BAB 3
METODOLOGI PENELITIAN

3.1 Metodologi Penelitian

Tahapan dalam Benchmarking Kontrak Cerdas yang Dihasilkan oleh Kecerdasan Buatan untuk DeFi adalah sebagai berikut:

1. Studi literatur
   Pada tahap ini melibatkan pembelajaran terhadap literatur yang relevan dengan *benchmarking* kontrak cerdas secara otomatis untuk DeFi, jenis kontrak cerdas DeFi dengan standar ERC untuk skenario DeFi dan pola penghematan *gas*. Tahap ini membantu dalam memahami konsep dasar, pendekatan dan teknik yang digunakan dalam *benchmarking* kontrak cerdas.

2. Penentuan skenario DeFi dan kriteria *benchmarking*
   Pada tahap ini melibatkan penentuan skenario DeFi untuk menentukan jenis kontrak cerdas *Solidity* yang akan dihasilkan oleh kedua model tersebut. Skenario DeFi yang ditentukan juga berdasarkan standar ERC yang tertera pada landasan teori. Lalu, menentukan sejumlah kriteria berdasarkan skenario dan pola penghematan *gas* yang akan menjadi faktor skor untuk evaluasi nanti.

3. Perancangan desain dan *workflow* program
   Pada tahap ini dirancang desain dan workflow dari program *benchmarking* yang akan dilakukan. Rancangan program tersebut dibagi menjadi dua, yaitu rancangan terhadap penghasilan kontrak cerdas secara otomatis dan rancangan terhadap sistem evaluasi untuk skor.

4. Implementasi LLM
   Pada tahap ini diimplementasikan kedua LLM ke dalam program *benchmarking*, yaitu model Code-LLaMa dan model Code-LLaMa - Python agar dapat menghasilkan kontrak cerdas secara otomatis menggunakan *library Python*.

5. Menghasilkan kontrak cerdas
   Pada tahap ini dihasilkan sejumlah kontrak cerdas berdasarkan skenario DeFi yang telah ditentukan menggunakan *Jupyter Notebook*. Kontrak cerdas yang

dihasilkan oleh kedua model akan di *export* menjadi .csv dan dievaluasi kemudian.

6. Evaluasi kontrak cerdas
   Pada tahap ini dievaluasi kontrak cerdas yang sudah dihasilkan berdasarkan skenario dan kriteria yang ditentukan. Metode yang digunakan untuk evaluasi kriteria adalah *Regex*. Program akan memberikan skor untuk setiap kriteria yang terpenuhi.

7. Dokumentasi
   Pada tahap ini mendokumentasi setiap proses penelitian mulai dari penentuan skenario dan kriteria, pengembangan program, penghasilan kontrak cerdas, evaluasi, hingga selesai.

8. Konsultasi
   Pada tahap ini konsultasi dengan dosen pembimbing mengenai proses dan hasil pengerjaan penelitian dengan tujuan untuk membahas dan mencari saran atau masukan demi kelancaran proses penelitian.

3.2 Penentuan Skenario DeFi

Penentuan skenario atau uji kasus yang relevan dengan DeFi dibutuhkan untuk menentukan jenis kontrak cerdas yang ingin dihasilkan dan metode-metode fondasi yang diperlukan pada perancangan program *benchmarking* karena program ini hanya tertuju pada DeFi. Kegiatan yang termasuk pada DeFi sangat beragam. Pada penelitian ini, ditentukan tiga skenario umum pada DeFi untuk menjadi skenario uji kasus, yang dapat dilihat pada Tabel 3.1.

Tabel 3.1. Skenario DeFi

Skenario DeFi	Jumlah Percobaan	
	Code-LLaMa	Code-LLaMa - Python
Kontrak cerdas untuk membuat *token* yang disebut "*Xian Yearn Finance*" (XYF) dengan 18 desimal. Tujuan dari kontrak cerdas ini adalah untuk memungkinkan pemegang *token* untuk menyetor, menarik, dan bekerja dengan *token* tersebut.	20	20
Kontrak cerdas untuk memungkinkan penyetoran dan penarikan *token* ke/dari *pool Balancer*. Kontrak cerdas harus mengimplementasikan fungsi-fungsi yang diperlukan oleh antarmuka *InteractiveAdapter* dan *ProtocolAdapter*.	20	20
Kontrak cerdas untuk memfasilitasi peminjaman dan peminjaman *token*. Tujuan dari kontrak ini adalah untuk menyediakan *platform* peminjaman yang terdesentralisasi di mana pengguna dapat meminjam dan membayar *token*. Ini juga mencakup fungsionalitas untuk membeli dan menjual *token* dengan biaya tertentu.	20	20

Berdasarkan jumlah subjek yang disarankan dari penelitian yang bersifat kuantitatif [39], penelitian ini akan menghasilkan sebanyak 120 kontrak cerdas, 40 setiap skenario, 20 setiap model akan dihasilkan berdasarkan skenario DeFi tersebut. Kontrak cerdas yang dihasilkan secara otomatis adalah subjek utama yang akan dievaluasi dalam program benchmark untuk menentukan kinerja LLM.

3.3 Penentuan Kriteria Benchmarking

Pertimbangan pertama untuk menentukan kriteria program *benchmarking* adalah mengevaluasi *source code* kontrak cerdas. Lalu, mengevaluasi efisiensi *gas* kontrak cerdas yang dihasilkan secara otomatis berdasarkan pola penghematan desain *gas* dan standar ERC kontrak cerdas masing-masing untuk kriteria lainnya. Hasil dari *benchmarking* akan di *output* dalam bentuk sistem penilaian berbobot, yang menghasilkan skor rata-rata dari *benchmarking* kontrak cerdas. Kriteria yang menentukan keseluruhan sistem benchmarking dapat dilihat pada Tabel 3.2.

Tabel 3.2. Kriteria *Benchmarking* Kontrak Cerdas

Kriteria	**Nilai Kemungkinan**
Kelengkapan *source code*	Lengkap, Sebagian, Gagal
Pengepakan variabel	Ya, Tidak
Pengepakan boolean	Ya, Tidak
Tipe penyimpanan	*uint**, *uint256*, Campuran
Tipe data	*Mapping*, *Array*, Campuran
Tipe ukuran variabel	*Fixed*, *Dynamic*
Nilai *default*	*Default*, *Initialized*
functions	*Public*, *External*, Campuran
Pembatasan penyimpanan	Standar penyimpanan ERC
Minimalisir data *on-chain*	Standar penyimpanan ERC

Metode penilaian *benchmarking* ini adalah dengan meningkatkan skor setiap kriteria yang dipenuhi hingga maksimum 100. Setiap kriteria memiliki bobot sebesar 0,1 dengan total bobot penjumlahan 1 yang diinisialisasikan ke dalam sebuah variabel untuk modifikasi pembobotan nanti. Kriteria yang ditentukan dijelaskan sebagai berikut:

- Kelengkapan *source code*: Menentukan apakah LLM menghasilkan *source code* lengkap atau tidak. Kriteria ini memiliki tiga kemungkinan nilai. kontrak cerdas yang dibuat sepenuhnya (100), kontrak cerdas yang dibuat hanya sebagian (50), dan yang menghasilkan generasi yang gagal (0).

- Pengepakan variabel: Pengepakan atau pengemasan variabel adalah proses mengatur variabel sehingga lebih dari satu variabel dapat muat dalam satu

slot. Slot 32 *byte* (256 bit) digunakan untuk penyimpanan dalam kontrak cerdas *Solidity*. Mendeklarasi *uint256* sebelum *uint8* dihitung sebagai variabel yang tidak dikemas. Kriteria ini memiliki dua kemungkinan nilai. kontrak cerdas yang dihasilkan mengemas variabelnya (100) dan tidak mengemas variabel (0).

- Pengepakan boolean: Variabel boolean direpresentasikan sebagai *uint8* (*unsigned integer of 8 bits*) di *Solidity*. Menyimpannya hanya membutuhkan satu bit. Mendeklarasi boolean menggunakan sebuah *function* dihitung sebagai boolean yang dikemas. Kriteria ini memiliki dua kemungkinan nilai. Kontrak cerdas yang dihasilkan menggunakan sistem *packing* boolean (100) dan tidak menggunakan sistem *packing* boolean (0).

- Tipe penyimpanan: EVM menjalankan 256 bit pada satu waktu, sehingga menggunakan ukuran *unsigned integer* yang lebih kecil dari 256 bit lebih murah daripada menetapkan *uint256* secara langsung (menetapkan *uint256* tanpa angka merupakan *uint256*). Kriteria ini memiliki tiga kemungkinan nilai. Kontrak cerdas yang dihasilkan menggunakan bilangan bulat tidak bertanda tangan kecil di bawah 256 bit (100), menggunakan 256 bit tetapi juga menggunakan bilangan bulat tidak bertanda tangan campuran di bawahnya (50), dan setiap variabel hanya menggunakan 256 bit (0).

- Tipe data: *Solidity* memiliki dua tipe data, *Mapping* dan *array*. Tipe data *mapping* lebih murah daripada *array*. Kriteria ini memiliki tiga kemungkinan nilai. Kontrak cerdas yang dihasilkan hanya menggunakan *mapping* untuk tipe datanya (100), menggunakan tipe data campuran (50), dan hanya menggunakan *array* (0).

- Tipe ukuran variabel: Ukuran *array* tetap lebih murah daripada menggunakan *array* dinamis. Kriteria ini memiliki dua kemungkinan nilai. Kontrak cerdas yang dihasilkan hanya menggunakan ukuran *array* tetap (100) dan hanya menggunakan ukuran *array* dinamis (0).

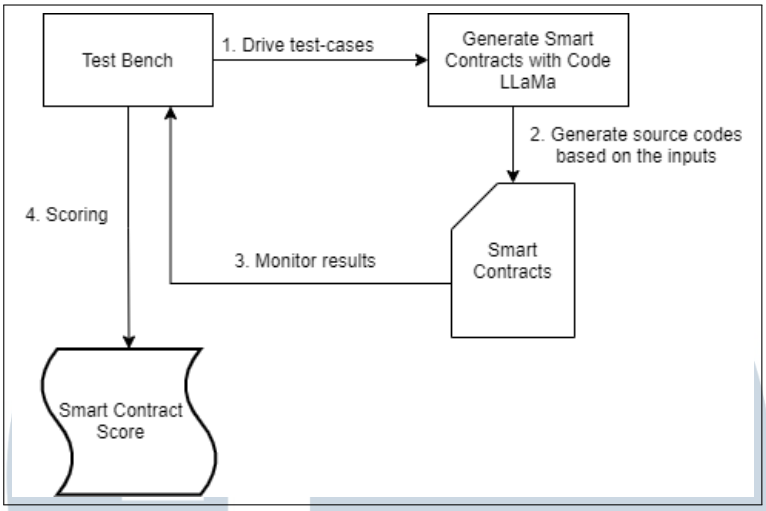
- Nilai *default*: Menginisialisasi semua variabel pada saat pembuatan adalah praktik yang direkomendasikan dalam rekayasa perangkat lunak. Tetapi di *Ethereum* membutuhkan biaya. Variabel yang dideklarasi dan tidak diberikan nilai otomatis akan mempunyai nilai 0. Kriteria ini memiliki dua kemungkinan nilai. Kontrak cerdas yang dihasilkan menggunakan

nilai *default* untuk variabel jika nilai yang dibutuhkan adalah 0 pada saat mendeklarasi (100) dan mendeklarasi variabel dan memberikan nilai 0 (0).

- *functions*: *Solidity* menyalin argumen *array* ke memori secara instan dalam fungsi publik, tetapi fungsi eksternal dapat membaca langsung dari *calldata*. Meskipun pengalokasian memori mahal, pembacaan dari *calldata* murah. Kriteria ini memiliki tiga kemungkinan nilai. Kontrak cerdas yang dihasilkan hanya menggunakan fungsi eksternal (100), menggunakan fungsi campuran (50), dan hanya menggunakan fungsi publik (0).
- Pembatasan penyimpanan: Karena penyimpanan adalah jenis memori yang paling mahal, yang terbaik adalah menggunakan sesedikit mungkin. Nilai kriteria ditentukan oleh standar ERC kontrak cerdas skenario DeFi masing-masing.
- Meminimalkan data *on-chain*: Meminimalkan data *on-chain* bermanfaat karena penyimpanan membutuhkan biaya yang sangat tinggi. Hanya data penting yang harus disimpan secara *on-chain*; semua data lainnya harus disimpan secara *off-chain*. Nilai kriteria ditentukan oleh standar ERC kontrak cerdas skenario DeFi masing-masing.

3.4 Workflow Program Benchmarking

Membuat dan mengevaluasi kontrak cerdas dapat dilakukan secara manual, tetapi risiko kesalahan manusia itu tinggi dan tidak akan efisien jika pembuatan dan evaluasi dilakukan secara tidak benar. Maka dari itu, dirancang sebuah program khusus untuk penelitian ini, sebuah program sederhana menggunakan bahasa *Python* untuk menghasilkan dan mengevaluasi kontrak cerdas DeFi secara otomatis. Program ini hanya membutuhkan *Jupyter Notebook*, sebuah aplikasi web *open-source*. *Data scientist* dapat menghasilkan dan mendistribusikan dokumen yang berisi persamaan, kode langsung, dan sumber daya multimedia lainnya. Alur kerja *testbench* dapat dilihat pada Gambar 3.1.

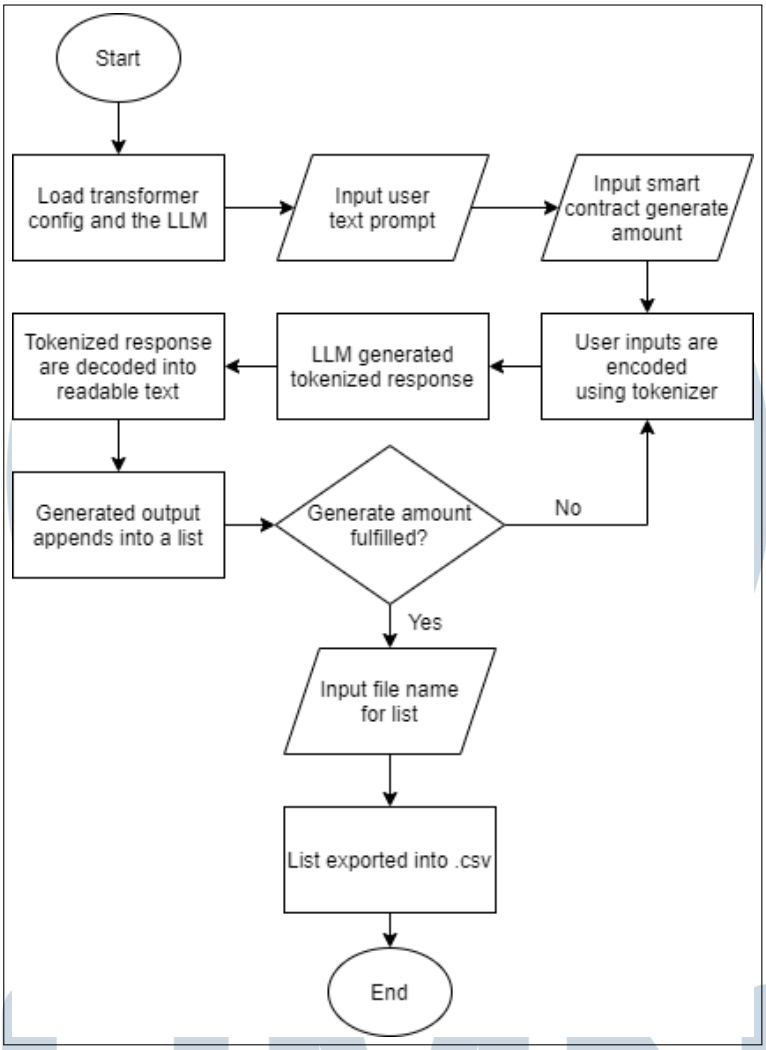


Gambar 3.1. *Workflow* Program *Benchmarking* Secara Keseluruhan

Pertama memasukkan uji coba input tiga skenario DeFi yang berbeda ke dalam LLM terlebih dahulu. LLM kemudian mencoba menghasilkan kontrak cerdas *Solidity* yang berbeda secara otomatis berdasarkan input kalimat atau masukan dari skenario. Setelah LLM selesai menghasilkan kontrak cerdasnya, seluruh kontrak cerdas tersebut dievaluasi dalam program *benchmarking* menggunakan kriteria yang telah ditetapkan. Kontrak cerdas dievaluasi berdasarkan jumlah set *regex* dari kriteria dari program *benchmark*. Terakhir, *testbench* menghitung skor berdasarkan kriteria yang dipenuhi dan mengeluarkan skor total untuk menentukan kinerja LLM dalam menghasilkan kontrak cerdas skenario DeFi.

3.4.1 Menghasilkan Kontrak Cerdas

Proses implementasi dan penghasian kontrak cerdas dapat dilihat pada Gambar 3.2.

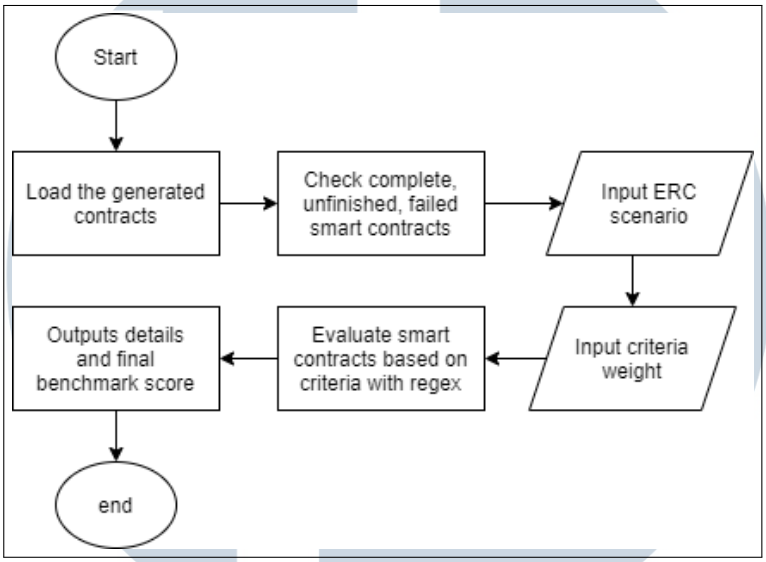


Gambar 3.2. Proses Penghasilan Kontrak Cerdas Secara Otomatis

Bagian pertama dari desain program ini adalah menghasilkan kontrak cerdas secara otomatis menggunakan LLM. Model pertama kali dimuat secara lokal atau melalui *Cloud*. Setelah model dimuat, pengguna kemudian diminta untuk memasukkan *prompt* teks dan berapa banyak kontrak cerdas yang ingin dihasilkan dalam satu sesi. Kemudian input dikodekan ke dalam *token* menggunakan *tokenizer* agar LLM dapat menghasilkan respons. LLM menghasilkan respons *token* yang kemudian diterjemahkan kembali ke dalam teks yang dapat dibaca oleh pengguna. Program ini akan terus menghasilkan respons lain berdasarkan berapa banyak yang ingin dihasilkan oleh pengguna dalam satu sesi. Terakhir, kontrak cerdas yang dihasilkan kemudian dimasukkan ke dalam fitur *list* dalam *Python* untuk diekspor oleh program ke dalam file .csv, yang kemudian akan dievaluasi di bagian kedua dari program *benchmark* tersebut.

3.4.2 Benchmarking Kontrak Cerdas

Proses *benchmarking* atau evaluasi kontrak cerdas dapat dilihat pada Gambar 3.3.



Gambar 3.3. Proses Evaluasi Kontrak Cerdas Secara Otomatis

Bagian kedua dan terakhir dari desain program adalah untuk mengevaluasi kontrak cerdas yang dihasilkan secara otomatis oleh LLM. Pertama, kontrak cerdas yang dihasilkan dimuat ke dalam program sebagai *list*. Setelah dimuat, secara manual memeriksa *source code* kontrak cerdas yang selesai, belum selesai, dan gagal untuk menandainya dengan kata kunci. Kemudian, pengguna diminta untuk memilih skenario ERC dan menetapkan bobot nilai kriteria untuk evaluasi di program *benchmarking*. Program ini kemudian mengevaluasi kontrak cerdas berdasarkan kriteria dengan set *regex* yang telah ditentukan. Terakhir, program menghitung skor *benchmark* dan mengeluarkan detil lengkap dan skor akhir yang menentukan kinerja LLM.

3.5 Metode Implementasi Code-LLaMa dan Code-LLaMa Python

Setelah mengumpulkan data dan mendesain seluruh program, proses pembuatan kontrak cerdas secara otomatis dilakukan dengan memanfaatkan generasi autoregresif. Model yang digunakan untuk penelitian ini adalah 7 miliar parameter Code LLaMa dan Code LLaMa - Python.

Penelitian ini menggunakan *Jupyter Notebook*. Sebuah *platform* yang

meminjam GPU *Backend Engine* dari *Google* untuk menghasilkan kontrak cerdas. Karena menghasilkan kontrak cerdas membutuhkan memori, dibutuhkan *library* dari *Python* untuk konfigurasi kuantisasi dan mengefisiensikan penggunaan memori, yaitu *BitsandBytes* (BnB) yang digunakan untuk konfigurasi kuantisasi. Pengaturan parameter BnB berdasarkan pengaturan pelatihan yang disesuaikan oleh Eduardo Muñoz [40]. Konfigurasi kuantisasi parameter BnB dapat dilihat pada Tabel 3.3.

Tabel 3.3. Konfigurasi kuantisasi *BitsandBytes* untuk model

Parameter	**Nilai**
use_4bit	True
bnb_4bit_compute_dtype	Float16
bnb_4bit_quant_type	nf4
use_double_nested_quant	True
load_in_8bit_fp32_cpu_offload	True

Model kemudian dimuat ke dalam program dengan *library transformer* dan diatur dengan pengaturan parameter BnB. *Tokenizer* digunakan untuk memproses input teks agar LLM dapat menghasilkan respons. Parameter untuk *tokenizer* dapat dilihat pada Tabel 3.4.

Tabel 3.4. Parameter *tokenizer* model untuk menghasilkan respon

Parameter	**Nilai**
return_tensors	pt
truncation	True

Ketika menghasilkan banyak teks dalam satu sesi, LLM pasti diharapkan untuk menghasilkan *output* yang berbeda. Berikut adalah parameter generasi respon untuk model untuk mencapai hal tersebut yang dapat dilihat pada Tabel 3.5.

Tabel 3.5. Parameter generasi respon untuk generasi hasil yang berbeda

Parameter	**Nilai**
do_sample	True
max_new_tokens	2000
top_p	0.9
temperature	0.9
pad_token_id	tokenizer.eos_token_id

3.6 Metode Implementasi Kriteria Evaluasi

Code LLaMa dan Code LLaMa - Python telah diimplementasikan untuk menghasilkan kontrak cerdas secara otomatis, model-model tersebut diuji dari tiga skenario DeFi yang berbeda yang telah ditentukan pada Tabel 3.1. Setelah model menghasilkan kontrak cerdas, kontrak tersebut akan dievaluasi melalui program *benchmark* dengan kriteria yang ditetapkan dari pola desain penghematan *gas* pada Tabel 3.2 yang akan menentukan kinerja LLM terhadap kontrak cerdas DeFi yang dihasilkan secara otomatis.

Tiga skenario yang akan diuji oleh model tersebut adalah pembuatan *cryptotoken*, brankas *tokenized*, dan *flash loan*. Masing-masing skenario ini mewakili standar ERC yang berbeda. Skenario pertama mewakili ERC-20, skenario kedua mewakili ERC-4626, dan skenario terakhir mewakili ERC-3156. Model-model ini diharapkan dapat menghasilkan kontrak cerdas secara otomatis yang memenuhi masing-masing standar ERC.

Tabel 3.6. Skenario DeFi yang akan diuji Model

	Tujuan	Hasil Ekspektasi	Tipe ERC
1	Membuat *cryptotoken* bernama ”Xian Yearn Finance” (XYF)	Kontrak cerdas *cryptotoken* yang memenuhi kriteria pola penghematan *gas* dan memenuhi standar ERC-20	ERC-20
2	Memungkinkan penyetoran dan penarikan *token* kedari *pool Balancer*	Kontrak cerdas brankas *tokenized* yang memenuhi kriteria pola penghematan *gas* dan memenuhi standar ERC-4626	ERC-4626
3	Memfasilitasi peminjaman dan peminjaman *token*	Kontrak cerdas *flash loan* yang memenuhi kriteria pola penghematan *gas* dan memenuhi ERC-3156	ERC-3156

Metode yang digunakan pada kriteria dalam *benchmarking* kontrak cerdas DeFi adalah dengan *regex*, kecuali kriteria pertama, yaitu pengecekan kelengkapan *source code*. Pada kriteria ini dimasukkan input manual dengan menambahkan kata kunci ke kontrak cerdas DeFi yang belum selesai dan gagal. Kemudian, kriteria kedua hingga kedelapan, yaitu pengepakan variabel, pengepakan boolean, tipe penyimpanan, tipe data, tipe ukuran variabel, nilai *default*, dan *functions* akan menggunakan *regex* yang sama untuk mengevaluasi semua kontrak cerdas skenario DeFi. Kata kunci kriteria pengecekan *source code* dapat dilihat pada Tabel 3.7.

Tabel 3.7. Kata kunci untuk Kriteria Pertama

Parameter	**Nilai**
source code lengkap	none
source code tidak lengkap	UNFINISHED_SC
Gagal menghasilkan *source code*	NOT_A_SC

Kumpulan *regex* dari kriteria umum yang akan digunakan untuk mengevaluasi semua kontrak cerdas skenario DeFi dapat dilihat pada Kode 3.1.

```
#Packing variable
(uint|uint(256))+ public [A-Za-z0-9]+ = [0-9]+;(\n)+\s+uint
    (8|16|32|128)

#Packing booleans
(uint|uint[0-9])+\([0-9]\)

#Storage type (Two regex)
(uint|uint256) public
uint(?!(?:256))\d+ public

#Data type (Two regex)
mapping\([^)]*\) public [A-Za-z0-9]+;
[A-Za-z0-9]+\[\] public [A-Za-z0-9]+;

#Variable size type (Two regex)
(uint|string|int)\[[0-9]+\]
[A-Za-z0-9]+\[\]

#Default value
public [A-Za-z0-9]+ = 0;

#functions (Two regex)
function [A-Za-z0-9]+\([^)]*\) external
function [A-Za-z0-9]+\([^)]*\) public
```

Kode 3.1: *Regex* untuk kriteria umum kontrak cerdas

Beberapa kriteria hanya membutuhkan satu *regex*, sementara beberapa kriteria lainnya membutuhkan dua *regex*. *Regex* akan dijelaskan lebih lanjut di bawah ini untuk memahami lebih mendalam kegunannya:

- *Regex* Pengepakan Variabel: *Regex* mencari variabel penyimpanan yang tidak dikemas. Contohnya, *uint256* atau *uint256* default dideklarasi sebelum ukuran yang lebih kecil dari *uint** seperti *uint8*.
- *Regex* Pengepakan Boolean: *Regex* ini mencari variabel boolean yang menggunakan fungsi packing boolean, yang dapat diidentifikasi dengan mencari variabel boolean dengan tanda kurung seperti *uint256()*
- *Regex* Jenis Penyimpanan: *Regex* pertama mencari variabel penyimpanan apapun yang menggunakan *uint256* atau *uint* yang dideklarasi, sedangkan regex kedua mencari *uint256* dengan ukuran lebih kecil di bawah 256 bit.

- *Regex* Tipe Data: *Regex* pertama mencari penggunaan *mapping* untuk tipe data, sedangkan *regex* kedua mencari penggunaan *array* untuk tipe data.
- *Regex* Tipe Ukuran Variabel: *Regex* pertama mencari variabel penyimpanan tetap atau *fixed*, sedangkan *regex* kedua mencari variabel penyimpanan dinamis.
- *Regex* Nilai *Default*: *Regex* ini mencari variabel apapun yang semestinya tidak perlu diinisialisasi dengan nilai *default*, yaitu nol.
- *Regex function*: *Regex* pertama mencari *function* apapun yang menggunakan pemanggilan eksternal, sedangkan *regex* kedua mencari *function* apa pun yang menggunakan pemanggilan publik.

Skor ditentukan oleh bagaimana kontrak cerdas memenuhi kriteria yang tertera di atas. *Regex* pembatasan penyimpanan dan meminimalkan data *on-chain* akan dijelaskan di setiap skenario karena ini adalah evaluasi berbasis standar ERC, yang menentukan apakah smart contract memenuhi standar ERC untuk penyimpanan dan data *on-chain*.

3.6.1 Skenario 1: Token ”Xian Yearn Finance” (XYF)

Dalam skenario ini, model harus menghasilkan kontrak cerdas yang menciptakan *token* yang disebut ”*Xian Yearn Finance*” (XYF). Tujuan dari kontrak cerdas adalah untuk memungkinkan pemegang *token* untuk menyetor, menarik, dan bekerja dengan *token* tersebut. Persediaan awal *token* diatur ke 1000 *token* XYF.

Kontrak cerdas dari skenario ini dievaluasi dengan set *regex* dari kriteria pada Kode 3.1 dan juga berdasarkan standar penyimpanan dan data *on-chain* ERC-20 dengan bobot yang sama, yaitu 0,1. *Regex* yang digunakan untuk membatasi penyimpanan dan meminimalkan data *on-chain* pada skenario yang akan dievaluasi dapat dilihat pada Kode 3.2 dan Kode 3.3.

```
#Daftar Regex (Total: 11)
function name\([^)]*\)
function symbol\([^)]*\)
function decimal\([^)]*\)
function totalSupply\([^)]*\)
function balanceOf\([^)]*\)
function allowance\([^)]*\)
function transfer\([^)]*\)
function approve\([^)]*\)
function transferFrom\([^)]*\)
event Transfer\([^)]*\)
event Approval\([^)]*\)
```

Kode 3.2: *Regex* untuk Standar Penyimpanan ERC-20

```
#Regex pencarian function diluar standar
function (?!totalSupply|_totalSupply|balanceOf|_balanceOf|
    allowance|_allowance|transfer|_transfer|approve|_approve|
    transferFrom|_transferFrom|decimal|_decimal|symbol|_symbol|name
    |_name)\w+\([^)]*\)

#Regex pencarian event diluar standar
event (?!Approval|Transfer)\w+\([^)]*\)
```

Kode 3.3: *Regex* untuk Mencari Data on-chain diluar standar ERC-20

Regex standar penyimpanan ERC-20 didasarkan pada standar inti ERC-20 dari Tabel 2.1 dan Tabel 2.2. *Regex* ini mencari *function* dan *event* yang disebutkan. Total skor untuk semua standar *function* dan *event* adalah 100 sehingga setiap *function* atau *event* yang hilang akan mengurangi skor. *Regex* untuk meminimalisasi data *on-chain* mencari *function* dan *event* yang berlebihan yang bukan merupakan standar inti ERC-20. Dalam penelitian ini, hanya 10 data berlebihan yang diperbolehkan. Skor awal dari kriteria ini adalah 100, jadi setiap data *on-chain* yang berlebihan akan mengurangi skor sebesar 10.

3.6.2 Skenario 2: Menyetor dan Menarik Token

Dalam skenario ini, model harus menghasilkan kontrak cerdas yang memungkinkan pengguna untuk menyetor *token* ke *pool Balancer* menggunakan *function* `deposit` dan menarik *token* dari *pool Balancer* menggunakan *function* `withdraw`. Kontrak cerdas juga harus menyediakan kemampuan untuk mendapatkan saldo *token* yang terkunci di *pool Balancer* oleh akun tertentu

menggunakan *function* `getBalance`. *Prompt* yang digunakan untuk menghasilkan kontrak cerdas berdasarkan skenario ini dapat dilihat pada Tabel 3.9.

Kontrak cerdas dari skenario ini dievaluasi dengan set *regex* dari kriteria pada Kode 3.1 dan juga berdasarkan standar penyimpanan dan data *on-chain* ERC-4626 dengan bobot yang sama, yaitu 0,1. *Regex* yang digunakan untuk membatasi penyimpanan dan meminimalkan data *on-chain* pada skenario yang akan dievaluasi dapat dilihat pada Kode 3.4 dan Kode 3.5.

```
#Daftar Regex (Total: 18)
function asset\([^)]*\)
function totalAssets\([^)]*\)
function convertToShares\([^)]*\)
function convertToAssets\([^)]*\)
function maxDeposit\([^)]*\)
function previewDeposit\([^)]*\)
function deposit\([^)]*\)
function maxMint\([^)]*\)
function previewMint\([^)]*\)
function mint\([^)]*\)
function maxWithdraw\([^)]*\)
function previewWithdraw\([^)]*\)
function withdraw\([^)]*\)
function maxRedeem\([^)]*\)
function previewRedeem\([^)]*\)
function redeem\([^)]*\)
event Deposit\([^)]*\)
event Withdraw\([^)]*\)
```

Kode 3.4: *Regex* untuk Standar Penyimpanan ERC-4626

```
#Regex pencarian function diluar standar
function (?!asset|totalAssets|convertToShares|convertToAssets|
    maxDeposit|previewDeposit|deposit|maxMint|previewMint|mint|
    maxWithdraw|previewWithdraw|withdraw|maxRedeem|previewRedeem|
    redeem)\w+\([^)]*\)

#Regex pencarian event diluar standar
event (?!Deposit|Withdraw)\w+\([^)]*\)
```

Kode 3.5: *Regex* untuk Mencari Data on-chain diluar standar ERC-4626

Regex standar penyimpanan ERC-4626 didasarkan pada standar inti ERC-4626 dari Tabel 2.3 dan Tabel 2.4. *Regex* ini mencari *function* dan *event* yang disebutkan. Total skor untuk semua standar *function* dan *event* adalah 100 sehingga

setiap *function* atau *event* yang hilang akan mengurangi skor. *Regex* untuk meminimalisasi data *on-chain* mencari *function* dan *event* yang berlebihan yang bukan merupakan standar inti ERC-4626. Dalam penelitian ini, hanya 10 data berlebihan yang diperbolehkan. Skor awal dari kriteria ini adalah 100, jadi setiap data *on-chain* yang berlebihan akan mengurangi skor sebesar 10.

3.6.3 Skenario 3: Meminjamkan dan Meminjam Token

Dalam skenario ini, model harus menghasilkan kontrak cerdas yang menyediakan *platform* peminjaman terdesentralisasi di mana pengguna dapat meminjam dan membayar *token*. Ini juga mencakup fungsionalitas untuk membeli dan menjual *token* dengan biaya tertentu. Kontrak tersebut memiliki fitur tata kelola untuk memungkinkan pengelolaan *platform*. Kontrak memastikan keamanan transfer *token* melalui fungsi transfer yang aman.

Kontrak cerdas dari skenario ini dievaluasi dengan set *regex* dari kriteria pada Kode 3.1 dan juga berdasarkan standar penyimpanan dan data *on-chain* ERC-3156 dengan bobot yang sama, yaitu 0,1. *Regex* yang digunakan untuk membatasi penyimpanan dan meminimalkan data *on-chain* pada skenario yang akan dievaluasi dapat dilihat pada Kode 3.5 dan Kode 3.6.

```
#Daftar Regex (Total: 4)
function maxFlashLoan\([^)]*\)
function flashFee\([^)]*\)
function flashLoan\([^)]*\)
function onFlashLoan\([^)]*\)
```

Kode 3.6: *Regex* untuk Standar Penyimpanan ERC-3156

```
#Regex pencarian function diluar standar
function (?!maxFlashLoan|flashFee|flashLoan|onFlashLoan)\w+\([^)]*\)

#Regex pencarian event diluar standar
event [A-Za-z0-9]+\([^)]*\)
```

Kode 3.7: *Regex* untuk Mencari Data on-chain diluar standar ERC-3156

Regex standar penyimpanan ERC-3156 didasarkan pada standar inti ERC-3156 dari Tabel 2.3 dan Tabel 2.4. *Regex* ini mencari *function* yang disebutkan. Total skor untuk semua standar *function* adalah 100 sehingga setiap *function* yang hilang akan mengurangi skor. *Regex* untuk meminimalisasi data *on-chain* mencari *function* dan *event* yang berlebihan yang bukan merupakan standar inti ERC-3156.

Dalam penelitian ini, hanya 10 data berlebihan yang diperbolehkan. Skor awal dari kriteria ini adalah 100, jadi setiap data *on-chain* yang berlebihan akan mengurangi skor sebesar 10.

3.7 Testing

Setelah implementasi LLM dan kriteria evaluasi sudah diterapkan, agar model dapat menghasilkan kontrak cerdas berdasarkan skenario-skenario tersebut, dibutuhkan sebuah input teks berupa bahasa manusia yang menggambarkan skenario yang ingin dihasilkan. *Prompt* yang digunakan untuk menghasilkan kontrak cerdas berdasarkan setiap skenario ini dapat dilihat pada Tabel 3.8 hingga Tabel 3.10.

Tabel 3.8. *Prompt* Pengguna untuk Skenario 1

Input
<<SYS>>This is a code generation. <<SYS>>Please write a full solidity smart contract to create a token called ”Xian Yearn Finance” (XYF). The purpose of the smart contract is to allow the token holders to deposit, withdraw, and work with their tokens. The initial supply of the token is set to 1000 XYF tokens. using pragma solidity 0.8.0;

Tabel 3.9. *Prompt* Pengguna untuk Skenario 2

Input
<<SYS>>This is a code generation. <<SYS>>Please write a full solidity smart contract to enable depositing and withdrawing tokens to/from a Balancer pool. The smart contract should implement the functions required by the InteractiveAdapter and ProtocolAdapter interfaces. It should allow users to deposit tokens to a Balancer pool using the ‘deposit‘ function and withdraw tokens from the Balancer pool using the ‘withdraw‘ function. The smart contract should also provide the ability to get the balance of a token locked on the Balancer pool by a specific account using the ‘getBalance‘ function. using pragma solidity 0.8.0;

Tabel 3.10. *Prompt* Pengguna untuk Skenario 3

Input
<<SYS>>This is a code generation. <<SYS>>Please write a full solidity smart contract to facilitate borrowing and lending of tokens. The purpose of the contract is to provide a decentralized lending platform where users can borrow and repay tokens. It also includes functionality for buying and selling tokens at a specified rate. The contract has governance features to allow for the management of the platform. The contract ensures the safety of token transfers through safe transfer functions. using pragma solidity 0.8.0;

Kontrak cerdas yang dihasilkan tersebut akan diubah menjadi bentuk file .csv untuk program *benchmark* evaluasi nanti.