method:** run() initiates experiments based on configurations.

**System and State Management**

**SystemManagerBase**

* Handles logging and finite state machine (FSM) logic.
* **Method:** run() ensures logging; \_run\_system\_logic() performs state transitions.

**StateManagerBase**

* Manages the execution of agent-level operations within system states.
* **Method:** run() executes logging and agent actions.

**Agents and Logic Execution**

**AgentBase**

* Responsible for executing agent-specific logic with logging.
* **Method:** run() manages logging; \_run\_agent\_logic() implements agent tasks.

**Prompt Generation**

**PromptGeneratorBase**

* Generates prompts needed by agents, ensuring logging.
* **Method:** generate\_prompt() oversees logging and calls \_generate\_prompt().

**Context and Symbol Management**

**ContextProviderBase**

* Supplies context using symbol graphs, enforcing comprehensive logging.
* **Method:** get\_context() handles logging; \_get\_context() manages symbol graph access.

**External Tool Integration**

**ToolProviderBase**

* Integrates and manages the execution of external tools with robust logging.
* **Method:** run() handles logging; \_run() executes tool-specific logic.

**Scoring and Evaluation**

**ScoringProviderBase**

* Evaluates agent performance using predefined metrics.
* **Method:** score() ensures logging; \_score() computes evaluation metrics.

**Logging Strategy**

Structured logging captures comprehensive data throughout each experiment run. Logs are categorized into:

* **Experiment Logs**
* **State and Transition Logs**
* **Prompt and Conversation Logs**
* **Scoring Logs**
* **Code Quality Logs**
* **Error Logs**

Each category has clearly defined triggers and schemas to ensure consistent and analyzable data.

**Unified Log Schemas**

* **ScoringLog:** experiment\_id, round, symbol, final, score, passed, evaluator\_name, evaluator\_version, diagnostics, tests\_total, tests\_passed, all\_tests\_passed, issue\_id, attempt\_number, parent\_attempt\_number, timestamp
* **CodeQualityLog:** experiment\_id, round, symbol, lines\_of\_code, cyclomatic\_complexity, maintainability\_index, lint\_errors, timestamp
* **ConversationLog:** experiment\_id, round, agent\_role, target, content, originating\_agent, intervention, intervention\_type, intervention\_reason, timestamp
* **PromptLog:** experiment\_id, round, system, agent\_id, agent\_role, agent\_engine, symbol, prompt, response, attempt\_number, agent\_action\_outcome, start, stop
* **StateLog:** experiment\_id, system, round, state, action, score, details, timestamp
* **StateTransitionLog:** experiment\_id, round, from\_state, to\_state, reason, timestamp
* **ErrorLog:** experiment\_id, round, error\_type, message, file\_path, timestamp
* **ExperimentLog:** experiment\_id, description, mode, variant, max\_iterations, stop\_threshold, model\_engine, evaluator\_name, evaluator\_version, final\_score, passed, reason\_for\_stop, start, stop

**Comprehensive Trigger Events**

* **ExperimentLog:** Experiment initialization/start, completion/end, errors or exceptions.
* **StateLog:** Entry into and completion of system states; errors during execution.
* **StateTransitionLog:** Transition between states.
* **PromptLog:** Generation attempts by agents, including successes, failures, partial completions, and exceptions.
* **ConversationLog:** Communications between agents, humans, and orchestrators; interventions; exceptions.
* **ScoringLog:** Scoring events, including start, completion of evaluations, and test suite execution; exceptions.
* **CodeQualityLog:** Completion of static code analysis and linting checks; exceptions.
* **ErrorLog:** Any encountered exceptions.

**Evaluation Metrics**

Each experiment evaluates performance against multiple metrics:

* **Bug-fix success rate**
* **Functional correctness**
* **Test pass rates**
* **Maintainability and complexity indices**
* **Linting compliance**
* **Iterations to convergence**
* **Intervention frequencies**
* **Role-specific success rates**
* **Retry and mediation effectiveness**

Detailed evaluation functions and metric computations are provided separately.

**Comprehensive End-to-End Engineering Specification for CodeCritic Experimentation (Part 3)**

**Conducting Experiments**

Follow these structured steps for systematically conducting and managing experiments:

1. **Configure Experiment Parameters**
   * Clearly define and document all experiment settings, including agent configurations, evaluation criteria, and logging preferences.
2. **Execution of Experiments**
   * Invoke the run() method of the Experiment class to initiate the configured experiments.
3. **Data Logging**
   * Verify that all logs are accurately captured and stored in the designated SQLite database and filesystem.
4. **Data Analysis**
   * Utilize provided analysis scripts and notebooks to evaluate and interpret the collected log data and experiment outcomes.

**Reporting Results**

Upon experiment completion, results should be systematically reported using:

* **Evaluation Notebooks:** Clearly documented notebooks to visualize metrics and performance outcomes.
* **Benchmark Comparisons:** Results should be benchmarked against established industry standards and known baselines such as SWE-Bench and HumanEval.

This structured reporting ensures clear visibility of experiment results and facilitates informed decisions on system optimizations.

**Detailed Implementation Plan**

**Phase 1: Infrastructure Setup**

**Tasks:**

* Implement abstract base classes.
* Define and validate folder structure.
* Set up registries and factories.

**Deliverables:**

* Initial class and directory structure.
* Test notebook confirming instantiation and structural correctness.

**Phase 2: FSM Logic and State Management**

**Tasks:**

* Develop finite state machine (FSM) transitions.
* Implement system and state management classes.

**Deliverables:**

* Functional FSM implementation.
* State manager execution logic.
* Test notebook for validating state transitions and logging.

**Phase 3: Agent Logic and Tool Integration**

**Tasks:**

* Implement agent logic for execution.
* Integrate tooling such as black, mypy, radon, ruff, sonarcloud.

**Deliverables:**

* Integrated agent logic with external tools.
* Test notebook demonstrating agent actions and tool outputs.

**Phase 4: Contextual and Prompt Generation**

**Tasks:**

* Implement context providers integrating symbol graphs.
* Develop prompt generators for agent and system prompts.

**Deliverables:**

* Operational context providers.
* Prompt generation modules.
* Test notebook confirming the accuracy of generated contexts and prompts.

**Phase 5: Scoring and Metric Evaluation**

**Tasks:**

* Implement scoring providers.
* Develop evaluation metric computation logic.

**Deliverables:**

* Fully functional scoring mechanisms.
* Test notebook demonstrating accurate computation of evaluation metrics.

**Phase 6: Full System Integration and Validation**

**Tasks:**

* Perform comprehensive integration of all system components.
* Implement multi-agent interaction logic.

**Deliverables:**

* Fully integrated and operational system.
* Final benchmarking notebook detailing system performance.

**Test Notebook Structure per Phase**

1. **Setup**
   * Environment initialization.
2. **Execution**
   * Task and agent execution.
3. **Validation**
   * Verification of states, outputs, and logging integrity.
4. **Metrics**
   * Computation and validation of evaluation metrics.

**Final Benchmark Notebook**

* Comprehensive evaluation and visualization against benchmarks.
* Detailed metrics performance report.

**Recommendations for Codex Implementation**

* Provide explicit, clear examples.
* Ensure structured and consistent logging practices.
* Conduct stepwise implementations to facilitate easy debugging and iterative refinement.

This document concludes the comprehensive guide to executing and optimizing CodeCritic experiments.