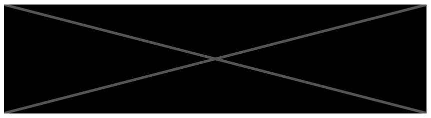

Analysis of the `deploy-avp.yml` GitHub Workflow

Overview

The **Deploy AVP (Finanzas)** workflow is a manual GitHub Actions pipeline that provisions an **Amazon Verified Permissions (AVP)** policy store and its prerequisites for the Finanzas service. It is designed to be run on demand (via **workflow_dispatch**) for a given environment (`dev`, `stg`, or `prod`) to create or update the AVP Policy Store, upload the Cedar authorization schema, and bind a Cognito identity source [1] [2]. This setup is a one-time (per environment) infrastructure provisioning step, meant to run **before** deploying the Finanzas API so that the API can integrate with the created Policy Store. The workflow is intentionally decoupled from the regular API/UI deployment triggers – it only runs when explicitly invoked – to avoid performing heavy infrastructure changes on every code push [3] [4].

In summary, the **Deploy AVP** workflow: - Uses OpenID Connect (OIDC) to assume an AWS IAM role and deploys a CloudFormation stack that creates an AVP Policy Store (with strict Cedar schema validation) [5] [6]. - Retrieves the new **Policy Store ID** from CloudFormation outputs [7] [8] and uses it to: - Upload a Cedar **schema** file to the store for policy structure validation [9]. - **Bind a Cognito user pool** as an identity source (if not already bound) so that JWT tokens from that user pool can be used for authorization decisions [10]. - Verifies that the policy store, schema, and identity source are properly in place, then outputs a Markdown **summary** with next steps (including instructing the user to add the Policy Store ID to repo variables and deploy the API) [11] [12].

This workflow is a critical preparatory step: the Finanzas API deployment workflow will look for the `POLICY_STORE_ID` variable and integrate AVP if present [13] [14]. If the AVP store hasn't been deployed or configured, the API workflow can skip or fail fast as appropriate. Below is a detailed line-by-line breakdown of `deploy-avp.yml`, identification of any issues/misuses, and recommendations for improvements and proper coupling with other pipelines.

Trigger and Inputs

The workflow is configured to run **only on manual dispatch**. The `on:` section specifies `workflow_dispatch` with an input parameter `stage` (environment stage) that must be provided by the user [1]:

- **Manual Trigger** – Because only `workflow_dispatch` is listed (no push or pull_request events), this pipeline runs *explicitly* when a user goes to the Actions tab and clicks "Run workflow." This prevents accidental execution on every code change.
- **Environment Input** – The `inputs.stage` parameter accepts one of `dev`, `stg`, or `prod` (defaulting to `dev`) [15]. This value is used to tailor resource names and configurations per environment. For example, the CloudFormation stack name is constructed with the stage suffix (`finanzas-avp-dev/stg/prod`) [16] and the Policy Store name is similarly suffixed [17]. The choice of stage does **not** automatically enforce any branch or approval conditions by itself – it's purely user-

provided, so running a `prod` deployment is possible from any branch unless additional guards are added.

Potential Improvement – Environment Guard: Given the critical nature of production changes, it may be wise to gate production runs behind GitHub Environments or branch conditions. For instance, we could require the workflow to only allow `stage: prod` when on the `main` branch or after an approval. This can be done by adding an `if` condition on the job (e.g. only allow `inputs.stage == 'prod'` if `github.ref == 'refs/heads/main'`) or by using GitHub **Environments** with required reviewers for prod deployments. Currently, no such guard exists, so caution relies on the user triggering it correctly.

Permissions and Environment Configuration

At the top level, the workflow declares required permissions and a default region:

- **Permissions**: It requests `id-token: write` and `contents: read` [18]. The `id-token: write` permission is needed for OpenID Connect authentication – the workflow uses the OIDC token to assume the AWS role specified in `OIDC_AWS_ROLE_ARN`. The `contents: read` permission allows checkout of the repository code (which is needed to access the CloudFormation template and schema file).
- **AWS Region**: Under `env:`, `AWS_REGION` is set to `us-east-2` [19]. This makes `AWS_REGION` available to all steps as the default region for AWS CLI commands. Notably, this is hard-coded in the YAML rather than referencing a variable. In the repository's Action variables, `AWS_REGION` is likely also defined (the runbook lists it as a required variable [20]), and the API deployment workflow uses `${{ vars.AWS_REGION || 'us-east-2' }}` to allow override [21]. **It would improve consistency** to define `AWS_REGION: ${{ vars.AWS_REGION || 'us-east-2' }}` here as well, so that if the region is ever changed via repo variables, the AVP workflow automatically uses the updated value. As is, it will always deploy to `us-east-2` unless manually edited.

Aside from AWS_REGION, the workflow does **not** globally export the Cognito configuration variables. Instead, it references them via the `${{ vars.VARIABLE_NAME }}` context at point of use (in the identity source step and in the summary). The required variables (Cognito User Pool ARN and App Client ID) must be configured in the repository's settings before running this workflow [22]. This is documented in the guides, but the workflow doesn't explicitly validate their presence. We address this in the breakdown and recommendations below.

Workflow Steps Breakdown

The job is named `deploy-avp` and runs on Ubuntu. It defines an **output** `policy_store_id` that captures the created Policy Store ID [23]. (This output could be used if another workflow calls this one as a reusable workflow, but currently it's mainly used internally and for the summary.) The steps proceed as follows:

1. **Checkout Repository** – Uses the standard checkout action [24] to ensure the repository code is available. This is necessary because the CloudFormation template (`avp-policy-store.yaml`) and the Cedar schema file (`avp/schema.cedar`) reside in the repo (under `services/finanzas-api`). Without checkout, subsequent file references would fail.

2. **Normalize Line Endings** – A utility step runs `dos2unix` over files in the local actions directory [25]. This converts any Windows-style line endings to UNIX format. It's likely included to prevent issues executing any custom actions or scripts that might have incorrect line endings. In this case, the workflow uses a local action `.github/actions/oidc-configure-aws`, so normalizing ensures the entrypoint script of that action (and others in `.github/actions`) has proper LF line endings for Linux. This step is non-critical (it's run with `|| true` to ignore errors) but good practice for cross-platform consistency.

3. **Configure AWS (OIDC)** – This step uses a local composite action `.github/actions/oidc-configure-aws` to assume the AWS role via OIDC [26]. It passes:

4. `role_arn: ${{ secrets.OIDC_AWS_ROLE_ARN }}` – the ARN of the IAM role in AWS that the GitHub runner will assume.
5. `aws_region: ${{ env.AWS_REGION }}` – the region (us-east-2).
6. `session_name: deploy-avp-${{ github.run_id }}` – a unique session name.

The OIDC action presumably exchanges the GitHub OIDC token for temporary AWS credentials (setting environment variables like `AWS_ACCESS_KEY_ID`, etc.). After this step, all AWS CLI commands in later steps run under the assumed role's permissions. **Note:** It's crucial that the IAM role has the necessary AVP permissions (as documented) – e.g., ability to create policy store, put schema, create identity source, etc. [27] [28]. If not, subsequent AWS CLI calls would fail. The documentation snippet confirms the OIDC role is expected to have those Verified Permissions actions allowed [28].

1. **Verify AWS Identity** – A simple `aws sts get-caller-identity` is executed [29]. This confirms that AWS credentials are in place and prints the account and ARN being used. It's essentially a sanity check after the OIDC step. If this fails or returns the wrong account, the workflow would stop here, avoiding unintended deployments.

2. **Deploy AVP Policy Store (CloudFormation)** – This is a critical step that uses AWS CLI to deploy a CloudFormation stack [30] [31]:

3. It sets `set -euo pipefail` to ensure any error causes an immediate exit [32].
4. Defines `STACK_NAME="finanzas-avp-${{ inputs.stage }}"` [33], so for example "finanzas-avp-dev."
5. Runs `aws cloudformation deploy` with the template file `avp-policy-store.yaml` [34], located in `services/finanzas-api` (the step has `working-directory: services/finanzas-api` [35], so the CLI runs in that folder).
6. Parameters passed:
   - `PolicyStoreName="FinanzasPolicyStore-${{ inputs.stage }}"` – a human-readable name for the policy store [17].
   - `ValidationMode=STRICT` – enforces Cedar policy schema validation [36].
7. The deploy command includes `--capabilities CAPABILITY_IAM` (required because the template may create IAM roles/policies) and `--no-fail-on-empty-changeset` [34]. The latter is important: if the stack already exists and there are no changes in the template, the CLI will not treat that as an error. This makes the deployment idempotent – re-running the workflow for the same stage won't fail if nothing changed, allowing safe repeated runs.

8. **CloudFormation Template**: The referenced `avp-policy-store.yaml` defines not just the Policy Store but also a set of **static policies** and **policy templates** that get created as part of the stack [37] [38]. For example, it includes AWS::VerifiedPermissions::Policy resources like *HealthPolicy*, *CatalogRulesReadPolicy*, etc., and template resources like *ProjectMemberAccessTemplate*, *FinanceWriteAccessTemplate*, etc. These define baseline authorization rules (e.g., permit certain user groups to view or modify data) [39] [40]. All those are created transactionally by CloudFormation. The stack's outputs include the **PolicyStoreId** (and some template IDs) [8]. By deploying via CF, the workflow ensures that the policy store comes pre-loaded with those policies and templates.

9. On success, this step outputs `stack_name=$STACK_NAME` to the GitHub step output [41]. If the CloudFormation deployment fails (for example, if the template is invalid or AWS permissions are missing), the step and workflow will error out here. One subtle point: deploying with `ValidationMode=STRICT` means the policy store expects a Cedar schema to be attached. **The CloudFormation static policies are created before the schema is uploaded**, which raises the question of schema enforcement. In practice, AWS Verified Permissions allows policy creation before a schema is present (the policies use the Cedar language but schema validation is effectively deferred until a schema is attached) [42]. The verification steps later ensure the schema is added and that the policies then validate correctly.

10. **Get Policy Store ID** – After stack deployment, the next step retrieves the Policy Store's unique ID from the CloudFormation outputs [43] [44]:

11. It uses the `aws cloudformation describe-stacks` command, querying for the output where `OutputKey` `==` `PolicyStoreId` [45]. The JMESPath query `'Stacks[0].Outputs[? OutputKey==\` PolicyStoreId`].OutputValue'` `returns the PolicyStoreId value` [45] `. This should yield a string like` ps-XXXXXXXX`.

12. If the query result is empty or `"None"`, the script prints an error and exits with failure [46]. This guards against a scenario where the stack deploy succeeded but somehow the output isn't there (which would be unexpected). On success, it echoes the ID and exports two things:

    - `policy_store_id=$POLICY_STORE_ID` to `$GITHUB_OUTPUT` [47], making it available as `steps.output_store_id.outputs.policy_store_id` in later steps.
    - `POLICY_STORE_ID=$POLICY_STORE_ID` to `$GITHUB_ENV` [47], which sets an environment variable for subsequent steps in this job. In practice, the workflow doesn't directly rely on the `$POLICY_STORE_ID` env var in later shell commands (each step re-fetches the ID via the step output), but setting it in the env could be useful for debugging or if any future step/script expects it. This duplication is harmless, though not strictly necessary.

13. **Upload AVP Schema** – With the Policy Store created and ID obtained, the workflow now uploads the Cedar schema definition:

14. It changes directory to `services/finanzas-api` (same as CF step) [48] to locate the schema file.
15. Exports `POLICY_STORE_ID="${{ steps.output_store_id.outputs.policy_store_id }}"` into the shell context [49] for convenience.
16. Runs `aws verifiedpermissions put-schema` with the policy store ID and the schema file path [50]. The schema file (`avp/schema.cedar`) contains the entity definitions, actions, and context types for the policies (ensuring strict validation can succeed). If this call fails (e.g., due to a syntax

error in the Cedar schema or misnamed types), the step will exit on error. A common failure here would be a `ValidationException` if the schema text is invalid; the documentation suggests checking the schema syntax if that occurs.

17. On success, it prints a confirmation " **Schema uploaded successfully"** [50] . This means the policy store now has a Cedar schema attached. **This is crucial** – with `STRICT` mode, any subsequent creation or evaluation of policies will enforce that they conform to this schema. The earlier static policies created via CloudFormation will now be associated with this schema. (If those policies had definitions incompatible with the schema, this call might fail or those policies would be non-functional; however, since the same team wrote both the policies and schema, they should align.)

18. **Bind Cognito Identity Source** – Next, the workflow integrates Amazon Cognito as an identity provider for the policy store [51] [52] :

19. No specific working directory is needed here (it runs in the default workspace root).
20. It pulls in the `POLICY_STORE_ID` from the previous step's output again into the shell environment [53] .
21. It echoes a message about creating the Cognito identity source and then proceeds to check if one already exists [54] .
22. **Idempotence Check:** The script lists existing identity sources with `aws verifiedpermissions list-identity-sources` for the given store, and uses a JMESPath query to filter those whose `configuration.cognitoUserPoolConfiguration.userPoolArn` matches the Cognito User Pool ARN configured in our repo variables [10] . The query string is:

```
--query 'identitySources[?
configuration.cognitoUserPoolConfiguration.userPoolArn==`$
{{ vars.COGNITO_USER_POOL_ARN }}`]'
```

This is wrapped in single quotes in the command, using backticks to enclose the ARN value for an exact match [52] . This clever quoting ensures the ARN is treated as a literal in the query (backticks in JMESPath denote a literal value). The result (`EXISTING_SOURCES`) will be a JSON array of any identity sources already using that Cognito user pool (likely at most one).
23. The script then uses `jq 'length'` to count how many such sources were found [55] . If the count is zero, it proceeds to create a new identity source; if not zero, it skips creation.
24. **Creation:** To create the identity source, it calls `aws verifiedpermissions create-identity-source` with:
    - `--policy-store-id "$POLICY_STORE_ID"`.
    - `--configuration "cognitoUserPoolConfiguration={userPoolArn=${{ vars.COGNITO_USER_POOL_ARN }},clientIds=[${{ vars.COGNITO_WEB_CLIENT }}]}"` [56] . This in-line JSON sets the Cognito User Pool ARN and an array of allowed client IDs (one of which should match the Cognito app client used by the frontend). It also specifies:
    - `--principal-entity-type "Finanzas::User"` [56] – meaning that Cognito identities will be mapped to the Cedar entity type `Finanzas::User` in policies.
    - The region (`--region $AWS_REGION`) is included as well [56] .

25. If creation succeeds, it prints **" Cognito identity source created"** [57] . If it finds an existing source, it prints an info message that it's skipping creation [58] . Notably, this step uses the `${{ vars.COGNITO_USER_POOL_ARN }}` and `${{ vars.COGNITO_WEB_CLIENT }}` **directly inside the run command**. GitHub Actions will substitute those with the actual values from repo variables at runtime. If those variables were not set, this command would end up malformed (e.g., an empty ARN or client ID) and likely fail. **Potential Issue:** The workflow does not verify that `COGNITO_USER_POOL_ARN` and `COGNITO_WEB_CLIENT` are non-empty before this step. If someone forgets to set them, the `list-identity-sources` call might return all sources (if ARN filter is empty) or fail, and the `create-identity-source` call would definitely fail (empty ARN). This would cause the job to exit with an error. To improve resiliency, a **pre-flight check for required variables** at the start of the workflow is recommended (similar to what the API deploy does for its variables [59]). We could add a step to echo and fail if those specific `vars.` are not provided, giving a clear error early (e.g., " COGNITO_USER_POOL_ARN not set").

26. **Verify Policy Store** – This step performs a comprehensive verification of the deployed resources [60] [61] [62] :

27. It again sets `POLICY_STORE_ID` from outputs in the shell [63] .
28. Retrieves and prints basic store info with `aws verifiedpermissions get-policy-store` (grabbing ARN, validation mode, and creation date) [64] . This confirms the store is accessible and shows if `ValidationMode` is indeed STRICT as expected.
29. **Verify Schema**: It calls `aws verifiedpermissions get-schema` for the store and captures the output [65] . If the returned schema JSON is non-empty and not `"None"`, it prints " Cedar schema attached"; otherwise, prints an error and exits [66] . This double-checks that our `put-schema` step succeeded and that the store has a schema. Failing here would indicate the schema is missing (which would be a serious issue given STRICT mode – policies wouldn't be usable). On a normal run, this should pass (since we just uploaded the schema).
30. **Verify Identity Sources**: It lists all identity sources (`list-identity-sources` without filters) and counts them using `jq` [67] . If at least one identity source exists, it prints " Cognito identity source bound (N source(s))" and even lists each source ID and its user pool ARN [68] . If none are found, it prints " No identity sources found" and exits with failure [69] . This ensures our Cognito source was created (or was pre-existing). In practice, if the create step was skipped due to an existing source, this will still find that source and pass. If the create step failed unexpectedly, this would catch it and stop the workflow.
31. **Count Policies**: It then prints the count of static policies by calling `aws verifiedpermissions list-policies` with a `--query 'length(policies)'` [70] . This doesn't enforce a pass/fail criterion; it's informational. We expect this number to reflect the static policies created via CloudFormation (e.g., ~7 or 8 policies). The output "Total policies in store: X" [71] appears to be captured (though line 182 in our view is split; the script likely echoes the count).
32. **Count Policy Templates**: Similarly, it counts policy templates with `list-policy-templates` [72] . This too is informational ("Total policy templates in store: Y"). No threshold check here – even 0 would just be reported (but we expect some templates from CF, e.g., 3 templates).

33. This verification step is thorough and will cause the workflow to **fail** if either the schema or the identity source is not in place. That failure would prevent a false "success" when the setup is incomplete. It does not fail on missing policies/templates, since those are optional (but in our case

CF created them, so the counts will be non-zero). Everything through this step ensures the AVP store is fully operational: policy store exists, schema attached, identity provider linked, and policies loaded.

34. **Summary** – The final step writes a detailed summary to the GitHub Actions **job summary** (a rich markdown output visible on the run page) [73] [11]:
    - It uses the special `$GITHUB_STEP_SUMMARY` file to append markdown content [74].
    - The summary includes a title " AVP Policy Store Deployed Successfully" and tables listing the **Deployment Status** of each component [75] and the **Deployment Details** (region, stack name, stage, store ID, Cognito pool and client) [76].
    - The status table reflects whether each component was created/attached, using the results from verification:
    - Policy Store: always " Created" (if we got this far).
    - Cedar Schema: either " Schema attached" or " Not verified" depending on a check very similar to the earlier one (here they do a quick `get-schema` and set `SCHEMA_STATUS` accordingly) [77].
    - Cognito Identity Source: either " Cognito identity source bound (N source(s))" or " Not configured", based on counting identity sources (with a safe default of 0 if the CLI call fails) [78].
    - Static Policies and Policy Templates: they are marked with   along with the counts (`$POLICY_COUNT policies`, `$TEMPLATE_COUNT templates`) [79]. These counts are gathered with CLI calls similar to above, using `|| echo "0"` to avoid failures [80] [81]. Even if those were 0, they still display " 0 policies" (since having zero policies might be acceptable in a fresh store scenario, it's not a breaking error for the workflow).
    - Next, a **Next Steps** section is included with enumerated instructions [82]:
    - **Update GitHub Variables** – It tells the user to add the newly created `POLICY_STORE_ID` as a repository variable [83]. (This is crucial for coupling with the API deployment. The summary even provides the exact line to add.) If multiple environments are used, they might use environment-specific variables like `POLICY_STORE_ID_DEV`, but the example uses a single `POLICY_STORE_ID`. The AVP deployment guide also emphasizes adding this variable after running the workflow [84] [85].
    - **Deploy API** – Instructs running the "Deploy Finanzas API" workflow next [86]. It notes that the API will automatically pick up the `POLICY_STORE_ID` variable and enable AVP authorization on protected endpoints. (We'll discuss coupling with the API in a moment.)
    - **Test Authorization** – Recommends testing the API with Cognito JWTs and observing that different user groups have appropriate access, and checking Lambda logs for AVP authorization decisions [87].
    - Finally, the summary provides a block of **verification commands** (AWS CLI commands to manually inspect the policy store, schema, identity sources, policies, etc.) [88] [89] and concludes with a note that the Policy Store is ready for integration, listing what has been configured (schema, identity source, static policies, templates) [90].
    - This summary is extremely useful for developers and auditors, as it documents the outcome and next steps in one place. It's formatted in Markdown for readability. All variables in the summary (IDs, counts, etc.) are pulled from the workflow context (ensuring accuracy).

One minor detail: the summary suggests there are " Static Policies" and lists a number. In our CloudFormation, we created 7 static policies (numbered 1,2,5,7a,7b,7c,8 in comments) [91] [92]. The documentation examples mention “8 static policies” [93] – possibly counting differently or including an additional policy. In any case, the workflow doesn’t hardcode the number; it counts whatever is actually present. So if the template changes, the summary stays accurate.

With the above steps, the **Deploy AVP workflow completes**. If all went well, it ends in success, having created/updated the AVP store for the specified stage. If any critical part failed (store creation, schema, identity source), the workflow fails and stops, preventing partial configuration from being used unknowingly. Because this job is standalone and manually triggered, a failure here does not automatically rollback any API or UI deployment – it only affects the AVP stack. CloudFormation itself will roll back the stack on failure (meaning if, say, a static policy creation failed because of schema strictness, the PolicyStore resource would be deleted as part of the rollback) [46] [42], leaving no half-configured store around. Thus, a failure requires the developer to fix the issue (or set the correct variables) and re-run this workflow, but it does not directly impact other workflows unless they proceed without AVP.

Coupling with API/UI Deployments

It’s important to understand how this workflow ties into the broader deployment process: - The Finanzas **API deployment workflow** (`deploy-api.yml`) is set up to optionally use the results of this AVP deployment. It checks for a repository variable `POLICY_STORE_ID` and, if present, verifies the policy store exists before deploying the Lambdas with AVP enabled [94] [95]. Specifically, the API workflow will fail early if a `POLICY_STORE_ID` is set but not valid (telling you to run Deploy AVP first) [96]. If no `POLICY_STORE_ID` is set, it logs that AVP will be skipped [97], and proceeds to deploy the API without AVP integration (the Lambdas will not enforce fine-grained policies in that case). This design means: - You **must** run Deploy AVP (and set the variable) before deploying the API **if you want AVP authorization active**. - If you deploy the API without doing so, it will just skip AVP (no authorization checks beyond the base Cognito JWT authorizer). - If you set `POLICY_STORE_ID` but forgot to deploy AVP, the API pipeline will catch it and halt, preventing a broken configuration. - The **UI deployment** (if any; likely uploading static frontend files or updating CloudFront) is largely decoupled from AVP. The UI might only need the CloudFront distribution updated to route API calls (as described in the CloudFront/AVP unified deployment guide) [98] [99]. The AVP workflow’s success or failure doesn’t directly affect UI artifact deployment. However, in a unified rollout (where a CloudFront distribution is configured to use the new API and AVP), the presence of AVP could indirectly matter (for example, the CloudFront cache behaviors forward the Authorization header expecting the API to handle it, and the API expects AVP store if configured). The project documentation suggests a coordinated sequence: update CloudFront, deploy AVP, then deploy API [100] [101]. There is no evidence of the UI workflow triggering the AVP workflow automatically, and given the manual nature, such coupling is avoided. Each step is run intentionally by the team.

Is Deploy AVP triggered incorrectly by other workflows? Currently, **no** – it is only run manually. We found no `workflow_run` triggers or `needs:` references in other workflows that call `deploy-avp.yml`. The coupling is **procedural** (through the `POLICY_STORE_ID` variable and documentation) rather than automated triggers. This is deliberate to keep the AVP provisioning separate from CI/CD of code. There was a consideration to unify deployments (the runbook calls it a unified deployment scenario), but even then, the instructions explicitly have the operator run the AVP workflow as a step in the sequence [100]. Therefore, there’s no risk of it running at unintended times, aside from user error (e.g., running it with the wrong stage or before CloudFront is ready).

Effect on API/UI deployments if AVP fails: If the `Deploy AVP` workflow fails or is not run: - The **API deployment** can still be run, but it will likely skip AVP integration (assuming no POLICY_STORE_ID is set). If `POLICY_STORE_ID` was already added (perhaps from a previous run) and the AVP store is now in a bad state or deleted, the API workflow will detect "store not found" and fail [96]. In such a case, the API deployment would roll back (no changes applied), which is a safe failure – it prevents deploying an API configured for a nonexistent store. The API and UI CloudFormation stacks themselves remain unaffected by an AVP workflow failure because they are deployed independently. There's no direct rollback link between them – only the logic in the API pipeline that optionally exits if AVP isn't ready. - The **UI deployment** (for static files or CloudFront config) doesn't depend on AVP at all, so it would not roll back if AVP fails. The only interplay is that if the API ends up without AVP, some UI features that rely on fine-grained authorization might behave differently (e.g., all authorized users might see all data if no AVP policies enforced). But that is an application logic outcome, not a pipeline failure. From a CI perspective, UI deploy is decoupled.

In short, **Deploy AVP is kept standalone** to isolate the complexity of provisioning security infrastructure. Its success enables the API to tighten authorization; its failure or absence means the API will deploy with coarse-grained auth (Cognito only) but clearly logs/skips the fine-grained step. This is a reasonable fallback strategy for development environments, but likely in production they will always run AVP first.

Identified Issues and YAML Flaws

Overall, the YAML is well-structured and logically correct. We did not find serious syntax errors or outright flaws, but we did identify a few **areas of concern or improvement**:

- **Missing Pre-Checks for Variables**: As noted, the workflow uses `${{ vars.COGNITO_USER_POOL_ARN }}` and `${{ vars.COGNITO_WEB_CLIENT }}` directly in a command [52] without verifying they are set. If these repository variables are not configured, the command will be malformed (e.g., `userPoolArn=,clientIds=[]`) and will fail. The failure might not clearly indicate "variable not set" to the user. A proactive solution is to add a step at the beginning to ensure these critical variables are present. For example:

```
- name: Preflight check variables
  run: |
    missing=0
    for var in COGNITO_USER_POOL_ARN COGNITO_WEB_CLIENT; do
      if [[ -z "${{ vars[$var] || '' }}" ]]; then
        echo "  Required variable $var is not set";
        missing=1;
      fi
    done
    if [[ $missing -eq 1 ]]; then exit 1; fi
    echo "  Cognito variables are set."
```

This would fail fast with a clear message if any required variable is empty. (In GitHub Actions, an alternative is to use `${{ vars.VAR }}` in an if condition, but since we want to check at runtime and possibly list multiple missing, a shell loop works.)

- **Inconsistent Region Variable Usage**: The workflow hardcodes `AWS_REGION: us-east-2` [19], whereas the API workflow draws it from `${{ vars.AWS_REGION }}` with a default [102]. If the repository has an `AWS_REGION` variable set (which it likely does, per docs [103]), the Deploy AVP workflow currently *ignores* it. This isn't a functional bug (it will always deploy to us-east-2 as intended for this project), but it's an inconsistency. A simple fix is:

  ```
  env:
    AWS_REGION: ${{ vars.AWS_REGION || 'us-east-2' }}
  ```

  This way, if the team ever needs to deploy to a different region (or clone the workflow for another project), they only need to change the variable in one place. It also aligns with the documented setup where AWS_REGION is a configurable variable.

- **Use of `$GITHUB_ENV` vs Step Output**: The workflow sets the `POLICY_STORE_ID` to both a step output and the job environment [47]. Using the step output is very handy for subsequent steps (as we see throughout). Setting `$GITHUB_ENV` is redundant in this single-job scenario – none of the later steps actually use `$POLICY_STORE_ID` from the environment; they either recompute it or use the output. This doesn't cause any issue, but it's extra work. If we were to streamline, we could remove the `echo "POLICY_STORE_ID=$POLICY_STORE_ID" >> $GITHUB_ENV` line, or conversely, use the env var directly in later steps to avoid repeating the substitution. For instance, in **Upload AVP Schema**, instead of

  ```
  POLICY_STORE_ID="${{ steps.output_store_id.outputs.policy_store_id }}"
  aws verifiedpermissions put-schema --policy-store-id "$POLICY_STORE_ID" ...
  ```

  we could rely on the env (if exported) and do:

  ```
  aws verifiedpermissions put-schema --policy-store-id "$POLICY_STORE_ID" ...
  ```

  having set `POLICY_STORE_ID` in the job env. However, clarity is also important – explicitly pulling from the output makes it obvious which value is being used. So this is a minor note. It doesn't *misuse* the environment, but it's a place to simplify. No user-facing bug here, just an internal clean-up.

- **Quoting and Path Issues**: We examined the YAML quoting in tricky areas (like the JMESPath queries with backticks) and path references:

- The **JMESPath query strings** using backticks are correctly enclosed in single quotes, which avoids YAML parsing issues. For example, '...?OutputKey==\ PolicyStoreId`].OutputValue' in the CF output query [45] and the identity source filter '...?userPoolArn==`${{ vars.COGNITO_USER_POOL_ARN }}`]' [52]. These are a bit hard to read but functionally correct. We confirm that in YAML, the backtick is treated as a literal character inside the single-quoted string (not as closing the quote).

`The GitHub Actions expression`${{ vars.COGNITO_USER_POOL_ARN }}` will be substituted before the script runs. One must ensure that the ARN value has no stray backtick characters, which it wouldn't. An alternative approach could be to use double quotes around the query and single quotes around the ARN value, but that would require careful escaping in YAML. The chosen method is fine and likely tested. No change needed unless this proves error-prone in practice.

- **File paths**: The use of `working-directory: services/finanzas-api` is correct for steps that need to access files in that service module (the CF template and schema). We verified that `services/finanzas-api/avp-policy-store.yaml` exists and includes the expected resources [104], and `services/finanzas-api/avp/schema.cedar` exists (implied by usage; also the docs mention its content). The paths are referenced relative to the working directory, so `avp-policy-store.yaml` and `avp/schema.cedar` are found. There's no apparent path error. One improvement could be to parameterize the file paths (if, say, the location might change), but given the repository structure, it's stable.

- All Bash scripts use `set -euo pipefail`, which is good practice to catch unset variables (`-u`) and pipeline failures. We should ensure that any variable used in those scripts is defined. In these scripts, `$AWS_REGION` is defined in the env, `$POLICY_STORE_ID` is always set right before use, and `$STACK_NAME` is set. So they won't trip `-u`. The use of `${{ vars.COGNITO_USER_POOL_ARN }}` inside the script doesn't create an undefined shell var; it's substituted to a value or empty string *before* the script runs (if it were empty, it would still be a quoted empty string in the command, which wouldn't trigger `-u`, but would cause AWS CLI error).

- **Conditional Logic**: The Bash conditional logic in the steps appears sound:

- Checking for empty `$POLICY_STORE_ID` after CF deploy [46] ensures we don't proceed with a blank ID.
- Identity source creation check uses `jq length` and numeric comparison correctly [55].
- The verification step smartly handles command success/failure. Notably, they use constructs like:

```
if aws verifiedpermissions get-schema ... >/dev/null 2>&1; then
  SCHEMA_STATUS="  Schema attached"
fi
```

under `set -e`. In bash, a command in an `if` conditional will not cause the script to exit on failure (because the exit code is captured by `if`). This means the script continues even if `get-schema` fails, which is intended here (we want to set a status instead of exiting). Similarly, they append `|| echo "0"` when capturing counts to avoid `set -e` killing the script if the AWS call fails [80] [81]. Instead, it will return "0" and continue. This is thoughtful, ensuring the summary still gets produced even if the list commands fail for some reason (maybe a transient permission issue or eventual consistency delay). All these indicate the conditional logic is carefully done.

- We did not find mistakes like using the wrong context (e.g., `$vars` inside a run – which would be wrong – they correctly use `${{ vars.X }}` for GitHub context vs `$VAR` for shell env). One thing to double-check: in the preflight example we gave above, `${{ vars[$var] }}` is pseudo-code; one cannot use GitHub expressions inside the run that way. Instead, one would likely just check the environment variables if we export them, or do separate checks. The key point is to ensure those values exist via some means.

- **Outputs and Reuse**: The job defines an output `policy_store_id` [23]. However, since this workflow isn't configured as a reusable workflow (no `on: workflow_call`), and it doesn't have multiple jobs depending on each other, this output currently isn't used elsewhere. It doesn't hurt anything – in fact, the summary uses the step output directly (which is available within the job without needing the job output). If there was an intention to call this workflow from another (using the `workflow_call` trigger), then keeping the output is useful. If not, it's extraneous. This is a very minor note. It could be left as is for future flexibility or removed if cleaning up.

In conclusion, **no critical YAML flaws** (like syntax or indentation errors) were found. The issues are mostly about hardening and consistency: adding upfront variable checks, using variables uniformly, and minor simplifications. We'll incorporate these into the recommendations next.

Recommendations for Improvement and Resiliency

1. **Add Preflight Variable Validation**: Implement a step to verify that required GitHub Action **Variables** and **Secrets** are present before making AWS calls. In particular, check that `COGNITO_USER_POOL_ARN` and `COGNITO_WEB_CLIENT` (and potentially `OIDC_AWS_ROLE_ARN` secret) are provided. This will turn a cryptic AWS CLI failure into a clear error message. Example:

```
- name: Verify required inputs
  run: |
    for var in COGNITO_USER_POOL_ARN COGNITO_WEB_CLIENT; do
      if [ -z "${{ vars[$var] || '' }}" ]; then
        echo "  Missing repository variable: $var"
        exit 1
      fi
    done
    echo "  All required variables are set."
```

   (Using shell to iterate isn't the only way; one could also leverage `${{ vars.VAR }}` *in an if expression at the job level to fail fast. The above is illustrative.)* This guard ensures the workflow fails early with a message like "Missing repository variable: COGNITO_USER_POOL_ARN" if those aren't set, prompting the user to fix the configuration as documented [22]. The **Deploy API** workflow has a similar check for critical env vars [59] (though it notably misses verifying `COGNITO_WEB_CLIENT`, which could likewise be added there for completeness).

2. **Use Repository Variables for Region (and others)**: Align the workflow to use GitHub variables where available, for maintainability. Specifically, set `AWS_REGION: ${{ vars.AWS_REGION || 'us-east-2' }}` at the top [102] so that the region is not hardcoded. Likewise, consider loading the Cognito ARN and Client ID into the job environment for easier reference:

```
env:
  AWS_REGION: ${{ vars.AWS_REGION || 'us-east-2' }}
  COGNITO_USER_POOL_ARN: ${{ vars.COGNITO_USER_POOL_ARN }}
  COGNITO_WEB_CLIENT: ${{ vars.COGNITO_WEB_CLIENT }}
```

Then in the run steps, you could use `$COGNITO_USER_POOL_ARN` and `$COGNITO_WEB_CLIENT`. This doesn't change functionality but makes the YAML cleaner and ensures all required inputs are declared in one place. It also means the *Preflight check* can simply check `$COGNITO_USER_POOL_ARN` env var instead of the `${{ vars }}` context, since those will be populated or empty strings.

3. **Safety Gating for Production**: Introduce measures to prevent accidental execution in the wrong context, especially for `prod`. Two approaches:

4. **GitHub Environments:** Define environments like "Production" (and maybe "Staging") in the repository settings, with required approvers. In the workflow YAML, add:

```
jobs:
  deploy-avp:
    environment:
${{ inputs.stage == 'prod' && 'Production' || inputs.stage == 'stg' &&
'Staging' || 'Development' }}
```

This will assign the job to an environment based on the stage. If `stage: prod`, it uses the Production environment – GitHub will then require whatever approval is configured (or at least record the deployment). This also allows using environment-scoped variables (so you could have different `POLICY_STORE_ID` vars per environment if needed in the future, or different OIDC roles, etc.). For dev, it uses 'Development' environment in this example (assuming no special protection).

5. **Conditional Checks in YAML:** Alternatively, use an `if` on the job or early step. E.g., to ensure prod deployments only run on main:

```
if: ${{ inputs.stage != 'prod' || github.ref == 'refs/heads/main' }}
```

This condition means "if stage is prod, then require the source branch is main, otherwise allow." If someone tries to manually run prod from a feature branch, the job would be skipped or fail with a message. We could also incorporate actor checks or other controls as needed. Either of these would help prevent misuse. Given this is a manual workflow, the risk is lower, but adding a safety net for production is good practice. This addresses the **environment coupling** aspect – making sure prod infrastructure is only deployed when appropriate.

6. **Implement Concurrency Control** (optional): To avoid overlapping runs that could interfere, use the `concurrency` keyword. For example:

```
concurrency:
  group: deploy-avp-${{ inputs.stage }}
  cancel-in-progress: false
```

This would ensure that if someone triggers two Deploy AVP runs for the same stage (say, dev) concurrently, they won't conflict with each other – only one will run at a time. Given that CloudFormation and the AWS CLI calls could conflict if run in parallel on the same resources, this is a safe guard. However, running the same stage twice simultaneously is unlikely unless triggered by mistake. This concurrency grouping by stage prevents chaos (one could also set a single group for all to ensure only one AVP deployment at a time across all envs, if resource contention in AWS is a concern). This is a minor insurance policy.

7. **AWS Resource Tagging**: To improve operational clarity, consider adding **Tags** to the CloudFormation stack and/or the AVP Policy Store resource. Tags can include the environment, project name, and an owner. For example, in the CF template `FinanzasPolicyStore` resource, we could add:

```
Properties:
  ValidationSettings: ...
  Tags:
    - Key: Environment
      Value: !Ref PolicyStoreName        # or use StageName if passed
    - Key: Project
      Value: FinanzasSD
```

Or when calling `aws cloudformation deploy`, use the `--tags` option:

```
aws cloudformation deploy ... \
    --tags Project=FinanzasSD Environment=${{ inputs.stage }}
```

These tags would propagate to the PolicyStore and related resources. Tagging isn't about preventing failures but aids in monitoring and cost tracking. It can also be useful if multiple environments share an AWS account – you can quickly filter resources by tag to see, say, all dev vs prod policy stores.

8. **Keep Workflow Decoupled (Final Decision)**: We recommend **keeping this AVP deployment as a standalone workflow**, rather than trying to fully integrate it into the API or UI pipelines. The current design – manual trigger and explicit variable handoff – is intentional and provides control over when infrastructure changes occur [105] [106]. Integrating it (for example, automatically running AVP deployment as a step inside API deployment if needed) might seem convenient but could introduce complexity and delay to every API deployment:

9. The AVP stack only needs to be created once per environment (or updated when policies change). Running it on every code deploy would be overkill and potentially risky.

10. By keeping it separate, you ensure that infrastructure (policy store and policies) is set up first, and application deployment can be done multiple times thereafter using the established store. This separation aligns with infrastructure-as-code best practices.
11. The only coupling needed is the `POLICY_STORE_ID` variable, which is a simple and clear interface between the workflows. The documentation and summary explicitly instruct updating that variable after running AVP deployment [12], and the API workflow reads it to toggle AVP features [13]. This is a reasonable handoff point. It also means you can choose to not set the variable in a dev/test scenario to deploy the API without AVP (effectively bypassing fine-grained auth) – useful for debugging.
12. If in the future, the team wants to streamline the process, one could create a meta-workflow or a dispatch that runs AVP then API then maybe UI in sequence. This could be done with one button click (through repository dispatch events or a script). However, automating that sequence in YAML might complicate error handling (e.g., if AVP fails, do we automatically stop API deploy? Currently yes due to variable check). Given that the checks are in place, a human-driven sequential run (possibly following a checklist like the runbook [107]) is safer.

In conclusion, **the workflow should remain standalone but hardened**. By adding the guards and improvements above, we enhance its reliability and prevent it from running under wrong conditions or with missing config. The workflow's logic itself (AVP creation, schema upload, etc.) is sound and effectively establishes a ready-to-use Policy Store as evidenced by the final summary and the verification steps. Keeping it decoupled ensures that a failure in AVP setup doesn't directly take down application deployments – instead, it's handled as a prerequisite that can be retried or fixed independently. This separation, combined with clear documentation (which is provided in the repo guides [106] [108]), is a robust approach for introducing fine-grained authorization into the deployment pipeline.

---

[1] [2] [5] [7] [9] [10] [11] [12] [15] [16] [17] [18] [19] [23] [24] [25] [26] [29] [30] [31] [32] [33] [34] [35] [36] [41] [42] [43] [44] [45] [46] [47] [48] [49] [50] [51] [52] [53] [54] [55] [56] [57] [58] [60] [61] [62] [63] [64] [65] [66] [67] [68] [69] [70] [71] [72] [73] [74] [75] [76] [77] [78] [79] [80] [81] [82] [83] [86] [87] [88] [89] [90] deploy-avp.yml
https://github.com/valencia94/financial-planning-u/blob/2800f5e56c90aafc2c8c2885e1d50df13878a0ae/.github/workflows/deploy-avp.yml

[3] [4] [84] [85] [105] [106] AVP_DEPLOYMENT_GUIDE.md
https://github.com/valencia94/financial-planning-u/blob/2800f5e56c90aafc2c8c2885e1d50df13878a0ae/services/finanzas-api/AVP_DEPLOYMENT_GUIDE.md

[6] [8] [37] [38] [39] [40] [91] [92] [104] avp-policy-store.yaml
https://github.com/valencia94/financial-planning-u/blob/2800f5e56c90aafc2c8c2885e1d50df13878a0ae/services/finanzas-api/avp-policy-store.yaml

[13] [14] [21] [59] [94] [95] [96] [97] [102] deploy-api.yml
https://github.com/valencia94/financial-planning-u/blob/2800f5e56c90aafc2c8c2885e1d50df13878a0ae/.github/workflows/deploy-api.yml

[20] [98] [99] [100] [101] [103] [107] cloudfront-avp-unified-deployment.md
https://github.com/valencia94/financial-planning-u/blob/2800f5e56c90aafc2c8c2885e1d50df13878a0ae/docs/runbooks/cloudfront-avp-unified-deployment.md

[22] [27] [28] [93] [108] AVP_DEPLOYMENT_AUTOMATION_GUIDE.md
https://github.com/valencia94/financial-planning-u/blob/2800f5e56c90aafc2c8c2885e1d50df13878a0ae/AVP_DEPLOYMENT_AUTOMATION_GUIDE.md