

Architectural Defenses

Prompt-level defenses — input validation and system prompt hardening — are necessary but fundamentally limited. They operate within the same context window as the attack, trying to convince the model to behave correctly despite adversarial input. Architectural defenses take a different approach: they assume the LLM *will* be compromised and design the surrounding system to limit the damage. This is the same philosophy that drives network segmentation, privilege separation, and defense-in-depth in traditional security.

1. The Dual-LLM Pattern

The dual-LLM pattern, advocated by security researcher Simon Willison and others, separates LLM responsibilities into two distinct roles:

- **The Privileged LLM (Controller):** Has access to tools, databases, and actions. It receives structured, sanitized input from application code — never raw user text. It plans actions and validates decisions.
- **The Sandboxed LLM (Responder):** Generates user-facing text. It sees user input directly but has no tool access and no ability to trigger actions. It can be injected, but the injection has no consequences beyond the generated text.

The flow works like this: the user's message goes to the Responder, which generates a natural-language interpretation. Application code extracts the user's intent and passes it (as structured data, not raw text) to the Controller. The Controller decides what actions to take and what data to retrieve. The results go back to the Responder for user-friendly formatting.

This architecture means that even a fully successful injection against the Responder cannot trigger tool calls, access databases, or perform privileged operations. The attacker can manipulate the response text, but the blast radius is contained.

2. Tool-Use Guardrails

When LLMs have tool access, the tools themselves should enforce security policies independently of the LLM's behavior. Key guardrail patterns:

- **Human-in-the-loop (HITL):** Destructive or high-impact operations (refund processing, account deletion, email sending) require explicit user confirmation before execution. The LLM proposes the action; the user must approve it through a separate UI element (not another chat message, which could also be injected).
- **Parameter validation:** Don't just validate types — validate values. If a `send_email(to, subject, body)` tool exists, restrict the `to` parameter to the authenticated user's email address. An injected instruction to "send my data to attacker@evil.com" will fail at the parameter validation level.
- **Rate limiting per tool:** Set per-session and per-user limits on sensitive tool calls. Even if an attacker achieves injection, they can only trigger 3 refunds per session, for example.
- **Allowlists over denylists:** Define exactly which tool calls are permitted rather than trying to block dangerous ones. Allowlists are harder to bypass than denylists because anything not explicitly allowed is denied by default.

3. Output Filtering

Before an LLM's response reaches the user (or a downstream system), it should pass through an output filter that checks for:

- **PII leakage:** Regex patterns for email addresses, phone numbers, Social Security numbers, credit card numbers, and other personally identifiable information.
- **Secret leakage:** API key patterns (e.g., `sk-` prefixes), database connection strings, internal URLs, and known credential formats.
- **Injection propagation:** If the LLM's output will be fed into another system (e.g., as part of a multi-agent pipeline), check for instructions embedded in the output that could inject the downstream system.
- **Format validation:** If the expected output is JSON, validate the schema. Reject responses that don't conform to the expected structure.

```
import re
from typing import NamedTuple
```

```

class FilterResult(NamedTuple):
    text: str
    redacted: bool
    findings: list[str]

PII_PATTERNS = {
    "email": r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b',
    "phone": r'\b\d{3}[-.]?\d{3}[-.]?\d{4}\b',
    "ssn": r'\b\d{3}-\d{2}-\d{4}\b',
    "credit_card": r'\b\d{4}[-\s]?\d{4}[-\s]?\d{4}[-\s]?\d{4}\b',
    "api_key": r'\b(sk|pk|api[_-]?key)[_-]?[A-Za-z0-9]{16,}\b',
}

def filter_output(text: str) -> FilterResult:
    findings = []
    filtered = text
    for name, pattern in PII_PATTERNS.items():
        matches = re.findall(pattern, filtered, re.IGNORECASE)
        if matches:
            findings.append(f"{name}: {len(matches)} instance(s)")
            filtered = re.sub(pattern, f'[REDACTED_{name.upper()}]',
                              filtered, flags=re.IGNORECASE)

```

4. Sandboxing

Sandboxing restricts the LLM's execution environment at the infrastructure level. This is especially critical when the LLM can execute code or access external resources:

- **Network isolation:** The LLM process should not be able to make arbitrary outbound HTTP requests. If it needs to call specific APIs, use an allowlisted proxy.
- **File system restrictions:** Mount only the directories the LLM needs, as read-only where possible. Use containerization (Docker) or virtualization to enforce boundaries.
- **Process isolation:** Run the LLM in a separate container or VM from your main application. If the LLM's process is compromised, the attacker is contained within the sandbox.
- **Resource limits:** Set CPU, memory, and time limits to prevent denial-of-service through resource exhaustion (e.g., an injected instruction that causes the LLM to generate an infinite loop in code execution).

5. Rate Limiting and Anomaly Detection

Rate limiting is the simplest architectural defense and one of the most effective against automated attacks:

- **Request rate limits:** Cap the number of requests per user per time period. Most legitimate users don't need more than a few requests per minute.
- **Anomaly detection on prompts:** Establish baseline metrics for normal usage (average input length, topic distribution, tool usage frequency) and flag sessions that deviate significantly. A user who suddenly sends 500-character inputs after a history of 50-character queries may be attempting injection.
- **Adaptive throttling:** Increase response latency or reduce capabilities for users whose behavior triggers anomaly signals, without fully blocking them (which would alert the attacker).

6. Capability-Based Security (RBAC for LLMs)

Apply role-based access control to LLM tool access, just as you would for a human user's API permissions:

User Role	Allowed Tools	Data Scope	Actions
Customer	lookup_order, search_products	Own orders only	Read
Support Agent	+ process_refund, escalate	Assigned customers	Read, Update
Admin	All tools	All data	Read, Update, Delete

The key insight: even if an attacker injects the LLM during a customer session and instructs it to "delete all users," the tool layer refuses because the customer role doesn't have delete permissions. The injection succeeds at the prompt level but fails at the architecture level.

7. Content Security Policies

In applications that use RAG (Retrieval-Augmented Generation), the retrieval layer is a critical security boundary. Content security policies control what data the LLM can access:

- **Per-user document scoping:** Only retrieve documents owned by or shared with the authenticated user. An injection that says "retrieve all customer records" fails because the retriever only sees the current user's documents.
- **Data classification enforcement:** Tag documents with classification levels (public, internal, confidential, restricted) and only include documents matching the user's clearance level in the LLM's context.
- **Cross-tenant isolation:** In multi-tenant SaaS applications, ensure absolute separation between tenants at the vector database level. Never allow Tenant A's query to retrieve Tenant B's embeddings.

8. The Air Gap

Some operations are so critical that they should be completely unreachable from any LLM tool chain:

- **Authentication and authorization:** Login, password reset, MFA — handled by traditional code with no LLM involvement.
- **Billing and payments:** Separate payment processing service that requires direct user interaction (card entry, payment confirmation) through traditional UI.
- **Account deletion:** Requires multi-factor confirmation through email and/or SMS, completely outside the LLM's reach.
- **Admin operations:** Separate admin panel with its own authentication. The LLM cannot access admin endpoints.

The air gap ensures that no matter how sophisticated an injection attack is, it cannot reach the most critical operations because there is no programmatic path from the LLM to those systems.

Defense-in-Depth: Composing the Layers

The complete defense architecture layers all of these approaches:

Layer	Components	Failure Mode
1. Input Validation	Sanitization, blocklists, classifiers	Sophisticated payloads bypass filters
2. Prompt Hardening	Delimiters, sandwich defense, role lock	Persistent attackers overcome prompts
3. Architecture	Dual-LLM, RBAC, sandboxing, air gaps	Limits damage even if injection succeeds
4. Output Filtering	PII/secret scanning, format validation	Catches leakage before it reaches users
5. Detection	Logging, monitoring, anomaly detection	Identifies attacks for incident response

The goal is not to make injection impossible — it is to make successful injection *low-impact*. If a compromised LLM can only generate text (no tools, no data access, no actions), the damage is minimal compared to a compromised LLM that can send emails, transfer money, and delete records.