

Chapter 4

The Guard at the Door

Program 01: PII Input Guardrail

The PII input guardrail in Program 01 is a pattern guardrail at the input position. It runs before the model and modifies the text.

The Message You Cannot Unsend

A customer is frustrated. Their payment failed, they have tried twice, and now they are typing into the support chat. Halfway through the message, trying to be helpful, they include something nobody asked for:

My SSN is 123-45-6789. Why was my payment declined?

Nothing has gone wrong yet. Everything that happens in the next second decides whether this becomes an incident.

If that message goes straight to the model, the number is now in the prompt. The prompt goes to a model provider. It appears in your tracing tool, where every engineer can read it. It lands in your application logs. If the agent keeps memory, a summary of this conversation may store it for months. One customer message has just created five copies of a Social Security Number in systems that were never designed to hold one. None of those copies are easy to delete. Some of them you do not even control.

The fix is not clever. **Look at the message before anyone else does, find the number, and replace it. That is the whole idea behind an input guardrail**, and it is the first program in the companion repo.

What This Program Teaches

The agent here answers customer questions. A customer writes a message, and before that message reaches the model, a guardrail looks through it for personal information and blanks out anything it finds. The model answers the cleaned copy and never sees the original.

An input guardrail is a check that runs on incoming text before the model reads it.

The word input is doing the work here. It does not ask the model to be careful with sensitive data. It removes the data before the model gets the chance to be careful or careless with it. That is the difference between a rule the model is told about and a rule the model cannot reach. The data being removed is usually called PII, short for personally identifiable information. Names, account numbers, card numbers, email addresses, phone numbers, national identity numbers. The detail that matters for us is simple: it is data your agent may need to talk about, but almost never needs to see.

The program is built with LangGraph, and it is the simplest agent in the book. It introduces the pieces every later program reuses: state, nodes, edges, a start, an end, and running the graph. Along the way this program introduces four things you will use in every later chapter.

What you learn	Where you see it
State	One shared object that every step reads from and writes to
Nodes	Small Python functions that each do one job
Edges	The wiring that decides which node runs next
START and END	Where a run begins and where it finishes
Running the graph	Turning the wiring into something you can call

Three Nodes in a Straight Line

LangGraph calls each step a node. This program has three, joined in a line. There is no branching. Every message takes exactly the same path.

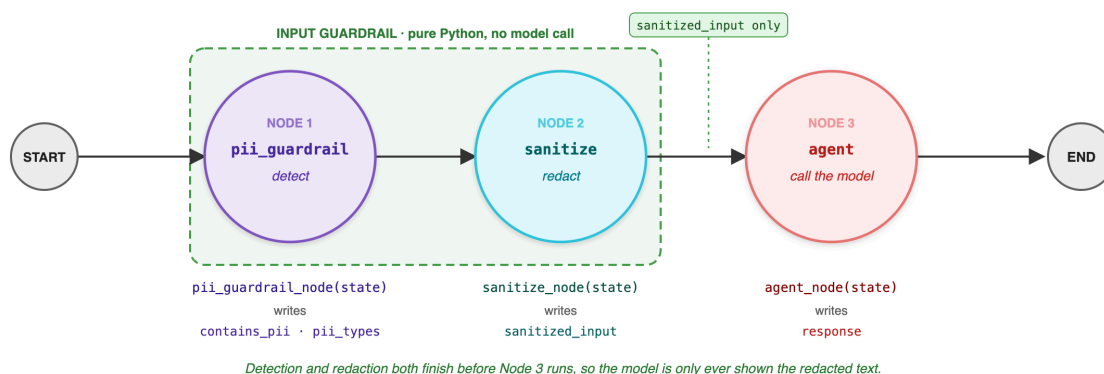


Fig 4.1: Three nodes, four edges, no branching. Each node writes its own keys into the shared state.

Node	Function	What it does	What it writes to state
1. pii_guardrail	pii_guardrail_node	Looks at the raw message and reports whether it contains PII, and which kinds	contains_pii, pii_types
2. sanitize	sanitize_node	Makes a copy of the message with every piece of PII replaced by a placeholder	sanitized_input
3. agent	agent_node	Sends only the cleaned copy to Claude and keeps the reply	response

The dashed box in the figure is the guardrail, and it is the reason this program comes first in the book. Both nodes inside it are plain Python using pattern matching, so they are fast, free, predictable, and impossible to talk out of doing their job. No customer message can persuade a regular expression to make an exception.

The dotted arrow matters just as much. Node 3 reads only `sanitized_input`. The raw message stays inside your application and never reaches the model.

What LangGraph Does, and What It Does Not

It is easy to assume the framework is doing the clever part. It is not.

LangGraph does not detect PII. It does not redact PII. It does not call Model. Those three jobs are done by ordinary Python functions that the program writes itself. LangGraph coordinates: it holds the shared state, calls each node in the order the edges describe, merges each result back into the state, and stops at END.

The wiring is nine lines.

```
builder = StateGraph(AgentState)

builder.add_node("pii_guardrail", pii_guardrail_node)
builder.add_node("sanitize", sanitize_node)
builder.add_node("agent", agent_node)

builder.add_edge(START, "pii_guardrail")
builder.add_edge("pii_guardrail", "sanitize")
builder.add_edge("sanitize", "agent")
builder.add_edge("agent", END)

graph = builder.compile()
```

That is worth remembering when a framework starts to feel like magic. The framework connects the pieces. Your own functions are the guardrail.

The Shared State

Every node reads from and writes to one shared notebook called the state. Here it is a typed dictionary with five entries.

Field	Meaning
<code>user_input</code>	The raw message exactly as the user typed it
<code>sanitized_input</code>	The same message with PII replaced by placeholders
<code>contains_pii</code>	True if at least one kind of PII was found

pii_types	The kinds found, such as ssn, or email and phone
response	Claude's reply to the sanitized message

A node does not return the whole state. It returns only the fields it changed, and LangGraph merges those in before calling the next node.

Here is the state for the demo message before and after the run.

Before:

```
{
  "user_input": "My SSN is 123-45-6789. Why was my payment declined?",
  "sanitized_input": "",
  "contains_pii": False,
  "pii_types": [],
  "response": ""
}
```

After:

```
{
  "user_input": "My SSN is 123-45-6789. Why was my payment declined?",
  "sanitized_input": "My SSN is [REDACTED_SSN]. Why was my payment declined?",
  "contains_pii": True,
  "pii_types": ["ssn"],
  "response": "...Claude's answer..."
}
```

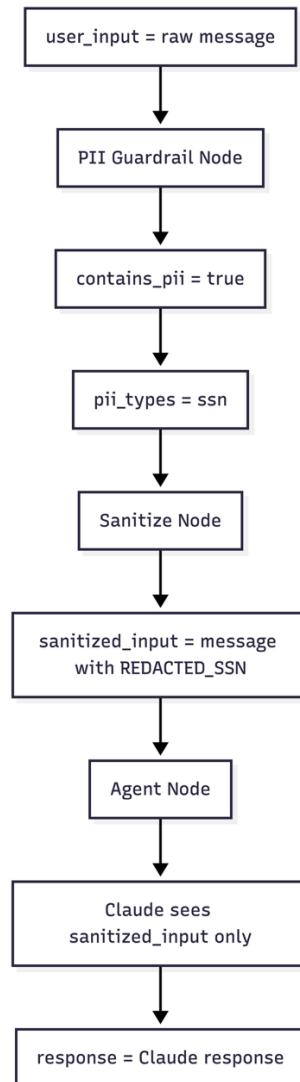


Fig 4.2: How the state fills up as the message moves through the graph

Notice that `user_input` never changes. The original is kept inside the application so a developer can confirm the guardrail worked. It is never part of the prompt.

What Counts as PII, and What It Becomes

The guardrail logic lives in `guardrail.py`. It defines four patterns and two small functions.

- `detect_pii(text)` returns a list of the kinds of PII present, such as `ssn` and `email`. An empty list means the message is clean.
- `sanitize_pii(text)` returns a new copy of the text with every match replaced by a labelled placeholder.

Kind	What the pattern looks for	Example	Becomes
------	----------------------------	---------	---------

SSN	Three digits, two digits, four digits	123-45-6789	[REDACTED_SSN]
Credit card	A run of 13 to 16 digits	4111 1111 1111 1111	[REDACTED_CARD]
Email	Something like name@domain.tld	jane@example.com	[REDACTED_EMAIL]
Phone	A US style number	(555) 123-4567	[REDACTED_PHONE]

The detection function is as plain as it looks:

```
def detect_pii(text):
    pii_types = []

    if SSN_PATTERN.search(text):
        pii_types.append("ssn")

    if CREDIT_CARD_PATTERN.search(text):
        pii_types.append("credit_card")

    if EMAIL_PATTERN.search(text):
        pii_types.append("email")

    if PHONE_PATTERN.search(text):
        pii_types.append("phone")

    return pii_types
```

And the sanitizer simply substitutes each pattern in turn:

```
def sanitize_pii(text):
    text = SSN_PATTERN.sub("[REDACTED_SSN]", text)
    text = CREDIT_CARD_PATTERN.sub("[REDACTED_CARD]", text)
    text = EMAIL_PATTERN.sub("[REDACTED_EMAIL]", text)
    text = PHONE_PATTERN.sub("[REDACTED_PHONE]", text)
    return text
```



Fig 4.3: A raw SSN goes in, a placeholder comes out

Both functions follow the contract from Chapter 3. They take text, they return a value, they know nothing about LangGraph, and they can be tested in one line.

Why Detection and Sanitization Are Two Jobs

The program keeps these as two nodes rather than one, and the reason is worth understanding, because it shows up again in every later program.

Detection answers what is in this message. Sanitization answers what should we do about it.

Today the answer to the second question is always replace it. But that is a policy, and policies change. A stricter version might redact emails and phone numbers while refusing to process a message at all if it contains an SSN or a card number. A regulated version might redact, store a record of what was found, and notify a compliance queue.

Every one of those changes is a change to the second node. The detector stays exactly as it is. Splitting the two keeps your policy cheap to change, which is the difference between a guardrail people improve and a guardrail people work around.

Why the Placeholders Have Labels

The placeholders are labelled on purpose. Claude sees [REDACTED_SSN] rather than a blank space or nothing at all.

That matters for the answer quality. The model can still tell that the customer mentioned their SSN, so it can reply sensibly, for example by reassuring them that it does not need the number to look into a declined payment. Strip the text silently and the model sees a strange half sentence and may ask the customer to repeat the very thing you just removed.

This is a small example of a large idea in guardrail design. **A good guardrail removes the risk without removing the meaning.** When a check makes the agent useless, people turn the check off.

A Complete Run

Running `python agent.py` uses the built in demo message.

Stage	Value
Original input	My SSN is 123-45-6789. Why was my payment declined?
Contains PII	True
PII types	['ssn']
Sanitized input, which is what Claude receives	My SSN is [REDACTED_SSN]. Why was my payment declined?
Agent response	Claude's answer about why payments get declined and what to do next

The only line that ever leaves the application is the sanitized input. The original is printed to the console purely so a developer can confirm the guardrail worked.

One production note on that last point. Printing the raw message is fine on your laptop and dangerous in a running service, because your console output usually becomes your log file. If you keep that print statement, you have moved the PII from the model into your logs, which is a different problem and not a smaller one.

Testing It

The tests never call a model. They run in milliseconds.

```
@pytest.mark.parametrize(
    "text, expected",
    [
        ("My SSN is 123-45-6789.", ["ssn"]),
        ("Call me at (555) 123-4567", ["phone"]),
        ("Card 4111 1111 1111 1111", ["credit_card"]),
        ("Email me at a@b.co", ["email"]),
        ("Why was my payment declined?", []),
        ("Meeting on 2024-01-15 at 10am", []),
    ],
)
def test_detect_pii(text, expected):
    assert g.detect_pii(text) == expected
```

Two of those rows are doing quiet but important work. The last two are negative tests: a clean message must produce an empty list, and a date must not be mistaken for an SSN. Without them, a stricter pattern added next month could start redacting every date in every message and nobody would notice until a customer complained that the agent could not read their invoice.

The sanitizer has its own test, and it checks both halves of the job:

```
def test_sanitize_redacts_all_pii():
    text = "SSN 123-45-6789, card 4111 1111 1111 1111, mail a@b.co"
    out = g.sanitize_pii(text)
    assert "123-45-6789" not in out
    assert "[REDACTED_SSN]" in out
```

The first assertion says the sensitive value is gone. The second says something meaningful took its place. A guardrail test should always check both, because a redactor that deletes the whole message passes the first assertion perfectly.

What This Guardrail Does Not Catch

This is the honest part, and it belongs in the chapter rather than in a footnote.

- **Words instead of digits.** My social is one two three, four five, six seven eight nine sails straight through.

- **Other formats.** The patterns are US shaped. An Aadhaar number, an IBAN, a UK national insurance number, or a passport number is not matched.
- **Split values.** A number spread across two messages, or broken by stray characters, can slip past.
- **False positives.** The card pattern looks for a run of 13 to 16 digits. A long order number or a tracking code can match it, and then the agent is arguing with a customer about an invoice it just redacted.
- **Names and addresses.** These are PII too, and no simple pattern finds them reliably.

So the honest summary of Program 01 is this. It stops the common, careless, high volume case, which is a customer pasting a real number into a chat box. It does not stop a determined attempt to get sensitive data through. For production you keep this same architecture and replace the patterns with a detection library or a small classifier model. That is Chapter 16, and the point of building it this way is that the swap touches one file.

The architecture is the durable part. The patterns are the disposable part.

Five Ways This Goes Wrong

- **Logging before redacting.** The guardrail runs, and two lines earlier the raw message was already written to a log. Redact first, log second, always.
- **Guarding only the chat box.** Documents, tool responses, uploaded files, and email bodies all reach the model too. Chapter 9 and Part III cover those doors.
- **Redacting too much.** If support genuinely needs the last four digits of a card, redact to the last four rather than removing the whole thing. A guardrail that breaks the job gets removed.
- **Silent placeholders.** Unlabelled redaction confuses the model and produces worse answers than labelled redaction.
- **Redacting the user message but storing the original.** If the raw text is saved to memory or to a transcript, you have only delayed the problem. Chapter 8 puts a guardrail on that door.

Where This Goes Next

The current graph always follows the same straight path. It never asks a question and never chooses between routes. Even a perfectly harmless message still travels through the sanitize node.

The next step is a graph that decides. If no PII is found, skip sanitizing and go straight to the agent. If the PII is an email or a phone number, redact it and continue. If it is an SSN or a card number, block the request entirely and never call the model.



Fig 4.4: The same guardrail with conditional

That change turns a fixed pipeline into a policy, where different kinds of sensitive data get different treatment. Conditional edges arrive properly in Chapter 6 with risk routing, and everything you have learned here carries over unchanged.

Try It Yourself

1. Run the demo, then call `run_agent(...)` with a message containing an email address and a phone number. Watch `pii_types` change.
2. Send a message with a fourteen digit order number and watch the card pattern fire. That is your first false positive, and it is a useful one to see early.
3. Write the message my social is one two three four five six seven eight nine and confirm that nothing is redacted. This is the limit of pattern matching, in one line.
4. Change the policy: if `pii_types` contains `ssn` or `credit_card`, do not call the model at all. Return a fixed message asking the customer not to share that information. You have just written your first blocking guardrail.

Key Takeaways

- An input guardrail cleans untrusted text before the model sees it, because once the model sees it, copies of it exist in places you do not control.
- Detection and sanitization are separate jobs, so the policy can change without rewriting the detector.
- The guardrail is plain Python: predictable, free, and impossible to argue with.
- The model receives only the sanitized text. The raw message stays in the application, and it must not reach your logs either.
- Labelled placeholders keep the meaning while removing the risk.
- LangGraph coordinates the flow. Your own functions do the work.
- Patterns catch the careless case, not the determined one. Keep the architecture, upgrade the detector later.

Program 01 protects the model from the user. In the next chapter we turn around and protect the business from the model, by checking an action before it runs.