ChatGPT

AZR Planner Service – Design and Implementation

Background and Objective

The **AZR Planner Service** is a component of the Osiris project responsible for dynamically planning a training **curriculum** for the Absolute Zero Reasoner (AZR) algorithm. In the AZR paradigm, a single large language model (LLM) acts as both **task proposer and solver**, continuously self-generating new reasoning tasks and solving them to improve its capabilities [1]. This self-play approach allows the model to *"propose tasks that maximize its own learning progress and [improve] reasoning by solving them"* [2]. The planner service's objective is to **produce a curriculum plan** – a set of next tasks or "steps" for the model – that adapts to the model's current state and available compute resources. Ultimately, we need to ship a production-grade planning service that:

- **Allocates Compute Dynamically:** Examines **Alpha/Beta compute resources** (as defined in our infrastructure) and schedules tasks without exceeding available capacity.
- **Recommends Curriculum Steps:** Generates meaningful new training tasks (the curriculum) tailored to the model's learning needs, rather than returning a fixed dummy plan.
- **Outputs Contextually Varied Plans:** The `/plan` API should return plans that differ based on input **state** (e.g. model performance metrics, resource load), instead of a one-size-fits-all dummy response.

Current State: The existing implementation is rudimentary – it returns a hard-coded *dummy patch* and priority, without real logic. There are placeholders (stubs) for resource checks (denoted as "alpha/beta resource") and curriculum generation, which are not yet implemented. This means the service currently ignores actual compute availability and does not truly analyze the model's progress or adapt the curriculum. Our task is to replace these stubs with real, intelligent behavior.

Goal: Design and implement a planning algorithm within the FastAPI `/plan` endpoint that meets the above needs. We will also provide a design document for the algorithm, implement unit tests for various scenarios, and update configuration (Helm charts) to support new settings. By completion, the planner should **return variable, context-aware plans based on state inputs** and pass >90% of synthetic unit tests for planning logic (per success criteria).

Key Requirements and Challenges

Based on the project needs and success criteria, the following are the key requirements for the AZR Planner Service and the challenges involved in achieving them:

- **1. Real Alpha/Beta Resource Checks:** The planner must incorporate *actual compute resource availability* into its decision-making. The terms "Alpha" and "Beta" resources likely refer to two classes of compute (for example, high-performance vs. standard nodes, or GPU vs. CPU clusters). The planner should check how much of each resource is free or usable, instead of using dummy fixed values. **Challenge:** Determining a reliable way to obtain current resource utilization at runtime (e.g.,

querying Kubernetes APIs or reading metrics provided via the service's context/state). We must ensure the plan does not schedule more tasks than resources can handle, and adjust the plan if resources are constrained.

- **2. Curriculum-Generation Logic:** Implement the logic for generating new training tasks ("curriculum steps") based on the model's current state and past performance. In AZR, the model *"self-evolves its training curriculum"* by proposing tasks that are neither too easy nor impossible [2] [3]. The planner should mimic this by selecting tasks of appropriate **type and difficulty** that maximize learning. **Challenge:** We need a way to quantify the model's current proficiency (perhaps via success metrics in the state) and to generate tasks that push the model's boundaries without being unsolvable. This may involve using heuristics or simplified versions of AZR's reward scheme (e.g. favor tasks where the model's success rate is around 50% for maximum learning signal [3]).

- **3. Dynamic Plan Synthesis:** The service must synthesize a plan that ties resource allocation to the curriculum steps. This means deciding how to distribute tasks given the available Alpha/Beta resources and possibly setting a **priority** for the plan. The output should not be static – it should reflect the input state context (e.g., if the model is struggling in math tasks, plan more of those; if Alpha compute is nearly full, perhaps delay or use Beta resources). **Challenge:** Designing a flexible plan data structure (often called a "patch" in this context) that can represent different numbers/types of tasks and their resource assignments. Also, defining what the "priority" field means in a dynamic way (maybe urgency or importance of this plan) and computing it from context.

Additionally, we must integrate these changes in a **maintainable, testable** way: - Write unit tests that simulate various scenarios (different state inputs for performance and resource levels) to verify that the planner responds correctly (fulfilling the 90% test pass criterion). - Update configuration/Helm charts to introduce any new **config flags** or parameters (for example, thresholds for resource utilization, default task counts, toggles for dummy vs real logic) so that the behavior can be tuned in production.

In the next sections, we outline the design of the planning algorithm addressing these requirements and then discuss implementation specifics including API design, testing, and configuration changes.

Planning Algorithm Design

The planning algorithm for AZR will be inspired by the Absolute Zero Reasoner's **self-play curriculum learning loop** [4]. In AZR, the model iteratively **proposes** new tasks and then **solves** them, using feedback to improve. We will adapt this idea into a planner that, given the current state, produces a set of tasks (the curriculum for the next training iteration) and suggests how to allocate resources to execute them. The design balances **resource awareness** with **learning potential**:

1. Resource Availability Checks (Alpha/Beta)

First, the planner will perform **Alpha/Beta resource checks** to gauge available compute. This ensures we do not plan more work than the system can handle. There are two scenarios for obtaining this info: - **State-Provided Metrics:** Ideally, the `state` input to `/plan` includes current resource utilization or free capacity for both Alpha and Beta resources. For example, `state` might contain fields like `"alpha_free": <N1>, "beta_free": <N2>` (or utilization percentages). If so, the planner will use

these values directly. - **External Query:** If such metrics are not in `state`, the planner might call out to a resource manager (e.g., Kubernetes API or a monitoring service) to get current free resources. In production, integrating with Kubernetes metrics (via the Downward API or a custom endpoint) could provide number of free GPU pods, CPU usage, etc.

Using these inputs, the planner logic will: - **Determine Capacity:** Assess how many tasks or how much workload can be scheduled. For instance, if "Alpha" corresponds to GPU nodes and only 1 Alpha node is free, the plan might limit itself to one heavy GPU task. If Beta (perhaps CPU workers) has more headroom, lighter tasks can fill those. - **Adjust Task Counts:** Scale the number of proposed tasks based on available compute. For example, if plenty of Alpha resource is free, we can schedule a larger batch of tasks (to utilize idle compute). If resources are scarce or nearly fully utilized, the planner might return a smaller plan (or even a plan indicating to wait). - **Allocate Task Types to Resources:** Certain tasks might be designated for Alpha or Beta. If “Alpha” denotes powerful compute (GPU-intensive tasks like large model inference), those tasks should be assigned there, whereas simpler verification or background tasks could use Beta. The planner will tag each proposed curriculum step with the intended resource type or requirements. (E.g., a complex coding task requiring model inference might be marked for Alpha, while a simple test or data preprocessing step might go to Beta.)

This resource-aware planning ensures efficient cluster utilization. It aligns with the principle of **dynamic resource allocation**, where scheduling decisions are driven by current utilization to optimize throughput (similar ideas are seen in cluster managers that *“flexibly request and allocate resources among tasks”* [5]). By implementing real checks, we move away from the dummy always-available assumption to a responsive system that can, for example, hold off on launching new tasks if both Alpha and Beta are at capacity.

2. Curriculum Generation Strategy

Next, the planner generates the **curriculum tasks** themselves. This is the core of AZR’s self-improvement loop – proposing new challenges for the model. We will incorporate key principles from the AZR research:

- **Autonomous Task Proposal:** The planner will create tasks optimized for the model’s learning progress, as opposed to relying on static pre-defined tasks [6]. This means examining the model’s performance data from the `state` (if provided) to identify areas that need improvement or further practice. For example, if the state indicates the model recently had a lower success rate on mathematical reasoning, the planner might generate more math-focused tasks next. Conversely, if the model is excelling in a certain domain, the planner might increase the difficulty or introduce a new domain to avoid stagnation.

- **Moderate Difficulty for Maximizing Learning:** We aim to choose tasks that are *neither too easy nor unsolvable* for the current model. According to the AZR paper, *“if a task is either trivial to solve… or unsolvable, the task provides little to no learning signal… In contrast, tasks of moderate difficulty, where the solver occasionally succeeds, offer the richest feedback and greatest potential for learning.”* [3]. Practically, the planner can use the model’s past success rate on similar tasks as a proxy for difficulty:

- If success rate is very high (task too easy), propose a harder variant of that task (e.g. a more complex problem or code snippet).
- If success rate is very low (task too hard), the planner should either simplify the task or ensure prerequisite simpler tasks are solved first.

- If success is intermediate (~40-60%), this indicates the task is at a good learning difficulty; the planner can include similar tasks to solidify learning, or push just slightly harder.

- **Task Diversity (Reasoning Modes):** AZR uses three reasoning modes – deduction, induction, abduction – each generating different types of coding tasks [7] [8]. Our planner should consider a diversity of task types to cover different reasoning skills and prevent overfitting to one pattern. For example, the planner might ensure that in each planning cycle, there is a mix of:

- *Deduction tasks*: Provide a program and input, ask the model to predict the output (logical reasoning) [8].
- *Abduction tasks*: Provide a program and an output, ask the model to infer the missing input (like reverse reasoning) [9].
- *Induction tasks*: Provide input-output pairs and ask the model to synthesize a program (generalizing patterns) [10].

Including multiple modes ensures the curriculum is **comprehensive**, echoing how AZR covers complementary reasoning tasks to fully exercise the model [7]. The specific implementation might simplify this (depending on what the model and environment support), but maintaining some variety (e.g. not all tasks are of one type) is ideal.

- **Validation of Tasks:** Once candidate tasks are generated (especially if they involve code), the planner should validate them using the available tools. In AZR's loop, Python execution is used to filter and verify tasks [11] [12]. In our service, this could mean:
- If a task is to write a piece of code or solve a problem, we might run a quick check (perhaps using a sandbox or the model itself in a "solver" role) to ensure the task is well-formed and not a dead-end. For example, if the task is a coding challenge, ensure that there is a known solution or that the code template runs without error.

- This step prevents proposing tasks that are impossible to evaluate or learn from (like tasks with no correct answer or with errors). It also mimics the *"environment provides verifiable feedback"* principle of Absolute Zero [13] – every task should have a way to be checked so the model can get a reward signal.

- **Learning Progress Estimation:** Optionally, incorporate a notion of **learnability reward** as in AZR [14]. In the paper, they actually roll out the model on proposed tasks to estimate success, but in a simpler planner, we might simulate this by using heuristics. For instance, we could predict the model's chance of solving a new task based on related past tasks. If we anticipate near 50% success, that task is ideal. This heuristic can guide which tasks get included in the plan (those with the highest expected learning benefit).

In summary, the curriculum generation logic will produce a list of tasks tailored to the model's needs and abilities, and each task will include metadata like its type, difficulty, and possibly an ID or description. This set of tasks forms the core of the "plan patch" to be returned.

3. Dynamic Plan Synthesis and Output Structure

Finally, the planner composes the **plan output** that will be returned via the API. The plan will combine the resource considerations and the generated curriculum tasks into a structured response. We define what the plan contains and how it is formatted (based on the existing dummy and project conventions):

- **Tasks List (Curriculum Patch):** A collection (list/array) of task specifications for the next training iteration. Each task entry might include fields such as:
- `task_id` or `description`: identifying the task or giving details (e.g., "Solve a deduction coding problem about sorting algorithms").
- `type`: the reasoning mode or category (deduction/induction/abduction or other domain classification).
- `difficulty` or expected `level`: perhaps an indicator of complexity or an estimated probability of success (for debugging/analysis).
- `resource`: which resource type it should run on (Alpha or Beta). For example, a task that requires the LLM to generate code and run it might be marked `resource: "Alpha"` if Alpha is a GPU node with the model loaded, whereas a simpler data preparation step might be `resource: "Beta"`.

This structure effectively acts as a **curriculum patch** – it's the delta or addition to the model's training curriculum for this cycle. It is dynamic: if we call `/plan` with different states, we should get a different list of tasks.

- **Compute Allocation/Counts:** In addition to listing tasks, the plan might summarize how resources should be allocated. If the tasks list is already annotated with resource tags, the service could also provide a high-level allocation summary. For example:
- `alpha_tasks_count`: N (number of tasks assigned to Alpha resource),
- `beta_tasks_count`: M (tasks for Beta),

- Or even specific compute requirements like `expected_gpu_hours` or similar metrics if relevant. (This depends on how detailed the planning needs to be for scheduling; initially, a simple count per resource is sufficient.)

- **Plan Priority:** The existing API returns a `priority` along with the patch. We will compute this priority based on context:

- Priority could indicate the urgency or importance of executing this plan. A higher priority might mean the model is at a critical learning juncture or resources are plentiful and we want to utilize them immediately. A lower priority might mean the plan is optional or could be deferred (e.g., if the system is busy).
- One strategy: If the planner finds that **resource utilization is low** (i.e., many free resources), it can assign a higher priority, implying "we should run these tasks now to use idle resources." If resources are very constrained, it might output a lower priority, implying "this plan can wait or be queued until resources free up." Another angle is the model's learning need: if the model has plateaued and needs new tasks urgently to improve a certain skill, mark the plan as high priority.

- We will define a clear scale or criteria for priority (for example, numeric scale 1-10 or a few levels like "low/medium/high"). In unit tests, we can then verify that given certain states (e.g., 0% free

resources vs 50% free, or very low performance in some area), the priority value changes appropriately.

- **Timestamp or ID (if needed):** In a production scenario, we might include a timestamp or plan ID for tracking, but this is optional. Given the current context, likely not required unless the consumer of this API expects it.

The **output JSON** might look like:

```
{
  "tasks": [
    { "id": "task1", "type": "deduction", "description":
"Predict output of code X for input Y", "resource": "Alpha" },
    { "id": "task2", "type": "induction", "description": "Write a function to
achieve Z given examples", "resource": "Alpha" },
    { "id": "task3", "type": "deduction", "description": "Simple math word
problem", "resource": "Beta" }
  ],
  "alpha_tasks_count": 2,
  "beta_tasks_count": 1,
  "priority": 8
}
```

(The exact schema will be based on how `dummy_patch` *was structured, but this illustrates a dynamic plan.)*

This synthesis step essentially **combines the curriculum and resource considerations** into one coherent plan. It ensures the plan is **contextually aware**: if called with a different state, the tasks list, counts, and priority should all adjust accordingly. This satisfies the requirement that the service *"returns contextually variable plans based on stubbed 'state' inputs"* (Success Criteria).

We will now discuss how this design translates into implementation, including integration into the FastAPI service, configuration via Helm, and testing.

Implementation Plan

With the algorithm defined, implementing it involves modifying several parts of the project: the `azr_planner` service code (particularly the FastAPI endpoint and any helper modules), writing unit tests, and updating deployment configuration.

FastAPI `/plan` Endpoint Logic

The core implementation will live in `services/azr_planner/main.py` (the FastAPI service code). We will replace the dummy return with calls to our new planning logic. A possible approach:

1. **Parse Input State:** The `/plan` endpoint likely receives a JSON payload representing the current state. We will define a Pydantic model or similar for this input if not already defined. For instance:

   ```
   from pydantic import BaseModel
   class PlannerState(BaseModel):
       alpha_free: float
       beta_free: float
       performance_metrics: dict  # e.g., {"deduction_success": 0.8,
   "induction_success": 0.5, ...}
       # ... other fields as needed
   ```

   Inside the endpoint function, we'll have something like:

   ```
   @app.post("/plan")
   def plan_endpoint(state: PlannerState):
       plan = generate_plan(state)
       return plan
   ```

   The `generate_plan` function (which we will implement) encapsulates the algorithm from the design section.

2. **Implement** `generate_plan(state)`: This function (could also be a method of a Planner class) will:

3. Read resource availability from `state` (e.g., `alpha_free = state.alpha_free` etc.) and apply **Resource Checks**. For example:

   ```
   max_tasks_alpha = int(state.alpha_free)  # or derive from free units
   max_tasks_beta = int(state.beta_free)
   ```

   (We assume `state.alpha_free` might be a number of free slots or a ratio; if it's a percentage, we convert it to a threshold decision.)
4. Read performance or curriculum-related info. For instance, `state.performance_metrics` might tell us recent success rates or reward signals for different task types.
5. **Generate tasks**: using the logic in Curriculum Generation Strategy:
   - Determine how many tasks to propose. Possibly `tasks_to_schedule = max_tasks_alpha + max_tasks_beta` as an upper bound, then maybe limit further if not needed.

- For each task to propose, decide its type and content:
  - Use performance metrics to decide type: e.g., if induction success is lowest, include an induction task.
  - Use difficulty heuristic: if success for a type is high, create a harder variant; if low, maybe create an easier or intermediate task.
  - Actually form the task description or payload. This could be as simple as a dict with type and a placeholder description if the actual content is generated elsewhere, or we might integrate with `research/azr/main.py` to generate a concrete problem. The **AZR research code** might have utilities for generating tasks (e.g., functions that produce a coding puzzle). We should check `research/azr/main.py` for any such logic to reuse. If found, we can call those functions to create realistic tasks.
  - Validate tasks if possible. For instance, if we generate a code task, perhaps call a function to execute it and get the correct output to include in the task (so the solver later has an answer to check against).
- Assign each task to a resource: e.g., heavy tasks default to Alpha until `max_tasks_alpha` is reached, then overflow to Beta, etc., or based on type. Perhaps deduction/induction tasks (involving the model heavily) go to Alpha, whereas simple abduction or tests go to Beta.

6. **Assemble the plan structure**: Format the tasks and any summary info into the returnable dict. Compute `priority`:
   - For priority, implement a rule as discussed (for example, if `state.alpha_free` or `beta_free` is above some threshold, set priority high). We might also incorporate performance urgency (if a certain metric is far below target, maybe high priority to address it).
   - Priority could be a numeric value. If the original dummy had a fixed number, we can use similar scale (say dummy was 5, now we can vary 1-10).
   - E.g.:

     ```
     if state.alpha_free > 0.5 and state.beta_free > 0.5:
         priority = 10  # lots of free resource, use immediately
     elif state.alpha_free < 0.1 and state.beta_free < 0.1:
         priority = 3  # very busy, low priority plan
     else:
         priority = 7  # moderate situation
     ```

     And possibly adjust up/down if performance metrics indicate critical learning needs.

7. Return the plan dict.

8. **Integrate Configurable Parameters:** To make this production-grade, certain “magic numbers” or behaviors should be configurable:

9. Thresholds for resource usage that determine priority levels or task scaling.
10. Default number of tasks to plan per available resource unit.

11. Flags to turn certain modes on/off (for example, if induction tasks are not fully supported yet, we could disable them via a config flag). We will use either environment variables or a config file (maybe via `research/azr/settings.py` or similar) to store these. For example,

`MAX_TASKS_PER_ALPHA=2` (so if 1 alpha unit free, plan up to 2 tasks on it) or `MIN_RESOURCE_FRAC_FOR_HIGH_PRIORITY=0.5`. The code will read these values (perhaps using `os.getenv` or a settings object).

12. **Logging and Error Handling:** Add logs that make it easier to debug/troubleshoot in production:

13. Log the received state and the generated plan (at an appropriate log level).
14. If resource info is missing or unusual, log a warning. For instance, if `state` doesn't have alpha_free, log "Alpha resource info not provided, assuming 1 task capacity".

15. If task generation fails (e.g., a validation step raises an error), catch exceptions to ensure the API still returns a sensible result (perhaps with fewer tasks or a message). The service should degrade gracefully rather than crash.

16. **Unit Testing Hooks:** We may structure some of the above logic into smaller functions to facilitate unit testing. For example, a function `choose_tasks(state)` that returns a list of tasks given a performance snapshot, and a function `allocate_resources(tasks, alpha_free, beta_free)` that assigns tasks to resources or prunes tasks if necessary. This way, each piece can be tested in isolation.

The **Dockerfile and requirements.txt** will be updated if needed. If we decide to use any external libraries (for example, a Kubernetes client to fetch resources or a math/coding library for generating tasks), those would be added to `requirements.txt`. At this time, it might be possible to avoid heavy dependencies by assuming resource info is passed in and by using simple Python for tasks (the `dummy_patch` resources might already contain a simple task example we can mimic). If we do add dependencies (like `kubernetes` Python client), we ensure the Dockerfile includes them. Given the deliverables, it's likely we can implement without new dependencies (using state input or basic logic), so changes to requirements.txt will be minimal.

Unit Testing the Planner

A suite of **unit tests** will be created (possibly under `services/azr_planner/` or a tests directory) to validate that our planner works as expected. We will craft **synthetic state inputs** to simulate various scenarios and assert that the output meets the criteria. Some test cases:

1. **Baseline Scenario:** Moderate resources free (e.g., 50% Alpha, 50% Beta), and balanced performance metrics. We expect the planner to return a mix of tasks for both resources, with a medium-high priority. For example, if success rates are all ~50%, tasks should still be generated (not skipped due to difficulty extremes), maybe one of each type, and priority around default (e.g. 7 as per our scheme).

2. **High Resource Availability:** e.g., `alpha_free=1.0, beta_free=1.0` (100% free). Planner should maximize tasks (up to some limit) and set a high priority. Test that the number of tasks >= in baseline and that `priority` is at the top of our scale (e.g., 10).

3. **Low Resource Availability:** e.g., `alpha_free=0.0, beta_free=0.2` (Alpha none free, Beta almost full). Planner should perhaps return fewer tasks (maybe even an empty list or minimal tasks

assigned to Beta) and a low priority. Verify that when `alpha_free` is zero, no tasks are assigned to Alpha in output, and perhaps priority is reduced.

4. **High Performance vs Low Performance:** Simulate the model being *too good* at tasks or *struggling*. For instance:

5. If `state.performance_metrics = {"deduction_success": 0.95, "induction_success": 0.9, "abduction_success": 0.9}`, the model is acing everything. The planner should then create tasks that are more challenging. We might expect the planner to still produce tasks, perhaps flagged as higher difficulty. In absence of an explicit difficulty field, we might infer difficulty via descriptions or variety. This test mainly ensures we *don't* output a plan that is empty just because it's doing well; rather we push new tasks (maybe from a different domain or higher complexity).

6. If `state.performance_metrics = {"deduction_success": 0.1, "induction_success": 0.1, "abduction_success": 0.1}`, the model is failing a lot. The planner might then either propose easier tasks or focus on one area to improve incrementally. We should verify the planner still gives some tasks (not too many), possibly with a note that they're simpler. In the plan, since the model is struggling, we might not increase difficulty; the tasks might be of the same type that it's trying to learn, to give it practice. We also might set priority high here, reasoning that the model needs intervention (depending on our design, this could be a case for high priority to address the failure).

7. **Edge Cases:**

8. If the state is missing some data (say no performance metrics given), ensure the planner can still produce a default plan. Perhaps default to a predetermined set of tasks (one of each type) with moderate difficulty.
9. If resource values are extremely out of expected range (e.g., negative or above 1.0 if they are fractions, or extremely large numbers if they represent counts), ensure the logic caps them or handles gracefully (maybe treat negative as 0, and large numbers as full capacity).
10. If an error occurs in task validation (simulate by maybe injecting a task that causes an exception), the planner should catch it and possibly exclude that task but still return the rest.

Each test will assert things like: - The structure of the returned plan (has keys `tasks` and `priority`, etc.). - The number of tasks fits within expected bounds for the given resource availability. - The content of tasks reflects the state (for example, if induction_success was lowest, check that at least one task of type "induction" is in the list). - Priority value correctness (monotonic behavior: more resources => higher priority, etc., if applicable).

Achieving **90%+ pass rate** means our tests cover a broad range of scenarios, and the logic handles them correctly. We will likely use a coverage tool to ensure most branches of `generate_plan` are tested.

Helm Chart and Configuration Updates

To deploy this new behavior, we need to update the configuration in the **Helm charts** (under `helm/osiris/`) so that the service can be tuned in different environments without code changes. Key updates:

- **New Config Values:** Introduce new entries in `helm/osiris/values.yaml` for any configurable parameters we identified. For example:

```
azrPlanner:
  enabled: true
  alphaThresholdHigh: 0.5
# example: threshold for high-priority resource availability
  alphaTasksPerUnit: 2      # how many tasks to schedule per free Alpha
unit
  betaTasksPerUnit: 1       # tasks per Beta unit
  minTasks: 1               # always propose at least 1 task
  enableInduction: true     # example flag to enable induction tasks
```

  These are illustrative; actual values and flags will match what we implement. We also include a toggle for the planner service itself if not already present (`azrPlanner.enabled`) to easily turn it on/off.

- **Deployment Template:** In `helm/osiris/templates/azr-planner-deployment.yaml`, we will pass the config into the container as environment variables or command-line args. For instance:

```
env:
  - name: ALPHA_THRESHOLD_HIGH
    value: "{{ .Values.azrPlanner.alphaThresholdHigh }}"
  - name: ALPHA_TASKS_PER_UNIT
    value: "{{ .Values.azrPlanner.alphaTasksPerUnit }}"
  # ... etc for each new config
```

  and similarly for Beta. These env vars will be read by the `main.py` or a config module (like `research/azr/settings.py`). If `settings.py` exists and uses something like `os.getenv` to set defaults, we ensure it pulls these values. We should double-check if `research/azr/settings.py` already defines any related constants that we can override.

- **Service Template:** The `helm/osiris/templates/azr-planner-service.yaml` likely defines the Kubernetes Service (network) for the planner. We may not need to change it unless we want to adjust ports or add annotations. Since our changes are internal to logic, the Service (exposing the API) likely remains the same. We ensure the `/plan` endpoint is still accessible on the same port.

- **ConfigMap or Secret:** If a lot of configuration is needed, alternatively we could use a ConfigMap. But given these are simple numeric flags, env vars in the deployment are fine.

- **Helm Documentation:** Update any relevant README or comments in `values.yaml` to explain the new fields, so operators know how to adjust them. For example, comment that `alphaThresholdHigh` is "if alpha_free > this fraction, planner marks plan high priority."

After these changes, deploying the Helm chart will set up the planner service with real logic. The **Helm values can be tuned** depending on environment (for instance, in a small test cluster set smaller task counts).

Conclusion and Success Criteria

In summary, we have designed a robust planning algorithm for the AZR Planner Service that accounts for compute resources and dynamically generates curriculum tasks based on the model's learning state. By drawing on the Absolute Zero Reasoner paradigm – where the model *proposes and solves its own tasks to self-improve* [1] – our planner will create **contextually appropriate, varied, and challenging tasks** for the model, rather than using a static plan. We integrated **resource-aware scheduling**, ensuring that tasks are aligned with available Alpha/Beta compute capacity, which is critical for smooth operation in a production cluster.

We have outlined the modifications to the FastAPI service code to implement this logic, including parsing input state, generating tasks (with references to AZR's induction/deduction/abduction modes), validating those tasks, and outputting a structured plan with a computed priority. Configuration is made flexible via Helm values and environment variables, allowing the system behavior to be tuned without code changes (e.g., adjusting difficulty or resource thresholds).

Success Criteria Check: - **Contextually Variable Plans:** The planner no longer returns a one-size dummy patch. Instead, it produces plans that change with the input. For example, given different stubbed states in tests (varying resource levels and performance metrics), the output tasks and priority will differ. This meets the criterion of "returns contextually variable plans based on state inputs." - **Unit Tests Passing:** We will create comprehensive synthetic tests for the planner. The target of 90% pass rate will be met by covering normal and edge cases, ensuring the planner behaves as expected. By design, our algorithm handles a wide range of scenarios (from resource saturation to model mastery and failure modes), which should translate to high test coverage and passing rates. Any bugs found will be addressed to improve test outcomes, aiming for that 90%+ threshold. - **Production-readiness:** With real resource checks and configurable parameters, the planner is ready for deployment. It will not overload the system (thanks to the Alpha/Beta checks) and will continually guide the model's training with meaningful curriculum steps. The updated Helm charts allow ops teams to configure the service and integrate it into the Osiris stack easily. We will verify that the deployment manifests include the new env vars and that the service starts with those configs.

By fulfilling these deliverables – the design document (this write-up), the implemented `/plan` logic with tests, and updated Helm configurations – we move the AZR Planner Service from a placeholder to a **production-grade component**. It embodies the innovative AZR approach from the research (Absolute Zero self-play learning) within a practical system that allocates resources efficiently and accelerates the model's learning curve.

Overall, this implementation will enable Osiris to autonomously craft and adjust the model's curriculum, driving continuous improvement in reasoning abilities **without human intervention**, which is exactly the promise of Absolute Zero Reasoner [15] and the goal of our project.

Sources:

- Zhao et al., *"Absolute Zero: Reinforced Self-play Reasoning with Zero Data"*, arXiv 2505.03335 (2025). – Provided conceptual foundation for AZR and curriculum generation [2] [1] [3].
- AZR Official Repository (LeapLab, 2025) – Algorithm flow and self-play loop description [4].
- Shrikhande, *"A Deep Dive into Absolute Zero"* (ADaSci, 2025). – Summarized key principles like autonomous task proposal [6].
- Kubernetes/Cluster Resource Management Docs – Inspiration for dynamic resource allocation concepts [5].

---

[1] [2] [3] [7] [8] [9] [10] [11] [14] [15] [2505.03335] 1 Absolute Zero Reasoner (AZR) achieves state-of-the-art performance with ZERO DATA. Without relying on any gold labels or human-defined queries, Absolute Zero Reasoner trained using our proposed self-play approach demonstrates impressive general reasoning capabilities improvements in both math and coding, despite operating entirely out-of-distribution. Remarkably, AZR surpasses models trained on tens of thousands of expert-labeled in-domain examples in the combined average score across both domains.
https://ar5iv.labs.arxiv.org/html/2505.03335v2

[4] [12] GitHub - LeapLabTHU/Absolute-Zero-Reasoner: Official Repository of Absolute Zero Reasoner
https://github.com/LeapLabTHU/Absolute-Zero-Reasoner

[5] About dynamic resource allocation in GKE - Google Cloud
https://cloud.google.com/kubernetes-engine/docs/concepts/about-dynamic-resource-allocation

[6] [13] A Deep Dive into Absolute Zero: Reinforced Self-play Reasoning with Zero Data
https://adasci.org/a-deep-dive-into-absolute-zero-reinforced-self-play-reasoning-with-zero-data/