ChatGPT

Plan to Modernize the Flask Learning App and Integrate ACA v4

1. Modernizing the Flask Application Architecture

Modular Project Structure with Blueprints and Factory Pattern

The current app is a single `app.py` file containing all routes and logic, which makes growth and testing difficult. We should refactor it into a **package** with multiple modules and use Flask **Blueprints** to separate concerns. Blueprints allow grouping related routes, templates, and static files into self-contained components [1] [2]. For example, we can have one blueprint for the learning content (lesson pages), another for code execution API, and another for progress tracking or admin. This separation follows Flask best practices for larger apps and improves code reuse [3] [4].

We will also introduce an **application factory** (`create_app`) instead of a global app instance. The factory creates and configures the Flask app at runtime, registering blueprints and extensions [5] [6]. This enables creating multiple app instances (e.g. for testing with different configs) and avoids the pitfalls of a global app state [7] [8]. In summary, the new structure will enforce loose coupling and testability.

Proposed Project Structure:

```
flask_learning_app/
├── app/                    # Application package
│   ├── __init__.py         # Application factory, extension init, blueprint
reg.
│   ├── config.py           # Configuration classes (Dev, Prod, Test)
│   ├── models.py           # Database models (e.g. User, Progress)
│   ├── routes/             # (Optional) Common or simple routes
│   ├── content/            # Blueprint: Learning content pages
│   │   ├── __init__.py     # Create blueprint (e.g. bp = Blueprint('content',
__name__))
│   │   ├── routes.py       # Routes: index, path overview, lesson view, etc.
│   │   └── templates/      # Templates for content (index, learning_path.html,
etc)
│   ├── execute/            # Blueprint: Code execution and playground API
│   │   ├── __init__.py     # Create blueprint (bp = Blueprint('execute',
__name__))
│   │   └── routes.py       # Route: /execute (POST), /playground (GET)
│   ├── progress/           # Blueprint: Progress tracking API (mark complete,
etc)
│   │   ├── __init__.py     # Create blueprint (bp = Blueprint('progress',
```

```
__name__))
│   │   └── routes.py       # Route: /mark-complete (POST), maybe /progress
overview
│   ├── templates/          # Global templates (if not placed in blueprint
folders)
│   │   ├── layout.html     # Base layout template
│   │   ├── lesson_view.html
│   │   ├── learning_path.html
│   │   ├── unified_index.html
│   │   └── ... etc ...
│   └── static/             # Static files (CSS, JS, images)
├── tests/                  # Test cases
│   ├── test_content.py     # Test content page routes and context
│   ├── test_execute.py     # Test code execution functionality
│   ├── test_progress.py    # Test progress tracking logic
│   └── ... etc ...
├── migrations/             # DB migration scripts (if using Alembic for
SQLAlchemy)
├── requirements.txt
└── run.py                  # Entry point (creates app via create_app and runs
it)
```

In this structure, `app/__init__.py` will define `create_app()` which initializes Flask, loads config, sets up extensions (database, etc.), and registers all blueprints [9] [6]. Each blueprint has its own routes and templates, making the app modular. For example, the **content blueprint** handles the UI pages (`/`, `/path/<id>`, `/lesson/<id>`), the **execute blueprint** handles the code execution logic (`/execute`, `/playground`), and the **progress blueprint** handles completion tracking endpoints (`/mark-complete`). This clear separation means each component can be developed and tested independently. As Miguel Grinberg notes, grouping related functionality into a blueprint (e.g. errors, auth, main features) prevents intermixing code and makes it easier to reuse or modify subsystems [10] [3].

The **app factory** ensures different configurations can be applied easily. For instance, we'll create `config.py` with classes for Development, Production, and Testing settings. This might include enabling debug, setting database URI, secret keys, Celery broker settings, etc. In `create_app`, we do `app.config.from_object(ConfigClass)` based on an environment variable. We also initialize extensions like database and possibly Celery within this function (see below). This pattern allows using `current_app` in place of a global `app` inside routes and tasks, which Flask supports via application context [11] [12].

Benefits: The result is a **scalable structure**: as new features are added (e.g. user accounts, new lesson modules), we can create new blueprints or modules without touching the core. The app is more maintainable and **test-friendly** – each blueprint and even the Flask app itself can be instantiated in isolation for unit tests [7] [8].

Database Integration for Progress Tracking

Currently progress is stored in a JSON file via the `progress.py` module. This should be migrated to a **database** for reliability and scalability. We can use **SQLAlchemy** (the popular ORM) to define a model like `Progress` (fields: user_id, path_id, lesson_id, completion_time, etc.) and possibly a `User` model if multi-user support is desired. Using a database ensures atomic updates and concurrency safety (unlike a flat file) and opens the door to multi-user progress tracking in the future. For example, instead of writing to `progress_data.json`, `mark_complete()` would create a Progress record or update a completion status in the DB. We can easily query progress percentages, etc., with SQLAlchemy queries instead of manually counting in JSON [13].

If the app is single-user, a simple SQLite database is sufficient (Flask can use SQLite by default for ease). Otherwise, a more robust DB (PostgreSQL etc.) can be configured. We'll use Flask-Migrate (Alembic) to handle schema changes.

Marshmallow for Validation/Serialization: We will introduce **Marshmallow** to define schemas for our data – for example, a schema for the execute-code request and response, and for progress data. Marshmallow allows declaring expected fields and types and will automatically validate input and serialize output to JSON [14] [15]. This reduces manual validation code and ensures consistent JSON responses. For instance, we can define a `CodeExecutionSchema` with a field `code` (string, required, max length 10000) to validate the incoming JSON for `/execute`, instead of manual length checks scattered around. Marshmallow also integrates well with Flask (via flask-marshmallow extension) for output schemas. Using schemas will enforce that our API outputs (like the JSON from `/execute` or `/mark-complete`) have the right structure and types. This is a **2026 best practice** for Flask APIs – treating input/outputs with declarative schemas to catch errors early [14] [16].

Template Rendering: The app already uses Jinja2 templates (`render_template`). We'll continue to leverage Jinja2 for HTML output, but we can organize templates better (possibly using a base layout and blocks for each page to avoid repetition). We might store blueprint-specific templates in sub-folders (as shown in structure) or keep them in a unified templates folder with naming conventions. Jinja2 in 2026 remains a standard for Flask; we ensure to use its features (macros, filters) to keep HTML maintainable.

Modern Python Tools and Features

We will ensure the code is updated to modern Python conventions: - **Python 3.10+** features like type hinting everywhere, f-strings (already used), and dataclasses for simple data structures if needed. Type hints will improve readability and allow use of static type checkers (mypy) to catch bugs early. - Use **Flask 2.x/3.x** (whichever is latest stable by 2026) which supports async routes if needed. If some operations can benefit from async (like making an external API call for analysis), we could turn certain view functions into `async def` and use asyncio. However, heavy compute like user code execution is CPU-bound, so async won't help that (we'll use background workers for those). - **Background Task Queue:** To keep the app responsive, long-running tasks (like code execution or multi-agent analysis) should run outside the request context. We propose integrating a **task queue** like **Celery** (with Redis or RabbitMQ as a broker) or a simpler alternative like **RQ** for background jobs. Flask's documentation recommends using a task queue so requests return immediately and a worker process handles the heavy lifting [17] [18]. For example, when a user submits code, the `/execute` route could enqueue a job to run the code and quickly return a task ID, while a separate worker executes the code and stores the result. The frontend can poll or wait via WebSocket/SSE

for the result. This prevents the Flask web worker from blocking on potentially slow or hung code. Celery is a robust choice ("a powerful task queue... for simple background tasks as well as complex multi-stage programs" [19]) and by 2026 has strong integration with Flask. We could start with the simpler approach (run code in a thread with a timeout) and move to full Celery if scaling up. - **Streaming and Real-time Updates:** To enhance user experience, we can implement **streaming responses** for the code execution. Instead of waiting until the code finishes to send all output, the server can stream output as it's produced. Flask can yield responses in chunks or use Server-Sent Events (SSE) for real-time logs. For example, as the code prints lines, we flush them to the client via an event stream in the `/execute` response (content type `text/event-stream`). This requires running the code in a way we can capture incremental output (which we already do with `StringIO` buffer). With SSE, we could send each newline from the buffer as an event. Alternatively, using **Flask-SocketIO** (which uses WebSockets) is an option if bi-directional communication is needed (like interactive execution), but SSE is simpler for one-way streaming of logs. - **Logging and Error Handling:** We'll set up structured logging (perhaps using Python's `logging` library with JSON or YAML logs for easier debugging). Also implement error blueprint for custom error pages or JSON error messages (e.g., a 500 error handler that returns JSON with error ID for troubleshooting, which can map to our Task Integrity logs – see ACA integration below).

Testing and Maintainability

We will prioritize **unit testing and integration testing** for all components: - Use **pytest** as the testing framework, which works well with Flask. With the app factory, we can create an app instance in test mode (e.g., `app = create_app(TestConfig)`) for each test, ensuring isolation [7] . Each test can use the Flask test client to simulate requests to routes and verify responses. - **Unit tests**: Each module (e.g., progress model, code execution function, ACA utility functions) will have tests. For example, test that `mark_complete` correctly updates the DB and that repeated calls don't duplicate entries, etc. Test the code execution sandbox with benign code vs malicious code to ensure safety checks work. - **Integration tests**: Simulate a full lesson flow – e.g., load a lesson page (should return 200 and contain lesson content), call the execute endpoint with some code (get a task ID or output), call mark-complete and verify progress updated. - Ensuring **coverage** for critical paths (target ~90%+ for core logic like progress and execution). Particularly, each ACA-related hook we add should have tests (e.g., a test for the safety guard blocking a dangerous input, a test for the refinement loop giving a second attempt on failure, etc.). - The application factory and blueprint structure inherently make testing easier: we avoid global state bleeding between tests (each test can create a fresh app and teardown) [20] [8] .

We will include a `tests/` directory (as shown) and perhaps use CI tools to run tests on each commit. By writing tests concurrently with each new feature, we ensure **no loss of functionality** during the refactor. For example, before refactoring progress storage, write tests against current JSON behavior, then refactor to DB and ensure tests still pass with the new implementation (the behavior from the user's perspective remains the same).

2. Integrating ACA v4 Cognitive Architecture into the Flask App

The Alvin Cognitive Architecture (ACA v4) defines a **modular, deterministic pipeline** of cognitive modules (M0 through M23). We will map these modules onto our Flask application's processing flow, so that each request (especially code execution or any intelligent response generation) goes through similar stages of guardrails, control, processing, and output alignment. The idea is to enforce the same *separation of concerns* – each ACA module's responsibility will be handled by a specific function or hook in our app – and to

preserve the **fixed sequence** of operations for predictability. Below is a mapping of ACA modules to implementation points in the Flask app, followed by how to integrate them efficiently:

Mapping ACA Modules (M0–M23) to Flask Implementation

Layer 0 – Guard & Setup:

- **M0: Safety & Memory Guard** – *Input validation and sandboxing.* This will be implemented at the very start of request handling. We will use Flask's `@app.before_request` (or blueprint-specific `before_request`) to run a safety check on user inputs. For example, in the `/execute` route, we already validate code length and restrict builtins [21] [22]. We'll expand this with a static analysis of the submitted code string to **ban unsafe operations** (e.g. file system access, infinite loops) before execution. We might scan for keywords like `import`, `open`, or dangerous builtins and refuse execution with an error if found (beyond what safe_globals covers). The "Memory Guard" aspect means the system should not retain any user prompt or chain-of-thought beyond the request – we ensure no state carries over between requests that could leak info or bias future ones. This aligns with using ephemeral sandbox (each code run is fresh, and we do not store the code or output except maybe in transient logs). Also, if we integrate any AI reasoning, ensure it doesn't store conversation history that persists (unless explicitly part of user progress). M0 will override processing if any safety trigger trips, consistent with ACA's "Safety overrides all reasoning" principle. Technically, this could be a function `safety_check(request)` called before processing, or a decorator on routes needing it. If it fails, we abort the request with a clear error (e.g., "Input not allowed").

- **M1: Identity Gate** – *Authentication/identity verification.* In a multi-user scenario, this module would confirm the user's identity and role at the start of a request. Implementation: use Flask-Login or similar to ensure a user is logged in (e.g., `@login_required`) for routes that modify progress or execute code. If not logged in or not the correct user, abort. If the app remains single-user, M1 can be minimal (e.g., verify a session token or skip). In the future, if roles (student vs admin) are introduced, M1 would enforce permissions (e.g., only admins can access certain diagnostic or admin routes). So, essentially, M1 would be integrated as either a **before_request hook** that checks `current_user` or an authentication decorator on relevant routes, ensuring the request has proper identity context (like ACA's identity check before proceeding).

- **M2: Preference Loader** – *Load user/system preferences.* We will load any user-specific preferences or global settings at request start. For example, user profile might contain preferred language, a "strict mode" toggle, or interface preferences. The system might have a config for how verbose the AI explanations should be. We can implement this by reading from the database or config file and storing values in Flask's application context or `g`. For instance, if a user has a preference for "guided mode" learning, we load that so later modules (Mode System, etc.) know about it. Technically, this can happen in `before_request` after identity verification (M1) – once we know who the user is, load their preferences into `g.preferences`. If no user context, we load default preferences (e.g., default mode = Teacher mode). This corresponds to ACA's idea of injecting preferences early so that downstream modules (M4, M6, etc.) can use them [23].

Layer 1 – Control Configuration:

- **M3: Meta-Controller** – *High-level orchestration of reasoning path.* In our app, the Meta-Controller will determine how to handle the request globally. This could be implemented within the route logic or as a dedicated controller class that our routes call. For example, when a code execution request comes in, the Meta-Controller decides: are we just running code and returning output, or are we also evaluating it and giving feedback? It could examine factors like the lesson type or user mode. It sets up a **control_config** that informs subsequent modules [24] [25]. For instance, Meta-Controller may say "use TEACHER_MODE, deep reasoning off, no multi-agent analysis for simple tasks" vs "for complex tasks, allow multi-agent analysis and refinement". In practice, this might be a Python function `configure_request()` that returns a dict or object with flags for mode, path depth, which evaluators to run, etc. The Meta-Controller will also handle **fallback logic**: if something fails down the line, it can decide to invoke Module 20 (Fallback Manager). For example, if code execution raises an unexpected error, Meta-Controller might switch the pipeline to a simpler mode or return a safe error message rather than nothing. ACA defines Meta-Controller as governing global reasoning and how deep/complex the process should be, so here it's the strategic brain of our request handling. We'll likely implement it at the *controller level* (i.e., in the view function or a function called by it), because it influences which services to call next. It's not a Flask hook per se, but embedded in our logic flow: right after preferences are loaded, call `meta_config = meta_controller.decide(request, prefs)` to get global settings for this request.

- **M4: Mode System (FSM)** – *Set the cognitive mode/state of the app.* ACA's Mode System is a finite state machine of modes like ALPHA (deep reasoning), BETA (fast/brief), ARCHITECT, TEACHER, SUPPORT modes [26] [27]. We will model these modes in our app so that responses can be tailored. For a learning app, **TEACHER_MODE** is very relevant – it would ensure outputs are pedagogical, explanatory, with clarity-first approach [27]. We might also define a **SUPPORT_MODE** for extra gentle encouragement, or a **DEBUG_MODE** if needed for diagnostics. Implementation: define an enum or constants for modes, and an object that holds current mode state (could be part of the meta_config or a global `g.mode`). The mode enforces certain behaviors: e.g., in TEACHER_MODE, after code execution, the system might always provide a hint or an explanation in addition to the raw output; in BETA_MODE (fast), it might just return the output with minimal commentary. The Mode System being an FSM means we should restrict transitions – e.g., the app might normally default to TEACHER_MODE, but if a user switches to a "Challenge Mode", that could correspond to BETA_MODE (just tests pass/fail without hints). We ensure illegal transitions don't happen (you wouldn't spontaneously change mode in the middle of processing unless a fallback triggers it). For now, we can start with one mode (Teacher) as default, but structure the code to allow adding modes easily. We can incorporate mode checks in controllers and in template logic (for example, if `mode == TEACHER_MODE`, the template may reveal more hint info). Downstream modules like Lands Mixer (M6) or Emotional Regulator (M7) will also use the mode to adjust their behavior [28] [29]. This mode state will operate at the **service/middleware level** in that once Meta-Controller chooses a mode, many subsequent functions will read this mode to decide how to act.

- **M5: Path Selector** – *Choose shallow vs deep reasoning path.* ACA allows different processing depths: e.g., **Fast path** (single pass), **Balanced**, **Deep path** (multiple refinement cycles). In our app context, this translates to deciding how much analysis to do for a given request. For a simple task or quick check, we take a fast path (just execute code and return output). For a complex problem or when high accuracy is needed, maybe we take a deep path (run code, then run additional checks, possibly

multiple attempts to refine an answer). The Path Selector can be implemented as part of the Meta-Controller's decision. For instance, meta_config might include `path_type = FAST_PATH` or `DEEP_PATH`. If `DEEP_PATH`, we might allow Module 16 (Refinement Loop) to iterate or allow more agents to weigh in. If `FAST_PATH`, we do minimal work (maybe skip style checks or hints generation to save time). We can base this on lesson difficulty or user request (user might explicitly request a thorough analysis). Implementation detail: a simple if/else in the processing pipeline that checks `path_type` and sets iteration counts or which modules to engage (e.g., skip refinement if fast). This ensures we **don't slow the app unnecessarily** for every request – only do heavy multi-cycle processing when needed or requested, aligning with ACA's idea of dynamic depth.

- **M6: Lands Mixer** – *Compute weights for cognitive aspects (clarity, depth, empathy, creativity)*. In ACA, the Lands Mixer outputs a profile (ground, growth, heart, play weights) based on mode, path, preferences, and safety flags [30] [31]. In our implementation, this would influence the style and tone of responses. For example, **ground (clarity)** might be prioritized for factual or troubleshooting tasks (meaning our feedback should be very direct and clear), **heart (empathy)** might be higher in support mode (meaning our messages should be encouraging), **play (creativity)** might be low unless we're in a brainstorming context (maybe not common in this app), and **growth (depth)** would be high if we want very in-depth explanations (like for an advanced topic or if the user is curious). We can implement M6 as a function that takes inputs (mode, task type, safety flags, etc.) and returns a dictionary of weights [31] [32]. Downstream, when generating any explanatory text or hints, we could use these weights to modulate the output. For example, if heart_weight is high, the feedback generator might include more praise or positive reinforcement; if ground_weight is high, it will stick strictly to the point. If we incorporate an AI component to generate hints/feedback, these weights could be parameters to that system (e.g., control temperature or style of an LLM, if used). Initially, since the app mostly executes code and returns output, Lands Mixer might not have an obvious effect, but as we integrate more automated feedback, it becomes useful. We can at least set up the structure: e.g., produce a `mixer_profile` each request (store in `g.mixer`) with default weights, and allow adjustments. This module would be implemented at the **service level** – it's part of the internal logic configuring the response style, not a Flask hook, but a pure function using the mode and prefs (makes it easy to unit test by feeding various scenarios).

- **M7: Emotional Regulation Layer** – *Maintain appropriate emotional tone.* This module ensures the system's outputs maintain a certain emotional balance (not too frustrated, not overly verbose due to emotional bias, etc.). In a teaching app, this translates to **keeping the feedback tone positive and supportive**. We will implement this as a filter on any user-facing messages. For example, if the user's code fails and we generate an error message or hint, we ensure the wording is constructive ("Have a look at X, you might need to…") rather than discouraging. If the user has failed the same exercise many times, the system might detect frustration and increase empathy (perhaps switching mode to SUPPORT_MODE automatically, which in ACA would increase heart_weight [32]). Implementation: a small function `adjust_tone(message, mixer_profile)` that tweaks the message according to the desired emotional tone. If heart (empathy) is high, maybe add a encouraging remark; if it's a neutral scenario, keep it factual. This could also ensure we don't unintentionally produce harsh text (especially if an AI is summarizing an error – we'd ensure it doesn't say "obviously you did it wrong", etc.). This check could be done right before final output (as part of Module 22 alignment checks) or during generation of the feedback. It's largely a **middleware service** – not directly a Flask extension, but part of output crafting that can be independently tested (feed in a sample negative message, expect a softened output).

- **M8: Bottleneck Monitor** – *Watch resource and performance bottlenecks.* In our app, this is critical because user-submitted code could hang or use excessive resources. We will incorporate a **timeout and resource limit monitor** when running code. For example, before executing code, start a timer thread that will terminate the execution if it exceeds, say, 2 seconds (or use `multiprocessing` to run code in a subprocess and kill if time or memory exceeds certain threshold). This ensures one request doesn't hog the CPU indefinitely (which would be a bottleneck and denial-of-service risk). We can also monitor memory by observing process memory before/after or using sandbox limits if running in a container. The Bottleneck Monitor might also collect metrics (how long each execution took, how many loops were run) and if a pattern of slowness is detected, inform Meta-Controller to maybe switch to a simpler path next time. But at minimum, it will **enforce timeouts** on code execution and possibly limit the size of outputs (to avoid huge payloads). Implementation: if using Celery or a separate thread for code, we can enforce a join with timeout, or use signals/ alarms in the same process. On timeout, we gracefully stop execution and mark the result as failed due to timeout (trigger fallback). This module likely operates at the **service level** (inside the code execution function). It's like a guard running in parallel with processing, ensuring the pipeline moves along. We will also have it log occurrences of slowness (for analysis).

- **M9: Sparse Attention (Dynamic Sparse Attention)** – *Limit focus to relevant data to conserve resources.* For the app, this means we should not over-process unnecessary information. In practice, we already do this by limiting code length and not storing history. We can further apply it by, for example, if a user's code output is extremely large, we might truncate it to the first N lines in the response with a note (to avoid overwhelming the UI). Or if there are many tests or checks, maybe run only the most relevant ones first to avoid combinatorial explosion. Essentially, **simplify the context** where possible. Another angle: if we integrate an LLM for explanation, feed it only the relevant pieces of code or error (like last error message) rather than the entire code and history, to keep context small (which is a direct analogy to sparse attention for efficiency). Implementation: define rules for ignoring or summarizing less important data. For example, if code has multiple functions but the error is in one function, focus only on that in feedback. This is more of a design guideline than a specific piece of code; we will incorporate it when writing analysis functions (like not analyzing every variable if not needed). It doesn't correspond to a specific Flask hook, but rather we'll ensure each analysis step is **purposefully constrained** (for unit testing, we can ensure performance by feeding large input and verifying our functions handle it within limits).

Layer 2 – Cognitive Processing Engine:

- **M10: Process Engine** – *Core execution of the request's primary task.* In our context, this is the actual **code execution engine** (or any primary content generation). We will refactor the code execution currently in `execute_code` route into a separate service function, e.g., `run_code(code_str) -> output`. This function embodies M10 by executing the code in the sandbox environment [33]. We already redirect stdout and restrict builtins as a safety measure. We'll keep that approach (potentially using an improved sandbox library in the future), and integrate the Bottleneck Monitor (M8) around it. The Process Engine function should be deterministic for a given input (no external side effects beyond what's allowed in safe_globals). By isolating this logic, we can **unit test** it easily by feeding code strings and checking outputs. It can also be reused by a background worker if we use Celery (the Celery task would call this function). If in the future the app supports other processing (like evaluating free-text answers or running different languages), that would fall under extending this Process Engine with appropriate handlers. For now, it's Python exec

specifically. This module sits at the **service layer** – triggered by the controller after initial checks, performing the heavy computation.

- **M11: Decision Tree Builder** – *Plan the solution approach or evaluation steps.* When a request comes in, especially if complex, we might need to decide on a series of actions. For example, if the user runs code, we might decide to: 1) run the code, 2) run some unit tests on the code's output (if the lesson has expected outputs or test cases), 3) analyze any exceptions to provide hints. This sequence can be seen as a decision tree or plan. We implement M11 by encoding such logic flows. Perhaps we create a function `plan_execution(context)` that returns a list of steps to perform. E.g., for a given lesson, if lesson defines some `expected_output` or `test_function`, then the plan includes a testing step. If user's code fails, plan might include a step to compare error to common mistakes database. Essentially, rather than hardcoding everything in one monolithic function, we structure the logic as discrete steps that can be dynamically chosen. For now, the "decision tree" might be simple (always execute code, then if success or failure handle accordingly), but as features grow (like giving automated hints or multiple solution attempts), having this planning layer ensures we can insert steps cleanly. This is an internal module (not a Flask hook) that orchestrates calls to the Process Engine (M10) and the evaluation agents (M14, etc.), depending on conditions. For testability, the planning function can be fed a scenario and we verify it returns the right steps. In code, this could just be straightforward if/else logic, but conceptually it aligns with ACA's idea of separating the *plan* from the actual execution.

- **M12: AIM Phase 1** – *Initial analysis and modeling.* In ACA, AIM might stand for some Analysis/Intuition/Model phase. For us, this could be a preliminary analysis of the user's request or code before actually executing it. For example, we could do a quick static code analysis to detect obvious errors (like syntax errors, or use of uninitialized variables) and maybe predict the outcome (though we typically just execute to see outcome). Another use: if the request was a question (in an AI Q&A scenario), Phase 1 would be gathering relevant info. In our code tutor scenario, Phase 1 might not be very extensive, but we can incorporate a **dry run or lint phase**. For instance, use `ast.parse` to check for syntax errors (which we get anyway from exec exceptions) or run a lint tool (like flake8 or pylint) for immediate style feedback. If we find errors here, we could short-circuit execution (since running will error anyway) and provide the syntax error message in a controlled way. This phase could also set up any necessary context for deeper reasoning: e.g., if we plan to ask an AI to explain the code, phase 1 might involve summarizing the code or extracting its structure. Implementation: This would be a function `pre_process(code)` that returns info (like "syntax_ok": True/False, "issues": [...] etc.). It would be called before the actual exec (M10). If it reports major issues, we incorporate that into the feedback. This makes our pipeline more robust (we don't rely solely on catching exceptions after execution; we anticipate some). In testing, we can feed malformed code and ensure `pre_process` flags it properly.

- **M13: Eve Supra** – *High-level coordination before entering multi-agent reasoning.* In ACA, Eve Supra likely prepares the inputs (supra_packet) for the Eve Core multi-agent engine [34]. In our case, if we adopt a multi-agent evaluation (coming up in M14), M13 would gather all necessary context to give to those agents. For example, after executing code, to evaluate it we need: the code itself, its output, any error or exception, perhaps the expected output for the lesson, and the user's progress (to see if they've tried many times). Eve Supra would bundle this context and maybe decide which agents should be activated. Implementation: we can represent this as creating an **evaluation context object** that holds everything needed for evaluation (code, output, error, lesson info, user info). The

Meta-Controller and Mode might influence what goes into it (e.g., if in a strict mode, include style guidelines as part of context so the style agent definitely runs). Then we pass this context to each evaluation agent (M14). Essentially, M13 is a preparatory step ensuring the multi-agent (or multi-check) process has what it needs. It can be a simple function that collates data into a dict. We will design it such that it's easy to add more data in future if needed.

- **M14: Eve Core (Multi-Agent Reasoning Engine)** – *Concurrent multi-dimensional evaluation.* This is a key part of ACA, where multiple "agents" (roles) analyze the content in parallel and produce "seeds" of reasoning [34] [35]. We'll translate this to our app by introducing **multiple evaluation functions/ agents** that assess the code submission from different angles:

- A **Correctness Agent**: checks if the code achieved the goal. For example, if the lesson has an expected output or known solution, this agent compares the actual output to expected or runs predefined unit tests. It produces a result like pass/fail and perhaps a message ("Output is correct!" or "The result is not as expected for input X.").

- A **Safety/Injection Agent**: verifies the code didn't do anything malicious or access forbidden resources. Since we restrict builtins, this might just confirm no security flags tripped during execution. It could examine the output or any logs for signs of misbehavior. (In ACA terms, a Safety agent ensures reasoning has no unsafe content [36]; here ensure code output is safe to display).
- A **Style/Quality Agent**: reviews the code for style, best practices, or efficiency (this could parse the code and run a style checker). If the code is functionally correct but written in a convoluted way, this agent might suggest improvements (e.g., "You used a global variable; consider using a function parameter instead").
- An **Empathy Agent**: in an AI conversation, would adjust tone [37]; for us, it might gauge the user's frustration level (maybe based on how many attempts in progress data) and suggest how gentle the feedback should be – though this overlaps with M7 emotional regulation.
- A **Clarity Agent**: if we were generating an explanation, this agent ensures it's clear (in our context, maybe not separate from correctness feedback).
- A **Creativity Agent**: likely not needed here unless we have an AI that could propose a creative solution or hint.
- A **Structure Agent**: ensures the solution or output follows the required format (if the lesson expects certain output structure, this agent checks formatting).

Each of these agents will run (potentially in parallel via threads or sequentially if simpler) on the **evaluation context** prepared by M13. They produce their own piece of feedback or score. For example, Correctness agent might output `{"correct": False, "error": "Test 3 failed: expected 10 got 7"}`; Style agent might output a PEP8 score or a note; Safety agent might output `{"safe": True}` or attach a warning if not. This design keeps each concern separate (mirroring ACA's separation of concerns).

We will implement each agent as a Python function or method in a class (for organization). They operate at the **service layer** (part of the request processing, but not Flask-specific). We can unit test each agent by feeding synthetic contexts (e.g., a context with a known wrong output to see CorrectnessAgent produces the expected message). They are conceptually concurrent "pseudo-agents" like ACA describes [34], although in code we might call them sequentially unless performance demands threading. Given our context,

sequential is fine (the overhead is small and easier to debug), but we could use Python's concurrency if needed.

- **M15: Seed Scoring Framework** – *Evaluate and score the outputs from agents.* After M14 gives us multiple "feedback pieces" or results, we need to prioritize and consolidate them. For example, if the code is completely wrong, the correctness feedback ("Your code didn't produce expected output") is more important to show the user than a minor style nitpick. The Seed Scoring module in ACA would score reasoning fragments on clarity, safety, etc., and pick top ones. In our case, we can define a simple scoring logic or priority order: Safety issues > Correctness issues > Others. We ensure any **safety-related feedback** (e.g., "Your code tried to open a file which is not allowed") would override showing other feedback, aligning with "Safety priority". Similarly, if the code failed, that feedback is crucial; style suggestions might be secondary.

We can implement this as a function that takes all agent outputs and computes a sorted list or a merged feedback message. For instance, it might return a combined feedback string that first addresses critical errors and then adds, say, "Also, note X style improvement." If multiple hints or solutions were generated, scoring would pick the best one to show. Because our scenario is not generating lots of free-form answers but rather evaluating specific aspects, this may be straightforward rule-based ranking rather than machine-learning scoring. But structuring it as a distinct step (M15) means if in the future the feedback generation becomes more complex (like multiple solution attempts), we have a place to decide which attempt is best to present.

Technically, this is inside the request processing pipeline, likely just after collecting agent results. It can be unit tested by mocking agent outputs and seeing if the scoring picks the intended outcome (e.g., if one agent says "safe" and another says "unsafe", ensure unsafe triggers a safety override to user message).

- **M16: Refinement Loop** – *Iteratively improve the outcome if needed.* In ACA, the refinement loop runs while quality < threshold and resources remain. We will incorporate a lighter version appropriate for a web app: for certain scenarios, if the first attempt doesn't meet criteria, we can try again or adjust and retry **within the same request** (if quick) or suggest the user to try again. For example, if we had an AI generating a solution, we could loop asking it to refine the solution until tests pass (bounded by a max number of attempts). In our current setup, the user is the one refining their code with each request, so we might not automatically modify their code. However, there are still uses:
- If using an LLM to provide a hint, we might find the hint unclear (via agent feedback) and regenerate once more (behind the scenes) before showing it.
- If we attempt some automated fix (like formatting the code to see if that solves an issue), we could loop: run code -> fails -> maybe the system realizes the error could be fixed by adding a missing initialization -> try that fix in a sandbox -> if it works, suggest that to user. This is advanced, but feasible.
- More practically, if the code execution **times out** (M8 triggers), the fallback (M20) might shift to a different strategy (like run with smaller input or simplified logic) if possible, which is a kind of refinement or alternate attempt.

Implementation: inside the code execution function or evaluation function, allow a loop with a counter and a quality metric. For example, quality metric could be "tests passed" (100% means success). If not all tests passed, and refinement budget > 0, we could try something: e.g., if there's a known common mistake, auto-modify the code or call a different agent to propose a fix, then re-run. This can get complex, so we might not implement a full auto-refinement of user code (as it might defeat the learning purpose to fix it

automatically), but we keep the architecture in mind. We might implement a simpler loop: if an AI-generated explanation is not coherent (M19 finds an issue), we regenerate it (one retry). Or if a particular agent fails to produce output (say style checker crashed), we could retry it once. These are forms of refinement to improve output integrity. We'll set a small limit (maybe 1 or 2 iterations) to avoid long waits, and log when refinements happen.

From a testing perspective, we can simulate a scenario needing refinement and ensure our loop does iterate and improve the outcome. The important part is that this loop is **controlled** by conditions and a budget to not loop indefinitely, per ACA guidelines. We also must ensure that if a refinement happens, it doesn't violate the "no memory of chain-of-thought stored" rule – i.e., each iteration's reasoning isn't kept beyond its use, and we don't accumulate hidden state except summary metrics.

- **M17: Conflict Resolution Unit** – *Resolve contradictions among agents' outputs.* If our multiple evaluation agents give conflicting feedback, we need to reconcile that before telling the user. For example, suppose one agent (based on a buggy heuristic) says "Great job, output looks correct" but another agent running actual tests says "Output is incorrect." We must resolve that – obviously, correctness should trump in this case, and we should not congratulate prematurely. Another example: a safety agent might flag something as unsafe whereas another agent overlooked it; we should err on the side of caution and treat it as unsafe until resolved. Conflict resolution can be implemented as part of the scoring (M15) or as a distinct step right after agent outputs, before finalizing feedback. It might set flags or remove inconsistent messages. In many cases, our scoring logic (which prioritizes certain outcomes) inherently resolves conflicts by weight (e.g., safety issues override others, as mentioned). We can formalize it: if any **critical agent** (safety, correctness) yields a negative result, we drop any positive messaging from other agents. If conflicts are less severe (e.g., one agent says "the code is efficient" and another says "the code could be more efficient"), we might merge them or choose one based on mode (maybe in teacher mode we present a balanced view).

Implementation: a simple set of rules applied to agent outputs, possibly integrated with M15's logic. We ensure by the time we generate final feedback, it's consistent (doesn't congratulate and criticize for the same thing). For traceability, we could log the conflict and how it was resolved as part of M18 (integrity log). Testing: create dummy conflicting outputs and run through resolution to see that the conflict is handled (expected outcome: the user sees a coherent message).

- **M18: Task Integrity Layer** – *Ensure traceability and integrity of the whole process.* This is about making sure the outcome is backed by the processing steps and everything is logged for debugging. We will implement M18 in a few ways:
- **Structured Logging**: Every critical step (input validation, code execution, agent evaluations, final output) will log an entry (could be to a file or console). This log can carry a request or session ID so we can reconstruct what happened for a given execution. For example, when a user runs code, we log: "Request X: Received code of length Y", "Ran code, output = … or error = …", "Correctness check result = …", "Final feedback = …". These logs ensure if something goes wrong or the user gets weird feedback, developers can trace the chain of events. We must be careful not to log sensitive info (like user code could be sensitive?), but since this is a learning app, it's probably fine, just treat logs securely.
- **Consistent IDs**: We may generate a unique ID for each execution (maybe use Python `uuid4`). This ID can be returned to the user if needed (perhaps not necessary here, but could be for polling results in async scenario). It will tag all log entries. In a multi-user scenario, also log user ID.

- **Data Integrity**: Ensure that the final output shown to the user is exactly derived from the process. For example, if the code produced output "Hello\n", the user should see "Hello\n" – we should not accidentally drop or alter it (except formatting for HTML). If we modify output (like truncating long output), we should indicate that and have the full output stored or accessible for integrity. If an AI generates an explanation, ensure it aligns with the actual code's behavior (we might verify some statements in the explanation against the code's result to ensure no hallucinations).
- **No silent failures**: If any module failed internally (maybe style agent threw exception), we catch it, log it (with the ID) and either exclude that part or fallback, rather than letting the whole request crash. This aligns with integrity – the system either fully succeeds or gracefully handles partial failures.

Technically, we might integrate this with Flask's `teardown_request` to log the outcome of every request (and maybe the time taken, to monitor performance integrity). Also, if using a database, we could store an entry for each code execution (user, code hash, success/fail, timestamp, etc.), which acts as a persistent record of what happened (helpful for auditing or even giving user a history feature). From a security perspective, we must avoid logging sensitive data inadvertently (like if user code had passwords – unlikely in learning context, but just mindful).

- **M19: Error & Coherence Checker** – *Final check for errors or incoherent output.* Before sending the response, this module ensures the response is free of internal errors and is logically coherent. Implementation in Flask: for HTML responses, ensure that all placeholders are filled (Jinja would error out if not, which we'd catch in testing). For JSON responses, ensure the JSON structure has all required fields (`success`, `output`, etc.) and no stack trace is being leaked unless intended. In our current code, when an exception happens during execution, we return the traceback [38]. Exposing raw traceback might not be ideal for a polished app (it could confuse users or reveal internal info). As an improvement, M19 could intercept that and produce a cleaner error message. For example, if the traceback indicates a `ZeroDivisionError`, we can return a user-friendly message like "Your code raised ZeroDivisionError (division by zero)". That is more coherent for a learner. We can still log the full traceback internally (M18 logs). So M19 will sanitize errors. Additionally, if we have combined multiple feedback messages, check that they don't contradict or repeat unnecessarily (this ties into conflict resolution from M17 as well). Essentially, **polish the final output**: correct any minor grammar issues in messages, ensure formatting is nice (e.g., for HTML, maybe converting newline to `<br>` in code output, etc.), and ensure consistency in tone. This check can be done in an `after_request` handler or just before returning within the route. Since it might need to modify the response, doing it in Python just before `return` is fine. (Flask `after_request` could also do something like always wrap JSON in a consistent envelope, but we already do that explicitly.)

- **M20: Fallback Manager** – *Gracefully handle failures by invoking fallback strategies.* If any stage fails or yields no result (and cannot be recovered in refinement), we activate a fallback. In our app, possible fallback scenarios:

- Code execution crashes the process or times out (e.g., user wrote an infinite loop that even our attempts couldn't stop in time). The fallback might be to kill that execution and then return a message: "Execution timed out. Try simplifying or check for infinite loops." This is better than hanging.
- An agent or analysis fails unexpectedly (say our style checker library throws an error) – fallback: skip that analysis and just return what we have.

- If using an external service (like an AI API for hints) and it doesn't respond, fallback: return the code output without hint rather than failing the whole request.
- If the *entire pipeline* fails (maybe an uncaught exception), we should have a generic error handler (HTTP 500) that returns a friendly message like "Sorry, something went wrong. Please try again." (and log details internally). This prevents the user from seeing a stack trace or broken page.

We will implement specific fallback logic at critical points. For example, around the code exec: use a try/except to catch any exception and instead of crashing, return a JSON with success=False and a simplified error [38]. This is already partly done: currently it returns the Python traceback as output on exception [38]. We will likely change that to a more user-friendly message in final version, but either way, it's a fallback from normal successful execution path. We also consider performance fallback – if something is taking too long (M8 triggers), maybe switch to a different approach (e.g., if a complex analysis was planned, abort it and just give basic output). The Meta-Controller can help decide fallback actions: ACA mentions fallback manager is conditional, meaning it might not always trigger unless certain faults occur. We'll design fallback triggers for specific failure types.

Implementation: could be integrated via try/except blocks in code, or Flask error handlers for broad cases. We'll define a function like `handle_fallback(error_type, context)` that can produce an appropriate response or recovery. For example, if `error_type == 'timeout'`, do above timeout message; if `'agent_failure'`, log and omit that agent's output; if `'system_error'`, use Flask's errorhandler to return a generic message. This keeps fallback handling organized.

Layer 3 – Output Structuring & Alignment:

- **M21: AIM Phase 2** – *Second-phase answer improvement.* If after the initial processing we have a draft answer or result, this phase would polish it further or add any final required structure. In a Q&A system, this might be where the final answer is formulated from the reasoning. In our app, once we have the code output and our feedback (if any), we might do a final improvement like formatting the output nicely. For instance, if the code output is very large or contains special characters, we might truncate or escape HTML in it. Or if multiple pieces of feedback were combined, perhaps ensure it forms a coherent paragraph. We can also insert any additional information: e.g., maybe a link to relevant lesson material if an error is common ("See Lesson 3 about loops for more info"). Essentially, treat the output (especially feedback text) as a draft that can be enhanced. Another example: ensure that if multiple points are made, they are in bullet form or numbered list for clarity (structured output). This phase can be a function `finalize_feedback(feedback_parts)` that returns a polished string or structured data ready for the user.

In terms of code placement, this likely happens just before rendering the template or JSON. For HTML, it might mean constructing the context variables to pass to Jinja such that everything is ready to display clearly (maybe splitting output lines, etc.). For JSON, it means the response fields are all set (maybe include an array of feedback messages rather than one string, if the frontend can handle it – that could be more structured). Since alignment (M22) is closely related, we might combine the steps: refine the output and simultaneously ensure alignment with policies.

- **M22: Safety & Alignment Layer** – *Final safety and alignment check on output.* This is the last guard before sending content out. We will verify the response does not violate any safety, ethical, or purpose alignment guidelines. For example, ensure we are not returning any profanity or disallowed content (shouldn't happen unless the user's code printed it – in which case, since this is an

educational context, maybe not a concern, but we could conceivably filter it). If the app had an AI generating text, definitely run that text through a moderation filter here. Alignment means the response should address the user's request and the educational goal appropriately. So we confirm that, for instance, if the user asked to run code, we indeed provide the code output or an error about their code – not something irrelevant. Or if the user is in a certain learning mode, the response aligns to that (e.g., in teacher mode we wouldn't just say "wrong", we'd align with the pedagogy).

Implementation: use Flask's `after_request` or just prior to return. If JSON, we can scan the response string for any disallowed patterns. If HTML, ensure no malicious script tags ended up in output (to prevent XSS if user printed `"<script>...</script>"` – we should have escaped that via Jinja autoescaping, but double-check). Essentially, treat the outgoing content as untrusted until this pass approves it. In many cases, this will be a no-op (pass-through) if nothing is wrong, but it's a safety net for edge cases.

Additionally, alignment can involve formatting the response to a consistent schema. For example, always returning JSON in `{success: ..., output: ..., feedback: ...}` format (which we do). We should ensure all paths (success, error, fallback) follow that format so the client handling is aligned (no surprises). This consistency is part of alignment with the app's interface contract. We may use Marshmallow schemas here too – e.g., a `ExecuteResponseSchema` to dump the final response, which inherently validates that all required fields are present and of correct type (if something is missing or wrong type, that's an integrity/alignment issue to fix).

- **M23: Interface Layer** – *Presentation to the user (UI/API response).* Finally, this corresponds to the Flask route returning the response – either rendering a template or sending JSON, thereby interfacing with the user. We should keep this layer very thin: ideally by the time we reach this point, all logic is done and we're just formatting data for output. For HTML pages, that means using Jinja2 templates to display the lesson content and any dynamic feedback or output. For API responses (like `/execute` and `/mark-complete`), it means returning a Flask `jsonify` or Response object with the final JSON.

We will apply standard best practices here: for HTML, use Jinja's escaping to avoid XSS (Flask auto-escapes by default in templates). For JSON, ensure it's serializable and not too large (the code output is bounded anyway). If streaming (SSE), this layer would handle setting the correct headers and streaming events. Essentially, the Interface Layer is where we integrate with Flask's output mechanism.

One improvement: currently `execute_code` returns JSON with just `success` and `output` [39]. We might extend that to include more structured info (especially if we add feedback). For instance, `{"success": false, "error_type": "RuntimeError", "message": "Division by zero error", "hints": ["Remember you cannot divide by zero"]}` for a failure – this gives the front-end more to work with (they can display error differently from normal output). Marshmallow can help define this schema and ensure alignment.

By mapping each ACA module to a concrete part of the Flask app as above, we **ensure every aspect of the cognitive architecture is addressed** in our system design. Some modules (especially in Layer 2) correspond to internal logic and can be implemented as plain Python classes/functions (making them independently unit-testable), while others in Layer 0/1 and Layer 3 tie into Flask request lifecycle (using before_request, after_request, and route logic). The execution order of modules in our implementation will mirror the ACA pipeline order, giving us a clear, deterministic flow: first global checks (M0-2), then configure

control (M3-9), then process and evaluate (M10-20), then finalize output (M21-23). This structured approach is straight from ACA's design of a "predictable pipeline", now applied to our app's request handling.

Integration Techniques and Hooks

To integrate these modules without turning the codebase into a tangled mess, we'll use Flask's hook mechanism and well-placed function calls: - **Flask Hooks**: Utilize `before_request` for M0-M2 global setup. For example, register a before_request that calls a function combining M0 (input sanitization) and M1 (auth check, if needed) at the very top. We can also use `after_request` or `teardown_request` for logging (part of M18) and final alignment checks (M22). Blueprint-specific before_request can apply certain modules only to certain routes (e.g., only the execute-code blueprint uses the code safety checks, whereas a general page blueprint might not need that). - **Decorators**: Create custom decorators for repeated patterns. For instance, a decorator `@safety_guard` could be applied to any route that takes user code, and inside it calls the safety check (M0) logic and returns an error if check fails. This keeps the route function clean. Similarly, if some routes require a certain mode or preference to be set, a decorator could ensure M2 is done. - **Service Layer Classes**: We can group some ACA modules into classes for clarity. For example, an `ACAController` class might have methods for M3 (configure), M4 (set_mode), etc., and a method `run_pipeline(request)` that executes M0 through M23 in order. However, implementing a single pipeline function for all requests might be overkill; we might have different pipelines for different actions (one for code execution, one for just page rendering which needs fewer steps). But having a structured approach (maybe a base class and specialized flows) could keep things organized. - **Independent Module Functions**: Many modules (like the agents in M14, or safety check M0, or mode selection M4) will be simple functions. We ensure each is **side-effect free** (except where intended like logging) and takes explicit inputs and returns outputs. This functional approach makes unit testing straightforward. For example, `safety_check_code(code_str) -> (safe: bool, reason: str)` can be tested with various code samples. - **Configuration and Dependency Injection**: The ACA modules often need config data (like preferences, thresholds for loops, etc.). We will store such config in one place (possibly in `app.config` or a dedicated config object passed through). For instance, the max refinement iterations or timeout durations can be config constants. The Mode System (M4) states might also be defined in config (or an enum class). By centralizing these, we can tweak behavior without changing code, and also adapt in tests (e.g., set shorter timeout in tests). - **Performance Considerations**: We will be careful to not introduce undue latency. Some ACA modules (like multi-agent evaluations) could be expensive if not managed. Our plan mitigates this by: - Making heavy parts optional or off by default unless needed (Path Selector chooses fast path unless conditions demand deep analysis). - Offloading where possible (e.g., actual code execution to a background thread/process, or potentially running style lint in a background thread in parallel with other checks). - Using efficient algorithms for analysis (e.g., our safety scan of code string will use simple regex or AST parse – which is quick, not an AI model on each request). - The overhead of running through these modules is mostly small Python function calls and if/else logic, which is negligible compared to running the user's code itself (which is the dominant cost). - We should, however, avoid doing something like calling an external API on each request unless absolutely needed (and if so, ensure it's asynchronous). - We will test performance by timing the execution pipeline on typical code to ensure it stays within acceptable limits (maybe target <100ms overhead for all our checks on top of the code run time).

Security Considerations with ACA Control

Integrating ACA modules in a code execution environment raises some security points: - **User Code Execution Risks**: Running arbitrary code is high risk. The current approach with restricted builtins reduces

risk of system damage (no open, no import by default) [22], but it's not foolproof. A crafty user might still consume resources (CPU/RAM) or find a loophole (like using `__builtins__['__build_class__']` to create objects or other magic). By adding ACA's Safety Guard (M0) and Bottleneck Monitor (M8), we mitigate many of these: we limit code length, check for banned tokens, and kill execution that runs too long or tries to allocate too much (we can perhaps intercept calls to `__builtins__` not allowed, or monitor memory if possible). - We'll run the code in a **clean namespace** (`exec(code, safe_globals, {})` as we do now [33]) so it cannot access the Flask app or other user's data. We must maintain that and perhaps further sandbox by executing code in a separate **process** to completely isolate memory and allow termination (threads share memory, so a malicious infinite loop in a thread would still tie up the Python GIL unless we interrupt it). - We should also ensure any file writes (if we allow `open` for the lesson on file handling) are done in a restricted directory (maybe a temp dir for that user) to prevent overwriting important files. Currently `open` isn't in safe builtins, so user's file handling lesson code wouldn't run unless we carefully allow it. One way is to intercept `open` calls by providing a custom `open` that only allows certain paths (like a sandbox folder). - **No Permanent State**: Ensuring "no memory of chain-of-thought is stored" is both a privacy principle and a security one (we don't want a malicious user to inject something into memory that affects the next user). We will not persist user code or internal reasoning beyond the request. Progress data is stored, but that's just completion flags, not the code itself or their outputs. We might consider clearing the execution environment each time. If using separate processes or Docker containers per execution in the future, that guarantees clean state. At minimum, our `safe_globals` is re-initialized for each exec, preventing leftover variables from a previous run affecting the next. - We need to caution that even read-only operations can be abused (like computing huge Fibonacci recursively to consume CPU). Our Bottleneck Monitor covers this with timeouts, but we also might pre-scan code for obvious infinite loops (though halting problem is undecidable generally, we can catch simple patterns or use runtime heuristics). - **ACA Modules Complexity**: The more code we add (for multi-agent, etc.), the more surface for bugs. We must ensure the ACA integration code itself doesn't introduce security issues. For instance, if we use an AI API for hints (not mentioned explicitly, but ACA suggests multi-agent reasoning, which might involve AI), that could return inappropriate content or be manipulated by user input (prompt injection). If so, our M22 (final safety filter) must catch that and sanitize or refuse output if it violates policy. We should also use the AI API's safety filters if available. - **Privacy**: If multi-user, one user's code or data should never leak to another. Our design with proper identity gating (M1) and context separation ensures each user's progress and code stays separate (e.g., if using a DB, always filter by current user ID). And we won't store their actual code long-term (unless maybe we log it in integrity logs, which should be access-controlled). - **Elevated Access**: Because our Flask app runs on a server, the exec environment should not run as a superuser in the OS. Running it under a lesser-privileged account or container is wise. If that's outside scope, at least limit what the Python process can do (no dangerous builtins, as done). Also, ensure the Flask `SECRET_KEY` remains secret (the new code uses a random key generation [40] which is good, and config in production will set a proper secret). - **Dependency Security**: If we introduce new packages (Marshmallow, Celery, etc.), keep them updated and be mindful of their vulnerabilities. E.g., Marshmallow primarily handles serialization in-process (should be fine), Celery will open network ports to a broker (use a protected Redis, not exposed publicly). - **Cross-Site Scripting (XSS)**: The app renders user-provided outputs in the browser (the code output). If a user prints HTML-like content, Jinja2 will escape it by default, preventing it from running as HTML. We must not mark it safe and unescaped unless intentionally. So we will ensure `{{ output }}` in templates is not unsafe. For the JSON API, if the front-end inserts the output into the page, it too should be escaped or shown in a `<pre>` tag to avoid interpretation. That's more of a front-end responsibility, but we can note it. - **CSRF and Auth**: The code already has CSRF protection enabled for forms and appropriately exempts the API endpoints [41]. After refactor, we should maintain CSRF protection for any new POST routes (progress, etc.), unless they're pure API consumed via XHR (then either use token-based auth or explicitly exempt but

accept only JSON). With user login, we'd use Flask-Login which has its own session management, and enforce secure cookies.

In summary, by combining the structured ACA approach with Flask's security features and careful sandboxing, we will **minimize risks** while adding powerful new capabilities. Each ACA module integration is designed such that it does not overly trust the user input or the output of any automated component without verification (multiple layers of checks and validations are present).

3. Implementation Timeline and Incremental Plan

To implement these improvements without breaking existing functionality, we will proceed in **phases**, validating at each step (with tests and possibly staging deployments):

Phase 1: Project Restructuring and Setup (Week 1-2)
- **Restructure codebase** into the new folder layout. Move routes from `app.py` into blueprint modules (`content.routes`, `execute.routes`, etc.) and implement `create_app` in `app/__init__.py` to register them [42]. During this move, keep the logic the same to avoid introducing bugs. Verify that the app still runs exactly as before (all routes responding, pages rendering). - **Integrate basic extensions**: initialize SQLAlchemy (but still use JSON for progress until next phase), set up Marshmallow (flask-marshmallow or standalone Marshmallow instance) for future use, and ensure CSRFProtect still applied (this can be done in app factory or in blueprint as needed). - **Testing**: Write tests for the core existing functionality *before* changing logic: e.g., a test that posting valid code to `/execute` returns success and output, posting code with `len>10000` returns the error about length (safety check) [21], marking a lesson complete updates progress (still using JSON file for now). These tests serve as a baseline. - **Static analysis**: Possibly add linting (flake8) and type checking configuration, to keep code quality high as we refactor.

Phase 2: Integrate Database and Core Improvements (Week 2-3)
- **Database for Progress**: Define a Progress model (and User model if needed now or in future). Migrate existing JSON progress data into the database (write a one-time script or just start fresh if data loss is acceptable during development). Update `mark_complete`, `get_progress`, etc., to use DB queries instead of file I/O. Ensure the progress functions are now methods or part of a service class for progress management. Add tests for the database model (e.g., mark complete twice -> only one record, progress percentage calculation matches expectations). - **Application Config**: Implement distinct config classes (Dev with `SQLALCHEMY_DATABASE_URI = 'sqlite:///app.db'` for example, Testing with SQLite in-memory, Prod maybe placeholder for now). Use environment variable to choose config in `create_app`. - **Blueprint and Factory Testing**: Ensure that using `create_app(TestConfig)` yields an app where we can simulate login (if added) and operations. Tests should use the app context and perhaps an in-memory DB to verify DB ops don't affect real data. - **Basic ACA Hooks**: Introduce the **Safety Guard (M0)** formally. This means expanding the `/execute` route logic: for instance, implement a `safety_check_code()` that scans for banned patterns (like `import`, `os.`, `sys.`) and call it before exec. If it fails, return a safe error JSON ("Unsafe code usage"). Add this to tests (attempt to import os should be blocked, etc.). Also configure execution time limits: possibly use Python's `signal.alarm` if running in main thread (works on UNIX) or implement the execution in a background thread with a timeout join. Test that an infinite loop code returns a timeout error rather than hanging. - Set up **before_request** hooks: e.g., if we have user accounts now, apply `@login_required` or custom before_request to lock down the execute and progress routes (maybe not needed if single-user, but structure it for future). - At this stage, the app's functionality is intact (maybe

slightly improved with safety) and more robust storage. We ensure all previous tests still pass (progress endpoints now working with DB but from user perspective no change except maybe faster and safer).

Phase 3: Introduce ACA Control Modules (Week 4)
- **Mode System & Preferences**: Implement a basic preference mechanism. Perhaps add a user setting for "assistant mode" vs "challenge mode" to simulate mode switching. If no user accounts, this could just be a toggle in config or session. Implement Mode System (M4) as a class or simple global state: define available modes (TEACHER_MODE default, maybe CHALLENGE_MODE), and enforce differences: in teacher mode, after code execution, the system will provide a hint if code is wrong; in challenge mode, it might simply say correct/incorrect with minimal hint. This can be done by branching logic in the evaluation section. Test by simulating each mode (perhaps expose an API to set mode in session for testing, or just call the evaluation functions with a mode parameter). - **Meta-Controller Decisions (M3 & M5)**: Create a function that given the request (lesson context, maybe user preference) decides `mode` and `path`. For now, maybe all normal requests use teacher mode and fast path (one execution, one evaluation). But if we detect (for example) a high difficulty lesson, we could choose a deeper path (maybe run an extra static analysis or allow multiple attempts internally). Implement a stub that always returns fast path unless a certain flag is set (we can simulate a deep path in a test scenario to ensure loop would work). - **Background Task integration**: If using Celery/RQ, set it up now. For example, configure Celery with Redis (if easily available) and convert the code execution function into a Celery task. This involves moving the actual `exec` call out of the Flask context. If this is too early, we can postpone Celery to Phase 4 or do a simpler thread method now. The goal is to ensure our architecture is ready: maybe have `/execute` either call the function directly (for now) or kick off a task. We can hide this behind a config flag (synchronous vs asynchronous execution) to ease development and testing. For now, likely keep synchronous (to not complicate testing), but ensure the design allows plugging a task queue easily. - **Multi-Agent Evaluation (M14)**: Implement at least two agent functions to demonstrate multi-perspective analysis: - CorrectnessAgent: If possible, create a small set of test cases for some lessons (e.g., known inputs and outputs). If lesson data includes expected output (some of the provided lessons might not have explicit tests, but we can add simple ones manually for demonstration, like for a sum function exercise, test that function works). The agent runs the user code in a restricted way or inspects output to determine correctness. Alternatively, if we can't have formal tests, the agent can simply check if no exception occurred (that's a minimal notion of success) or if output is non-empty when expected. Implement accordingly. - StyleAgent: Use a linter like `pyflakes` or `ast` to catch things (for example, detect if there are unused variables, or if the code is very inefficient like using nested loops if not needed – though auto judging style might be complex, we can at least check line length or other simple style points). Even a trivial style check like "did the user name any variable 'foo' or 'bar'? If so, suggest using meaningful names" – just to have something. - We can add more simple agents: maybe a **HintAgent** that, if code failed, tries to give a generic hint ("Check the conditions in your if statements" for logic errors, etc.). That might require knowing the lesson context (like expecting them to use a certain concept). We could encode a few rule-based hints per lesson for now (the lesson content might have hints listed; we can incorporate those as automated output if the user is stuck). - **Agent Integration**: Modify the `/execute` route flow: after running the code (M10), instead of immediately returning output, go through these agents (M14) to gather feedback. Then do conflict resolution (if any) and prepare a combined feedback. Update the JSON response to include a field for feedback or errors separate from raw output. On the frontend (if applicable), update how results are displayed (e.g., show feedback messages). - **Testing**: Write tests for each agent function with sample inputs (possibly mock an execution context). Then an integration test: e.g., input code that we know produces wrong output – the response JSON should contain `success: false` and a feedback message from CorrectnessAgent. Another test: a perfectly correct code – response should `success: true` and maybe feedback says "Great, correct!" or at least no error. This

ensures our pipeline is working. Also test that style suggestions appear when expected. - **Performance check**: If running these agents makes the request slow, measure it. If an agent (like running an external linter) is slow, consider marking it as optional (only run if quick or if deep mode). Since our initial agents are simple, it should be fine.

Phase 4: Advanced Reasoning & Output Alignment (Week 5)
- **Refinement Loop (M16)**: Implement a scenario to use the refinement loop. For example, if we integrate an AI for hints (optional), we could have it attempt an explanation and then refine if certain keywords missing. Without AI, perhaps implement a simple retry for code execution: e.g., if execution timed out (maybe the code had an infinite loop), as a refinement maybe run the code with an input limit or safe mode (this might not apply easily). Another angle: If multiple test cases and some fail, maybe the system could try to guess a fix for a common mistake and test again (very ambitious). As a simpler approach, we might simulate refinement by calling the agents again after initial feedback to see if any further insights (likely not much changes though). We may defer heavy refinement until a later version, but ensure the structure allows it (maybe keep a loop in the code with `while attempt < max_attempts and not quality_met:` that currently runs only once). - **Finalize Presentation (M21-M23)**: Refine the final output composition: - For HTML pages (lesson_view), maybe display a summary of the user's code run result nicely formatted. Possibly include a section "Feedback" that shows hints or messages from our agents, styled with icons (like a checkmark for correct, a lightbulb for a hint, etc.). - For JSON, we finalize a consistent response format. Possibly:

```
{
  "success": false,
  "output": "",
  "error": "NameError: 'x' is not defined",
  "feedback": "It looks like you tried to use a variable before defining it."
}
```

or if success:

```
{
  "success": true,
  "output": "Hello, World!\n",
  "feedback": "Good job! Your program printed the expected greeting."
}
```

Ensure that in both cases the fields exist (maybe feedback is an empty string or null if none). Use Marshmallow schema to enforce this structure and validate in tests. - Apply final safety filter: e.g., if `error` message contains file paths or weird info, strip it. Also, double-check that the feedback doesn't reveal solution outright unless that's intended. - Alignment check: Confirm that, for instance, in challenge mode, we don't accidentally give the full answer in feedback. We can write a test simulating challenge mode where user code is wrong and ensure feedback does *not* give them the correct answer (perhaps just a nudge). Conversely in teacher mode, maybe we do give a stronger hint. So we verify the mode is respected in output content. - **Security audit**: Do a thorough review of the integrated system. Try intentionally malicious code strings in a safe environment to see if anything slips through (like a code that tries to import os via

some obscure method). Ensure timeouts and memory limits are working by testing a known infinite loop. Also test for potential XSS by having code print `<script>alert('x')</script>` and confirming it appears escaped on the page (not executed). - **Documentation and Developer Guidance**: By end of phase 4, document the new architecture: perhaps in a README or internal docs, describe how the modules map and how to add new lessons or new ACA modules. This helps any developer (especially systems-focused devs as mentioned) to understand the design and build further.

Phase 5: Optimization and Future Enhancements (Week 6, ongoing)
- **Performance Tuning**: If the multi-agent evaluations slowed things down, consider enabling asynchronous execution. Possibly move agents that can run in parallel to threads (e.g., run style check while tests are running). Or integrate Celery fully now: the `/execute` route posts the job and immediately returns a task ID, and we add a new route or use a web-socket to push results when ready. This is a bigger change to UX (user has to wait differently), so only do if needed. Many educational apps opt for synchronous small runs, which is okay if under a couple seconds. - **Expand ACA Integration**: Add more of the ACA nuances if desired: - Emotional adaptation (the app could detect if a user failed the same exercise multiple times via progress records and then automatically switch to a more supportive mode or offer an extended hint). - Logging & Analytics: Use the integrity logs (M18) to analyze usage patterns. E.g., how often are fallbacks triggered? This could highlight if some lesson causes lots of timeouts (maybe a hint needed in content to warn about infinite loops). - Possibly incorporate an LLM for generating hints or explanations, which would heavily leverage the ACA pipeline (the reasoning chain and alignment modules would be critical here). This would be a whole project in itself, but our groundwork with modes, safety, and multi-agent structure would facilitate safe usage of an LLM. - **Continuous Testing**: As new features added, continue adding tests. Aim for comprehensive coverage of all ACA-related code (since it's new and could be complex). Particularly test edge cases like: a user submits no code (empty string) – safety check should handle that gracefully; a user code prints a very large output – ensure maybe truncated and no performance issue; simulate a database failure (maybe use a wrong connection string in config for a test) to see if fallback returns a 500 gracefully without sensitive info. - **Deployment considerations**: When deploying, run in production config with debug off (ensuring any debug routes or diagnostic info is protected). Possibly deploy the background worker if Celery is used. Monitor resource usage with real users to adjust timeouts or limits.

By following this phased approach, we ensure at each stage the app remains functional (tests from earlier phases keep passing) and we progressively enhance it with ACA features. Stakeholders can verify after each phase that the goals are being met (e.g., after Phase 3, they should see more informative feedback on code execution; after Phase 4, system should feel more "intelligent" and aligned with teaching goals, without sacrificing reliability). The modular design means we can even deploy these improvements gradually (feature flags to turn on the new evaluation pipeline once fully tested, for example).

Throughout these steps, our focus is on creating a **clean, maintainable, and extensible** architecture. We adopted modern Flask best practices (blueprints, factory, ORM, Marshmallow) to make the app scalable and testable, as evidenced by established sources [7] [14]. At the same time, we carefully mapped the conceptual ACA v4 modules to concrete mechanisms in the app, ensuring the final system benefits from ACA's structured reasoning, safety enforcement, and alignment guarantees. This plan provides a solid foundation for the team to implement and for future developers to build upon, with clear file structure and testing strategy to catch regressions early. Each component is justified with best-practice reasoning or ACA specification references, so the team can be confident in the rationale as they proceed to coding.

Sources:

- Miguel Grinberg, *Flask Mega-Tutorial* – recommendations on large app structure (blueprints, app factory, testing) [7] [8]
- Flask Official Documentation – using Blueprints for modular design [1]; using Celery for background tasks [19]
- Cameron MacLeod – using Marshmallow for input validation in Flask (cleaner than manual checks) [14]
- ACA v4 Master Spec – cognitive architecture modules and principles (deterministic pipeline, safety overrides)
- Current App Code – examples of existing behavior (safe builtins for exec, length check) [21] [22] which we will extend, and progress JSON handling [13] to be replaced.

---

[1] Modular Applications with Blueprints — Flask Documentation (3.1.x)
https://flask.palletsprojects.com/en/stable/blueprints/

[2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [20] [42] The Flask Mega-Tutorial, Part XV: A Better Application Structure - miguelgrinberg.com
https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-xv-a-better-application-structure

[13] progress.py
https://github.com/mokuzaitenko-source/support-automation-learning-platform/blob/a92b7e0b0b0a9a6385c1cf216a5d1db01eaf47f0/progress.py

[14] [15] [16] Better parameter validation in Flask with marshmallow - Cameron MacLeod
https://www.cameronmacleod.com/blog/better-validation-flask-marshmallow

[17] [18] [19] Background Tasks with Celery — Flask Documentation (3.1.x)
https://flask.palletsprojects.com/en/stable/patterns/celery/

[21] [22] [33] [38] [39] [40] [41] app.py
https://github.com/mokuzaitenko-source/support-automation-learning-platform/blob/a92b7e0b0b0a9a6385c1cf216a5d1db01eaf47f0/app.py

[23] [24] [25] [30] [31] [32] ACA_v4_Module_6_Lands_Mixer.pdf
file://file-Hrcd6DUgPFktyCUnQEPf5E

[26] [27] [28] [29] ACA_v4_Module_4_Mode_System.pdf
file://file-BnRpbKk4t7fDJQhn5Cvptk

[34] [35] [36] [37] ACA_v4_Module_14_Eve_Core_Detailed.pdf
file://file-DgU1rvcTtEEML7dVnjk2Qc