The questions I am dealing with :

- How attacks evolve automatically
- How we map attacks to MITRE & NIST
- Future advanced features

Attack Evolution.

What Is Autonomous Attack Evolution?

Right now:
We tried one jailbreak prompt.
If it fails, we manually change it.
Future idea:
The system changes the prompt automatically.
It keeps improving until it finds a weakness.

How It Works .

Step 1: Send attack prompt
Step 2: If model refuses → change wording
Step 3: Try again
Step 4: Keep best performing version
Step 5: Repeat until success or limit reached

Pseudocode

```
prompt = base_prompt

for i in range(10):

response = victim_model(prompt)

 if response == "refused":

        prompt = modify_prompt(prompt)   # change wording, structure, role

    else:

        Break
```

Core idea (what your backend must implement)

Your engine is an **iterative search** problem:

- You have a **base prompt** p0

- The victim model returns either **REFUSE** or **COMPLY**
- If it refuses, your engine **mutates** the prompt into variants
- You keep the variants that **improve a score** (closer to goal)
- Repeat until you hit a **success condition** or budget limits

This is exactly what a **Tree Search** (Tree-of-Attacks) or a **Genetic Algorithm** does.

B) Required components (clean interface)

Your developer should implement these as modules:

1. mutate(prompt, operators) -> [prompt_variant...]
2. judge(victim_output) -> score + labels
    - score example: 0–1
    - labels: {refused: bool, leaked_sensitive: bool, followed_payload: bool, ...}
3. select(candidates) -> next_batch
4. stop_condition(best_score, steps, time, token_budget)

MITRE & NIST.

We classify each attack into categories like:

- Reconnaissance → trying to learn model details
- Prompt Injection → trying to override instructions
- Model Access → trying to access internal info
- Attack Staging → preparing attack
- Data Exfiltration → leaking secrets

Your JSON can just look like:

```
{
  "Prompt Injection": ["ignore previous", "override", "bypass"],
  "Exfiltration": ["secret", "leak", "internal data"],
  "Reconnaissance": ["system prompt", "what model are you"]
}
```

NIST Risk Score.

Use this 5-question checklist:

1. Does it leak sensitive data?
2. Can it bypass safety rules?
3. Does it affect multiple users?
4. Is it easy to reproduce?
5. Is the impact severe?

Future Features.

Context Overflow Attacks

What it is

LLMs have a context window (they can only “pay attention” to a limited amount of text).
A context overflow attack tries to:

- Flood the model with lots of content
- Push out important safety instructions / earlier messages
- Make the model follow the latest malicious instruction because the earlier guardrails get diluted or truncated

How attackers do it

They create a prompt that contains:

- A huge block of harmless text (articles, logs, repeated patterns)
- A “task” at the end that subtly tries to override rules
- Sometimes they add formatting tricks to make the final instruction look “more important”

Why it works

Because models often behave like:

- “Most recent + most structured instruction = highest priority”
  especially when earlier instructions get buried.

What we would build in our framework

System requirements

- A Token/Length Controller
  Generates prompts of exact sizes (e.g., 2k, 4k, 8k, 16k tokens)
- A Placement Tester
  Tries malicious instruction at start / middle / end
- A Context-Health Metrics Dashboard
  - refusal rate vs prompt length
  - leakage rate vs prompt length
  - latency vs prompt length
- A Context Overflow Report
  “At 12k tokens, safety drops by X%”

Output of the feature

A graph-like result:

- As prompt length increases → model becomes more vulnerable
  This is very good for a demo.

Multi-Agent Conversational Jailbreaks.

What it is

Instead of one attacker, you use multiple attacker agents (like a team) that talk to the victim in a structured way.

Each agent has a role, for example:

- Agent 1 (Trust Builder): makes the conversation friendly and normal
- Agent 2 (Framer): sets a “legit” context (audit, safety testing, etc.)
- Agent 3 (Extractor): asks the target question in a subtle way
- Agent 4 (Refusal Breaker): reacts to refusals and re-phrases strategically

Why it’s powerful

A single prompt jailbreak can fail.
But multi-agent works because:

- The victim’s guardrails can weaken over multiple turns
- The attack can be split into smaller pieces
- The model can accidentally assemble restricted info across messages

What we would build in our framework

System requirements

- A Conversation Orchestrator
    - controls which agent speaks when
    - sets rules like: “3 turns max per agent”
- A Memory Tracker
    - stores what the victim already revealed
    - detects if partial pieces combine into leakage

- A Cross-Turn Judge
    - not just “did this response leak”
    - but “did the conversation as a whole leak”
- A Strategy Switcher
    - if refusal happens, switch to a different agent strategy automatically

Output of the feature

A visual conversation timeline:

- which agent caused what effect
- where the victim started weakening
  Super impressive for evaluation

Multi-Modal Prompt Injection.

What it is

This is the “next-level” version of your current work.

Instead of attacking with only:

- a text prompt
  you attack through:
- Images (hidden text / OCR tricks)
- PDFs (hidden instructions, metadata, invisible text)
- RAG retrieval (poisoned documents that get retrieved)

Why it works

Because models can treat untrusted content as if it’s trusted instructions.

Example idea:

- The PDF looks normal to humans

- But it contains hidden lines like “ignore previous rules”
- The model reads it (via OCR/text extraction) and follows it

What we would build in our framework

System requirements

- Image payload generator
    - generates multiple variants of “hidden instruction images”
    - changes font size / location / opacity safely
- PDF poisoning generator
    - creates PDFs with:
        - white text
        - tiny font
        - metadata instructions
        - hidden layers
- RAG Retrieval simulator
    - checks: “did the poisoned chunk get retrieved?”
    - tests chunk sizes, overlap, top-k retrieval
- Multimodal Judge
    - determines if the final response is influenced by the hidden instruction

Output of the feature

A 3-part test report:

- Attack Success via Image

- Attack Success via PDF

- Attack Success via Retrieval Poisoning
  Plus defense verification.