Cross-Chapter Handout: The Phase-1.6 LLM-Tutor 3-Layer Architecture (AI-301 / Belt 5; substrate-versus-language bridge)

5,236 words · ~24 min read

Camera-ready handout, 2026-05-07. Audience: VCA-AI-301 students (Belt 5; capstone register; substrate-versus-language thesis). Authoritative source for the LIVE LLM-tutor architecture treated as a programming-language artifact rather than a chatbot. Use this handout to recognize that an LLM tutor is itself a multi-tier interpreter for natural-language programs, to author capstone-quality substrate-versus-language analyses, and to bring the academy's CSA-side silicon literacy into productive contact with the AI-side language-runtime literacy.

Anchored in the live implementation serving virtuscyberacademy.org/tutor/. Companion handout: `cross-chapter-llm-tutor-3-layer-architecture-ai-201.md`*. Belt-5 students should read the AI-201 companion first for the architectural surface walk; this handout treats the LLM-tutor as a thesis vehicle, not as an architectural tour.*

Petzold weaves: deliberately none. Petzold's CODE 1st edition does not reach the modern LLM stack; the right Belt-5 anchors are Pierce TAPL (Types and Programming Languages) and Krishnamurthi PLAI (Programming Languages: Application and Interpretation). Toolchain Diary entries §10 below name both. The substrate-side anchor from Petzold lives in CSA-101 and CSA-201, where the academy's silicon-side argument is built.

---

§0. What this handout is for

You are a Belt-5 student. You finished AI-201 by red-teaming a tutor of this class through the engagement-quality lens; you finished CSA-101 by building an RV32I-Lite CPU in HDL on a Tang Primer 25K; you finished CSA-201 by extending that CPU with privilege levels, MMU, and a driver-writing track. You arrive at AI-301 with two literacies that the

rest of the cybersecurity training market does not pair: substrate (silicon) and language (LLM stack). The capstone register asks you to demonstrate that the pairing produces something neither half can produce alone.

This handout is the substrate-versus-language bridge. It treats the Phase-1.6 LLM tutor at https://virtuscyberacademy.org/tutor/ as the worked example. The architectural surface (three layers, ten tools, anonymous-cookie identity, Postgres persistence, sliding window, structured summarization) is covered in the AI-201 companion handout; this handout assumes you have read it and treats Phase-1.6 as a programming-language artifact rather than a chatbot. The system prompt is a constitution. The KB tools are a runtime's intrinsic functions. The agent loop is the eval-apply loop of a metacircular interpreter. Layer 2 compaction is garbage collection. Layer 3 cross-session memory is persistent heap. The anonymous cookie is a capability token. These are not metaphors; they are the actual category each piece occupies.

The capstone register asks you to walk this map for a tutor of your own design, then to red-team it, then to write the substrate-versus-language analysis that closes your AI-301 portfolio. This handout is the anchor for that arc.

---

§1. The 3-layer architectural contract (compressed; AI-201 companion has the full walk)

Phase-1.6 is three layers cooperating at the request boundary. Layer 1 is the per-message context: a skinny base prompt (~150 lines maximum), a per-persona overlay, a per-mode overlay, an optional lab-mode overlay, plus six knowledge-base tools (`get_chapter_section`, `get_filter_grammar`, `get_recipe_library`, `get_anchor_quote`, `inspect_packet` (client-dispatched), `get_fixture_meta`). Layer 2 is within-session memory: a sliding verbatim window (K=10 turns) plus a Pydantic-shaped structured summary that materializes when the window slides past a 50%-of-budget compression trigger. Layer 3 is cross-session per-student memory: a Postgres schema (four tables: `students`, `concept_events`, `lab_progress`, `session_summaries`) keyed by an opaque anonymous student ID minted into a first-party `virtus_student_id` cookie, with four read/write tools the model uses to access it (`get_student_state`, `get_session_history`, `record_concept_mastered`, `record_misconception`).

The contract that ties the layers is the tool-mediated-state-versus-prompt-stuffing discipline: state that could live in the prompt instead lives behind tool calls the model elects to issue. Implementation paths are in the AI-201 companion §1-§5; the source is at `/media/laptop/data4t/laptop/jupyter/virtus-academy/pcap-tools/backend/tutor/`.

§2. Layer 1 (compressed)

Six Layer-1 KB tools surface the canonical-knowledge-lookup paths the tutor should not paraphrase from training data. The tool definitions live at `pcap-tools/backend/tutor/tools/__init__.py` (467 lines) as an OpenAI-tool-shape `TOOLS` list. Five dispatch server-side (`tools/dispatch.py`); `inspect_packet` dispatches client-side (`CLIENT_TOOLS = frozenset({"inspect_packet"})` at `tools/__init__.py:448`). The tool descriptions explicitly favor tool-call over training-recall: "Prefer this over recalling from training data -- academy chapters are the authoritative source."

The prompt envelope: ~600-1200 system-prompt tokens before any tool dispatches anything. A prompt-stuffing alternative on the same use case would run 8000-15000 tokens of system prompt per request. The tradeoff is the architecture's first pedagogical claim: invest in tools, not in fat prompts.

§3. Layer 2 (compressed)

Sliding window K=10 turns; 50%-of-budget compression trigger; structured `SessionSummary` schema with seven fields (`concepts_introduced`, `misconceptions_flagged`, `fixtures_loaded`, `filter_history`, `tool_calls_consumed`, `preferred_pacing`, plus session metadata). Two summarizer backends behind one interface: rules-based (default; ships in Phase-1.6; deterministic regex-and-keyword scan; sub-10ms; `session_compressor.py:33-95`) and LLM-based (Phase-1.6.1 second wave; cheaper-tier model call; ~200-500ms; not yet shipped).

Re-inflation discipline: re-call the dropped tool, do not re-probe from the structured summary. The architecture nudges the model toward the re-call path through the `ToolCallConsumed.result_summary` field, which is informative enough to decide whether to re-call but not informative enough to substitute for re-call.

§4. Layer 3 (compressed)

Postgres schema at `store/migrations/0001-initial.sql` (four tables: `students`, `concept_events`, `lab_progress`, `session_summaries`). Anonymous-cookie identity via `student_id.py:get_or_mint_student`: cohort bearer token plus persisted `virtus_student_id` cookie are the two identity inputs; the cookie carries an opaque `stu_<urlsafe-16>` (~96 bits of entropy via `secrets.token_urlsafe(16)`); the cookie attributes are `HttpOnly` plus `Secure` plus `SameSite=Lax`. Four Layer-3 tools the model

uses (`get_student_state`, `get_session_history`, `record_concept_mastered`, `record_misconception`); all four are constrained by `LAYER3_TOOLS = frozenset({...})` at `tools/__init__.py:453`, which causes the dispatcher to refuse to run them without a `ToolContext` carrying the resolved `StudentIdentity`. The model never sees student_id values; it asks for "the current student's state" and the dispatcher resolves which student that is from the cookie.

Cross-cohort portability via `students.all_cohorts[]` (a Postgres `TEXT[]` array): the student_id is durable; the cohort is decorative metadata over the durable identity.

---

§5. The tool-mediated-state contract (compressed)

The contract that ties the three layers: state that COULD live in the prompt lives instead behind tool calls. Token economics, auditability, and honesty about training-data drift are the three reasons the contract earns its place (full treatment at AI-201 companion §5). The Belt-5 adds a fourth reason: the contract turns the LLM tutor into a programming-language artifact whose semantics are inspectable. That fourth reason is the bridge to §6.

---

§6. The substrate-versus-language thesis through the Phase-1.6 lens

The critical section. AI-301's pedagogical move: a software system is a stack of substrates and languages, and the right way to reason about either half is to know which is which at every level. CSA-101 and CSA-201 give you the substrate side: silicon, ISA, machine code, assembler, linker, VM, compiler, OS. AI-301 gives you the language side at the LLM tier: prompts, tool definitions, agent loops, structured summaries, persistence layers. Phase-1.6 is the worked example that holds both sides in the same frame.

§6.1 The system prompt is a constitution

A constitution declares the polity's name, the scope of legitimate authority, the procedure for resolving disputes, and the obligations every member owes. It does not contain the laws; it contains the rules under which the laws will be made. The Phase-1.6 base prompt at `pcap-tools/backend/tutor/prompts/base.md` is ~35 lines that declare the tutor's identity ("an academy tutor for cybersecurity learners"), the scope of legitimate authority ("answers grounded in academy materials"), the procedure for resolving

disputes ("declines to fabricate; defers to chapter content over training data"), and the obligations the tutor owes the student ("follows a Socratic discipline when in lab mode"). The persona overlays are the polity's amendments. The mode overlays are the polity's procedural rules. The lab-mode overlay is the polity's emergency clause that activates under specified conditions.

The implication for a Belt-5 student authoring a tutor: the constitution is not where you put the knowledge. The constitution is where you put the rules under which the knowledge will be retrieved, summarized, and surfaced. Put the knowledge in the corpus and reach it through tools.

§6.2 The KB tools are a runtime's intrinsic functions

Every programming language has intrinsics: operations the language exposes directly to the program rather than through a library. C has `+`, `-`, `*`, `/` as intrinsics over integers and floats; the standard library is a separate concern. Python has dictionary indexing, attribute access, function call as intrinsics; the standard library lives outside. The Phase-1.6 KB tools are the runtime's intrinsics for the LLM-program: `get_chapter_section` is the chapter-fetch intrinsic; `get_filter_grammar` is the grammar-resolve intrinsic; `get_recipe_library` is the recipe-search intrinsic; `get_anchor_quote` is the verbatim-quote intrinsic; `inspect_packet` is the packet-dissect intrinsic (dispatched by the frontend, which is the runtime's I/O subsystem in this metaphor); `get_fixture_meta` is the fixture-metadata intrinsic.

The implication for a Belt-5 student: the choice of intrinsics is the choice of what the language treats as primitive. Add an intrinsic and you have changed what the program can express directly. Phase-1.6's six KB tools are a deliberate primitive set; the four Layer-3 tools are a deliberate state-mutation set. A tutor in another domain (medical, legal, financial) would have different intrinsics shaped by the canonical-knowledge paths of that domain. The intrinsic set IS the language's domain-specificity.

§6.3 The agent loop is the eval-apply loop of a metacircular interpreter

A metacircular interpreter implements a language inside itself: Lisp interpreting Lisp; Scheme's `eval` and `apply` cooperating across a recursive descent into the AST; the interpreter's state being the program's environment plus the program's continuation. The Phase-1.6 agent loop, in `pcap-tools/backend/tutor/llm_client.py` (545 lines) and `main.py` (583 lines), is structurally identical: the LLM's response is parsed for `tool_use` chunks (the AST nodes); the dispatcher applies them (`dispatch_tool(name, args)`); the

result is appended to `messages` (the environment); the next `litellm.acompletion` call extends the recursion (the continuation). When the response is a final text rather than a tool call, the recursion bottoms out and the result returns to the caller.

The implication for a Belt-5 student: the agent loop is the language's eval-apply. Knowing this lets you reason about the language's termination, soundness, and performance using the same frames you would use for any interpreter. Termination: does the loop always converge on a final text response? Phase-1.6's `budget.py` budget cap is the language's termination guarantee. Soundness: does the dispatcher faithfully execute what the model asked? Phase-1.6's enum-restricted parameters and `LAYER3_TOOLS` discipline are the soundness anchors. Performance: how many recursion-levels does a typical request take? Phase-1.6's `cost_log.py` per-tool attribution is the performance instrumentation.

§6.4 Layer 2 compaction is garbage collection of working memory

Garbage collection identifies live objects, retains them, and reclaims the rest. The reclaimed memory is reused for new allocations; the retained objects keep their bindings. Phase-1.6's Layer 2 compaction does the same thing at the conversation tier: identifies the live signal in the dropped turns (concepts introduced; misconceptions flagged; fixtures loaded; filters tried; tool calls consumed; pacing preferences); retains them as a `SessionSummary`; reclaims the verbatim text. The reclaimed prompt-budget is reused for new turns; the retained structured signal keeps its semantics.

The classical GC choice is mark-and-sweep versus generational versus reference-counting. Phase-1.6's compaction is closest to generational: the verbatim window is the young generation (everything starts there; most of it gets reclaimed quickly); the structured summary is the old generation (the survivors of the young-generation pass; cheap to keep around because the schema is small). The rules-based summarizer is a deterministic mark phase; the planned LLM-based summarizer (Phase-1.6.1) is a smarter but more expensive mark phase. Both reclaim the same young-generation territory; both produce the same old-generation shape.

The implication for a Belt-5 student: knowing the GC analogy lets you reason about the architecture's pathological cases. A turn that introduces a high concept density would mark heavily; the survivors would dominate the structured summary; the next compression boundary might find very little to reclaim. The runtime designer's response is the same: tune the trigger threshold; the academy's response is the same: tune the K=10 window or the 50%-of-budget threshold per the cohort's empirical conversation lengths.

§6.5 Layer 3 cross-session memory is persistent heap with namespaced regions

A persistent heap survives the end of program execution. Erlang's mnesia, Smalltalk's image, Common Lisp's image-based development; the heap-on-disk pattern is a recognized programming-language design point. Phase-1.6's Layer 3 is a persistent heap keyed by student-id: each student has their own region (`students.student_id` is the namespace); the regions do not alias (the dispatcher resolves student_id server-side and refuses Layer-3 tools without a resolved StudentIdentity). The four Layer-3 tools are the heap's read/write API; the schema is the heap's type system.

The classical question for a persistent heap is "what guarantees does the heap make against concurrent access?" Phase-1.6's posture: each student is a single concurrency unit (one browser, one cookie, one session at a time per the academy's normal access pattern); cross-student concurrency is not a concern because the regions do not alias; cross-session concurrency for a single student is rare (tabs sharing a cookie, perhaps; the architecture does not commit to ACID across that case). The Postgres-level concurrency primitives (transactions, row-level locks) are available if a future cohort tier needs them; Phase-1.6 chooses not to spend the complexity in v1.6.

The implication for a Belt-5 student: persistent-heap-with-namespaced-regions is a recognizable pattern from the language-runtime literature; the academy's choice is a particular point on a recognized design space, not an ad-hoc decision.

§6.6 The anonymous cookie is a capability token

Object-capability security (Mark Miller's E language; Capn Proto's Cap'n Proto RPC; the Pony language's capability-secure type system) treats a capability as an unforgeable reference that grants a specific authority. The cookie at `pcap-tools/backend/tutor/student_id.py` is exactly this shape: an opaque random token (~96 bits of entropy) that grants the bearer the authority to read and mutate the keyed student-state; the cookie is unforgeable in the sense that guessing one is computationally infeasible; the cookie is delegable in the sense that copying it transfers the authority (which is also why the academy commits to "medium" privacy posture, not "high"). The cohort bearer token is a separate capability that grants temporal-scope authority over the cohort's resources; the two compose to authorize a session.

The implication for a Belt-5 student: the privacy posture's foundational assumption (the cookie is held only by its rightful owner) is the standard object-capability assumption, and the architecture's threat model is the standard one for capability-secure systems.

When you author your capstone tutor, you should be able to state your capability model explicitly: what tokens are in play, what authority each grants, how they compose, what attacks against the composition produce what failures.

§6.7 Cohort tokens are compilation units / linkage scope

A compilation unit is a self-contained piece of source that links into a larger artifact: a `.c` file in C, a `.py` module in Python, a crate in Rust. The cohort token in Phase-1.6 is the linkage scope for an academy-side cohort: it grants access to the cohort's resources (chapters, fixtures, recipes, the tutor's lab-mode bindings); a student-id can have multiple cohort tokens accumulated over time (`students.all_cohorts[]`), and the tutor's behavior can vary by which cohort context is current. The student is the durable artifact (the student-id); the cohort is the linkage tier.

The implication for a Belt-5 student: when you design a tutor for multiple cohort tiers (free trial, paid cohort, alumni access), the cohort token is the right place to put the per-cohort policy variation. The student's learning data does not change shape with cohort; the policies do.

§6.8 Tool-mediated-state-versus-prompt-stuffing recapitulates lexical-versus-dynamic scope

Lexical scope (the variable's binding is determined by where the variable is declared in the source text) and dynamic scope (the variable's binding is determined by the call stack at runtime) are the two classical scoping disciplines. Common Lisp uses dynamic scope by default for special variables; Scheme and most modern languages use lexical scope. The debate has been adjudicated in favor of lexical scope for most cases because lexical scope is easier to reason about: you can read the source and know what binds where.

Phase-1.6's tool-mediated state is the lexical-scope move at the LLM tier. The state binds where the tool definition lives; the model fetches it on demand; the model's behavior is determined by what the source (the tool definition) declares plus what the tool actually returns. Prompt-stuffing is the dynamic-scope move: the state binds at the call site (this request's prompt); the model's behavior is determined by what the call site embedded plus what the model recalls. The lexical approach gives you reading-the-source determinism; the dynamic approach gives you the convenience of "everything is here" at the cost of opacity. Phase-1.6 chooses lexical.

§6.9 The prompt → tool-call → response loop is a small-step operational semantics

Small-step operational semantics describes a language by giving the rules under which a single computation step transitions to the next: the configuration is the program's state; the rule says how each kind of state transitions; the computation is the chain of transitions until a final state. Phase-1.6's agent loop has small-step shape: a configuration is `(messages, model_response_in_progress)`; the rules are (a) if the model is composing text, append the text and continue, (b) if the model emits a `tool_use`, apply the dispatcher and append the `tool_result`, (c) if the model emits a final text, terminate. A request is a chain of these transitions until rule (c) fires.

The implication for a Belt-5 student: knowing this lets you write down the architecture's semantics in a paper rather than in prose. Pierce's TAPL chapter on operational semantics is the standard reference for the formal frame; Phase-1.6's agent loop fits the frame well enough that a Belt-5 student authoring a substrate-versus-language analysis can use the formalism without inventing it.

§6.10 Substrate at the bottom; language at the top; correspondence at every level

The substrate-versus-language thesis closes with a reminder. The CSA-101 graduate has built the bottom: NAND gates to ALU to register file to RAM to RV32I-Lite CPU to assembler to linker to VM translator to Jack-equivalent compiler to Virtus OS. The AI-301 graduate has read the top: prompt-as-constitution, tools-as-intrinsics, agent-loop-as-eval-apply, compaction-as-GC, persistent-memory-as-heap, cookies-as-capabilities, cohort-as-linkage. Every level has a substrate that does the work and a language that names what the work is for. A bug at any level is a mismatch between substrate and language at that level. A capstone-quality finding at any level is a precise statement of the mismatch and the correction.

The Belt-5 is this thesis applied at AI-tier without losing the discipline that the CSA-tier graduate brought. You are not just red-teaming a chatbot; you are reading a programming-language artifact and producing a finding-and-correction that an academy operator can put into the next deploy.

---

§7. Architecture Comparison Sidebar: Phase-1.6 vs three other LLM-tutor architectures

Across LLM-tutor architectures: where does the knowledge live?

Phase-1.6 is one design point on a populated map. Three other points: a naive prompt-stuffing tutor that embeds the corpus in the system prompt, a vector-DB RAG tutor that retrieves by similarity-search at request time, and a fine-tuned per-cohort tutor that bakes the corpus into the model's weights. Each point honors a different cost model, audit shape, and red-team surface.

The naive prompt-stuffing tutor is the simplest to build. The corpus is in the system prompt; every request pays for the full corpus in tokens; the audit trail is opaque (you cannot tell from a transcript whether the model used a given chapter or just sounded like it did). The cost grows linearly with corpus size and with cohort scale. The red-team surface is concentrated on prompt-injection and prompt-leakage: a successful injection can make the model ignore the corpus; a successful leakage exposes the corpus to a competitor. The architecture is the right choice for prototypes and small one-cohort runs; it is not the right choice for an academy operating at any scale.

The vector-DB RAG tutor is the contemporary default in most enterprise contexts. The corpus is embedded into a vector database (FAISS, Pinecone, pgvector); the request triggers a similarity-search retrieval; the retrieved chunks are concatenated into the prompt; the model composes from the concatenation. The cost is bounded per request (a fixed retrieval count regardless of corpus size); the audit trail is partial (you can see which chunks were retrieved but you cannot easily see whether the model used them faithfully); the red-team surface adds RAG-poisoning (corrupt the embeddings; the retrieval surfaces poisoned chunks) and embedding-collision (craft a query whose embedding is close to a poisoned chunk's embedding). The architecture is the right choice for unstructured corpora and informal knowledge; it is overkill for an academy with a structured chapter corpus and FTS5-indexable content.

The fine-tuned per-cohort tutor goes the opposite direction: bake the corpus into the model's weights via fine-tuning (LoRA, full fine-tune, prefix-tuning). The cost is high upfront (fine-tuning hours; compute; quality-eval cycles) and low per-request (no corpus tokens; just the user's question); the audit trail is the worst of the four (the corpus is not retrievable from the model's behavior; you can probe but not verify); the red-team surface adds model-extraction and training-data-extraction attacks.

> The architecture is the right choice for proprietary corpora that should never appear verbatim in transcripts; it is the wrong choice for an academy whose curriculum value depends on transcripts being verifiable.
>
> Phase-1.6 sits at a fourth point: tool-mediated retrieval against a structured corpus, with explicit `tool_use` and `tool_result` messages in the transcript. The cost is bounded (only the elected fetches cost tokens); the audit trail is the best of the four (every retrieval is in the transcript); the red-team surface is distributed across the tool boundary (which the AI-201 companion §6 walks). The architecture's cost-per-cohort scales sub-linearly because most students do not exercise most tools per request; the empirical hit-rate per request is lower than the request rate by roughly an order of magnitude.

Facet	Phase-1.6 (skinny + tools + Postgres)	Naive prompt-stuffing	Vector-DB RAG	Fine-t
KB layer	Six KB tools fetch on demand	Whole corpus in system prompt	Vector DB; similarity-search retrieval per request	Corpus model
Context layer	Sliding window K=10 + structured summary	Verbatim history; trim by max-turn count	Verbatim history; trim by token budget	Verbat by tok
Memory layer	Postgres tables; anonymous-cookie ID; 4 tools	Per-request ad-hoc	Per-request ad-hoc; sometimes paired with vector DB for memory	Per-req somet separa layer
Cost model	Sub-linear in corpus size; linear in actual fetch rate	Linear in corpus size per request	Bounded per request; linear in corpus size at index time	High u per-req
Audit surface	Transcript shows tool_use + tool_result; cost-log attributes per tool	Opaque (corpus is in system prompt; usage is implicit)	Partial (retrieved chunks visible; usage implicit)	Worst weight require
Red-team surface	Tool-boundary; dispatcher schema; corpus boundary; agent-loop; cookie identity	Prompt-injection; prompt-leakage; model behavior on full-corpus context	RAG-poisoning; embedding-collision; retrieval-poisoning; same model-behavior class	Model- trainin same class
Cohort scaling	Excellent; per-student data is namespaced; per-cohort is decorative	Poor; cost grows linearly with corpus and cohort	Good; one index per corpus; per-cohort isolation requires care	Poor; c cohort per N isolati
Operator-side complexity	Medium; FTS5 indices; Postgres ops; tool-dispatch discipline	Low; just the prompt	Medium; vector DB; retrieval pipeline	High; f quality deploy

The sidebar's claim, in one sentence: the four points are not "good versus bad"; they are "different commitments about where knowledge lives, what auditability you keep, and what attack surface you accept." A Belt-5 student authoring a substrate-versus-language analysis should be able to walk this sidebar against any tutor they encounter and place it on the map.

---

§8. Capstone scaffolding

The AI-301 capstone arc has four moves. Phase-1.6 is the scaffold for each.

§8.1 Build your own 3-layer tutor on a domain of your choice

Pick a domain that has a structured-knowledge corpus you can author or curate (medical first-aid; small-business tax; home-electrical safety; classical music theory; a programming language whose standard library has identifiable canonical-lookup paths). Author the four artifacts:

1. **A skinny base prompt (~150 lines or less)** declaring tutor identity, scope, fabrication-discipline, and pedagogical posture. No domain content in the prompt.
2. **Three to six KB tools** covering the canonical lookups your domain treats as primitive. Use OpenAI tool-shape; route through LiteLLM (`pcap-tools/backend/tutor/llm_client.py` is the reference). Hard-enumerate every parameter where domain semantics permit; bind every free-form parameter through a parameterized query layer; refuse the schema if a parameter is structurally invalid.
3. **A sliding window plus structured summary at Layer 2.** Use the `pcap-tools/backend/tutor/session_state.py` Pydantic schema as the starting shape; extend the fields where your domain demands extension; document the additions in your capstone artifact.
4. **A Postgres schema plus four read/write tools at Layer 3.** Mirror the four-table shape from `store/migrations/0001-initial.sql`. Document where you diverge and why.

The deliverable is a working tutor a peer can interact with, plus a substrate-versus-language analysis (per §8.3 below) that places your tutor on the §7 map.

§8.2 Red-team a peer's tutor end-to-end

Take the AI-201 register into the capstone register: pick a peer's tutor; run the §6 attack-surface map of the AI-201 companion against it; produce findings keyed to MITRE ATLAS technique IDs; produce a remediation plan with prioritization. The Belt-5 elevation over

Belt-4: the engagement memo includes a substrate-versus-language analysis of where each finding sits in the architecture's category map. A finding that is "tool-call poisoning at Tool 3" is a Belt-4 finding; a finding that is "tool-call poisoning at Tool 3, which corresponds to an unsoundness at the runtime's intrinsic-function tier" is a Belt-5 finding.

§8.3 Author a substrate-versus-language analysis

The portfolio centerpiece. Pick a deployed tutor (your own per §8.1; a peer's per §8.2; a third-party tutor with an inspectable architecture). Write a 15-to-25-page analysis with the following structure:

1. **Architectural surface walk.** What are the layers? What are the tools? What is the cost model? What is the audit surface?
2. **Substrate-versus-language category map.** For each architectural element, name its language-tier category (constitution, intrinsic, eval-apply, GC, persistent heap, capability token, linkage scope, lexical-versus-dynamic-scope move, small-step semantics).
3. **Operational-semantics formalism.** Write down the agent loop's small-step rules. Include a termination argument citing the budget cap; a soundness argument citing the dispatcher discipline; a performance argument citing the cost log.
4. **Threat model with capability-secure framing.** Name the capabilities in play (cohort token; student cookie; any other tokens); name the authorities each grants; name the attacks against the composition.
5. **Findings.** A table of architectural findings with severity, ATLAS mapping, and remediation. The findings register is the AI-201 register; the framing register is the AI-301 register.
6. **Forward-pointer to substrate.** Close with a paragraph that connects the analysis back to the silicon: the tutor's tool dispatcher runs on a CPU you could have built in CSA-101; the Postgres database runs on a kernel CSA-201 graduates have read line-by-line; the cookie's entropy comes from a CSPRNG whose construction is in CSA-201's privilege-level discipline. The academy's pedagogical arc closes here.

§8.4 Defensive remediation prioritization

The capstone's final artifact: a prioritized remediation plan for the deployed tutor you analyzed. The Belt-5 adds the cost-of-fix dimension that AI-201 typically does not: each remediation gets an estimated operator-side cost (engineering hours; ongoing operational burden; risk of regression). The prioritization is not "highest severity first"

but "highest severity-per-cost-unit first." Capstone-quality remediation plans are the artifacts the academy can carry into a paid client engagement; the prioritization discipline is what distinguishes an academy graduate from a fresh-out-of-school red-teamer.

---

§9. CSA-101 + CSA-201 forward-callbacks

The substrate-versus-language thesis is incomplete if it stays on the language side. Phase-1.6 runs on a CPU; the CPU runs an OS; the OS provides syscalls the LiteLLM client uses; the syscalls touch a network stack the academy's CSA-201 graduates have built abbreviated versions of in the driver-writing track. The pedagogical arc closes only when the AI-301 student traces Phase-1.6's full stack down to the silicon.

§9.1 The agent loop runs on instructions you have encoded

When `litellm.acompletion` returns, the Python interpreter resumes at the `await` point; the interpreter's bytecode dispatcher fetches the next opcode; the opcode is decoded; the operands are read from the Python frame; the operation runs. CSA-101 graduates have built a CPU that does fetch-decode-execute on RV32I-Lite. The interpreter's bytecode loop is the same shape at a higher tier. CSA-101's worked example (Ch 4 §4.9 "The Program Counter and the Fetch-Decode-Execute Loop") is the substrate-side analog of §6.3's agent-loop-as-eval-apply.

§9.2 The Postgres database runs on a kernel CSA-201 graduates know

asyncpg sends queries over a Unix socket or a TCP connection to the Postgres backend. The kernel's network stack handles the bytes; the file descriptor table holds the socket; the kernel's scheduler interleaves the asyncio event loop with the Postgres process. CSA-201 graduates have built privilege-level discipline (M/S/U mode), MMU-backed virtual memory (Sv32 or Sv39), and a syscall interface that user-mode processes use to ask the kernel for resources. The Postgres connection runs on top of all of that; the academy's CSA-201 chapter on syscalls is the substrate-side analog of §6.5's persistent-heap framing.

§9.3 The cookie's entropy comes from a CSPRNG

`secrets.token_urlsafe(16)` calls into Python's `secrets` module, which calls into the OS's CSPRNG (`/dev/urandom` on Linux; `getrandom(2)` on modern kernels). The CSPRNG is built from a seed plus a hash construction (HMAC-DRBG, ChaCha20, or similar). CSA-201's

mitigation-toggle work has students enabling and observing the kernel's CSPRNG; the academy's CSA-201 chapter on entropy and randomness is the substrate-side analog of §6.6's capability-as-unforgeable-token claim.

§9.4 Closing the arc

A Belt-5 student finishing AI-301 can close the academy's pedagogical arc by writing one paragraph that traces a single tutor request from Phase-1.6's `main.py:chat` endpoint down to a single fetched cycle on the CPU the student built in CSA-101. The paragraph is short, but writing it requires every belt below: NAND gates to ALU to register file to ISA to assembler to linker to VM translator to compiler to OS to syscall to network stack to TLS to HTTP to FastAPI to LiteLLM to provider API to tool-dispatch to Postgres query to row-level lock to disk write to byte on a platter or a flash cell. The student who writes the paragraph has integrated the academy's two halves into a single literacy. That integration is what AI-301 uses.

---

§10. Toolchain Diary additions (Belt-5)

Belt-5 students should add the following anchors to `toolchain-diary.md`. The prior belts cover the production-pentest tools (LiteLLM, FastAPI, asyncpg, MITRE ATLAS, PyRIT, HarmBench at the AI-201 register). Belt-5 adds the substrate-versus-language framework anchors:

- **Pierce, *Types and Programming Languages* (TAPL).** The standard reference for type systems and operational semantics; the small-step operational-semantics formalism §6.9 cites lives in Chapter 3. Use TAPL as the framework for §8.3's operational-semantics section of the capstone analysis. Entry-pointer: https://www.cis.upenn.edu/~bcpierce/tapl/.
- **Krishnamurthi, *Programming Languages: Application and Interpretation* (PLAI).** The companion to TAPL for the implementation side. Walks the construction of an interpreter from the ground up, covering metacircular interpreters, environments, continuations, garbage collection. Use PLAI as the framework for §6's category map. Entry-pointer: https://www.plai.org/.
- **Miller, "Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control" (the E thesis).** The capability-secure framework §6.6 cites; the cookie-as-capability framing has its formal grounding here. Entry-pointer: http://www.erights.org/talks/thesis/.

- **MITRE ATLAS at the red-team-frame depth** (https://atlas.mitre.org/). Belt-4 cites individual technique IDs; Belt-5 reads the matrix as a knowledge structure and produces multi-technique chains as findings. ATLAS Navigator's chain-construction surface is the canvas.
- **An LLM-eval harness the student stands up themselves.** HarmBench (https://harmbench.org/) is the recommended starting point; PyRIT (https://github.com/Azure/PyRIT) is the orchestration layer above. The Belt-5 elevation: the student adapts the harness to their capstone tutor's domain; produces a corpus of probes; runs the corpus; reports pass/fail-rates against the §6 attack-surface map.
- **Pony (https://www.ponylang.io/).** Optional but illustrative. A capability-secure type system implemented in a contemporary systems language. Reading the Pony docs is the fastest way to absorb the object-capability discipline from a working language rather than from a thesis.
