



# CoqPilot

a plugin for LLM-based generation of proofs

Andrei Kozyrev

Gleb Solovev

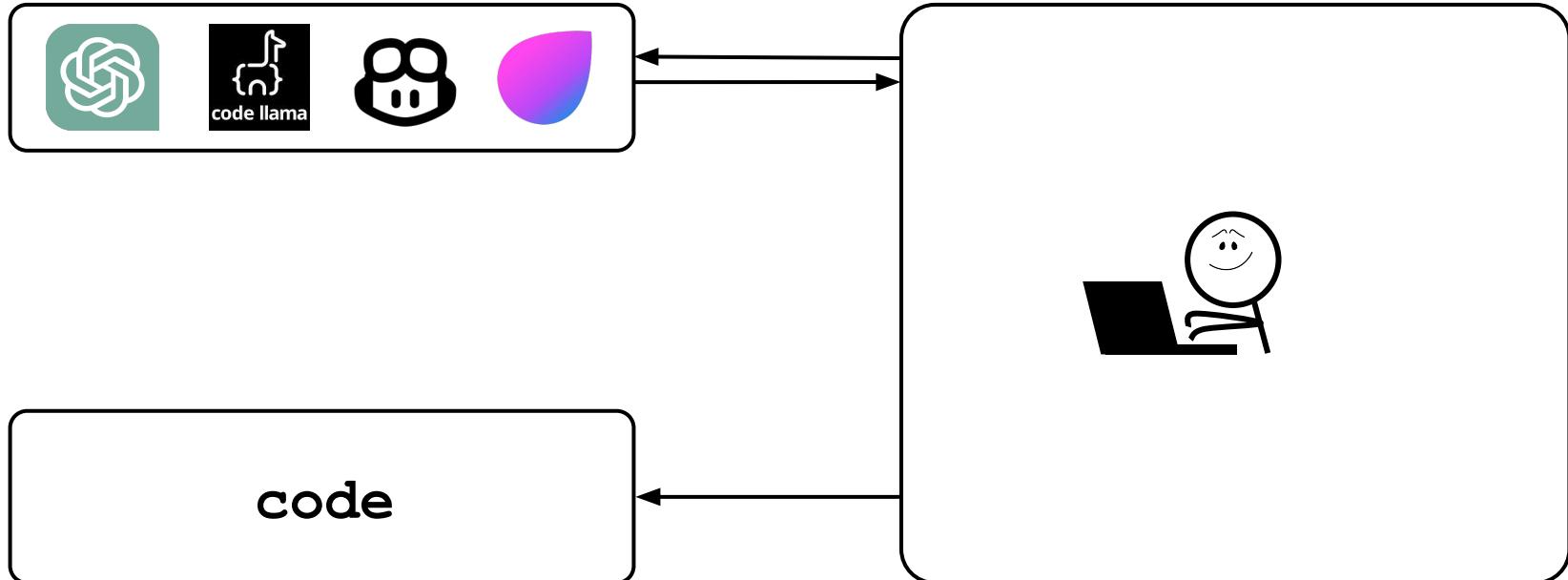
Nikita Khramov

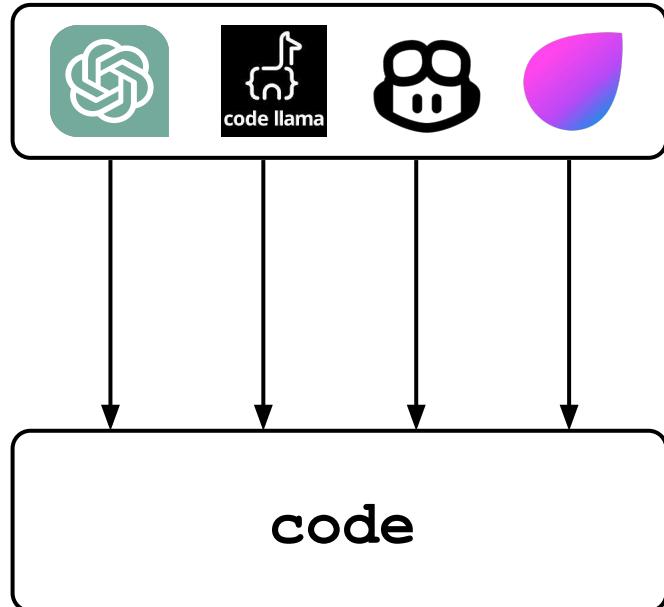
Anton Podkopaev

Programming Languages and Program Analysis Lab (PLAN), JetBrains Research

September, 2024

# LLMs are used more and more for code generation



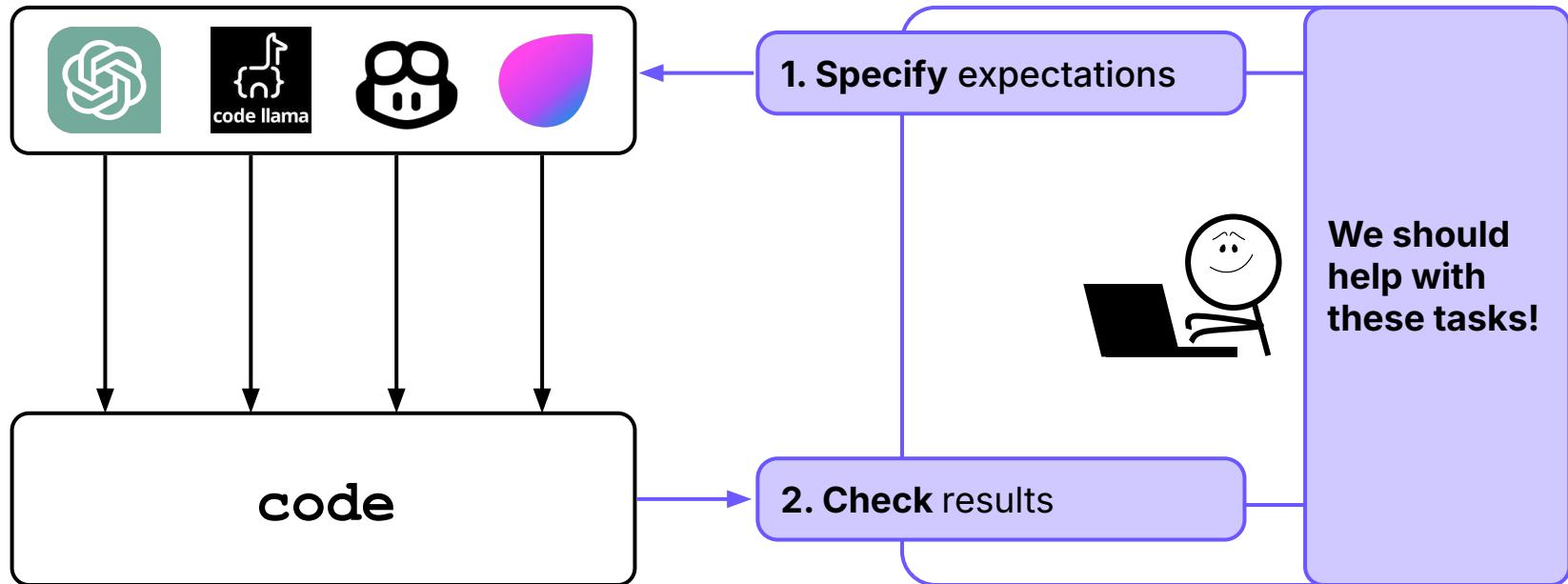


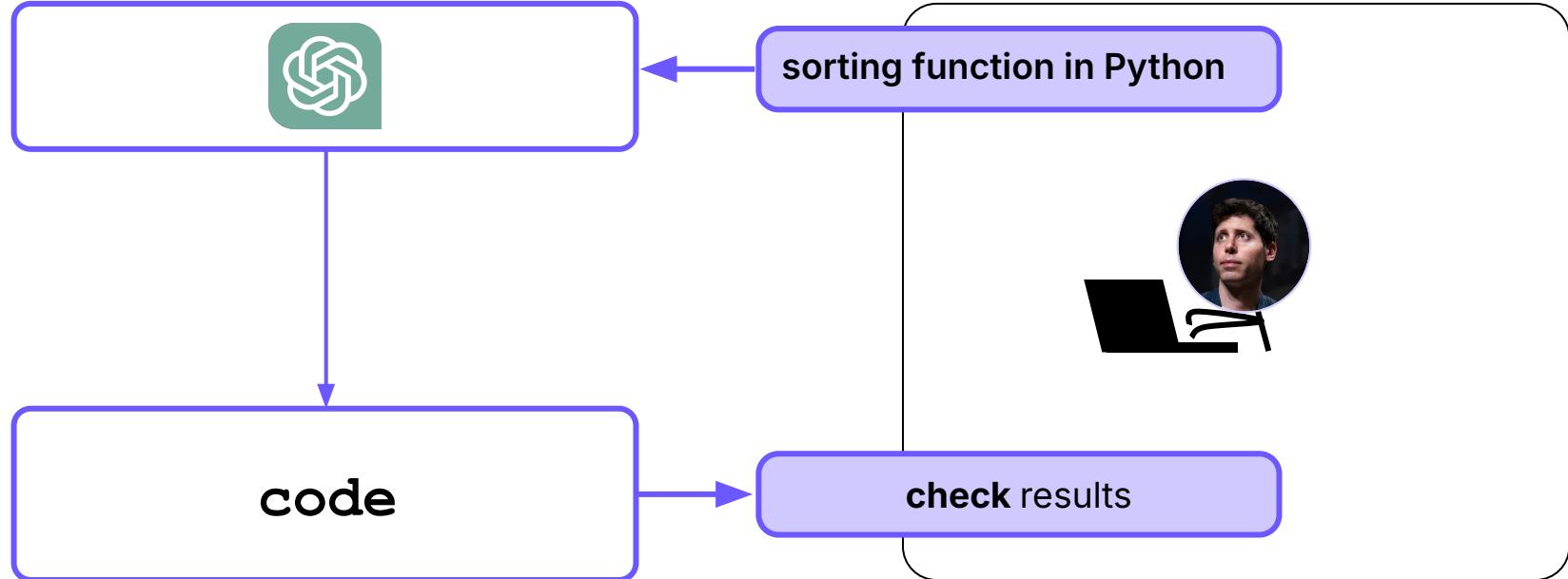
1. Specify expectations



2. Check results

# Developer — LLM supervisor







Generate a list sorting function in Python without using any library functions



```
def custom_sort(arr):
    n = len(arr)
    for i in range(n):
        # Last i elements are already sorted
        for j in range(0, n - i - 1):
            # Traverse the array from 0 to n-i-1
            # Swap if the element found is greater
            # than the next element
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr
```

# **Do Users Write More Insecure Code with AI Assistants?**

Neil Perry \*

Stanford University

Megha Srivastava \*

Stanford University

Deepak Kumar

Stanford University

Dan Boneh

Stanford University

# Do Users Write More Insecure Code with AI Assistants?

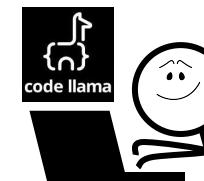
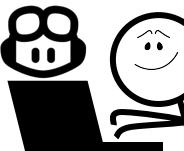
Neil Perry \*  
Stanford University

Megha Srivastava \*  
Stanford University

Deepak Kumar  
Stanford University

Dan Boneh  
Stanford University

introduce security **vulnerabilities**



# Do Users Write More Insecure Code with AI Assistants?

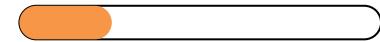
Neil Perry \*  
Stanford University

Megha Srivastava \*  
Stanford University

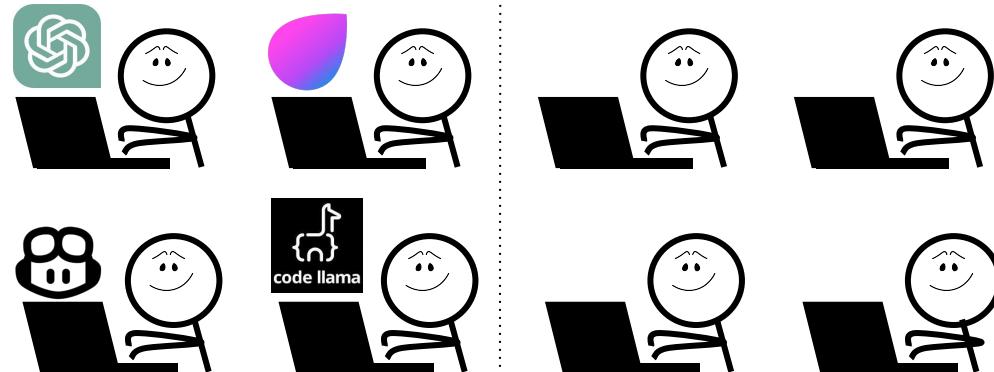
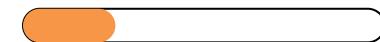
Deepak Kumar  
Stanford University

Dan Boneh  
Stanford University

introduce security **vulnerabilities**



rate their **insecure** code **as secure**





Generate a list sorting function in Python without using any library functions



```
def custom_sort(arr):
```

**Not** only **familiar** code is needed

```
# Last i elements are already sorted
for j in range(0, n - i - 1):
    # Traverse the array from 0 to n-i-1
    # Swap if the element found is greater
    # than the next element
    if arr[j] > arr[j + 1]:
        arr[j], arr[j + 1] = arr[j + 1], arr[j]
return arr
```



Generate a list sorting function in Python without using any library functions



```
def custom_sort(arr):
```

**Not** only **familiar** code is needed

**Testing** may **not** be **sufficient** (eg, concurrency)

```
# Last i elements are already sorted
for j in range(0, n - i - 1):
    # Traverse the array from 0 to n-i-1
    # Swap if the element found is greater
    # than the next element
    if arr[j] > arr[j + 1]:
        arr[j], arr[j + 1] = arr[j + 1], arr[j]
return arr
```



Generate a list sorting function in Python without using any library functions



```
def custom_sort(arr):
```

**Not** only **familiar** code is needed

**Testing** may **not** be **sufficient** (eg, concurrency)

Plain **English** is **hard** to debug and **imprecise**

```
# Traverse the array from 0 to n-1  
# Swap if the element found is greater  
# than the next element  
if arr[j] > arr[j + 1]:  
    arr[j], arr[j + 1] = arr[j + 1], arr[j]  
return arr
```

**Is there a better way?**

# The Coq programming language

Definition sort (l : list nat) : {l' : list nat | Permutation l l' & is\_sorted l'}.

# The Coq programming language

```
Definition sort (l : list nat) : {l' : list nat | Permutation l l' & is_sorted l'}.
```



Argument

# The Coq programming language

```
Definition sort (l : list nat) : {l' : list nat | Permutation l l' & is_sorted l'}.
```

↑  
Argument

↑  
Returning type

# The Coq programming language

```
Definition sort (l : list nat) : {l' : list nat | Permutation l l' & is_sorted l'}.
```

↑  
Argument

↑  
Returning type

- + Any **implementation** is a **correct** sorting function

# The Coq programming language

```
Definition sort (l : list nat) : {l' : list nat | Permutation l l' & is_sorted l'}.
```



- + Any **implementation** is a **correct** sorting function
- + **Specification** is a type-automatic **checking**

```
94 Definition sort l : {l' | Permutation l l' & is_sorted l'}.
95 Proof.
96 induction l.
97 { admit. }
98 destruct IHl as [l'].
99 edestruct (insert_sorted a l') as [l''].
100 { admit. }
101 exists l''.
102 2: { admit. }
103 transitivity (a::l').
104 { admit. }
105 admit.
106 Admitted.
107 Eval compute in (p1 (sort [3;2;4;1])).
```

Not in proof mode

s.v M X

src > s.v > sort

```
94 Definition sort l : {l' | Permutation l l' & is_sorted l'}.
95 Proof.
96 induction l.
97 { admit. }
98 destruct IHl as [l'].
99 edestruct (insert_sorted a l') as [l''].
100 { admit. }
101 exists l''.
102 2: { admit. }
103 transitivity (a::l').
104 { admit. }
105 admit.
106 Admitted.
107 Eval compute in (p1 (sort [3;2;4;1])).
108
109
110
111
112
113
114 Main buffer
115
116
117
118
119
120
```

Not in proof mode

ProofView: s.v X

main\* Environment: default.nix Ready ✓ coqilot "s.v" 152L 2817C written

Ln 94, Col 59 Spaces: 2 UTF-8 LF Coq

The screenshot shows a Coq proof assistant interface. On the left, a code editor displays a proof script in Coq's Gallina language. The script defines a function `sort` and proves its correctness. A large blue arrow points upwards from the bottom of the code editor towards the top of the screen, indicating the flow of the proof. The right side of the interface shows a "ProofView" window titled "ProofView: s.v X" which displays the state of the proof. The status bar at the bottom indicates the file is "Not in proof mode".

```
s.v M X
src > s.v > sort
94 Definition sort l : {l' | Permutation l l' & is_sorted l'}.
95 Proof.
96 induction l.
97 { admit. }
98 destruct IHl as [l'].
99 edestruct (insert_sorted a l') as [l''].
100 { admit. }
101 exists l''.
102 2: { admit. }
103 transitivity (a::l').
104 { admit. }
105 admit.
106 Admitted.
107 Eval compute in (p1 (sort [3;2;4;1])).
108
109
110
111
112
113
114
115
116
117
118
119
120
```

# Interactively typechecked part

The screenshot shows a Coq proof assistant interface with two main panes. The left pane displays a proof script in Coq's Gallina language, while the right pane shows the current proof state.

**Left Pane (Proof Script):**

```
94 Definition sort l : {l' | Permutation l l' & is_sorted l'}.
95 Proof.
96 induction l.
97 { admit. }
98 destruct IHl as [l'].
99 edestruct (insert_sorted a l') as [l''].
100 { admit. }
101 exists l''.
102 2: { admit. }
103 transitivity (a::l').
104 { admit. }
105 admit.
106 Admitted.
107 Eval compute in (p1 (sort [3;2;4;1])).
```

**Right Pane (Proof State):**

MAIN 1 SHELVED 0 GIVEN UP 0

```
l: list nat
```

(1/1)  
{l' : list nat | Permutation l l' & is\_sorted l'}

A large blue arrow points upwards from the bottom of the left pane towards the top of the right pane, indicating the flow of the proof process.

# Interactively typechecked part

Ln 95, Col 7 Spaces: 2 UTF-8 LF Coq

s.v M X

src > s.v > ...

```
94 Definition sort l : {l' | Permutation l l' & is_sorted l'}.
95 Proof.
96 induction l.
97 { admit. }
98 destruct IHl as [l'].
99 edestruct (insert_sorted a l') as [l''].
100 { admit. }
101 exists l''.
102 2: { admit. }
103 transitivity (a::l').
104 { admit. }
105 admit.
106 Admitted.
107 Eval compute in (p1 (sort [3;2;4;1])).
```

MAIN 1 SHELVED 0 GIVEN UP 0

l: list nat

(1/1)  
{l' : list nat | Permutation l l' & is\_sorted l'}

# Current state buffer

s.v M X

src > s.v > ...

```
94 Definition sort l : {l' | Permutation l l' & is_sorted l'}.
95 Proof.
96 induction l.
97 { admit. }
98 destruct IHl as [l'].
99 edestruct (insert_sorted a l') as [l''].
100 { admit. }
101 exists l''.
102 2: { admit. }
103 transitivity (a::l').
104 { admit. }
105 admit.
106 Admitted.
107 Eval compute in (p1 (sort [3;2;4;1])).
```

MAIN 1 SHELVED 0 GIVEN UP 0

l: list nat

(1/1)  
{l' : list nat | Permutation l l' & is\_sorted l'}

Assumptions

Current state buffer

Environment: default.nix    0 △ 0    0 Ready    ✓ coqilot "s.v" 152L 2817C written

Ln 95, Col 7    Spaces: 2    UTF-8    LF    Coq

The image shows a Coq proof session with two main windows. The left window displays the source code of a file named `s.v`. The right window shows the proof view for the current goal.

**Source Code (`s.v`):**

```
src > s.v > ...
94 Definition sort l : {l' | Permutation l l' & is_sorted l'}.
95 Proof.
96 induction l.
97 { admit. }
98 destruct IHl as [l'].
99 edestruct (insert_sorted a l') as [l''].
100 { admit. }
101 exists l''.
102 2: { admit. }
103 transitivity (a::l').
104 { admit. }
105 admit.
106 Admitted.
107 Eval compute in (p1 (sort [3;2;4;1])).
```

**Proof View:**

MAIN 1 SHELVED 0 GIVEN UP 0

**Assumptions:**

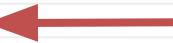
`l: list nat` ← **Assumptions**

**(1/1)**  
`{l' : list nat | Permutation l l' & is_sorted l'}`

**Current goal:**

**Current state buffer:**

Ln 95, Col 7 Spaces: 2 UTF-8 LF Coq

```
94 Definition sort l : {l' | Permutation l l' & is_sorted l'}.
95 Proof.
96 induction l. |  Tactic
97 { admit. }
98 destruct IHl as [l'].
99 edestruct (insert_sorted a l') as [l''].
100 { admit. }
101 exists l''.
102 2: { admit. }
103 transitivity (a::l').
104 { admit. }
105 admit.
106 Admitted.
107 Eval compute in (p1 (sort [3;2;4;1])).
```

```
(1/2) {l' : list nat | Permutation [] l' & is_sorted l'}
(2/2) {l' : list nat | Permutation (a :: l) l' & is_sorted l'}
```

The screenshot shows a Coq proof assistant interface with two main panes. The left pane displays a proof script in Coq's Gallina language, and the right pane shows the current proof state.

**Left Pane (Proof Script):**

```
94 Definition sort l : {l' | Permutation l l' & is_sorted l'}.
95 Proof.
96   induction l.
97   { admit. }
98   destruct IHl as [l'].
99   edestruct (insert_sorted a l') as [l''].
100  { admit. }
101  exists l''.
102  2: { admit. }
103  transitivity (a::l').
104  { admit. }
105  admit.
106 Admitted.
107 Eval compute in (p1 (sort [3;2;4;1])).
```

**Right Pane (Proof State):**

MAIN 1 SHELVED 0 GIVEN UP 0

(1/1)

```
{l' : list nat | Permutation [] l' & is_sorted l'}
```

At the bottom of the interface, there is a status bar with the following information:

main\* Environment: default.nix 0 0 0 Ready ✓ coqilot "s.v" 152L 2817C written

Ln 97, Col 4 Spaces: 2 UTF-8 LF Coq

```
94 Definition sort l : {l' | Permutation l l' & is_sorted l'}.
95 Proof.
96   induction l.
97   { admit. }
98   destruct IHl as [l'].
99   edestruct (insert_sorted a l') as [l''].
100  { admit. }
101  exists l''.
102  2: { admit. }
103  transitivity (a::l').
104  { admit. }
105  admit.
106 Admitted.
107 Eval compute in (p1 (sort [3;2;4;1])).
108
109
110
111
112
113
114
115
116
117
118
119
120
```

There are unfocused goals

The screenshot shows a Coq proof session with two main panes. The left pane displays the source code for a proof, and the right pane shows the proof state and goals.

**Left Pane (Source Code):**

```
src > s.v > ...
94 Definition sort l : {l' | Permutation l l' & is_sorted l'}.
95 Proof.
96   induction l.
97   { admit. }
98   destruct IHl as [l'].
99   edestruct (insert_sorted a l') as [l''].
100  { admit. }
101  exists l''.
102  2: { admit. }
103  transitivity (a::l').
104  { admit. }
105  admit.
106 Admitted.
107 Eval compute in (p1 (sort [3;2;4;1])).
```

**Right Pane (Proof View):**

MAIN 1 SHELVED 0 GIVEN UP 1

Warning: You have given up goals

```
a : nat
l : list nat
IHl : {l' : list nat | Permutation l l' & is_sorted l'}
```

(1/1)

```
{l' : list nat | Permutation (a :: l) l' & is_sorted l'}
```

Environment: default.nix 0 0 0 Ready ✓ coqilot "s.v" 152L 2817C written

Ln 97, Col 13 Spaces: 2 UTF-8 LF Coq

s.v M X

src > s.v > ...

```
94 Definition sort l : {l' | Permutation l l' & is_sorted l'}.
95 Proof.
96   induction l.
97   { admit. }
98   destruct IHl as [l'].
99   edestruct (insert_sorted a l') as [l''].
100  { admit. }
101  exists l''.
102  2: { admit. }
103  transitivity (a::l').
104  { admit. }
105  admit.
106 Admitted.
107 Eval compute in (p1 (sort [3;2;4;1])).
```

ProofView: s.v X

MAIN 1 SHELVED 0 GIVEN UP 1

Warning: You have given up goals

```
a : nat
l, l' : list nat
p : Permutation l l'
i : is_sorted l'
```

(1/1)

```
{l'0 : list nat | Permutation (a :: l) l'0 & is_sorted l'0}
```

main\* Environment: default.nix Ready coqilot -- NORMAL --

Ln 98, Col 24 Spaces: 2 UTF-8 LF Coq

```
src > s.v > ...
94 Definition sort l : {l' | Permutation l l' & is_sorted l'}.
95 Proof.
96 induction l.
97 { admit. }
98 destruct IHl as [l'].
99 edestruct (insert_sorted a l') as [l''].
100 { admit. }
101 exists l''.
102 2: { admit. }
103 transitivity (a::l').
104 { admit. }
105 admit.
106 Admitted.
107 Eval compute in (p1 (sort [3;2;4;1])).
```

MAIN 2 SHELVED 0 GIVEN UP 1

Warning: You have given up goals

a: nat  
l, l': list nat  
p: Permutation l l'  
i: is\_sorted l'

(1/2)  
is\_sorted l'

(2/2)  
{l'0 : list nat | Permutation (a :: l) l'0 & is\_sorted l'0}

```
src > s.v > ...
94 Definition sort l : {l' | Permutation l l' & is_sorted l'}.
95 Proof.
96 induction l.
97 { admit. }
98 destruct IHl as [l'].
99 edestruct (insert_sorted a l') as [l''].
100 { admit. }
101 exists l''.
102 2: { admit. }
103 transitivity (a::l').
104 { admit. }
105 admit.
106 Admitted.
107 Eval compute in (p1 (sort [3;2;4;1])).
```

MAIN 1 SHELVED 0 GIVEN UP 2

Warning: You have given up goals

```
a: nat
l, l': list nat
p: Permutation l l'
i: is_sorted l'
l'': list nat
i0: is_inserted a l' l''
i1: is_sorted l''
```

(1/1)

```
{l'0 : list nat | Permutation (a :: l) l'0 & is_sorted l'0}
```

```
src > s.v > ...
94 Definition sort l : {l' | Permutation l l' & is_sorted l'}.
95 Proof.
96 induction l.
97 { admit. }
98 destruct IHl as [l'].
99 edestruct (insert_sorted a l') as [l''].
100 { admit. }
101 exists l''.
102 2: { admit. }
103 transitivity (a::l').
104 { admit. }
105 admit.
106 Admitted.
107 Eval compute in (p1 (sort [3;2;4;1])).
```

MAIN 2 SHELVED 0 GIVEN UP 2

Warning: You have given up goals

```
a: nat
l, l': list nat
p: Permutation l l'
i: is_sorted l'
l'': list nat
i0: is_inserted a l' l''
i1: is_sorted l''
```

```
(1/2)
Permutation (a :: l) l''

(2/2)
is_sorted l''
```

src > s.v > ...

```
94 Definition sort l : {l' | Permutation l l' & is_sorted l'}.
95 Proof.
96   induction l.
97   { admit. }
98   destruct IHl as [l'].
99   edestruct (insert_sorted a l') as [l''].
100  { admit. }
101  exists l''.
102  2: { admit. }
103  transitivity (a::l').
104  { admit. }
105  admit.
106 Admitted.
107 Eval compute in (p1 (sort [3;2;4;1])).
```

MAIN 1 SHELVED 0 GIVEN UP 3

Warning: You have given up goals

```
a: nat
l, l': list nat
p: Permutation l l'
i: is_sorted l'
l'': list nat
i0: is_inserted a l' l''
i1: is_sorted l''
```

(1/1)  
Permutation (a :: l) l''

s.v M X

src > s.v > ...

```
94 Definition sort l : {l' | Permutation l l' & is_sorted l'}.
95 Proof.
96 induction l.
97 { admit. }
98 destruct IHl as [l'].
99 edestruct (insert_sorted a l') as [l''].
100 { admit. }
101 exists l''.
102 2: { admit. }
103   transitivity (a::l').
104   { admit. }
105   admit.
106 Admitted.
107 Eval compute in (p1 (sort [3;2;4;1])).
```

108  
109  
110  
111  
112  
113  
114  
115  
116  
117  
118  
119  
120

ProofView: s.v X

MAIN 2 SHELVED 0 GIVEN UP 3

Warning: You have given up goals

```
a: nat
l, l': list nat
p: Permutation l l'
i: is_sorted l'
l'': list nat
i0: is_inserted a l' l''
i1: is_sorted l''
```

(1/2)  
Permutation (a :: l) (a :: l')

(2/2)  
Permutation (a :: l') l''

Ln 103, Col 24 Spaces: 2 UTF-8 LF Coq

```
src > s.v > ...
94 Definition sort l : {l' | Permutation l l' & is_sorted l'}.
95 Proof.
96 induction l.
97 { admit. }
98 destruct IHl as [l'].
99 edestruct (insert_sorted a l') as [l''].
100 { admit. }
101 exists l''.
102 2: { admit. }
103 transitivity (a::l').
104 { admit. }
105 admit.
106 Admitted.
107 Eval compute in (p1 (sort [3;2;4;1])).
```

MAIN 1 SHELFED 0 GIVEN UP 4

Warning: You have given up goals

```
a: nat
l, l': list nat
p: Permutation l l'
i: is_sorted l'
l'': list nat
i0: is_inserted a l' l''
i1: is_sorted l''
```

(1/1)  
Permutation (a :: l') l''

```
94 Definition sort l : {l' | Permutation l l' & is_sorted l'}.
95 Proof.
96 induction l.
97 { admit. }
98 destruct IHl as [l'].
99 edestruct (insert_sorted a l') as [l''].
100 { admit. }
101 exists l''.
102 2: { admit. }
103 transitivity (a::l').
104 { admit. }
105 admit.
106 Admitted.
107 Eval compute in (p1 (sort [3;2;4;1])).
```

There are some goals you gave up. You need to go back and solve them, or use `Admitted`.

Not in proof mode

```
94 Definition sort l : {l' | Permutation l l' & is_sorted l'}.
95 Proof.
96 induction l.
97 { admit. }
98 destruct IHl as [l'].
99 edestruct (insert_sorted a l') as [l''].
100 { admit. }
101 exists l''.
102 2: { admit. }
103 transitivity (a::l').
104 { admit. }
105 admit.
106 Admitted.
107 Eval compute in (p1 (sort [3;2;4;1])).
```



s.v M X

src > s.v > ...

```
94 Definition sort l : {l' | Permutation l l' & is_sorted l'}.
95 Proof.
96 induction l.
97 { admit. }
98 destruct IHl as [l'].
99 edestruct (insert_sorted a l') as [l''].
100 { admit. }
101 exists l''.
102 2: { admit. }
103 transitivity (a::l').
104 { admit. }
105 admit.
106 Admitted.
107 Eval compute in (p1 (sort [3;2;4;1])).
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS Notices

```
= let (x, _, _) := sort [3; 2; 4; 1] in x
: list nat
```

main\* Environment: default.nix Ready ✓ coqilot -- NORMAL --

ProofView: s.v X Not in proof mode

Ln 107, Col 39 Spaces: 2 UTF-8 LF Coq

**How do we fill those  
admits?**

**Coq is very suitable for code generation**

# Coq is **very suitable** for code generation

- + Type-checking validates **deep** program's **properties**

## Coq is **very suitable** for code generation

- + Type-checking validates **deep** program's **properties**
- + Tactics do **not** enforce **strict** structure

# Coq is **very suitable** for code generation

- + Type-checking validates **deep** program's **properties**
- + Tactics do **not** enforce **strict** structure
- + Easy to **guide** generation with providing **structure and insights**

# Coq is **very suitable** for code generation

- + **Type-checking** validates **deep** program's **properties**
- + **Tactics** do **not** enforce **strict** structure
- + Easy to **guide generation** with providing **structure and insights**
- + **Holes** are independent and might be **filled in parallel**

CoqPilot



src &gt; S.V

```

43 Lemma is_inserted_perm a l l' (INS : is_inserted a l)
44 (* Hint: perm_swap *)
45 Proof.
46   generalize dependent l'.
47   generalize dependent a.
48   induction l; ins; inv INS.
49   apply IHl in INS0.
50   etransitivity.
51   { by apply perm_swap. }
52   | by constructor.
53 Qed.

54

55 Lemma insert_sorted a l (SORT : is_sorted l) :
56   {l' | is_inserted a l l' & is_sorted l'}.
57 (* Hint: le_gt_dec *)
58 Proof.
59   induction l; eauto with myconstr.
60   edestruct IHl as [l'].
61   { clear -SORT. inv SORT. auto with myconstr. }
62   destruct (le_gt_dec a a0).
63   { exists (a::a0::l); auto with myconstr.
64     apply sorted_cons; auto.
65     eapply smallest_head; eauto.
66     inv SORT. auto with myconstr. }
67   exists (a0::l'); auto. constructor; auto.
68
69   clear -SORT i i0 g.
70   induction i; auto.
71 
```



## Proof

Main 1

Shelved 0

Given up 1

## Goal 1

a : nat  
l, l' : list nat  
p : Permutation l l'  
i : is\_sorted l'

(1 / 1) —  
is\_sorted l'

## Messages

src &gt; s.v

```
43 Lemma is_inserted_perm a l l' (INS : is_inserted a l)
44 (* Hint: perm_swap *)
45 Proof.
```

```
46 generalize dependent l'.
47 generalize dependent a.
48 induction l; ins; inv INS.
49 apply IHl in INS0.
50 etransitivity.
51 { by apply perm_swap. }
52 | by constructor.
```

```
53 Qed.
```

```
54
55 Lemma insert_sorted a l (SORT : is_sorted l) :
56 | {l' | is_inserted a l l' & is_sorted l'}.
57 (* Hint: le_gt_dec *)
58 Proof.
```

```
59 induction l; eauto with myconstr.
60 edestruct IHl as [l'].
61 { clear -SORT. inv SORT. auto with myconstr. }
62 destruct (le_gt_dec a a0).
63 { exists (a::a0::l); auto with myconstr.
64 | apply sorted_cons; auto.
65 | eapply smallest_head; eauto.
66 | inv SORT. auto with myconstr. }
67 exists (a0::l'); auto. constructor; auto.
68
69 clear -SORT i i0 g.
70 induction i; auto.
```

## Proof

Main 1      Shelved 0      Given up 1

Goal 1

```
a : nat
l, l' : list nat
p : Permutation l l'
i : is_sorted l'
```

(1 / 1) —

```
is_sorted l'
```

## Messages



src &gt; S.V

```
43 Lemma is_inserted_perm a l l' (INS : is_inserted a l)
44 (* Hint: perm_swap *)
45 Proof.
```

```
46 generalize dependent l'.
47 generalize dependent a.
48 induction l; ins; inv INS.
49 apply IHl in INS0.
50 etransitivity.
51 { by apply perm_swap. }
52 | by constructor.
53 Qed.
```

```
54
55 Lemma insert_sorted a l (SORT : is_sorted l) :
56 | {l' | is_inserted a l l' & is_sorted l'}.
57 (* Hint: le_gt_dec *)
```

```
58 Proof.
59 induction l; eauto with myconstr.
60 edestruct IHl as [l'].
61 { clear -SORT. inv SORT. auto with myconstr. }
62 destruct (le_gt_dec a a0).
63 { exists (a::a0::l); auto with myconstr.
64 | apply sorted_cons; auto.
65 | eapply smallest_head; eauto.
66 | inv SORT. auto with myconstr. }
67 exists (a0::l'); auto. constructor; auto.
68
69 clear -SORT i i0 g.
70 induction i: auto.
```

## Proof

Main 1

Shelved 0

Given up 1

## Goal 1

```
a : nat
l, l' : list nat
p : Permutation l l'
i : is_sorted l'

(1 / 1) —
is_sorted l'
```

## Messages

```
92  
93 Definition sort l : {l' | Permutation l l' & is_sorted l'}.
```

```
94 Proof.
```

```
95 induction l.
```

```
96 { admit. }
```

```
97 destruct IHl as [l'].
```

```
98 edestruct (insert_sorted a l') as [l''].
```

```
99 { admit. }
```

```
100 exists l''.
```

```
101 2: { admit. }
```

```
102 transitivity (a::l').
```

```
103 { admit. }
```

```
104 admit.
```

```
105 Admitted.
```

```
106
```

```
107 Eval compute in (p1 (sort [3;2;4;1])).
```

```
108
```

```
109
```

```
110
```

```
111
```

```
112
```

```
113
```

```
114
```

```
115
```

```
116
```

```
117
```

```
118
```

```
119
```

```
120
```

## Proof

Not in proof mode

## Messages

```
= let (x, _, _) := sort [3; 2; 4; 1] in x  
: list nat
```



```
92
93 Definition sort l : {l' | Permutation l l' & is_sorted l'}.
```

```
94 Proof.
```

```
95 induction l.
```

```
96 { admit. }
```

```
97 destruct IHl as [l'].
```

```
98 edestruct (insert_sorted a l') as [l''].
```

```
99 { admit. }
```

```
100 exists l''.
```

```
101 2: { admit. }
```

```
102 transitivity (a::l').
```

```
103 { admit. }
```

```
104 admit.
```

```
105 Admitted.
```

```
106
```

```
107 Eval compute in (p1 (sort [3;2;4;1])).
```

```
108
```

```
109
```

```
110
```

```
111
```

```
112
```

```
113
```

```
114
```

```
115
```

```
116
```

```
117
```

```
118
```

```
119
```

```
120
```

## Proof

Not in proof mode

## Messages

```
= let (x, _, _) := sort [3; 2; 4; 1] in x
: list nat
```



src > S.V

```
89 Definition p1 {l} (x : {l' | Permutation l l' & is_sorted l'}) :=
```

```
destruct x as [l']. exact l'.
```

```
Defined.
```

```
92
```

```
93 Definition sort l : {l' | Permutation l l' & is_sorted l'} :=
```

```
94 Proof.
```

```
95 induction l.
```

```
96 { admit. }
```

```
97 destruct IHl as [l'].
```

```
98 edestruct (insert_sorted a l') as [l''].
```

```
99 { admit. }
```

```
100 exists l''.
```

```
101 2: { admit. }
```

```
102 transitivity (a::l').
```

```
103 { admit. }
```

```
104 admit.
```

```
105 Admitted.
```

```
106
```

```
107 Eval compute in (p1 (sort [3;2;4;1])).
```

```
108
```

```
109
```

```
110
```

```
111
```

```
112
```

```
113
```

```
114
```

```
115
```

```
116
```

```
117
```

## Proof

Main 1

Shelved 0

Given up 1

## Goal 1

a : nat

l, l' : list nat

p : Permutation l l'

i : is\_sorted l'

(1/1) —

is\_sorted l'

## Messages

The screenshot shows the Coq Goals interface with the following proof script:

```
src > S.V
89 Definition p1 {l} (x : {l' | Permutation l l' & is_sorted l'}) :=
90   destruct x as [l'']. exact l''.
91 Defined.
92
93 Definition sort l : {l' | Permutation l l' & is_sorted l'}.
94 Proof.
95   induction l.
96   { admit. }
97   destruct IHl as [l''].
98   edestruct (insert_sorted a l'') as [l'''].
99   { admit. } exists l'''.
100 2: { admit. }
101  transitivity (a::l'').
102  { admit. }
103  admit.
104 Admitted.
105
106
107 Eval compute in (p1 (sort [3;2;4;1])).
```

The screenshot shows the Coq Goals interface. At the top, there's a header bar with a back arrow, the title "Coq Goals X", and three status indicators: "Main 1", "Shelved 0", and "Given up 1". Below the header, the word "Proof" is displayed. The main area contains a "Goal 1" section with the following hypotheses:

```
a : nat
l, l' : list nat
p : Permutation l l'
i : is_sorted l'
```

Underneath the hypotheses, a goal is listed:

```
(1/1) _____
is_sorted l'
```

The bottom section is titled "Messages" and is currently empty.

## Few-shot prompt

**Lemma** insert\_sorted ...  
**Proof.**  
...  
**Defined.**

• • •

**Lemma** is\_inserted\_perm ...  
**Proof.**  
...  
**Defined.**

## Query

**Lemma** unsort\_sorted ...  
**Proof.**  
???  
**Admitted.**

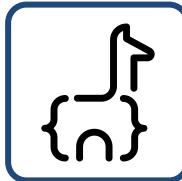
## Few-shot prompt

**Lemma** insert\_sorted ...  
**Proof.**  
...  
**Defined.**

**Lemma** is\_inserted\_perm ...  
**Proof.**  
...  
**Defined.**

## Query

**Lemma** unsort\_sorted ...  
**Proof.**  
???  
**Admitted.**



...

...



## Few-shot prompt

**Lemma** insert\_sorted ...  
**Proof.**  
...  
**Defined.**

**Lemma** is\_inserted\_perm ...  
**Proof.**  
...  
**Defined.**

## Query

**Lemma** unsort\_sorted ...  
**Proof.**  
???  
**Admitted.**



...

...



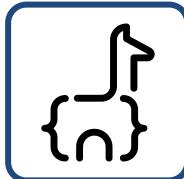
## Few-shot prompt

**Lemma** insert\_sorted ...  
**Proof.**  
...  
**Defined.**

**Lemma** is\_inserted\_perm ...  
**Proof.**  
...  
**Defined.**

## Query

**Lemma** unsort\_sorted ...  
**Proof.**  
???  
**Admitted.**



...

...



## Few-shot prompt

**Lemma** insert\_sorted ...  
**Proof.**  
...  
**Defined.**

**Lemma** is\_inserted\_perm ...  
**Proof.**  
...  
**Defined.**

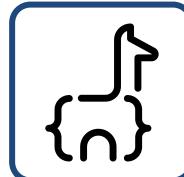
## Query

**Lemma** unsort\_sorted ...  
**Proof.**  
???  
**Admitted.**

## Checker

**Lemma** unsort\_sorted  
**Proof 1.**  
...  
**Defined.**

**Lemma** unsort\_sorted  
**Proof 2.**  
...  
**Defined.**



...

...



## Few-shot prompt

**Lemma** insert\_sorted ...  
**Proof.**  
...  
**Defined.**

**Lemma** is\_inserted\_perm ...  
**Proof.**  
...  
**Defined.**

## Query

**Lemma** unsort\_sorted ...  
**Proof.**  
???  
**Admitted.**

## Checker

**Lemma** unsort\_sorted  
**Proof 1.**  
...  
**Defined.**

**Lemma** unsort\_sorted  
**Proof 2.**  
...  
**Defined.**



...

...



## Few-shot prompt

**Lemma** insert\_sorted ...  
**Proof.**  
...  
**Defined.**

**Lemma** is\_inserted\_perm ...  
**Proof.**  
...  
**Defined.**

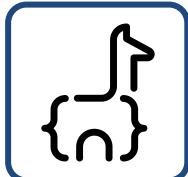
## Query

**Lemma** unsort\_sorted ...  
**Proof.**  
???  
**Admitted.**

## Checker

**Lemma** unsort\_sorted  
**Proof 1.**  
...  
**Defined.**

**Lemma** unsort\_sorted  
**Proof 2.**  
...  
**Defined.**



...

...



The screenshot shows a CoqIDE interface with the following components:

- Top Bar:** Includes icons for file operations (New, Open, Save, etc.) and settings.
- Left Panel:** A code editor showing a Coq proof script. The current line is highlighted in yellow. The code defines a sort function and proves its properties using induction and destruct tactics.
- Middle Panel:** A vertical stack of tabs. The top tab is "Coq Goals" (highlighted in blue). Below it are tabs for "Proof", "Main", "Shelved", and "Given up".
- Right Panel:** The "Proof" tab is active, displaying the proof state:
  - Main:** 1 goal
  - Goal 1:**  $l : \text{list nat}$
  - Preconditions:**  $\{l' : \text{list nat} \mid \text{Permutation } l l' \& \text{is_sorted } l'\}$
- Bottom Panel:** Shows the "Messages" tab, which is currently empty.

The code in the editor:

```
92
93 Definition sort l : {l' | Permutation l l' & is_sorted l'}.
94 Proof.
95   induction l.
96   { admit. }
97   destruct IHl as [l'].
98   edestruct (insert_sorted a l') as [l''].
99   { admit. }
100  exists l''.
101  2: { admit. }
102  transitivity (a::l').
103  { admit. }
104  admit.
105 Admitted.
106
107 Eval compute in (p1 (sort [3;2;4;1])).
```

At the bottom of the interface, there are status indicators for the main file ("main\*"), the language server ("vscoq-language-server 2.1.0, coq 8.19.0"), and various code analysis tools like Prettier and Go Live.

# Research Questions

# Research Questions

- **RQ1:** How well general purpose LLMs can write Coq proofs?

# Research Questions

- **RQ1:** How well general purpose LLMs can write Coq proofs?
- **RQ2:** To which extent does CoqPilot improve the LLM approach to Coq generation?

# Research Questions

- **RQ1:** How well general purpose LLMs can write Coq proofs?
- **RQ2:** To which extent does CoqPilot improve the LLM approach to Coq generation?
- **RQ3:** What is the additional value CoqPilot contributes to other Coq automation tools such as CoqHammer and Tactician?

# **Informal Evaluation**

# Informal Evaluation

Reference proof length	≤ 4	5 – 8	9 – 20	Total
Group size	131	98	71	300
firstorder auto with *	11%	2%	1%	6%
OpenAI GPT-3.5	29%	17%	6%	20%
OpenAI GPT-4o	50%	26%	15%	34%
LLaMA-2 13B Chat	2%	0%	0%	0.5%
Anthropic Claude	21%	7%	7%	13%
All models together	57%	32%	18%	39%
Tactician	45%	23%	10%	29%
CoqHammer	23%	4%	0%	11%
All methods together	71%	45%	23%	51%

# Informal Evaluation

Reference proof length	≤ 4	5 – 8	9 – 20	Total
Group size	131	98	71	300
firstorder auto with *	11%	2%	1%	6%
OpenAI GPT-3.5	29%	17%	6%	20%
OpenAI GPT-4o	50%	26%	15%	34%
LLaMA-2 13B Chat	2%	0%	0%	0.5%
Anthropic Claude	21%	7%	7%	13%
All models together	57%	32%	18%	39%
Tactician	45%	23%	10%	29%
CoqHammer	23%	4%	0%	11%
All methods together	71%	45%	23%	51%

# Informal Evaluation

Reference proof length	≤ 4	5 – 8	9 – 20	Total
Group size	131	98	71	300
firstorder	11%	2%	1%	6%
OpenAI GPT-3.5	29%	17%	6%	20%
OpenAI GPT-4o	50%	26%	15%	34%
LLaMA-2 13B Chat	2%	0%	0%	0.5%
Anthropic Claude	21%	7%	7%	13%
All models together	57%	32%	18%	39%
Tactician	45%	23%	10%	29%
CoqHammer	23%	4%	0%	11%
All methods together	71%	45%	23%	51%

# Informal Evaluation

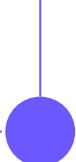
Reference proof length	≤ 4	5 – 8	9 – 20	Total
Group size	131	98	71	300
firstorder auto with *	11%	2%	1%	6%
OpenAI GPT-3.5	29%	17%	6%	20%
OpenAI GPT-4o	50%	26%	15%	34%
LLaMA-2 13B Chat	2%	0%	0%	0.5%
Anthropic Claude	21%	7%	7%	13%
All models together	57%	32%	18%	39%
Tactician	45%	23%	10%	29%
CoqHammer	23%	4%	0%	11%
All methods together	71%	45%	23%	51%

# **Improvement directions**



# Improvement directions

Integrate more Coq generation tools, such as Copra, Graph2Tac, TacTok etc.



# Improvement directions

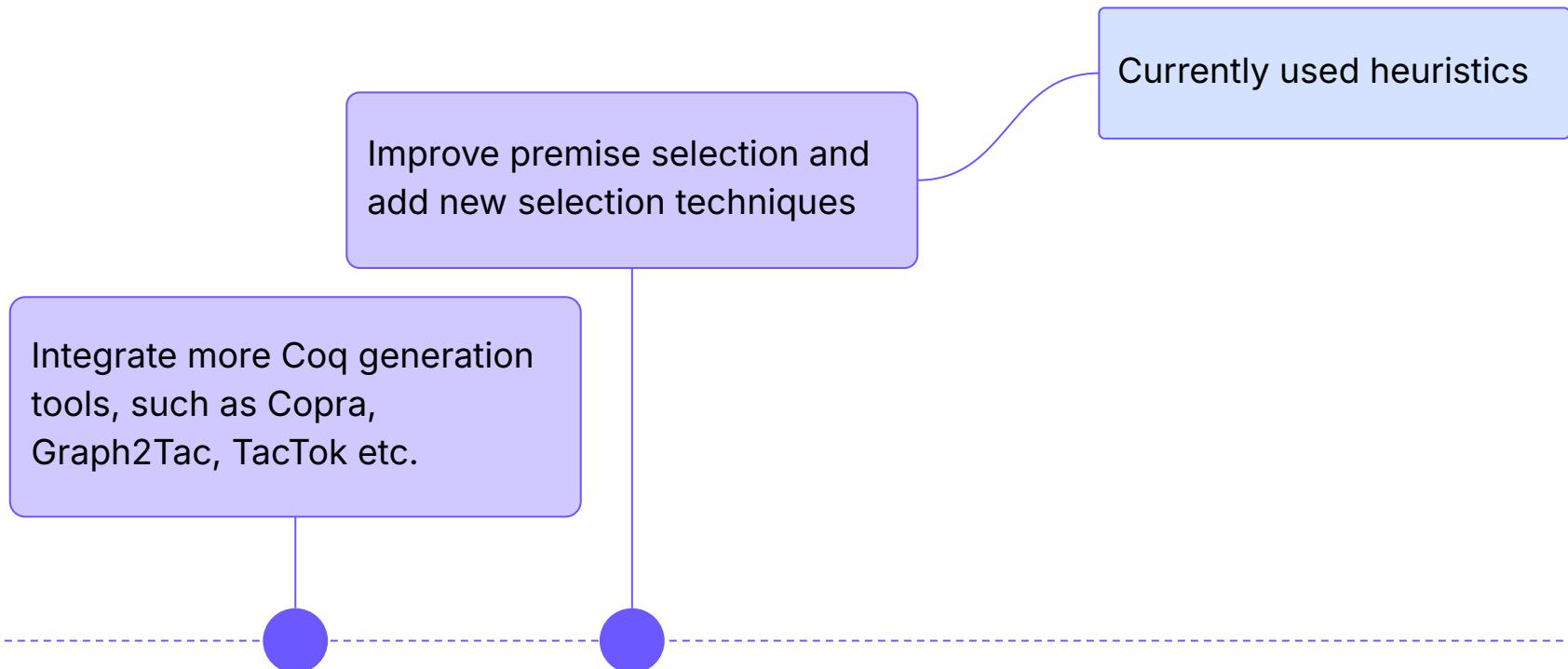
Improve premise selection and  
add new selection techniques

Integrate more Coq generation  
tools, such as Copra,  
Graph2Tac, TacTok etc.

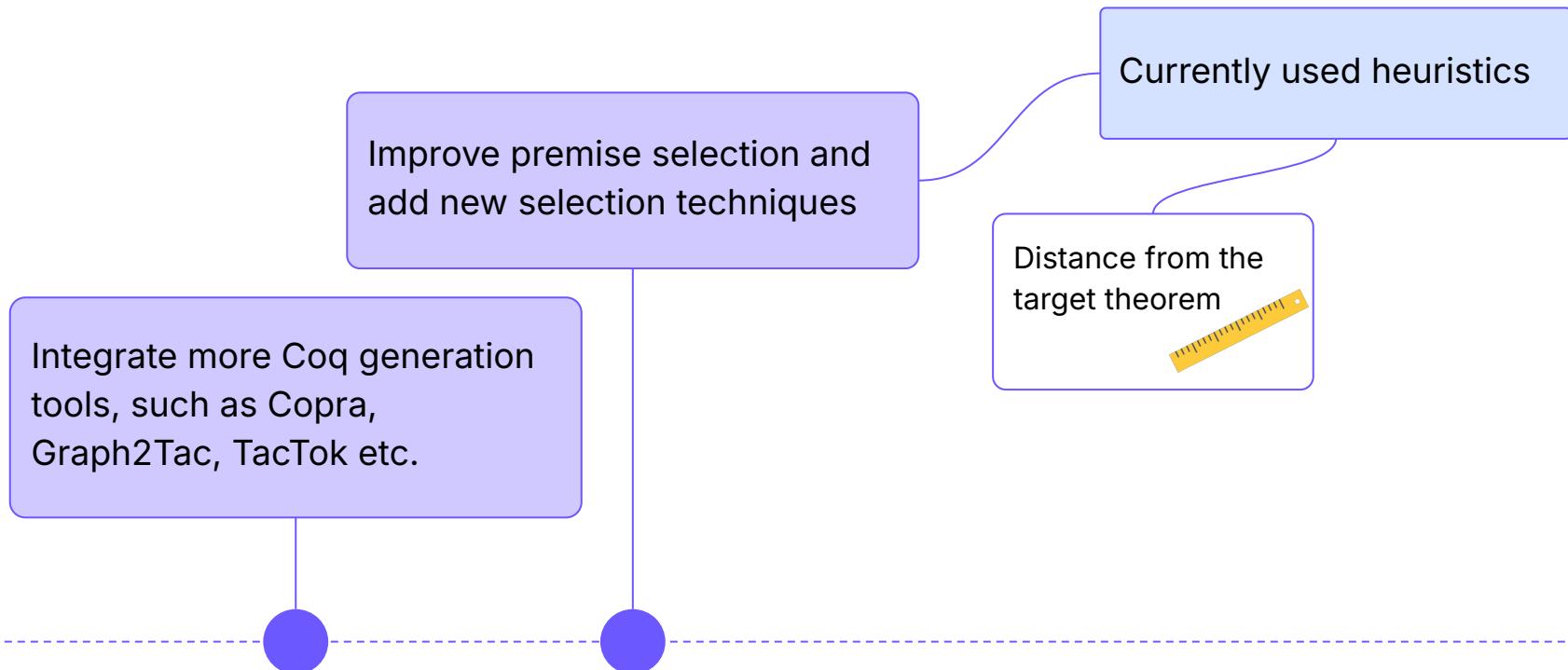


-----

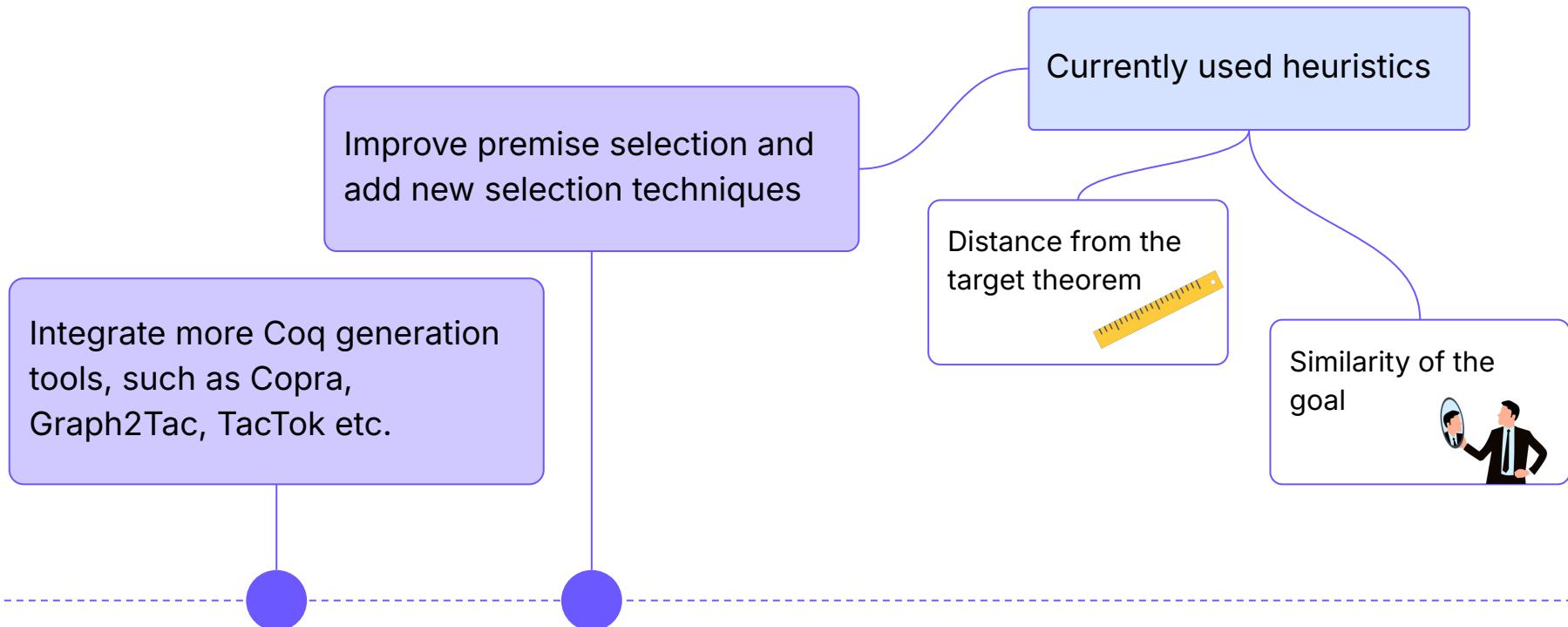
# Improvement directions



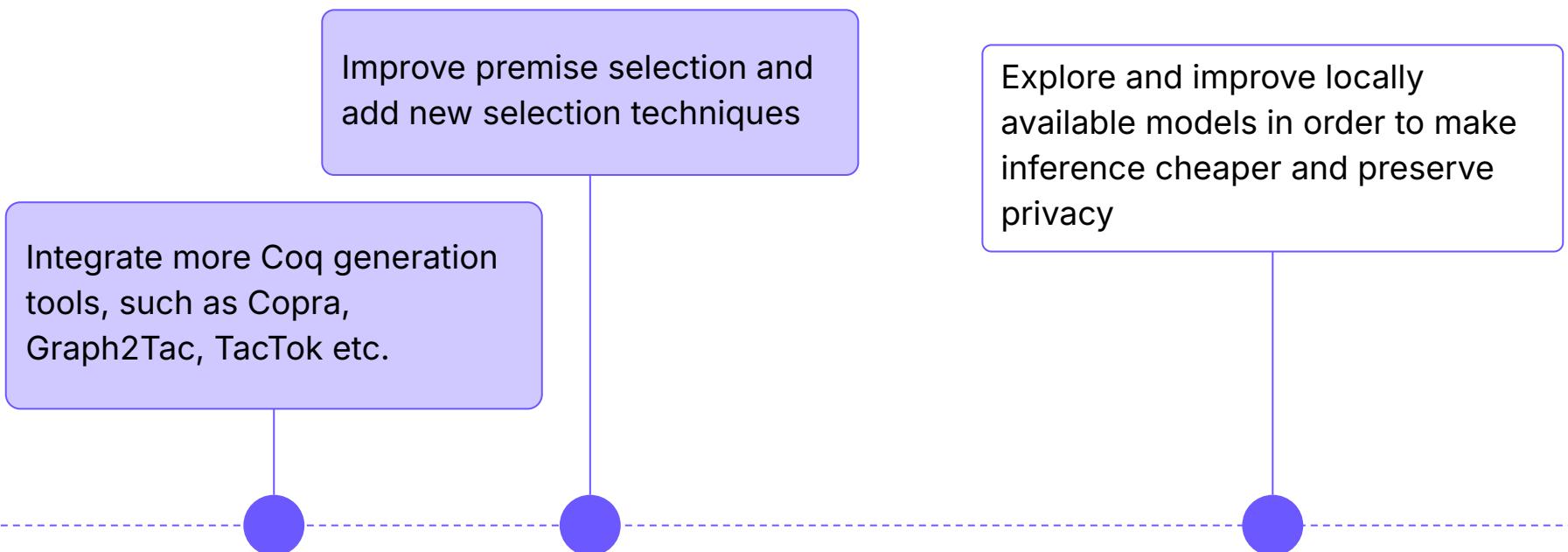
# Improvement directions



# Improvement directions



# Improvement directions



# Improvement directions

Improve premise selection and add new selection techniques

Integrate more Coq generation tools, such as Copra, Graph2Tac, TacTok etc.

Please talk to us if you have ideas!





CoqPilot: a plugin for LLM-based generation of proofs



JetBrains-Research/**coqpilot**



extension: **coqpilot**



{*andrei.kozyrev, gleb.solovev, nikita.khramov, anton.podkopaev*}@jetbrains.com

Programming Languages and Program Analysis Lab (PLAN), JetBrains Research