Serverless Applications with AWS SAM

- create auto-scaling web APIs
- handle background processes
- secure APIs
- inspect and monitor serverless applications
- manage deployments using AWS CloudFormation and AWS SAM
- design applications to get the most out of this new type of architecture

Code/slides will be @ serverless.pub in a few days

Two-day coding workshop at Crisp 28-29 March (www.crisp.se/kurser)

gojko@gojko.com @gojkoadzic

Why serverless?

- time to market
- significant reduction for operational costs
- good when throughput is more critical than latency

Why SAM?

- Rapidly maturing
- Provided by Amazon directly
- Integrated nicely with other Amazon dev tools
- Easy to extend (just CloudFormation under the hood)

Billing actual usage, not reserved capacity

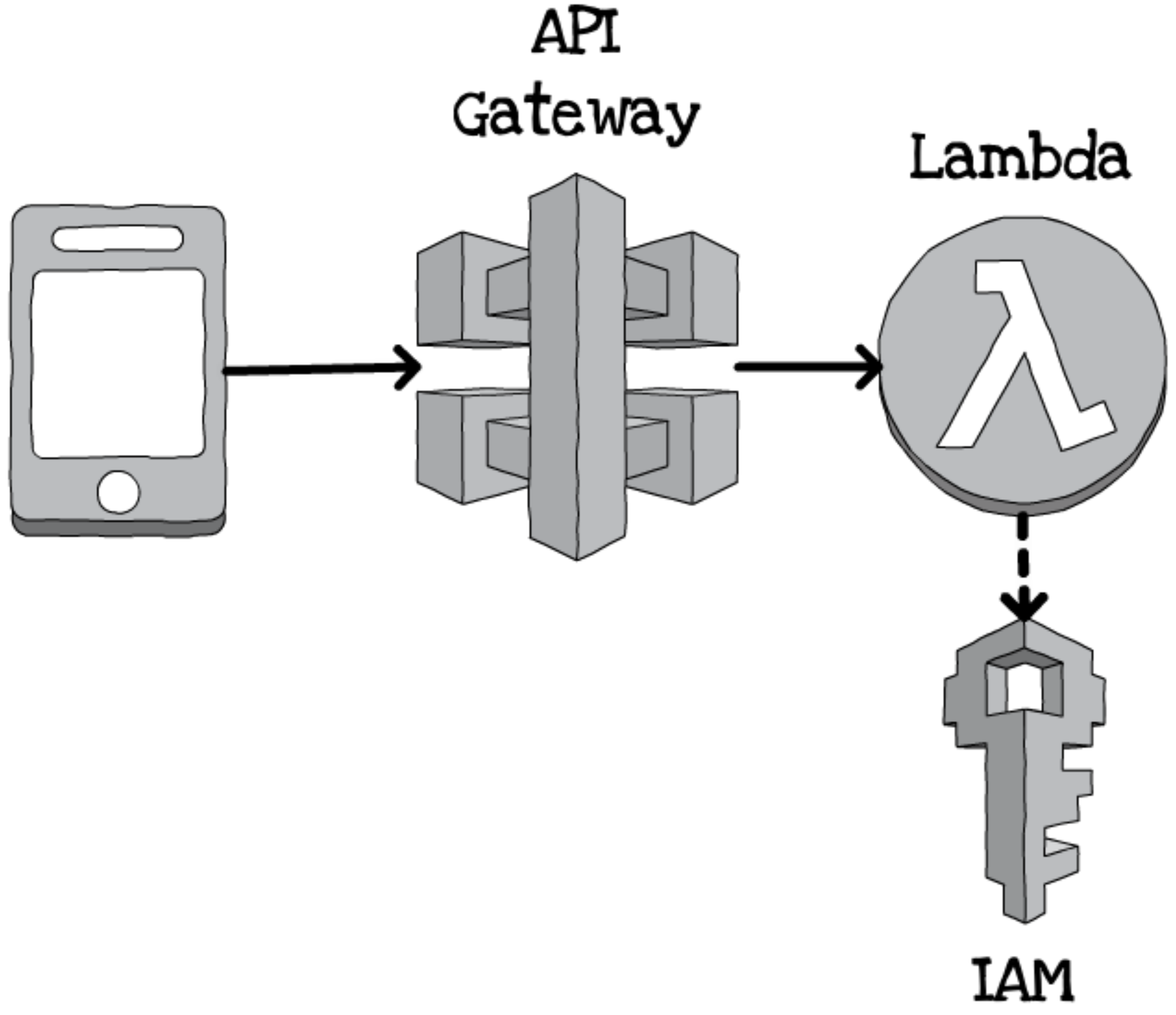
- $0.0000002 per request
- $0.000000834 for 100ms @ 512MB
- First 1 million requests per month are free

Code with batteries included

- Scaling
- Monitoring
- Recovery
- Versioning
- Logging

SAM Basics: initialise a new app

```
sam init --runtime java8
sam package ...
sam deploy ...
```


API Gateway
Lambda
IAM

"Time to recover"

<u>no longer important</u>

Multi-versioning

is amazing

It's not stateless, but

Share-nothing

CloudFormation basics: infrastructure as code

- **YAML/JSON template + links to project code**
- `package` **uploads project code to S3 and updates deployment config**
- `deploy` **using transformed config, or upload, or give to CI...**

AWS SAM: means two things

- `Transform: AWS::Serverless-2016-10-31`
- `sam` **command line tool**

`Transform: AWS::Serverless-2016-10-31`

- **adds new resources to CloudFormation**
- **implicitly creates IAM roles and event wiring**
- **reduces boilerplate code significantly**

SAM command line tool

- test locally using `docker`
- convenient templates for apps and events
- aliases/wrappers for common CloudFormation commands

CF basics: create a deployable template

```
aws cloudformation package
 --template-file <input template>
 --output-template-file <deployable template>
 --s3-bucket <asset bucket>
```

SAM extra: bundle source and dependencies cleanly

```
sam build
```

— **for nodejs, python, go... (not yet Java)**

SAM extras: pack either main or built template

```
sam package
  --output-template-file <deployable template>
  --s3-bucket <asset bucket>
  # not necessary --template-file <input>
```

ACTION	VERSION	ALIASES
create	1	$LATEST = 1
update	2	$LATEST = 2
update --version prod	3	$LATEST = 3, prod = 3
update --version dev	4	$LATEST = 4, prod = 3 dev = 4
set-version --version prod	4	$LATEST = 4, prod = 4 dev = 4

SAM extras: gradual deployment

```
DeploymentPreference:
 Type: Canary10Percent10Minutes
 Alarms:
   - !Ref CheckForDropInSales
   - !Ref CheckForDropInConversion
 Hooks:
   PreTraffic: !Ref ClearStatisticsLambda
   PostTraffic: !Ref NotifyAdminsLambda
```

gradual deployment options

- Canary10Percent30Minutes
- Canary10Percent5Minutes
- Canary10Percent10Minutes
- Canary10Percent15Minutes
- Linear10PercentEvery10Minutes
- Linear10PercentEvery1Minute
- Linear10PercentEvery2Minutes
- Linear10PercentEvery3Minutes

CF basics: get stack resources

```
aws cloudformation describe-stack-resources
   --stack-name <stack name>
```

CF basics: get stack outputs

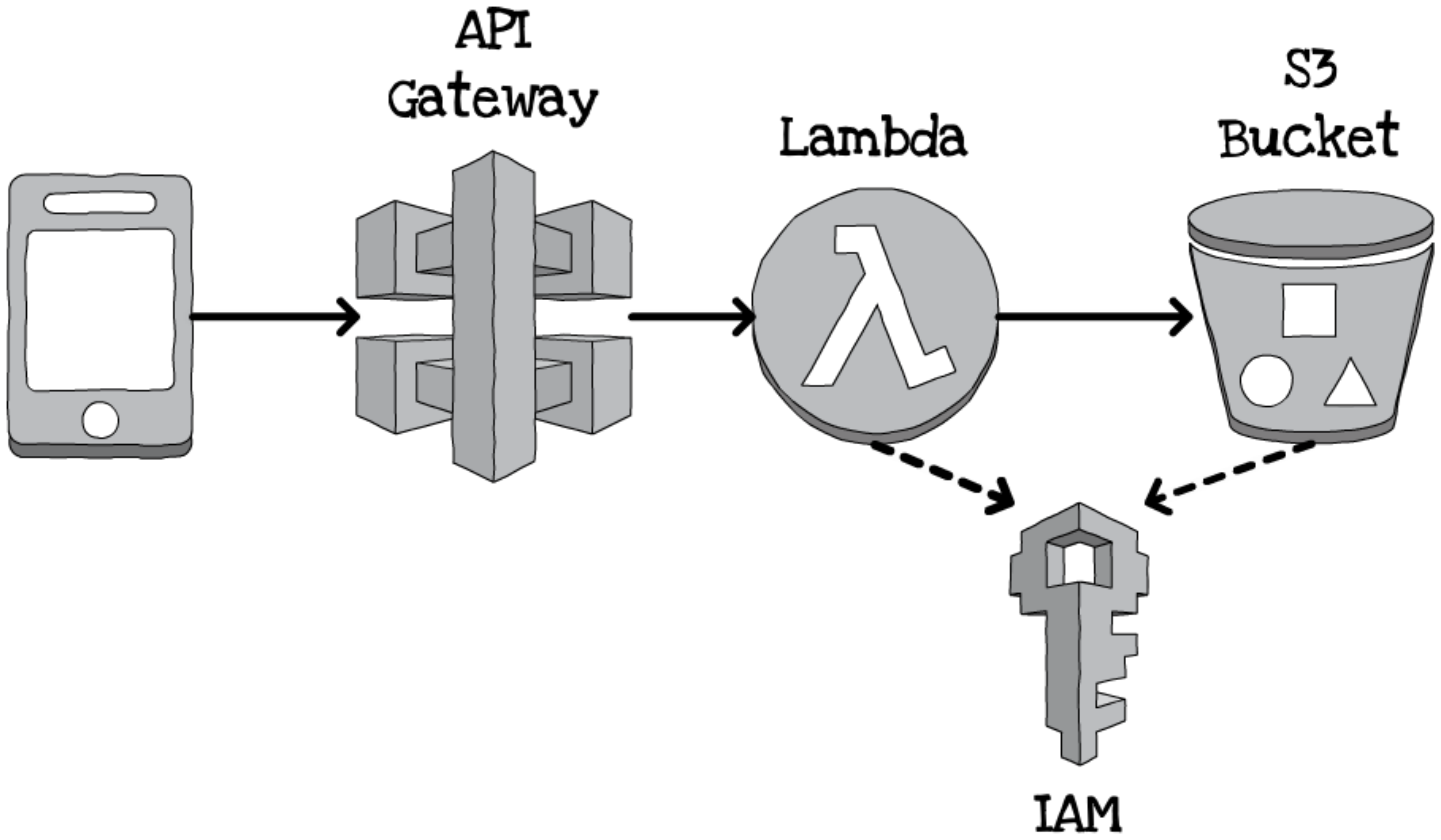
```
aws cloudformation describe-stacks
  --stack-name <stack name>
  --query 'Stacks[].Outputs[]'
  --output table
```

SAM extras: run with API locally

```
sam local start-api
```

SAM extras: read logs

```
sam logs -n <LAMBDA_FUNCTION_NAME>
```


API Gateway
Lambda
S3 Bucket
IAM

Talking to other AWS services

- set up IAM access policies
- use AWS SDK APIs with implicit authentication from Lambda
- use environment vars to pass references to resources
- use `context.awsRequestId` for unique-per-request values
- consider timeouts

SAM extras: generate sample events

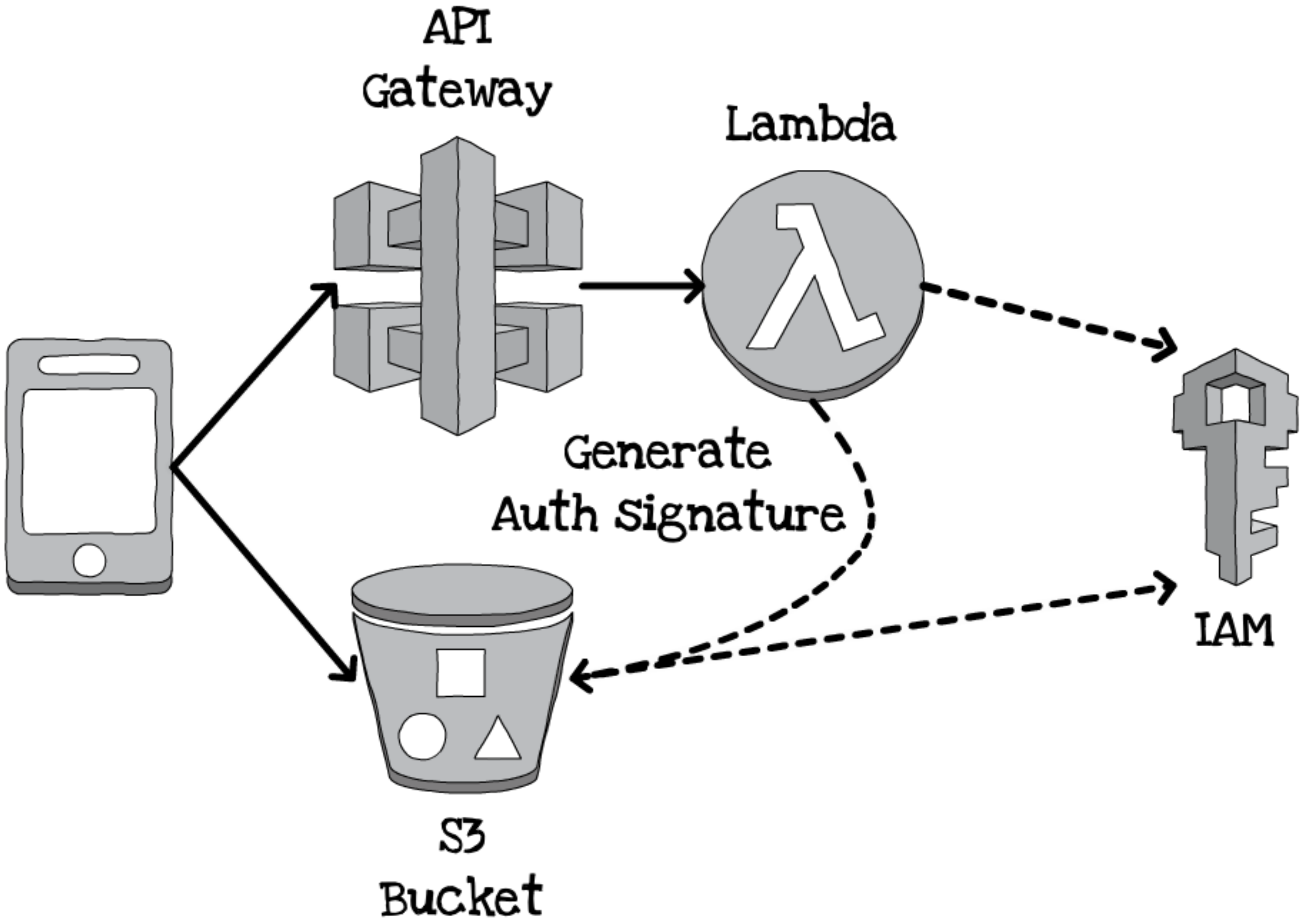
```
sam local generate-event apigateway aws-proxy
```

Give the platform traditional server roles

- Gatekeeper ➤ Distributed Auth
- Scaling point ➤ Containers
- Orchestration ➤ Client or workflow engines

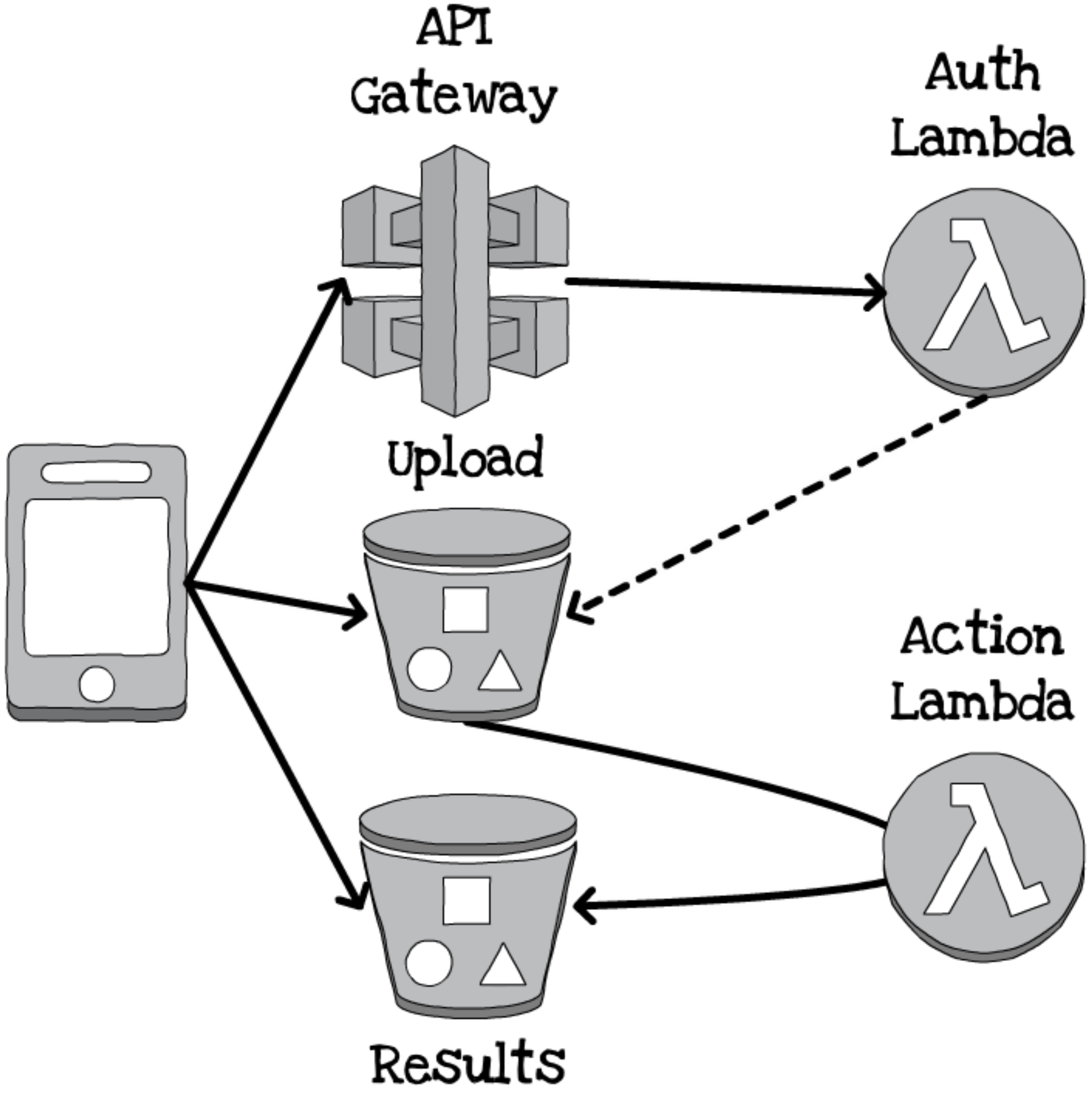
Serverless authentication

- IAM: individual named (internal) services and users
- SIG V4: temporary request grants, using your credentials
- Cognito: anonymous and named (external) users, with own IAM policies


API Gateway
Lambda
Generate Auth Signature
IAM
S3 Bucket

Triggering lambdas from other sources

```
Events:
  FileUpload:
    Type: S3
    Properties:
      Bucket: !Ref UploadBucket
      Events: s3:ObjectCreated:*
```


API Gateway
Auth Lambda
Upload
Action Lambda
Results

Two types of calls

- Synchronous: errors reported back
- Asynchronous: retry 3 times

Dead-letter queues

— fallback when Lambda gives up retrying

```
DeadLetterQueue:
  Type: SNS
  TargetArn: !Ref NotifyAdmins
```

Service integration patterns

- SNS: transient, all consumers get everything, Lambdas auto-scaled
- Kinesis: persistent, sequential, guaranteed max one Lambda per shard
- SQS: persistent, compete with other consumers, Lambdas auto-scaled

Lambda limits

- Max 15 minutes
- No way to keep open connections
- No sticky sessions

Delegate for better latency/length

- Fargate (run autoscale containers but pay per usage)
- Step functions (run programmable workflows for up to 1 year)

How to protect against abuse?

- set usage alerts with Cloudwatch
- set API usage plans (with keys)
- set Lambda concurrency limits (per function/per account)

SAM Benefits

- Atomic deployments for multiple resources
- Version control for infrastructure/wiring
- Integration with AWS code deployment services
- One-click deploy once it's polished
- Local docker-based testing

SAM Downsides

- Very fiddly with templates/transformes
- "Magic" YAML
- No knowledge about platform packaging (NPM)
- No knowledge of language-specific validation
- Good for complex stuff, but painful for simple tasks

Strengths

- Time to deploy minimal
- Time to recover irrelevant
- Multi-versioned
- Forces small, isolated code modules
- Fine-grained, transparent, cost of operation
- Use readily-available services built for massive scale

Weaknesses

- Non-deterministic Latency
- "Only" 99.95% SLA
- No way to keep open connections
- Requires complete rethink on many common practices
- Configuration becomes a challenge

Opportunities

- Skip a generation of technology/process upgrades
- Rethink architectural and operational "best practices"
- Change billing models
- Marketplaces for digital services
- Fine-grained monitoring and optimisation
- A/B testing throughout