A?

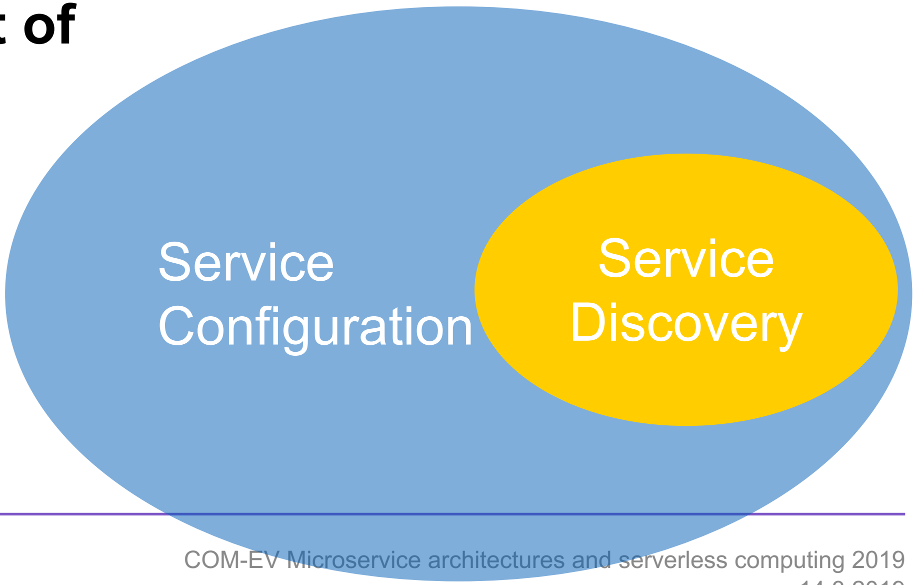
Aalto University
School of Electrical
Engineering

Service Configuration

14.3.2019
Santeri Paavolainen

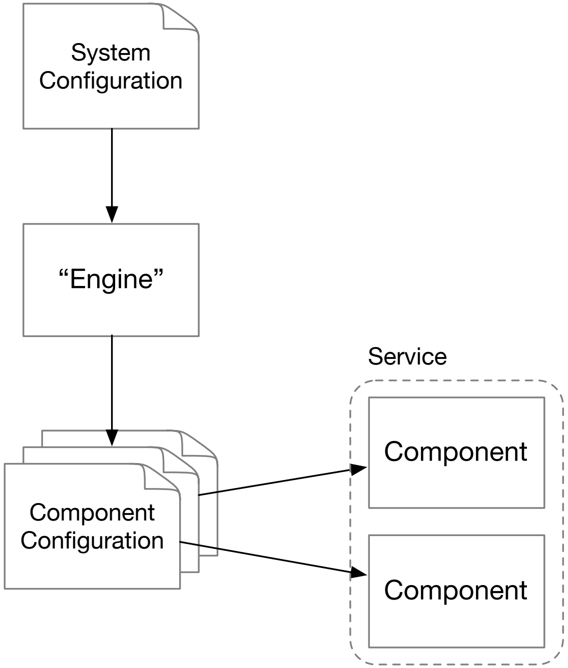
Previously …

- **Discussed service discovery**
  - How to "plumb" the pipes between services
  - Injection, host-based discovery, directory services
- **Discovery is just one aspect of <u>service configuration</u>**
  - E.g. not only about plumbing
  - Settings, secrets, …



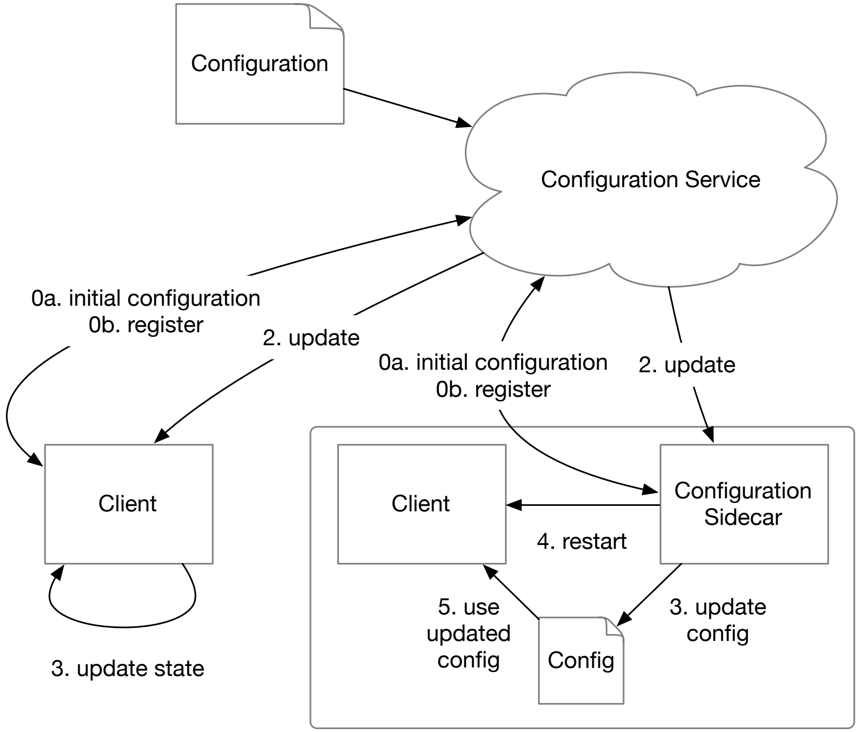
Techniques pretty similar to discovery

- **Static configuration**
  - System deployment
  - Service start



Techniques pretty similar to discovery

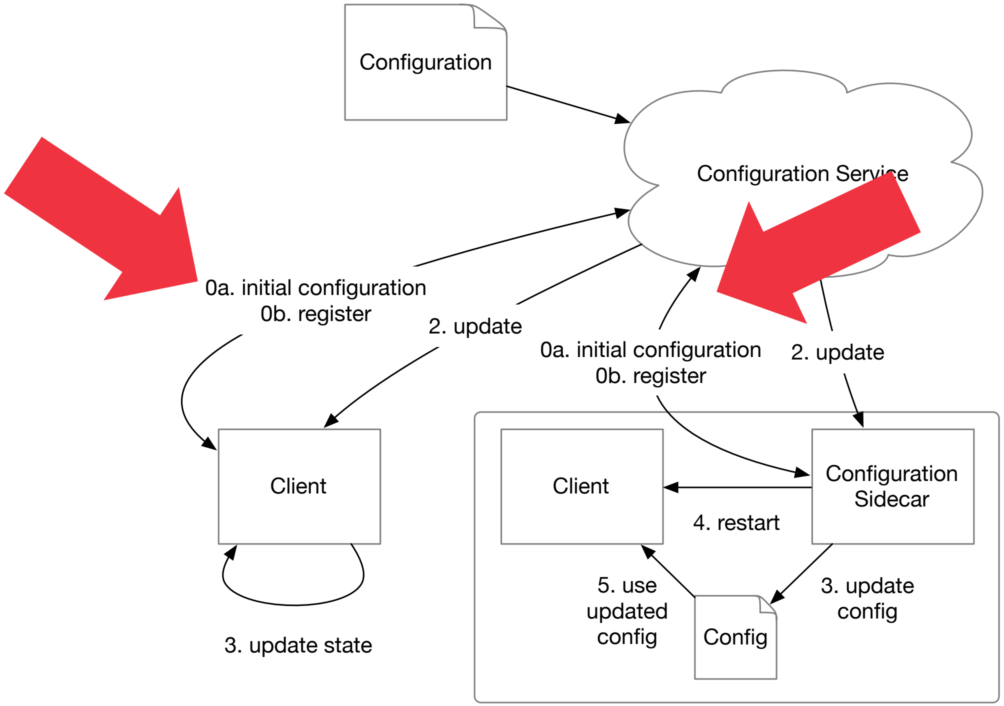
- **Static configuration**
  - System deployment
  - Service start
- **Dynamic configuration**
  - Integrated into service
  - Sidecar managed



Some new considerations

- **Configuration delivery and bootstrapping**
  - Somewhat sidestepped this on discovery …
- **Secrets**
  - Shared secrets (HMACs, JWT tokens, …)
  - Access keys (external services)
    - *(For IaaS internal access, should use service or instance roles instead)*
  - Private keys (TLS server, TLS client authentication)
- **How to handle these?**

Bootstrapping problem



How the node knows where to fetch configuration from?

Bootstrapping

- **Instances e.g. virtual machines**
  - Built into machine image (AMI etc.) – very static and cumbersome to change!
  - ”User script” – inject configuration as a runnable script defined when instance requested (but use cloud-init, see next page)
    - *Almost all machine images have user script support by default*
- **Containers**
  - Build into container image (a bit easier than full machine image)
  - Via environment or configuration script via volume mount

Instance configuration

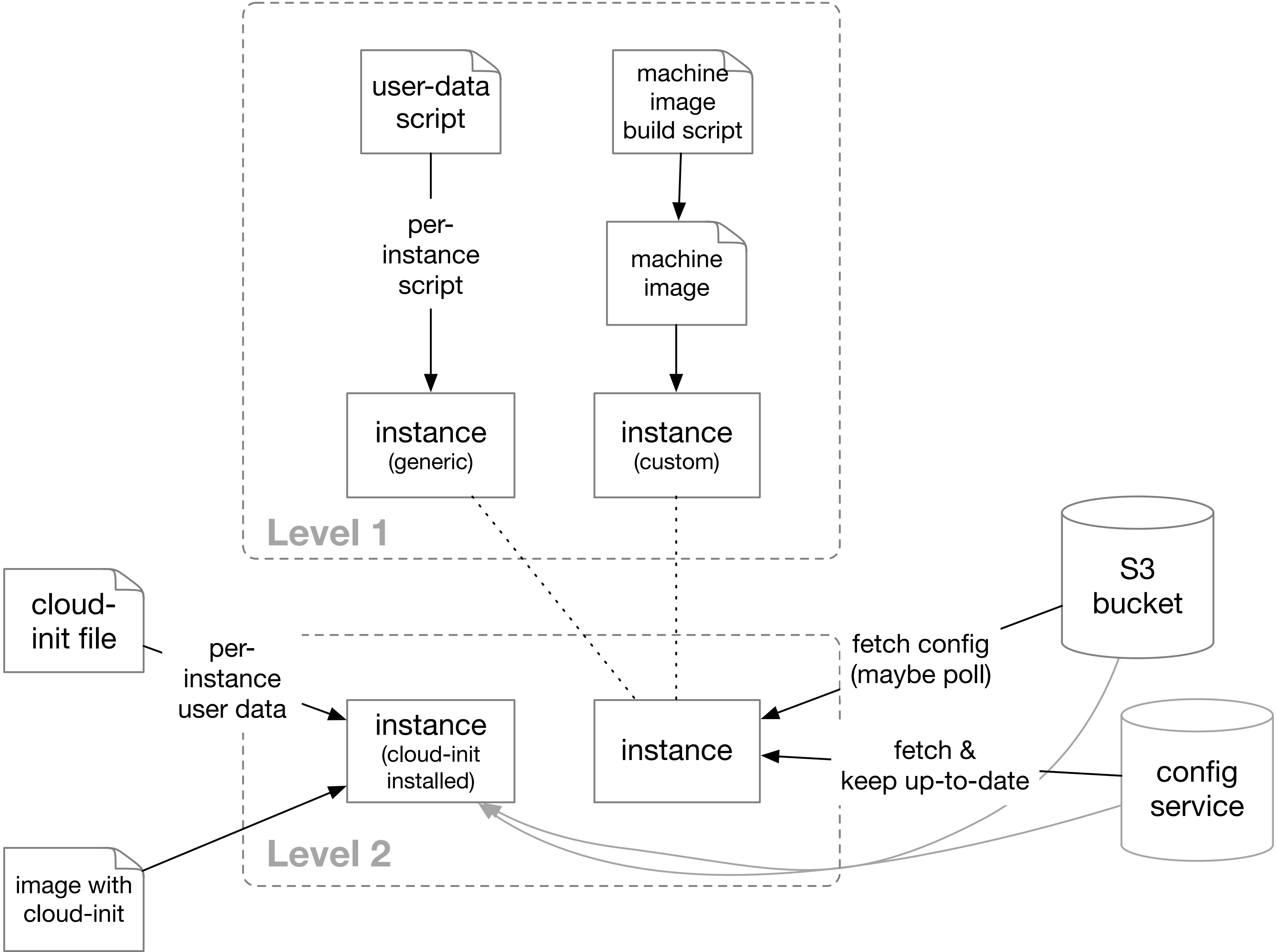
- **Simple bottom line answer: just use cloud-init**
  - Installed in Ubuntu images on AWS, Azure, GCE, ...
  - Support in many other tools (Terraform etc.)
  - Helps avoid many common mistakes
- **Build from there**

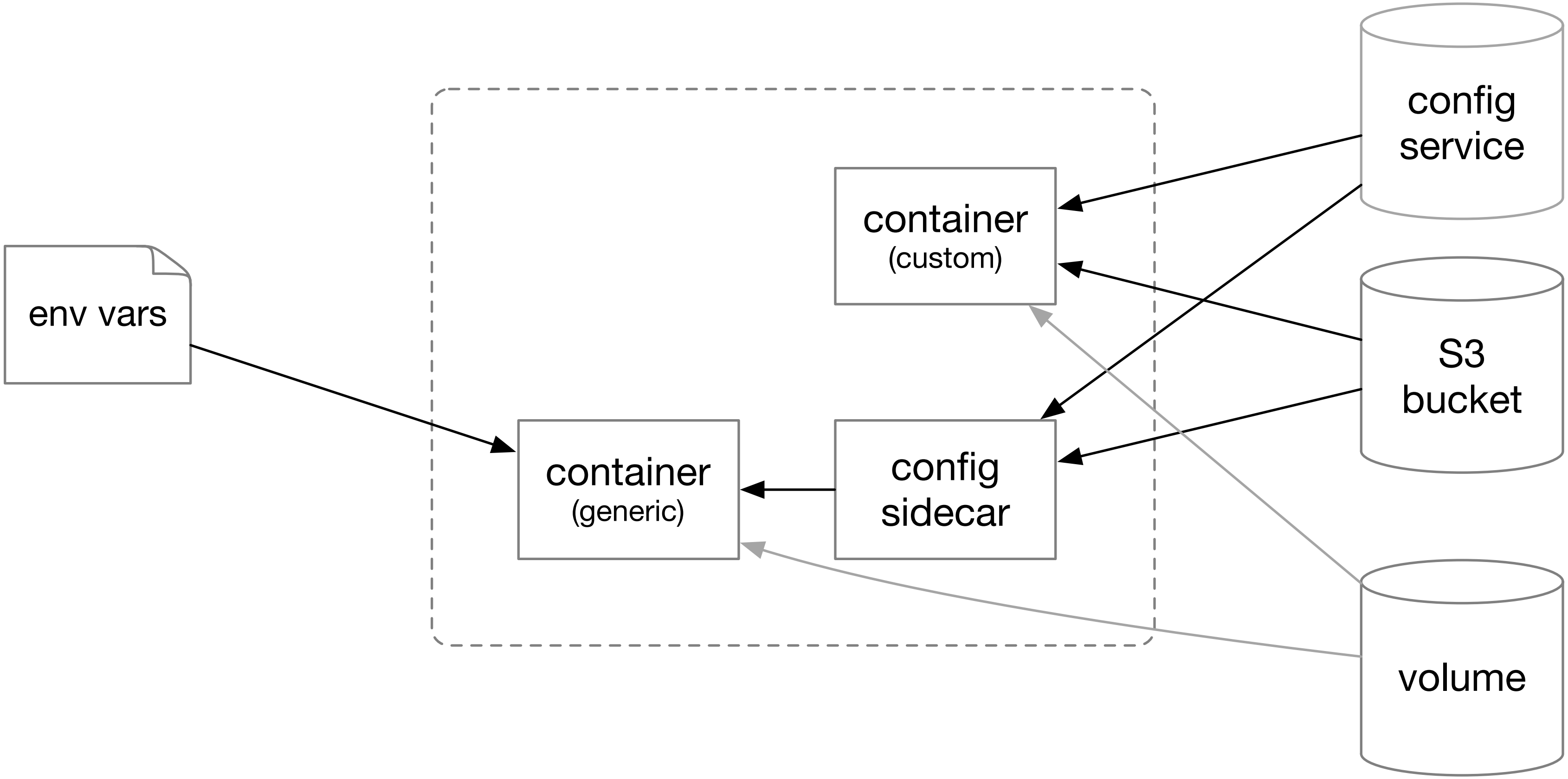
```
# -*- yaml -*-
package_update: true
package_upgrade: true
packages:
  - nfs-common
  - docker.io
swap:
    filename: /swap.img
    size: "auto"
    maxsize: 5373952000
write_files:
  - encoding: b64
    content: CiMgVGhpcyBma...
    owner: root:root
    path: /etc/sysconfig/selinux
    permissions: '0644'
```

Level 2 bootstrapping

- **Bootstrapping can be iterative**
  - First level hardcoded
    `curl http://config.local/config.sh`
    - *config.local different host on different environments*
  - Evaluate second level loader
  - … which does something else
- **Very much like cloud-init**

- **Common sources of 2nd level bootstrap**
  - S3 bucket
  - Separate configuration host or service
  - Often with host-based discovery for the 2nd level location


user-data script
per-instance script
machine image build script
machine image
instance (generic)
instance (custom)
Level 1
cloud-init file
per-instance user data
instance (cloud-init installed)
image with cloud-init
instance
S3 bucket
fetch config (maybe poll)
fetch & keep up-to-date
config service
Level 2


env vars
container
(generic)
container
(custom)
config
sidecar
config
service
S3
bucket
volume

Best practice?

- **Nothing that applies to all cases …**
- **Instances: Use of cloud-init recommended**
  - Custom machine image (AMI) a good idea if lots of commonality between instances – still don't put specific configuration in there, parameterize!
- **Containers: Environment**
  - Works best if number of configuration items low
  - If complex configuration, sidecar pattern preferable
  - Bootstrap sidecar config via environment

Dynamic configuration

- **Facebook and Google extreme examples**
  - Feature flags dynamically enable/disable functionality
    `if (feature_x_enabled) { … } else { … }`
  - Feature flags are dynamically configurable (via some directory)
  - Multivariate flags: on/off based on complex criteria
- **Potentially change large portions of service functionality without code changes or redeployment**
  - We'll come back to "dark launches" later on deployments
- **(Not without its own problems)**

Secrets and sensitive information

- **What are ”secrets”?**
  - Cloud infrastructure and 3rd party service access keys
  - Keys used for HMAC and encryption (signed session token)
  - Passphrases for asymmetric cryptography private keys (e.g. TLS)
    - *For any other kind of keystore (Java, Bitcoin, …)*
  - On-disk encryption keys
- **“Secrets” are runtime information**
  - Should never go into actual service code or configuration
  - Injected only when service started, or pulled in as needed

Secret management approaches

- **Simplest: inject at instantiation**
  - E.g. have separate deployment repository (limited access?)
  - Secrets injected as user script, environment, etc.
  - Problems: user script, env etc. visibility and accessibility (by others)
- **Inject via orchestration**
  - Kubernetes secrets
- **Separate service**
  - AWS KMS (Key Management System), Azure Key Vault, Google KMS
- **Extreme end is hardware-based systems (PKCS#11)**
  - Key material never leaves hardware enclosure
  - Limited for signatures, encryption and decryption

Typical KMS usage

1. **Create managed key → key identifier**
2. **Encrypt secret data using key**
   - The actual key does not leave KMS!
   - E.g. "encrypt 'supasekrit' with key id 1234' → '7ab76dfe67af77'"
3. **Put encrypted secret into configuration (plus key id)**
   - via environment, user script, directory service etc.
4. **Decrypt secret using key**
   - E.g. "decrypt '7ab76dfe67af77' with key id 1234 → 'supasekrit'"
   - KMS checks whether requestor has permissions to use the key

- **Details vary … (f. ex. direct integration to other cloud services)**

Access keys (within IaaS)

- **Cloud provider APIs accept access keys (id + secret)**
  - `AWS_ACCESS_KEY_ID=… AWS_SECRET_ACCESS_KEY=… aws ec2 run-instances …`
  - Possible to pass these via previously mentioned configuration methods
- **Not recommended to pass access keys directly**
- **Use instance (or container) roles instead**
  - Create a role that has required rights
  - Assign the role to runtime resource
  - Resource can now use APIs as the role!
  - (ok actually not that simple, see documentation)

Summary

- **Service configuration ⊇ service discovery**
  - Share many of the tools (etcd, zookeeper, consul, …)
- **Methods vary from static injection to dynamic configuration**
  - Applicability depends on requirements and constraints
- **Management of sensitive information (“secrets”)**
  - Operational security aspects
  - Separation of secrets from code, also KMS tools
  - Local cloud provider’s access keys