2 years with Python and Serverless

Hector Canto

Or what I would love to know 2 years ago

Background

- Python developer for 12 years
- Teacher and speaker for some time
- Scraping for IP protection
- Core services and integrations for video services for automation

[ ] SMART PROTECTION

Topics

- Stack and technologies
- Why and why not Serverless
- Code samples
- Deploy and Configuration
- Life Cycle
- Real Scenarios
- Costs
- Testing
- Logging and monitoring
- Reusability


Stack & technologies

Stack

- Python 3.8 and 3.9
- boto, botocore, aiobotocore
- FastApi, Pydantic, SQLAlchemy, pynamo
- AWS Lambda power tools
- pytest, structlog, handsdown

Backing services

- **API Gateway**
- **SQS**: message queue
- **RDS** Aurora: Managed DB services
- **DynamoDB**: Cache, KeyValueStore, Fast DB
  - **SSM Parameter Store**: Configuration
  - **Cloudwatch**: Logging and alerts
  - **S3**: File Storage

Backing technologies

Gitlab CI

Terraform & Terragrunt

Serverless Framework

Other interesting backing technologies

- Monitoring: AWS X-Ray, Epsagon
- ChaliceLib: Great for REST APIs, also other triggers
- Zappa, Kubeless, OpenWhisk
- Lamba power tools (see next)

Lambda power tools

- Pydantic event parser for each service
- Router for API GW
- Simple clients for SSM, Dynamo ...
- Decorators for exceptions and middlewares
- Logging and metrics
- 🌐 awslabs powertools

What I built so far

- APIs: full, small, webhook, partial
- DB procedures: maintenance, re-indexing, reporting
- Crons: Token refreshers
- Triggers: Alerting, thumbnails
- Task workers: Cross-region synchronization, Media post-processing
- Replaces Celery, Airflow, EC2 ...

Why serverless

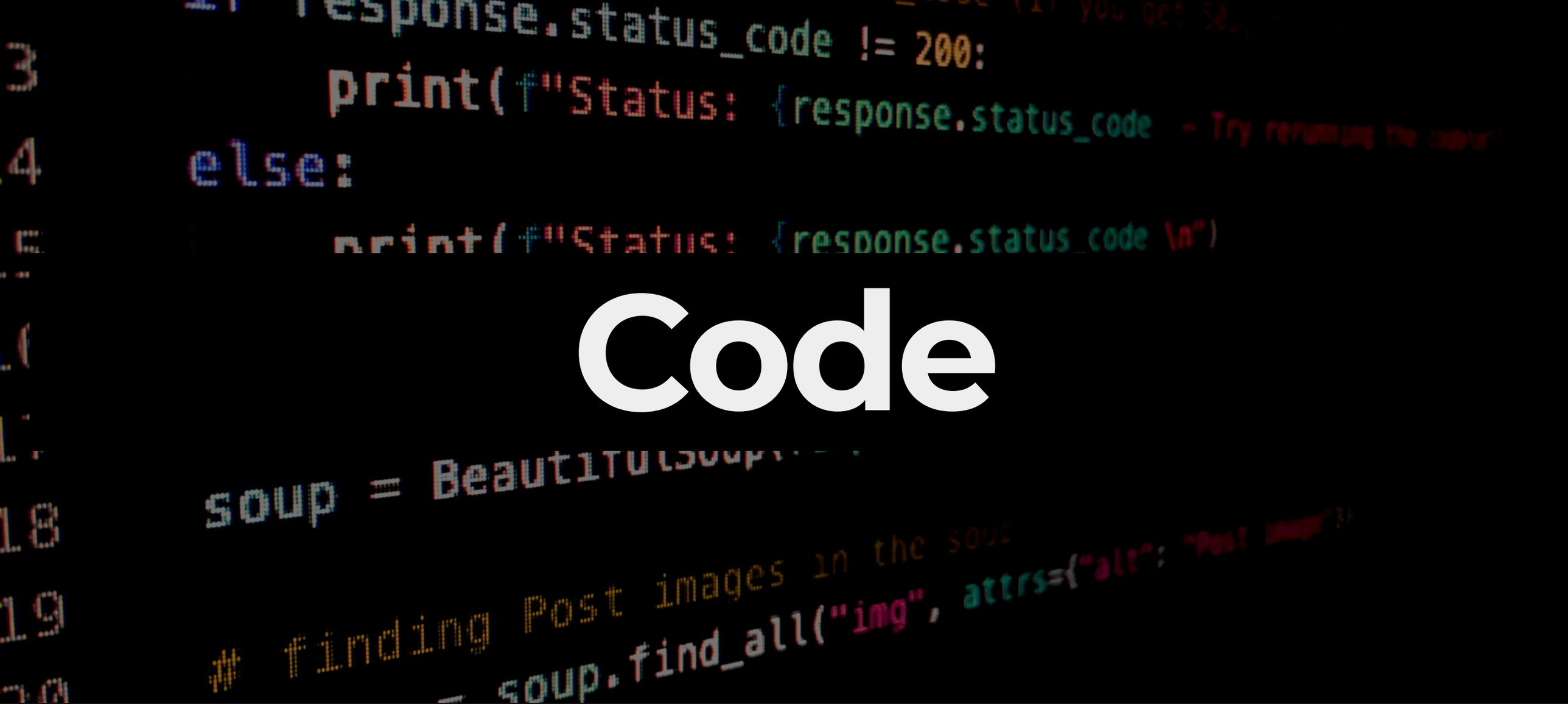
- reduce costs (a lot)
- reduced development scope (less local complexity)
- break out THE monolith
- task or event-oriented
- scalability

Why NOT serverless

- less Ops ... different Ops
- simpler architecture ... more small pieces
- quicker development ... focused on small pieces

Serverless caveats

- more general complexity
- a lot of “magic”
- worse/expensive monitoring
- some latency
- vendor lock-in


response.status_code != 200:
print(f"Status: {response.status_code
else:
soup = Beauti
# finding Post images in the soup
soup.find_all("img", attrs={"alt":

Code

Code: is Lambda different?

- Not really
- Same repo as any other application
- Avoid big libs: numpy, pandas ..
- More functions than classes (FaaS)

Lambda init is cached

```
import boto3
from service import conf

# cached artifacts
CONFIG = config.get_settings()
SSM = boto3.client("ssm", region_name=CONFIG.aws.region_name, config=**CONFIG.ssm.dict())
update_settings_from_ssm(SSM, CONFIG)

SQS = boto3.resource("sqs", region_name=CONF.aws.region_name)
QUEUE = SQS.Queue(SQS.get_queue_by_name(QueueName=CONFIG.app.queue_name).url)
```

Lambda handler or entrypoint

```python
# project/service/worker_entrypoint.py
@event_parser(model=event_models.CoolQueueEvent)
def worker_main(event: event_models.CoolEvent, _context: LambdaContext) -> list:
    batch_size = len(event.Records)
    result_list = []
    for index, record in enumerate(event.Records):
        result_list += process_record(event_record)
    return result_list
```

Lambda context object

- function_name
- function_version
- aws_request_id
- memory_limit_in_mb
- get_remaining_time_in_millis()

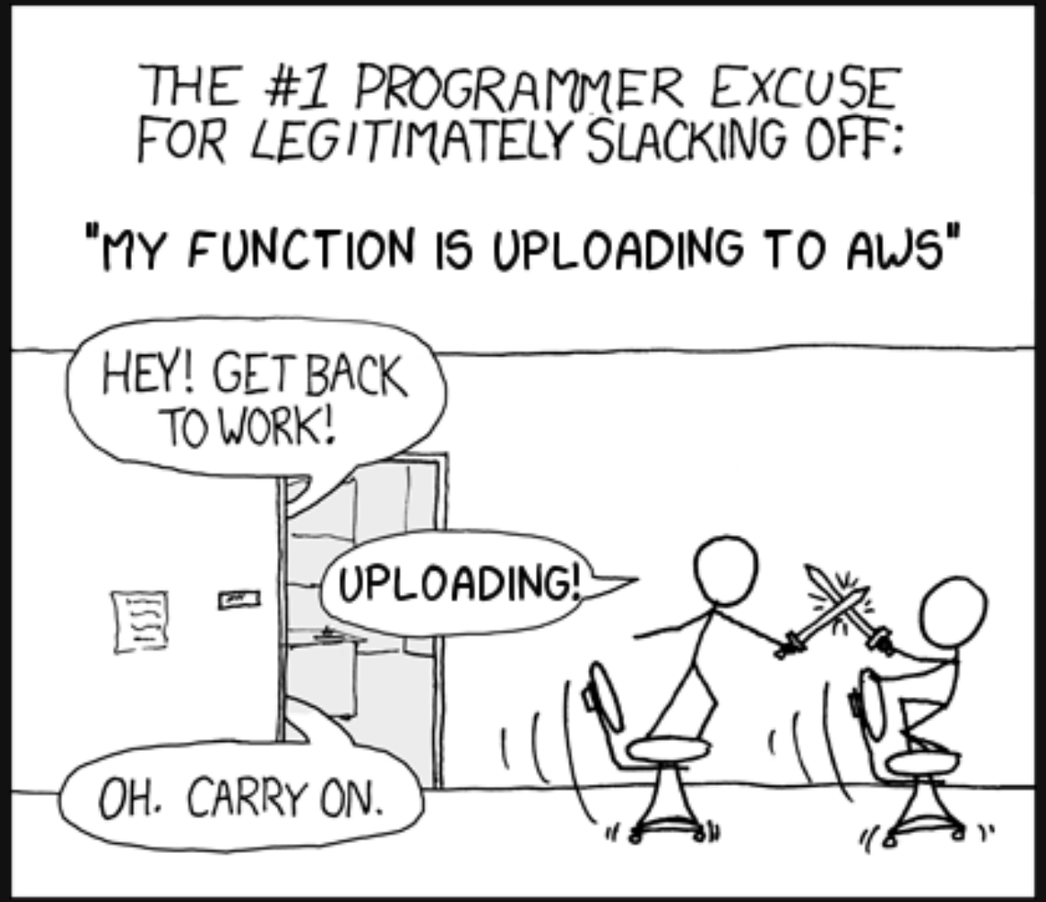
Lambda API resolver

```python
MAIN_ROUTE = "/integration/"
ENDPOINT_ONE = "/endpoint/<identifier>"
router = APIGatewayHttpResolver(strip_prefixes=[MAIN_ROUTE])

def api_entrypoint_or_handler(event: dict, _context: LambdaContext) -> Dict[str, Any]:
    """Entrypoint for all crud operations"""
    return router.resolve(event, _context)

@router.get(ENDPOINT_ONE)
def retrieve_entity(identifier) -> Union[str, Response]:
    """Retrieve entity with given identifer"""
    logger.info(inspect.currentframe().f_code.co_name)
    # Do stuff
    msg = "Item not found"
    logger.error(msg)
    return response_builder(msg, 404)
```

Lambda code surprises

- Session is reused, so it has to have a clean state
  - Rollbacks!!
- Don’t modify CONFIG globally
- Clean up `/tmp`
- Async tasks (alerts) are not awaited
  - sleep 1s?
- Memory leak: each run stores additional data

Deploy


THE #1 PROGRAMMER EXCUSE FOR LEGITIMATELY SLACKING OFF:
"MY FUNCTION IS UPLOADING TO AWS"
HEY! GET BACK TO WORK!
UPLOADING!
OH. CARRY ON.

Serverless Framework

- Configure and Deploy lambdas
- Specify permissions & triggers
- Link it with existing infrastructure
- Deploy even to localstack

Deploy with SLS

```
serverless deploy --stage production --region eu-central-1
serverless doctor
serverless remove --stage staging --region eu-west-1
metrics, config, tests, rollback, ...
```

Deploy surprises

- All versions are kept: 75 GB in 7MB bundles
- Private lambdas are very private (alerting blocked)
  - NAT IPs are very expensive: trigger another lambda
- requirements missing >> deploy without libs
- 1000 lambdas per account

Configuration

Configuration

- Resources
- Timeout
- Concurrency
- Batch size
- Triggers
- Permissions

Configure permissions

```
iam:
  role:
    statements:
      # Logs CloudWatch
      - Effect: 'Allow'
        Action:
          - 'logs:CreateLogGroup'
          - 'logs:CreateLogStream'
          - 'logs:PutLogEvents'
        Resource:
          - "*"
```

Configure triggers

```
functions:
  worker_alpha:
      handler: service.worker_entrypoint.worker_main
      timeout: 60
      memorySize: 512
      batchSize: 5
      events:
        - sqs:
            arn:aws:sqs:eu-central-1:2233445678:service1-task-alpha
        - httpApi:
          method: GET
          path: /integration/endpoint_one/{identifier}
          documentation: ${file(docs/functions.doc.yml):endpoints.endpoin_one}
          authorizer:
            name: integrations_api
            scopes:
              - integration:one
```

Configuration surprises

- double RAM, 1/4 execution time or more
  - more VCPUs, more BW, less cost
- default retries, extra waste
- 2 vCPU until 3 GB RAM

Timeout defaults

- Max general timeout: 15 min
- Default timeout: 3 min
- Max timeout for API GW triggers: 29 seconds

Configuration pydantic-settings

```python
class AppSection(YamlSection):
    """Configuration related to the Business logic or the application proper"""

    name: str = Field("globalapi", const=True)
    """Name of the application this module belongs to"""
    log_level: LogLevel = "INFO"
    """A standard log level name in uppercase"""
    log_visibility: Optional[dict]
    """Log namespace against its level"""
    slack_enabled: bool
    """Whether to sent alerts to Slack"""
    report_channel: Optional[str]
    """A Slack channel to report the CI summary"""
    ssm_enabled: bool = True
    """Whether SSM should be used, always True in live envs, but false in CI and local"""

    class Config(MetaConfig):
        env_prefix = "APP_"
```

Configuration per environment

```
Settings = Annotated[
    Union[ProdSettings, TestSettings, LocalStackSettings], Field(discriminator="app_environmen
]
def get_settings(app_environment=None) -> Settings:
    the_env = app_environment or os.environ.get("APP_ENVIRONMENT")
    if the_env not in ["test", "ci", "prod", "live", "localstack"]:
        raise ValueError(f"Invalid App environment {the_env}")
    return parse_obj_as(Settings, {"app_environment": the_env})
```

Configuration loading

- hierarchy: hardcoded > SSM* > secrets > env_vars > yaml* > pydantic class
- SSM loading: app > service > default

SSM configuration

- general `/default/log_level`
- service-wise `/service/default/log_level`
- app specific `/service/app_name/log_level`
- another specific `/integration/partner/token`
- per section general `/default/db/host`

```
ssm_client.get_parameters_by_path(Path="/default")
update_settings_from_ssm(ssm_client, CONFIG)
db_client = DbClient(**CONFIG.db.dict_no_secrets())
```

Lambda life cycle

Lambda life cycle

1. Queue 1-N msgs in Batch Window
2. Initialization
    1. Cold Start (download and unzip code)
    2. Warm start container creation, runs global)
3. Run entrypoint and resolve batch Container reuse
4. Another batch
    1. Now / short wait, goto 3
    2. Max batches, goto 2.2
    3. Long wait / new version, goto 1 (th. 2.2)

How to mitigate cold starts

- Provisioned concurrency
- Empty events

Empty event trick

- Forces Cold and Warm start
- Loads config and artifacts (cache)
- Test deploy, import, logging, and alerting

Empty event catch

```python
@lambda_handler_decorator
def preparation(
        handler: Callable, event: dict, ctx: LambdaContext, prepare: Optional[Callable] = None
) -> Callable:
    if prepare:
        prepare(handler.__module__, CONFIG.app.log_level)
    logger.info(f"Current version is {handler.__module__.__version}")
    if not event:
        raise EmptyEvent(f"Received and empty event for lambda `{ctx.aws_request_id}`")
    response = handler(event, ctx)
    return response
```

Scenarios

Lambda invocations

- Triggers: API GW, SQS, CW, EventBridge ..
- Sync invocation: invoking lambda sleeps
- Async invocation: invoking lambda keeps going

Does lambda fail? Runtime errors

- Import error
- Code error: Unexpected exception
- Logging error
- Timeout

What happens after it fails

- Retries (default 2)
- DLQ: message saved for later (auto)
- Cloudwatch trigger
- Waste: repeated logs, repeated writtes, more lambdas
- Data corruption!
- Increased costs

Prevent corruption and cost

- Avoid loops with the DLQ
- Limit or remove retries
- Save process hash and don’t repeat
- Trigger retry from the outside (on demand)
- Check remaining time on the lambda


Costs

Costs caused by

- Invocations and execution time and resources
- Timeouts
- DynamoDB operations and permanent data
- Duplicated or looped messages
- Big messages

Cost mitigation

- Keep messages small, send Ids, not entities
- Filter messages soon
- Useless messages
  - Filter in streams and SQS
- Timeouts, Check pending time within lambda
- Reduce client timeouts (DB, APIs)
- Avoid lambda, use direct calls
- Avoid NAT IPs, call another public lambda

Cost evolution

Faster lambda

- Lambdas have at least 2 VCPUs (2+ threads and idle time)
- Use `asyncio` and `aiofiles` to: download/upload in parallel
- Use `aiobotocore`: get/sent data in parallel
- Save partial results in DynamoDB
- SQS send_batch_messages

Faster lambdas examples

- 8 images download and upload: 2 min
- 60 images async, 2 min (x7)
- 3367 events one by one: 10 min
- 3500 events in batches of 10: 98 s (15%)

Testing with λ

How to test

- Moto : mocked boto
- localstack/localstack
  - awscli-local
- amazon/dynamodb-local
- amazon/aws-lambda-python
- minio/minio : Object storage like S3
- mysql/mysql:8-debian

Test AWS services with Moto

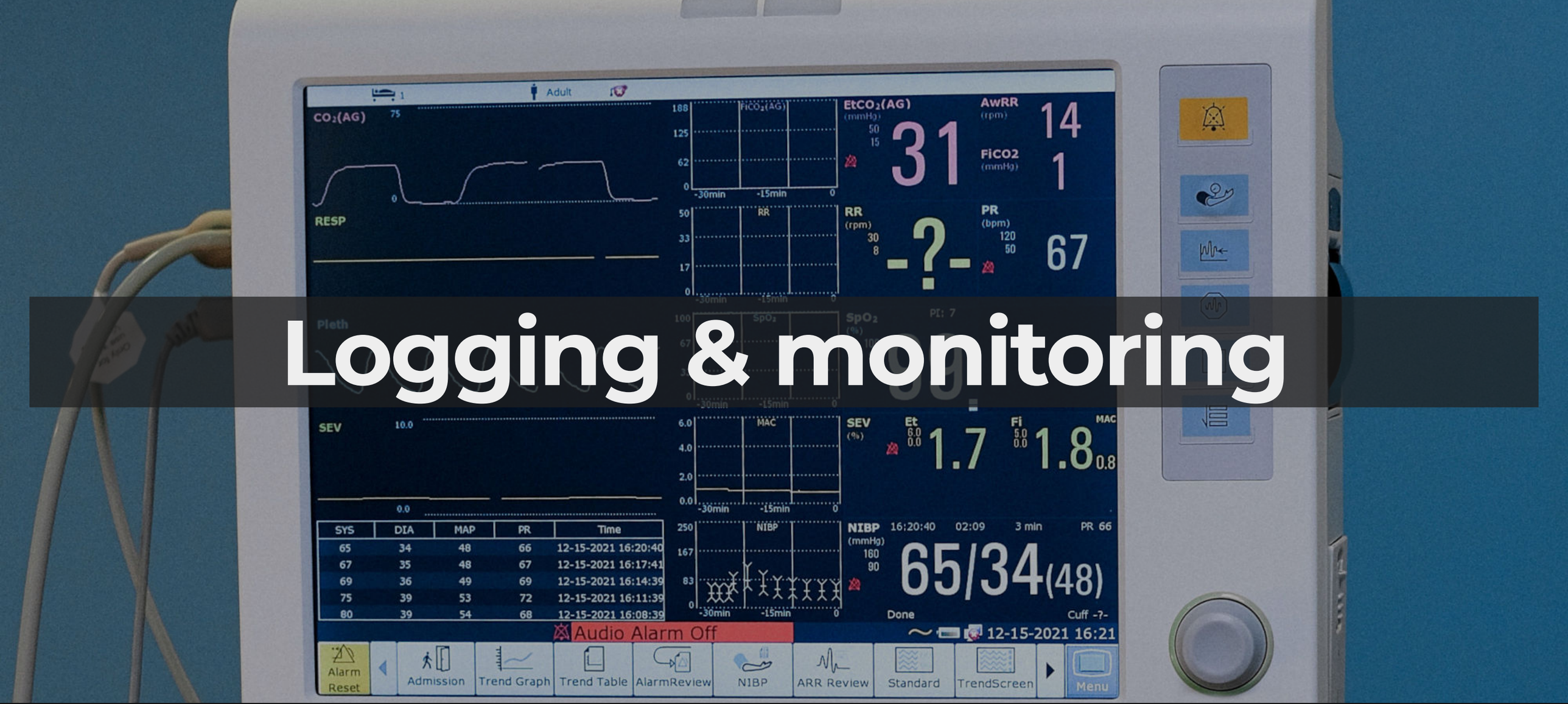
```python
@pytest.fixture(scope="function", name="mocked_sqs")
def mocking_sqs(test_settings):
    with mock_sqs():
        client = boto3.resource("sqs", region_name=test_settings.aws.region)
        queue_name = f"{test_settings.queue.prefix}-{test_settings.app.queue_name}"
        queue = client.create_queue(QueueName=queue_name)
        yield queue

@pytest.fixture(scope="function")
def test_dyno(test_settings):
    with mock_dynamodb():
        LocalDynamoDB.Meta.region = test_settings.aws.region
        LocalDynamoDB.Meta.table_name = test_settings.dyno.table_name
        LocalDynamoDB.create_table()
        yield LocalDynamoDB  # we return the table
        LocalDynamoDB.delete_table()

@pytest.fixture(scope="session", autouse=True)
def mocked_ssm():
    """SSM is mocked globally"""
```

Test AWS services with localstack

```
import boto3
from .conf import get_settings
config = get_settings(env="localstack")
client = boto3.client("ssm", region_name="eu-west-1", endpoint_url="http://localhost:4556")
client = boto3.client("ssm", **config.ssm.dict())
```

Testing tips

- Use powertools and factory-boy to create events and context
- Test also starting on the entrypoint
- Prefer localstack over moto
- Silent noisy logs
- Adapt json logging with KeyValueRender



Logging & monitoring

How to lambda-log

- json-based log (structured)
- f.i. structlog
- add identifiers and tracing
- keep partial timers

Log for humans and machines

```
logging.info(f"Download {num} images took {dhms(value)}, duration_s=value`
```

```
INFO event=Download 20 images took 1m3s550ms duration_s=63.550
```

More logs

```
from structlog import get_logger, contextvars

logger = get_logger(name=__name__)
contextvars.bindcontextvars(RequestId=context.aws_request_id, trace_id=event.trace_id, identif
logger.error(msg,  debug=event, exc_type=exc_type, exception=stacktrace, duration_s=duration,
```

Traceability

```
domain_event = event.Records[0].body
bindcontextvars(
  RequestId=context.aws_request_id,
  trace_id=event.headers["X-Amzn-Trace-Id"] # Root=1-5759e988-bd862e3fe1be46a994272793
  event_name=domain_event.event_name,
  identifier=domain_event.event_id
)
```

Log insights filtering

```
fields @timestamp, logger, RequestId, event, event_name, level, trace_id, identifier, exc_type
  | sort @timestamp desc
  | limit 100
  | filter ispresent(level)
```

Log insights stats

```
fields @timestamp, level, event_name, logger, event, @message
| stats avg(propagation_s), max(propagation_s)
| stats avg(@maxMemoryUsed/1000000) by bin(1h)
| filter propagation_s < 1000  # seconds
| filter type like "REPORT"
```

Log insights another example

Live logging

```
aws logs tail "/aws/lambda/my-lambda-function" \
  --color on --profile debug@prod_account --region eu-central-1 --follow
```

Logging surprises

- Cloudwatch logs take 1 to 15 minutes to appear
- Retention costs are big
- X-Ray is expensive, but external platforms too
- default logs START, END, REPORT, ERROR cannot be structured (?)

Reusability

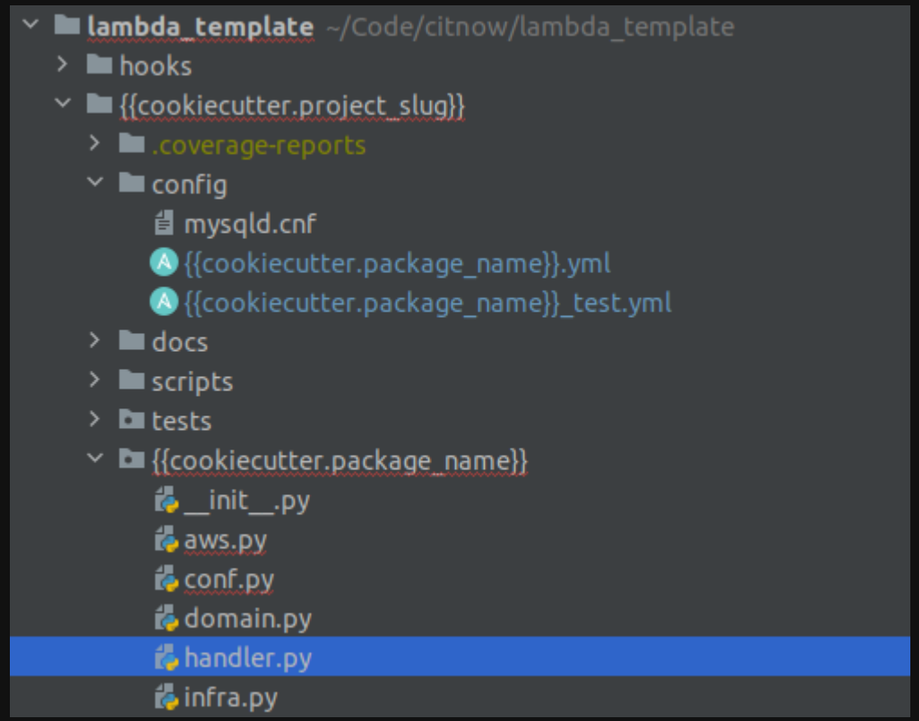
What is reusable

- Architecture recipes: how we connect lambdas and services
- Documentation
- Private libraries: domain events, db tables, company common, config and logging utils
- Repository and Code templates
- Deploy and permission templates

Reusability: deploys

- Many lambdas in the same repo: NO!
- deploy super-repository + git submodules
- multiple deploy files, CI magic
- Deploy templates
- Move fixed thing to Terraform or CloudFormation

Cookiecutter example



TLDR

- **Great libs**: powertools & aiobotocore
- **Deploy**: Terraform & Serverless
- **Configuration**: SSM & pydantic
- **Testing**: pytest, moto & localstack
- **Logging**: structlog & CW Log insights

Thanks for your attention

@hectorcanto_dev

slideshare.net/HectorCanto

linkedin.com/in/hectorcanto