ARCHGUARD
AWS Well-Architected Review

synthr-api - production

Date	4 May 2026
Prepared for	rost@rostcamp.com

High Risk Overall

15	Security
42	Reliability
92	Cost Optimization
92	Performance Efficiency
69	Operational Excellence
97	Sustainability

15 total findings 3 Critical 6 High 5 Medium 1 Low

SECTION 01

Executive Summary

The synthr-api production workload has pervasive critical security and reliability gaps - including a publicly accessible database, plaintext secrets in environment variables, single-AZ architecture with no backups, and absent observability - that make it unsuitable for production use in its current state.

What Is Working Well

+ The processed S3 bucket (synthr-processed-production) has a correctly configured aws_s3_bucket_public_access_block resource with all four block settings enabled, demonstrating awareness of S3 public-access controls.
+ CloudWatch log groups for both Lambda functions are explicitly declared with a 90-day retention policy, ensuring logs are retained long enough for incident investigation without accumulating indefinitely.
+ The ECS task definition uses awslogs log driver with a dedicated log group, providing structured container-level logging for the nightly reporting workload.

SECTION 02

Pillar Scorecard

15

Security

High Risk

42

Reliability

High Risk

92

Cost Optimization

Low Risk

92

Performance Efficiency

Low Risk

69

Operational Excellence

Medium Risk

97

Sustainability

Low Risk

SECTION 03

Detailed Findings

Security

[CRITICAL] RDS instance publicly accessible with security group open to 0.0.0.0/0

Security | Effort: Medium | Phase: Immediate

FINDING

aws_db_instance.main has publicly_accessible = true and aws_security_group.rds allows inbound TCP 5432 from 0.0.0.0/0, making the production PostgreSQL database directly reachable from the entire internet.

WHY IT MATTERS

Any attacker who obtains or brute-forces the database password - which is also exposed as a plaintext Lambda environment variable - can connect directly to the synthr production database, exfiltrate all customer data, or destroy it entirely. This is a single-step path to a catastrophic breach.

BLAST RADIUS

Full compromise of all data stored in the synthr PostgreSQL database, including any PII or business-sensitive records. Because the same db_password variable is reused across Lambda and ECS environments, a credential leak from any one surface exposes the database directly.

RECOMMENDATION

1. Set publicly_accessible = false on aws_db_instance.main immediately.
2. Replace the RDS security group ingress rule with one that references only aws_security_group.lambda as the source security group.
3. Move the RDS instance and Lambda functions into private subnets with a NAT gateway for outbound traffic.
4. Rotate the database password and store it in AWS Secrets Manager, removing it from all environment variables.

SECURE CONFIGURATION EXAMPLE

```
resource "aws_security_group" "rds" {
  name   = "synthr-rds-sg"
  vpc_id = aws_vpc.main.id

  ingress {
    from_port       = 5432
    to_port         = 5432
    protocol        = "tcp"
    security_groups = [aws_security_group.lambda.id]
  }
}

resource "aws_db_instance" "main" {
  # ... other config ...
  publicly_accessible    = false
  db_subnet_group_name   = aws_db_subnet_group.main.name
  vpc_security_group_ids = [aws_security_group.rds.id]
}
```

CIS AWS Foundations v3.0 §5.6	NIST 800-53 SC-7	PCI DSS Requirement 1.3

[CRITICAL] Database password and API key stored as plaintext Lambda and ECS environment variables

Security | Effort: Medium | Phase: Immediate

FINDING

aws_lambda_function.api_handler, aws_lambda_function.worker, and aws_ecs_task_definition.reporter all inject var.db_password and var.api_key_secret directly into container/function environment variables. These values appear in the Terraform state file, CloudWatch Logs, and are readable by anyone with lambda:GetFunctionConfiguration or ecs:DescribeTaskDefinition IAM permissions.

WHY IT MATTERS

Secrets stored as environment variables are visible in the AWS Console, Terraform state (which may be stored in S3 or a remote backend without encryption), and any logging that captures environment dumps. A single over-permissioned IAM user or a compromised CI/CD pipeline can harvest all credentials silently.

BLAST RADIUS

Compromise of the database password grants direct database access (compounded by SEC-001). Compromise of the API key secret may allow impersonation of the synthr API to downstream integrations or customers. Both secrets are shared across Lambda and ECS, so a single exposure affects all compute surfaces.

RECOMMENDATION

1. Store db_password and api_key_secret in AWS Secrets Manager as separate secrets.
2. Grant the Lambda execution role and ECS task execution role secretsmanager:GetSecretValue on only those specific secret ARNs.
3. Remove the plaintext environment variable references and instead reference the Secrets Manager ARN in the ECS task definition secrets block and retrieve the value at Lambda startup via the AWS SDK.
4. Rotate both secrets immediately after migration.

SECURE CONFIGURATION EXAMPLE

```
resource "aws_ecs_task_definition" "reporter" {
  # ... other config ...
  container_definitions = jsonencode([{
    name  = "reporter"
    image = "123456789.dkr.ecr.eu-west-1.amazonaws.com/synthr-reporter:sha256:abc1...
    secrets = [
      { name = "DB_PASSWORD", valueFrom = aws_secretsmanager_secret.db_password.ar...
    ]
  }])
}

resource "aws_lambda_function" "api_handler" {
  # ... other config ...
  environment {
    variables = {
      DB_PASSWORD_SECRET_ARN = aws_secretsmanager_secret.db_password.arn
    }
  }
}
```

NIST 800-53 IA-5	CIS AWS Foundations v3.0 §1.19	SOC 2 CC6.1

[HIGH] **Lambda IAM role grants wildcard s3:*, sqs:*, and dynamodb:* across all resources**

Security | Effort: Medium | Phase: Immediate

FINDING

aws_iam_role_policy.lambda_policy grants s3:*, sqs:*, and dynamodb:* with Resource = "*" to the shared Lambda execution role used by both api_handler and worker functions. This violates least-privilege and means either function can read, write, or delete any S3 bucket, SQS queue, or DynamoDB table in the account.

WHY IT MATTERS

If either Lambda function is compromised via a dependency vulnerability or injection attack, the attacker inherits full S3, SQS, and DynamoDB access across the entire AWS account - not just the synthr workload. This could enable exfiltration of data from unrelated buckets or destruction of other workloads sharing the account.

BLAST RADIUS

All S3 buckets, SQS queues, and DynamoDB tables in the AWS account are readable and writable by a compromised Lambda. The shared role between api_handler and worker means a vulnerability in the lightweight API handler grants the same permissions as the privileged document worker.

RECOMMENDATION

1. Create separate IAM roles for api_handler and worker with distinct permission sets.
2. Scope S3 actions to only the required operations (e.g., s3:PutObject on synthr-uploads-production ARN) and restrict to specific bucket ARNs.
3. Scope SQS actions to sqs:SendMessage on the jobs queue ARN only for api_handler, and sqs:ReceiveMessage, sqs:DeleteMessage for worker.
4. Scope DynamoDB actions to specific operations (GetItem, PutItem, DeleteItem) on the synthr-sessions table ARN only.

SECURE CONFIGURATION EXAMPLE

```
resource "aws_iam_role_policy" "api_handler_policy" {
  name = "synthr-api-handler-policy"
  role = aws_iam_role.lambda_api_exec.id

  policy = jsonencode({
    Version = "2012-10-17"
    Statement = [
      {
        Effect   = "Allow"
        Action   = ["s3:PutObject"]
        Resource = "${aws_s3_bucket.uploads.arn}/*"
      },
      {
        Effect   = "Allow"
        Action   = ["sqs:SendMessage"]
        Resource = aws_sqs_queue.jobs.arn
      },
      {
        Effect   = "Allow"
        Action   = ["dynamodb:GetItem", "dynamodb:PutItem", "dynamodb:DeleteItem"]
        Resource = aws_dynamodb_table.sessions.arn
      }
    ]
  })
}
```

CIS AWS Foundations v3.0 §1.16	NIST 800-53 AC-6	SOC 2 CC6.3

[HIGH] uploads S3 bucket has no public access block, versioning, or server-side encryption

Security | Effort: Low | Phase: Immediate

FINDING

aws_s3_bucket.uploads has no aws_s3_bucket_public_access_block, no aws_s3_bucket_versioning, and no aws_s3_bucket_server_side_encryption_configuration resource associated with it, unlike the processed bucket which has public access controls applied.

WHY IT MATTERS

The uploads bucket receives user-submitted content via the POST /upload API endpoint. Without a public access block, a misconfigured bucket policy or ACL could expose all uploaded files publicly. Without encryption, data at rest is unprotected. Without versioning, accidental overwrites or deletions of uploaded files are unrecoverable.

BLAST RADIUS

All user-uploaded content in synthr-uploads-production could be publicly exposed or permanently lost. If uploads contain sensitive documents (consistent with a document-processing API), this represents a direct data-exposure risk for all synthr customers.

RECOMMENDATION

1. Add an aws_s3_bucket_public_access_block resource for the uploads bucket with all four settings set to true.
2. Add an aws_s3_bucket_server_side_encryption_configuration resource using aws:kms or AES256.
3. Enable versioning via aws_s3_bucket_versioning to protect against accidental deletion.
4. Add an S3 lifecycle rule to transition or expire old versions to control storage costs.

SECURE CONFIGURATION EXAMPLE

```
resource "aws_s3_bucket_public_access_block" "uploads" {
  bucket                  = aws_s3_bucket.uploads.id
  block_public_acls       = true
  block_public_policy     = true
  ignore_public_acls      = true
  restrict_public_buckets = true
}

resource "aws_s3_bucket_server_side_encryption_configuration" "uploads" {
  bucket = aws_s3_bucket.uploads.id
  rule {
    apply_server_side_encryption_by_default {
      sse_algorithm = "aws:kms"
    }
  }
}

resource "aws_s3_bucket_versioning" "uploads" {
  bucket = aws_s3_bucket.uploads.id
  versioning_configuration {
    status = "Enabled"
  }
}
```

CIS AWS Foundations v3.0 §2.1.5	NIST 800-53 SC-28	SOC 2 CC6.1

[HIGH] No WAF or throttling on API Gateway - API is unprotected against abuse

Security | Effort: Medium | Phase: 30-day

FINDING

aws_apigatewayv2_api.api and aws_apigatewayv2_stage.default have no throttling configuration (default_route_settings) and no AWS WAF WebACL association, leaving the public API endpoint unprotected against brute force, injection attacks, and volumetric abuse.

WHY IT MATTERS

The synthr API accepts file uploads and processes documents. Without WAF, malicious payloads (e.g., oversized uploads, SQL injection in job IDs) reach Lambda directly. Without throttling, a single client can trigger unbounded Lambda invocations, exhausting concurrency limits and causing a denial of service for all other customers.

BLAST RADIUS

Unrestricted API access can exhaust Lambda concurrency account-wide, affecting all Lambda-based workloads. A successful injection attack via the /upload or /job/{jobId} routes could compromise the database or worker pipeline.

RECOMMENDATION

1. Create an aws_wafv2_web_acl with AWS managed rule groups (AWSManagedRulesCommonRuleSet, AWSManagedRulesKnownBadInputsRuleSet) and associate it with the API Gateway stage.
2. Add default_route_settings to aws_apigatewayv2_stage.default with throttling_burst_limit and throttling_rate_limit appropriate for expected traffic.
3. Set reserved_concurrent_executions on aws_lambda_function.api_handler to cap blast radius from traffic spikes.

SECURE CONFIGURATION EXAMPLE

```
resource "aws_apigatewayv2_stage" "default" {
  api_id      = aws_apigatewayv2_api.api.id
  name        = "$default"
  auto_deploy = true

  default_route_settings {
    throttling_burst_limit = 100
    throttling_rate_limit  = 50
  }

  access_log_settings {
    destination_arn = aws_cloudwatch_log_group.api_access.arn
  }
}

resource "aws_wafv2_web_acl_association" "api" {
  resource_arn = aws_apigatewayv2_stage.default.arn
  web_acl_arn  = aws_wafv2_web_acl.api.arn
}
```

NIST 800-53 SC-5	SOC 2 CC6.6	PCI DSS Requirement 6.4

Reliability

[CRITICAL] **RDS has no backups, no Multi-AZ, and skip_final_snapshot = true**

Reliability | Effort: Low | Phase: Immediate

FINDING

aws_db_instance.main has backup_retention_period not set (defaults to 0, disabling automated backups), multi_az defaults to false, and skip_final_snapshot = true. The db_subnet_group references only a single subnet in eu-west-1a.

WHY IT MATTERS

The synthr production database has zero recovery capability. An AZ failure, accidental table drop, or storage corruption results in permanent, unrecoverable data loss. With no final snapshot, even destroying and recreating the Terraform stack leaves no restore point. For a document-processing API, this means all job history, session data, and processed results are permanently gone.

BLAST RADIUS

Total loss of all synthr production data on any failure event. All dependent services (Lambda API handler, worker, ECS reporter) lose their data source simultaneously with no recovery path.

RECOMMENDATION

1. Set backup_retention_period = 7 (minimum; 14-35 days recommended for production) on aws_db_instance.main.
2. Set multi_az = true to enable synchronous standby replication across AZs.
3. Add a second private subnet in eu-west-1b to aws_db_subnet_group.main.
4. Set skip_final_snapshot = false and provide a final_snapshot_identifier.
5. Enable storage_encrypted = true and point_in_time_recovery via the RDS console or Terraform.

SECURE CONFIGURATION EXAMPLE

```
resource "aws_db_instance" "main" {
  identifier        = "synthr-postgres-prod"
  engine            = "postgres"
  engine_version    = "15.4"
  instance_class    = "db.t3.medium"
  allocated_storage = 100

  db_name  = "synthr"
  username = "synthr_admin"
  password = var.db_password

  db_subnet_group_name   = aws_db_subnet_group.main.name
  vpc_security_group_ids = [aws_security_group.rds.id]

  multi_az               = true
  publicly_accessible    = false
  storage_encrypted      = true
  backup_retention_period = 14
  skip_final_snapshot    = false
  final_snapshot_identifier = "synthr-postgres-prod-final"

  tags = { Environment = "production" }
}
```

NIST 800-53 CP-9	SOC 2 CC9.1	CIS AWS Foundations v3.0 §2.3.1

[HIGH] **SQS visibility timeout shorter than worker Lambda timeout causes duplicate processing and message loss**

Reliability | Effort: Low | Phase: Immediate

FINDING

aws_sqs_queue.jobs has no visibility_timeout_seconds set (defaults to 30 seconds), but aws_lambda_function.worker has timeout = 300 seconds. When a worker invocation takes longer than 30 seconds, SQS makes the message visible again and another worker picks it up, causing duplicate processing. There is also no dead-letter queue defined.

WHY IT MATTERS

For a document-processing workload, duplicate processing means the same job is processed multiple times, potentially producing duplicate outputs, double-charging customers, or corrupting job state in the database. Messages that repeatedly fail have no DLQ to land in, so they are silently dropped after the maxReceiveCount is exhausted - causing undetected job loss.

BLAST RADIUS

All in-flight document processing jobs are affected. Duplicate processing can corrupt the synthr-sessions DynamoDB table and the processed S3 bucket. Silent message loss means customers receive no output and no error notification.

RECOMMENDATION

1. Set visibility_timeout_seconds on aws_sqs_queue.jobs to at least 360 seconds (6× the Lambda timeout per AWS guidance).
2. Create a separate aws_sqs_queue resource as a dead-letter queue and attach it via a redrive_policy on the jobs queue.
3. Set bisect_batch_on_function_error = true on aws_lambda_event_source_mapping.worker_sqs to isolate failing messages.
4. Add a CloudWatch alarm on the DLQ ApproximateNumberOfMessagesVisible metric to alert on processing failures.

SECURE CONFIGURATION EXAMPLE

```
resource "aws_sqs_queue" "jobs_dlq" {
  name = "synthr-jobs-dlq-production"
  message_retention_seconds = 1209600
}

resource "aws_sqs_queue" "jobs" {
  name                       = "synthr-jobs-production"
  visibility_timeout_seconds = 360

  redrive_policy = jsonencode({
    deadLetterTargetArn = aws_sqs_queue.jobs_dlq.arn
    maxReceiveCount     = 3
  })

  tags = { Environment = "production" }
}

resource "aws_lambda_event_source_mapping" "worker_sqs" {
  event_source_arn                   = aws_sqs_queue.jobs.arn
  function_name                      = aws_lambda_function.worker.arn
  batch_size                         = 10
  bisect_batch_on_function_error     = true
}
```

NIST 800-53 SI-13 | SOC 2 CC7.2

[HIGH] **Entire workload deployed in a single availability zone with no private subnets**

Reliability | Effort: High | Phase: 30-day

FINDING

Only aws_subnet.public_a exists, pinned to eu-west-1a. The RDS subnet group, Lambda VPC config, and ECS task all reference this single subnet. There are no private subnets, no NAT gateway, and no multi-AZ redundancy for any compute or data tier.

WHY IT MATTERS

An eu-west-1a AZ impairment - which AWS experiences periodically - takes down the entire synthr production API, all Lambda workers, the database, and the ECS reporter simultaneously with no automatic failover. For a production API serving customers, this represents an unacceptable single point of failure with no recovery path until the AZ recovers.

BLAST RADIUS

Complete synthr service outage affecting all customers. RDS has no standby in another AZ to promote. Lambda and ECS have no subnets in other AZs to reschedule into. Recovery requires manual intervention after AZ restoration.

RECOMMENDATION

1. Add private subnets in at least eu-west-1a and eu-west-1b for RDS, Lambda, and ECS.
2. Add a NAT gateway (or NAT gateway pair for HA) to allow private-subnet resources to reach the internet.
3. Update aws_db_subnet_group.main to include both private subnets and set multi_az = true on the RDS instance.
4. Update Lambda vpc_config and ECS task network configuration to reference both private subnets.

NIST 800-53 CP-7 | SOC 2 CC9.1

[MEDIUM] **DynamoDB sessions table has no TTL and no point-in-time recovery**

Reliability | Effort: Low | Phase: 30-day

FINDING

aws_dynamodb_table.sessions has no ttl block configured and no point_in_time_recovery block. Session tokens accumulate indefinitely, and there is no recovery capability if the table is accidentally deleted or corrupted.

WHY IT MATTERS

Without TTL, the sessions table grows without bound, increasing storage costs and query latency over time. Without PITR, accidental deletion of session records - or a bug that corrupts the table - cannot be recovered, forcing all synthr users to re-authenticate and potentially losing active session state.

RECOMMENDATION

1. Add a ttl block to aws_dynamodb_table.sessions with attribute_name set to the session expiry attribute and enabled = true.

2. Add a point_in_time_recovery block with enabled = true.
3. Ensure the application sets the TTL attribute on each session record at write time.

SECURE CONFIGURATION EXAMPLE

```
resource "aws_dynamodb_table" "sessions" {
  name         = "synthr-sessions"
  billing_mode = "PAY_PER_REQUEST"
  hash_key     = "token"

  attribute {
    name = "token"
    type = "S"
  }

  ttl {
    attribute_name = "expires_at"
    enabled        = true
  }

  point_in_time_recovery {
    enabled = true
  }
}
```

NIST 800-53 CP-9 | SOC 2 CC9.1

Operational Excellence

[HIGH] No CloudWatch alarms, dashboards, or alerting for any production resource

Operational Excellence | Effort: Medium | Phase: 30-day

FINDING

The Terraform code defines CloudWatch log groups for Lambda functions but zero aws_cloudwatch_metric_alarm resources. There are no alarms for Lambda errors, throttles, or duration; no SQS queue depth or age-of-oldest-message alarms; no RDS CPU, storage, or connection alarms; and no composite alarms or dashboards.

WHY IT MATTERS

The synthr API processes customer documents asynchronously. Without alarms, a Lambda error spike, a growing SQS backlog, or an RDS storage-full condition will go undetected until a customer reports a failure. For a production API, this means SLA breaches and customer-impacting outages are discovered reactively rather than proactively.

BLAST RADIUS

All production tiers - API, worker pipeline, database - operate without any automated detection of degradation or failure. Incidents that could be resolved in minutes instead persist for hours until customer complaints surface them.

RECOMMENDATION

1. Create CloudWatch alarms for Lambda Errors and Throttles (both functions) with SNS notification to an on-call topic.
2. Create alarms for SQS ApproximateAgeOfOldestMessage on the jobs queue (threshold: > 2× expected processing time) and ApproximateNumberOfMessagesVisible on the DLQ.
3. Create RDS alarms for CPUUtilization > 80%, FreeStorageSpace < 10GB, and DatabaseConnections approaching max.
4. Create a CloudWatch dashboard aggregating all key metrics for the synthr workload.

NIST 800-53 SI-4 | SOC 2 CC7.2

[MEDIUM] API Gateway stage has no access logging configured

Operational Excellence | Effort: Low | Phase: 30-day

FINDING

aws_apigatewayv2_stage.default has no access_log_settings block, so API request logs (caller IP, route, status code, latency) are not captured. Only Lambda execution logs are available, which do not include request-level metadata.

WHY IT MATTERS

Without API Gateway access logs, the synthr team cannot perform request-level debugging, identify abusive callers, audit which API keys accessed which endpoints, or reconstruct the sequence of events during a security incident. This gap is particularly significant given the absence of WAF logging.

RECOMMENDATION

1. Create an aws_cloudwatch_log_group for API Gateway access logs with an appropriate retention period.
2. Add an access_log_settings block to aws_apigatewayv2_stage.default referencing the log group ARN.
3. Grant API Gateway permission to write to CloudWatch Logs via the account-level CloudWatch Logs role if not already configured.

SECURE CONFIGURATION EXAMPLE

```
resource "aws_cloudwatch_log_group" "api_access" {
  name              = "/aws/apigateway/synthr-api-access"
  retention_in_days = 90
}

resource "aws_apigatewayv2_stage" "default" {
  api_id      = aws_apigatewayv2_api.api.id
  name        = "$default"
  auto_deploy = true

  access_log_settings {
    destination_arn = aws_cloudwatch_log_group.api_access.arn
  }
}
```

CIS AWS Foundations v3.0 §3.10 | NIST 800-53 AU-2 | SOC 2 CC7.2

[MEDIUM] ECS reporter task uses :latest image tag with no pinned digest

Operational Excellence | Effort: Low | Phase: 30-day

FINDING

The container image in aws_ecs_task_definition.reporter is referenced as synthr-reporter:latest. The :latest tag is mutable and will resolve to a different image on each task launch without any change to the Terraform configuration.

WHY IT MATTERS

A new image pushed to ECR with the :latest tag will be silently picked up by the next nightly reporter run without any deployment review or approval. This means a broken or malicious image can enter production without triggering a Terraform plan, making rollbacks difficult and deployments non-deterministic.

RECOMMENDATION

1. Pin the container image to an immutable digest (e.g., synthr-reporter@sha256:abc123...) in the task definition.
2. Update the CI/CD pipeline to update the Terraform variable or task definition with the new digest on each build.
3. Enable ECR image scanning on push to detect vulnerabilities before deployment.

SECURE CONFIGURATION EXAMPLE

```
container_definitions = jsonencode([{
  name  = "reporter"
  image = "123456789.dkr.ecr.eu-west-1.amazonaws.com/synthr-reporter@sha256:abc123...
  # ... rest of config
}])
```

NIST 800-53 CM-2 | SOC 2 CC8.1

ost Optimization

[MEDIUM] Lambda functions have no concurrency limits, risking runaway cost from traffic spikes or bugs

Cost Optimization | Effort: Low | Phase: 30-day

FINDING

Neither aws_lambda_function.api_handler nor aws_lambda_function.worker has reserved_concurrent_executions set. An upstream traffic spike, a retry loop bug, or a malicious caller can scale Lambda to account-level concurrency limits, generating unexpected costs and starving other workloads.

WHY IT MATTERS

The worker Lambda has a 300-second timeout and 1024 MB memory. Without a concurrency cap, a runaway SQS processing loop could spin up hundreds of concurrent worker invocations simultaneously, generating significant unexpected charges and potentially exhausting the account's Lambda concurrency quota.

RECOMMENDATION

1. Set reserved_concurrent_executions on aws_lambda_function.api_handler based on expected peak RPS (e.g., 100 for moderate traffic).
2. Set reserved_concurrent_executions on aws_lambda_function.worker based on the desired maximum parallel document processing throughput.
3. Monitor Lambda ConcurrentExecutions metrics and adjust limits based on observed usage patterns.

SECURE CONFIGURATION EXAMPLE

```
resource "aws_lambda_function" "worker" {
  function_name                  = "synthr-doc-worker"
  reserved_concurrent_executions = 20
  # ... other config
}

resource "aws_lambda_function" "api_handler" {
  function_name                  = "synthr-api-handler"
  reserved_concurrent_executions = 100
  # ... other config
}
```

NIST 800-53 SC-5

erformance Efficiency

[MEDIUM] **No caching layer for API responses or database queries**

Performance Efficiency | Effort: Medium | Phase: 60-day

FINDING

The API Gateway routes (GET /job/{jobId}, POST /upload) invoke Lambda directly with no caching configured on the API Gateway stage or within the application. There is no ElastiCache or DAX layer between Lambda and the RDS/DynamoDB data stores.

WHY IT MATTERS

The GET /job/{jobId} route is a status-polling endpoint that clients likely call repeatedly. Without response caching, every poll hits Lambda and then the database, increasing RDS connection pressure and latency. As job volume grows, this pattern will degrade database performance and increase costs proportionally with polling frequency.

RECOMMENDATION

1. Enable API Gateway response caching on the GET /job/{jobId} route with a short TTL (e.g., 5-10 seconds) to absorb polling bursts.
2. Evaluate adding an ElastiCache (Redis) cluster for session lookups from the DynamoDB sessions table if read volume is high.
3. Consider using DynamoDB DAX if DynamoDB read latency becomes a bottleneck for session validation.

NIST 800-53 SC-5

ustainability

[LOW] **ECS Fargate reporting task has no scheduling or auto-scaling - may run continuously or be over-provisioned**

Sustainability | Effort: Low | Phase: 60-day

FINDING

aws_ecs_task_definition.reporter is defined as a nightly reporting task (per the workload description) but there is no aws_scheduler_schedule, aws_cloudwatch_event_rule, or ECS service with scaling policies in the Terraform code. The task definition allocates 512 CPU units and 1024 MB memory with no evidence of right-sizing based on actual utilization.

WHY IT MATTERS

If the ECS task is run as a long-lived service rather than a scheduled task, it consumes Fargate compute continuously for a workload that only needs to run nightly. Even as a scheduled task, the fixed 1 vCPU / 1 GB allocation may be significantly over-provisioned for a reporting job, resulting in unnecessary carbon footprint and cost.

RECOMMENDATION

1. Implement the nightly reporter as an EventBridge Scheduler rule targeting ECS RunTask to ensure it only runs when needed.
2. Profile the reporter task's actual CPU and memory utilization and right-size the task definition accordingly.
3. Consider using AWS Graviton (ARM64) Fargate tasks for the reporter workload to reduce cost and energy consumption by up to 20%.

NIST 800-53 SA-8

SECTION 04

Remediation Roadmap

Immediate - Fix before next deployment

[CRITICAL]	SEC-001	RDS instance publicly accessible with security group open to 0.0.0.0/0
[CRITICAL]	SEC-002	Database password and API key stored as plaintext Lambda and ECS environment variables
[HIGH]	SEC-003	Lambda IAM role grants wildcard s3:*, sqs:*, and dynamodb:* across all resources
[HIGH]	SEC-004	uploads S3 bucket has no public access block, versioning, or server-side encryption
[CRITICAL]	REL-001	RDS has no backups, no Multi-AZ, and skip_final_snapshot = true
[HIGH]	REL-002	SQS visibility timeout shorter than worker Lambda timeout causes duplicate processing and message loss

30-Day - Address within one sprint cycle

[HIGH]	SEC-005	No WAF or throttling on API Gateway - API is unprotected against abuse
[HIGH]	REL-003	Entire workload deployed in a single availability zone with no private subnets
[MEDIUM]	REL-004	DynamoDB sessions table has no TTL and no point-in-time recovery
[HIGH]	OPS-001	No CloudWatch alarms, dashboards, or alerting for any production resource
[MEDIUM]	OPS-002	API Gateway stage has no access logging configured
[MEDIUM]	OPS-003	ECS reporter task uses :latest image tag with no pinned digest
[MEDIUM]	COST-001	Lambda functions have no concurrency limits, risking runaway cost from traffic spikes or bugs

60-Day - Plan in next quarter

[MEDIUM]	PERF-001	No caching layer for API responses or database queries
[LOW]	SUST-001	ECS Fargate reporting task has no scheduling or auto-scaling - may run continuously or be over-provisioned

SECTION 05

Risk Distribution

SECTION 06

Disclaimer

This report was generated by ArchGuard using AI-assisted analysis of submitted Terraform configuration files. Findings are based on static analysis and represent potential issues, not confirmed vulnerabilities. No live AWS environment was accessed. Results should be reviewed by qualified engineers before taking remediation action. ArchGuard makes no warranty as to the completeness or accuracy of findings.