Solidity Smart Contract Generator

A User Guide & Technical Documentation

Introduction

The Solidity Smart Contract Generator is an autonomous AI system that converts natural language descriptions into validated, production-ready Solidity smart contracts. It integrates several technologies to achieve this:

- MCP (Model Context Protocol) for structured tool execution
- IBM Agentics and CrewAI for multi-step agent reasoning and orchestration
- Google Gemini 2.0 Flash and an FSM fine-tuned TinyLlama model as language backends
- solcx and eth-tester for compilation and sandbox deployment
- Streamlit for an interactive, web-based user interface

This document describes the system objectives, architecture, pipeline behavior, setup process, extensibility model, and operational details.

System Objectives

The primary goal of the system is to provide an end-to-end pipeline that takes an informal, human-readable description of a smart contract and transforms it into a validated Solidity implementation. To support this, the system aims to:

1. Automate the creation of Solidity contracts from natural language specifications.
2. Ensure generated contracts compile and deploy successfully in a sandbox environment.
3. Provide iterative refinement using structured feedback from the compiler and deployment logs.
4. Produce structured metadata for the contract, such as extracted clauses and logical components.
5. Support multiple language models, including cloud-hosted and fine-tuned local models.
6. Offer a user-friendly web interface suitable for both technical and semi-technical users.
7. Allow straightforward extension by adding new MCP tools for security, analysis, and optimization.

System Architecture

The architecture is organized into three main layers: the MCP Tools Layer, the Agentic Pipeline Layer, and the Streamlit User Interface Layer.

MCP Tools Layer (agents/mcp_tools.py)

The MCP Tools Layer defines the executable tools used by the agent pipeline. Each tool is exposed via the MCP decorator and is designed to be stateless and composable.

The core tools are:

Tool Name	Description
`generate_smart_contract`	Generates Solidity code and clause metadata using Google Gemini 2.0 Flash.
`generate_smart_contract_pretrained`	Generates Solidity code using the FSM fine-tuned TinyLlama model.
`validate_smart_contract`	Compiles the contract with solcx, deploys it using eth-tester, and returns logs.
`refine_contract`	Uses LLM guidance to correct compilation or deployment errors.
`web_search`	Uses DuckDuckGo search for contextual or domain information.

The agentic pipeline uses these tools and is responsible for the core “actions” of generating, validating, refining, and retrieving contextual knowledge.

Agentic Pipeline Layer (agents/agent.py)

The Agentic Pipeline Layer is implemented using Agentics and CrewAI. It defines the multi-step reasoning process that turns user input into a final, validated smart contract.

The pipeline proceeds through several stages:

1. **Generation** – The system calls the generation tool using the selected LLM. The output includes a full Solidity contract, along with structured information such as clauses and high-level explanations.

2. **Validation** – The generated contract is compiled using solcx (Solidity 0.8.20) and then deployed using EthereumTester. Compiler warnings, errors, and deployment logs are collected and structured.

3. **Conditional Refinement** – If validation fails, the system calls the refinement tool. The exact error logs are passed to the LLM along with the previous contract. The LLM returns a corrected version that preserves the original intent while resolving the reported issues.

4. **Re-Validation** – The refined contract is recompiled and deployed. If it succeeds, the contract is marked as validated. If it fails, the refinement cycle can repeat up to a configurable limit.

5. **Finalization** – The pipeline returns a final result consisting of the validated contract, logs, clause metadata, and any refinement history.

The pipeline is configured with a fixed reasoning depth (e.g., 10 reasoning steps) and uses shared memory within CrewAI to maintain context across these stages. Tool routing is handled automatically based on the current pipeline state.

Streamlit User Interface Layer (app.py)

The User Interface Layer is implemented in Streamlit and serves as the main interaction surface for users. It provides:

- A chat-style interface where users can describe the smart contract they want generated.
- Real-time streaming of logs from the agentic pipeline, cleaned from ANSI escape codes for readability.
- Syntax highlighting for Solidity code.
- Indicators that show whether the contract was compiled and deployed successfully.
- Visualization of extracted clauses and key contract components.
- A set of example prompts for quick experimentation.

The UI communicates with the agent pipeline, forwards user prompts, and renders the pipeline's outputs in an accessible and interpretable way.

End-to-End Pipeline Behavior

This section describes the flow from user input to the final, deployable contract.

Input Stage

The user begins by describing the desired contract in natural language through the Streamlit interface. For instance:

> "Generate a mintable and burnable ERC-20 token using OpenZeppelin, with pause functionality and safe initialization of name, symbol, and initial supply."

This description is converted into a structured request object that the agent pipeline uses as input.

Generation Stage

The agent invokes either `generate_smart_contract` or `generate_smart_contract_pretrained`, depending on the selected model via configuration. The output from this stage includes:

- A complete Solidity contract.
- A breakdown of clauses, including roles, state variables, modifiers, events, and key functions.
- Explanations of the primary logic and design decisions.

This information is passed to the next stage for validation.

Validation Stage

The validation stage has two subcomponents: compilation and deployment.

1. **Compilation:** The contract is compiled using solcx targeting Solidity version 0.8.20. The compiler returns ABI, bytecode, and any warnings or errors. These are parsed and structured.

2. **Deployment:** If the compilation succeeds, the contract is deployed to an in-memory EVM instance via EthereumTester. This verifies that the constructor and any initial logic execute without reverting. Deployment errors are captured and attached to the validation result.

If both compilation and deployment succeed, the contract is considered valid, and the pipeline skips to the output stage. If any errors are present, the pipeline proceeds to refinement.

Refinement Stage

When validation fails, the agent enters the refinement stage. The system provides the LLM with:

- The original contract source code.
- The full compiler and/or deployment error logs.
- The original user intent.

The refinement tool generates a corrected version of the contract that addresses the reported issues while maintaining the requested behavior. The refined contract is then returned to the validation stage for re-checking.

Re-Validation Stage

The refined contract goes through the same compilation and deployment process as before. If it succeeds, it is marked as valid. If it fails again, the system may attempt additional refinement iterations depending on the configuration. The goal is to converge on a deployable contract without requiring manual debugging.

Output Stage

Once the pipeline produces a valid contract or reaches its refinement limit, the result is returned to the user. The final output includes:

- The latest version of the Solidity contract.
- Compiler and deployment logs.
- Structured clause information and explanations.
- A status summary indicating whether the contract compiled and deployed successfully, and whether refinement was required.

This information is displayed in the Streamlit UI.

Installation and Local Setup

Cloning the Repository

To begin, clone the repository and change into the project directory:

```
git clone
https://github.com/AgenticsFintekColumbia/smart-contracts.git

cd smart-contracts
```

Creating a Virtual Environment

Create and activate a Python virtual environment (example for Python 3.12):

```
python3.12 -m venv .venv

source .venv/bin/activate
```

Installing Dependencies

This project depends on IBM Agentics. Install it in editable mode, then install the main project:

```
git clone git@github.com:IBM/agentics.git

cd agentics

pip install -e .

cd ..

pip install -e .
```

Alternatively, you may use `pip install -r requirements.txt` after installing Agentics, depending on your environment's structure.

Environment Configuration

The application requires several environment variables. Create a file named `.env` in the project root with contents similar to the following:

```
SELECTED_LLM="gemini"

GEMINI_API_KEY="YOUR_API_KEY"

GEMINI_MODEL_ID="gemini/gemini-2.0-flash"
```

You can adjust `SELECTED_LLM` if you add or support alternative backends, such as the fine-tuned TinyLlama model.

Running the Streamlit Application

To launch the user interface, run:

```
streamlit run app.py
```

Once the app starts, open the displayed URL in your browser. You can then begin describing contracts in natural language and observe the full pipeline in action.

Project Structure

The core structure of the repository is as follows:

```
smart-contracts/
├── app.py                        # Streamlit UI entry point
├── agents/
│   ├── agent.py                # Agentic pipeline orchestration
│   └── mcp_tools.py            # MCP tool definitions and
implementations
├── data/
│   └── pretraining_code_checkpoints/  # FSM TinyLlama pretraining
checkpoints
├── requirements.txt            # Python dependency list
└── README.md                   # High-level project overview
```

Additional configuration files or deployment scripts may be added as the project evolves.

Example Usage Scenarios

The system supports a wide range of smart-contract types. Below are typical examples of prompts that work well.

ERC-20 Token

A user who wants a fungible token could specify:

> "Create a complete ERC-20 token using Solidity and OpenZeppelin. The contract should support minting and burning restricted to the owner, pausing transfers, and safe initialization of name, symbol, and initial supply."

The system will generate a contract using OpenZeppelin templates, incorporate owner-only permissions, and validate the compiled result.

DAO Governance

For decentralized governance:

> "Generate a Solidity-based DAO governance contract that supports proposal creation, weighted voting, automatic proposal lifecycle transitions, and execution of approved proposals once quorum is met."

The resulting contract will define proposal structures, voting mechanisms, and execution paths, and then undergo the same compilation and deployment steps.

NFT Marketplace

For non-fungible tokens:

> "Create a secure Solidity smart contract for an NFT marketplace supporting ERC-721 tokens with listings, purchases, cancellations, and royalty support under ERC-2981."

The system will generate marketplace logic, integrate royalty handling, and verify that the contract deploys correctly.

Feature Summary

The Solidity Smart Contract Generator provides:

- An end-to-end AI pipeline for transforming natural language descriptions into Solidity code.
- Automatic compilation and in-memory deployment testing using solcx and EthereumTester.
- A refinement loop that uses compiler and runtime error logs to fix issues automatically.
- Structured extraction of contract clauses and documentation for interpretability.
- Integration with multiple language model backends, including Google Gemini 2.0 Flash and a fine-tuned TinyLlama model.
- A Streamlit-based web interface with live logs, syntax highlighting and validation status indicators.
- A modular MCP-based tool system that can be extended with custom tools.

Extensibility

Adding New MCP Tools

To extend the system with additional analysis or processing capabilities, you can add new tools to `agents/mcp_tools.py`. Each tool is defined using the MCP decorator. A typical pattern looks like:

```
@mcp.tool(name="my_tool_name")

def my_tool_name(param: str) -> str:

    """Short description of what this tool does."""

    # Implement your logic here

    return "result"
```

Once added, the tool can be integrated into the agent pipeline by updating the agent configuration and specifying when the tool should be invoked.

Common Extension Ideas

Common directions for extension include adding tools for:

- Security auditing (for example, integrating Slither or Mythril for vulnerability scanning).
- Gas cost estimation and optimization suggestions.
- Static analysis and linting using tools such as SolHint.
- Compliance checking against specific ERC or protocol standards.
- Upgradeability analysis to suggest proxy patterns or upgrade-safe architectures.

Troubleshooting

Compilation Issues

If the generated contract fails to compile:

- Confirm that the contract is targeting Solidity 0.8.20, which is the version configured for solcx.
- Ensure that imported libraries (for example, OpenZeppelin) are available and referenced correctly.
- Avoid complex inline assembly or experimental features in the first iteration of generation, unless explicitly needed.

The system will surface compiler error messages directly in the UI and use them in the refinement stage. Still, issues may require adjusting the natural language prompt to be more explicit.

Deployment Issues

If deployment fails in EthereumTester:

- Check that the constructor does not contain logic that reverts based on the initial state.
- Verify that required parameters are correctly initialized.
- Confirm that no unsupported opcodes or environment assumptions are being used.

Deployment errors are also passed to the refinement stage, so the system will attempt to adjust the contract, but certain logic errors may still require prompt modification.

LLM Output Quality

If the generated contract is structurally valid but does not match the desired design:

- Add more specific requirements in the input prompt, such as exact function names, modifiers, or event structures.
- Specify library versions explicitly (for example, “use OpenZeppelin v5.x-compatible patterns”).
- Mention constraints or invariants that the contract must satisfy to provide the LLM with clearer guidance.

Deployment Information

The project is also configured for deployment under the following parameters:

- Project slug: `smart-contracts`
- Deployment URL: `https://d28ay7eykuuini.cloudfront.net/smart-contracts`
- Main entry file: `app.py`

These values may be used in an automated deployment pipeline (for example, via CI/CD or container orchestration).

Team and Acknowledgments

This project is developed by:

- Chaitya Shah | MS in Data Science, Columbia University
- Chunghyun Han | MS in Operations Research, Columbia University
- Nami Jain | MS in Data Science, Columbia University
- Yegan Dhaivakumar | MS in Data Science, Columbia University

Faculty advisors:

- Alfio Gliozzo | Chief Science Catalyst, IBM Research
- Agostino Capponi | Professor, Columbia University

Future Directions

Planned or potential future enhancements include:

- Integration of automated smart contract security auditing pipelines.
- Fuzz testing and property-based testing of generated contracts.
- Gas analysis dashboards within the UI.
- Support for generating multi-contract systems or full dApp backends.
- Integration with public test networks such as Goerli or Sepolia for real testnet deployments.
- Enhanced visualization of contract architecture and call graphs.