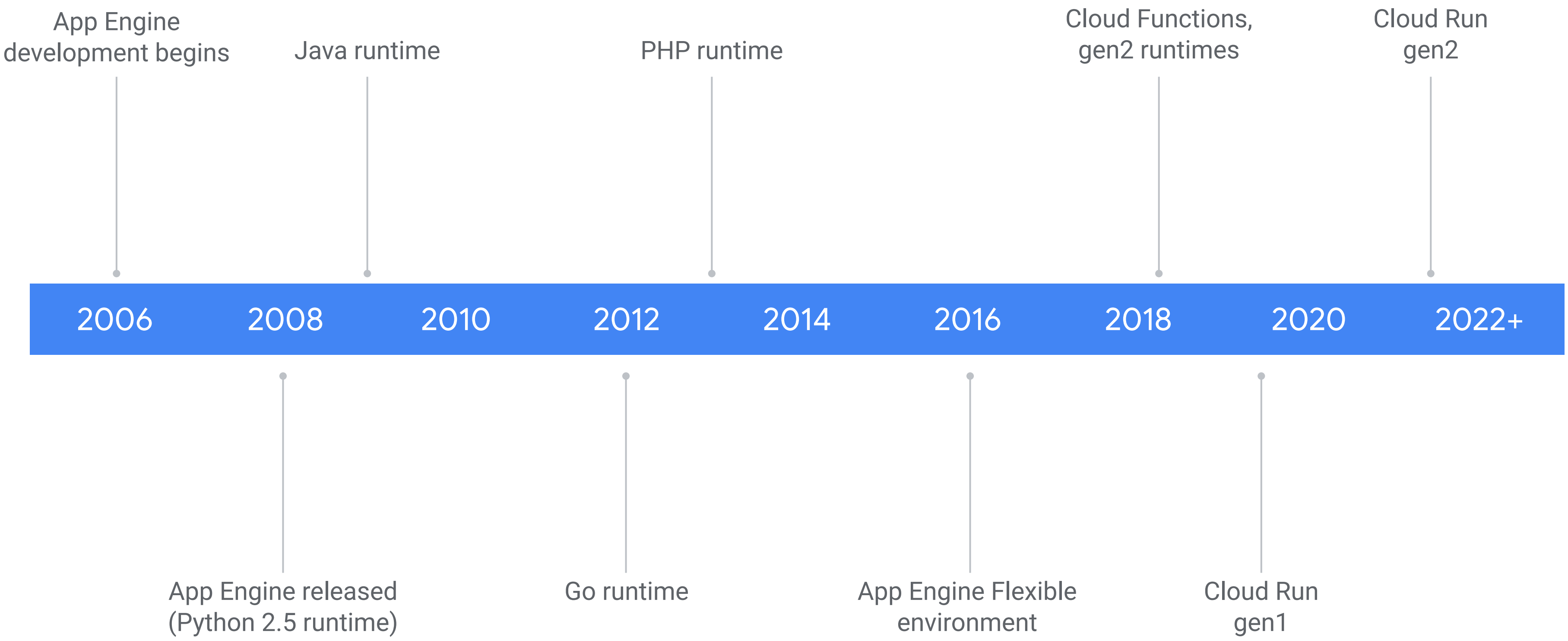
Serverless Platforms

Tradeoffs and Consequences

Dave Bailey / December 2021

Serverless at Google : 15 years and counting



App Engine development begins

App Engine released (Python 2.5 runtime)

Java runtime

Go runtime

PHP runtime

App Engine Flexible environment

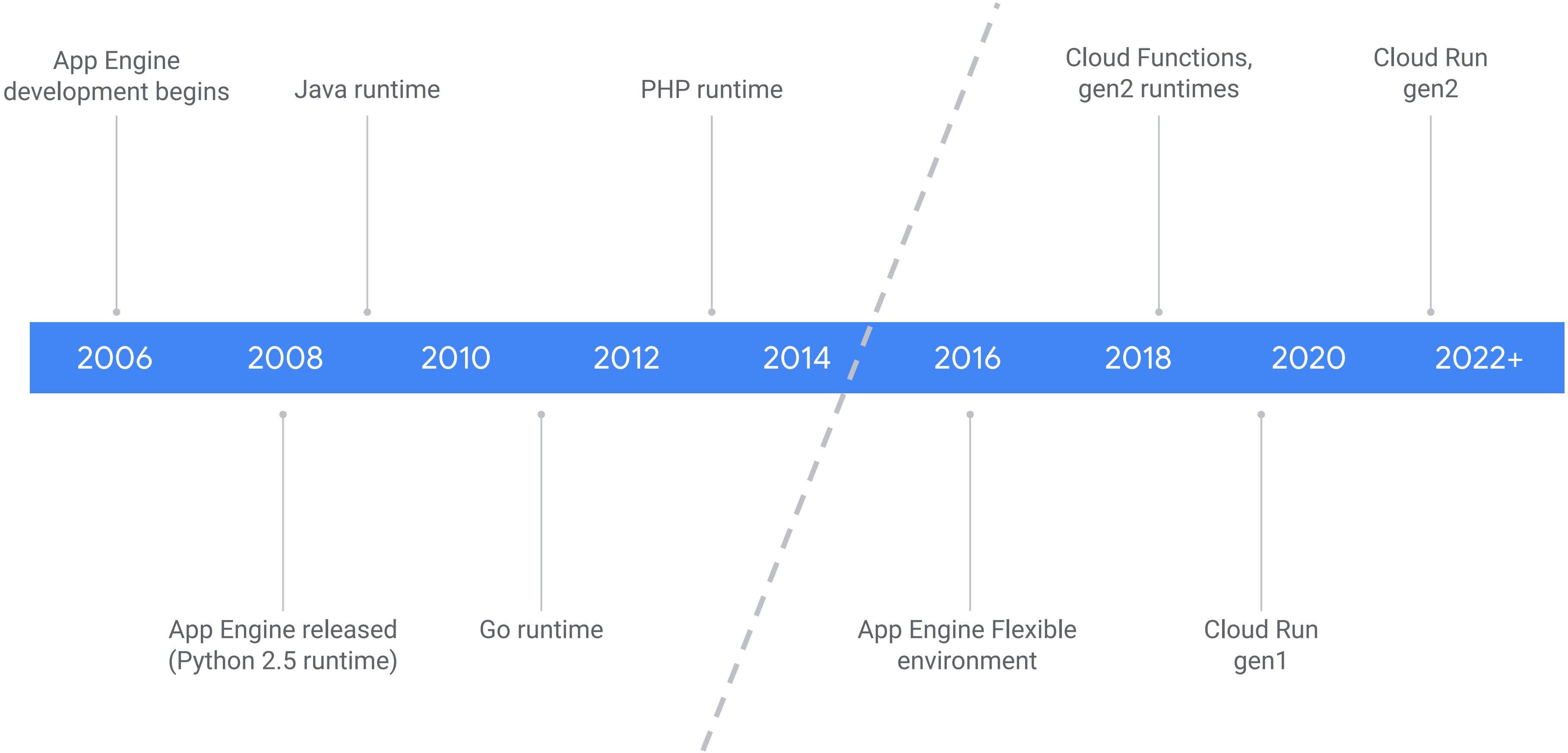
Cloud Functions, gen2 runtimes

Cloud Run gen1

Cloud Run gen2

2006 2008 2010 2012 2014 2016 2018 2020 2022+

Specialized runtimes → Container-based execution



App Engine development begins

Java runtime

PHP runtime

Cloud Functions, gen2 runtimes

Cloud Run gen2

2006 2008 2010 2012 2014 2016 2018 2020 2022+

App Engine released (Python 2.5 runtime)

Go runtime

App Engine Flexible environment

Cloud Run gen1

An earlier version of this evolution… going in the opposite direction.

Let’s go back to the 1990’s…

- CGI: fork/exec, env vars / stdin / stdout
- Apache: pre-forked free pool of child processes
- mod_perl: preload Perl interpreter and selected packages into pre-forked child processes

CGI scripts introduced the cold start problem

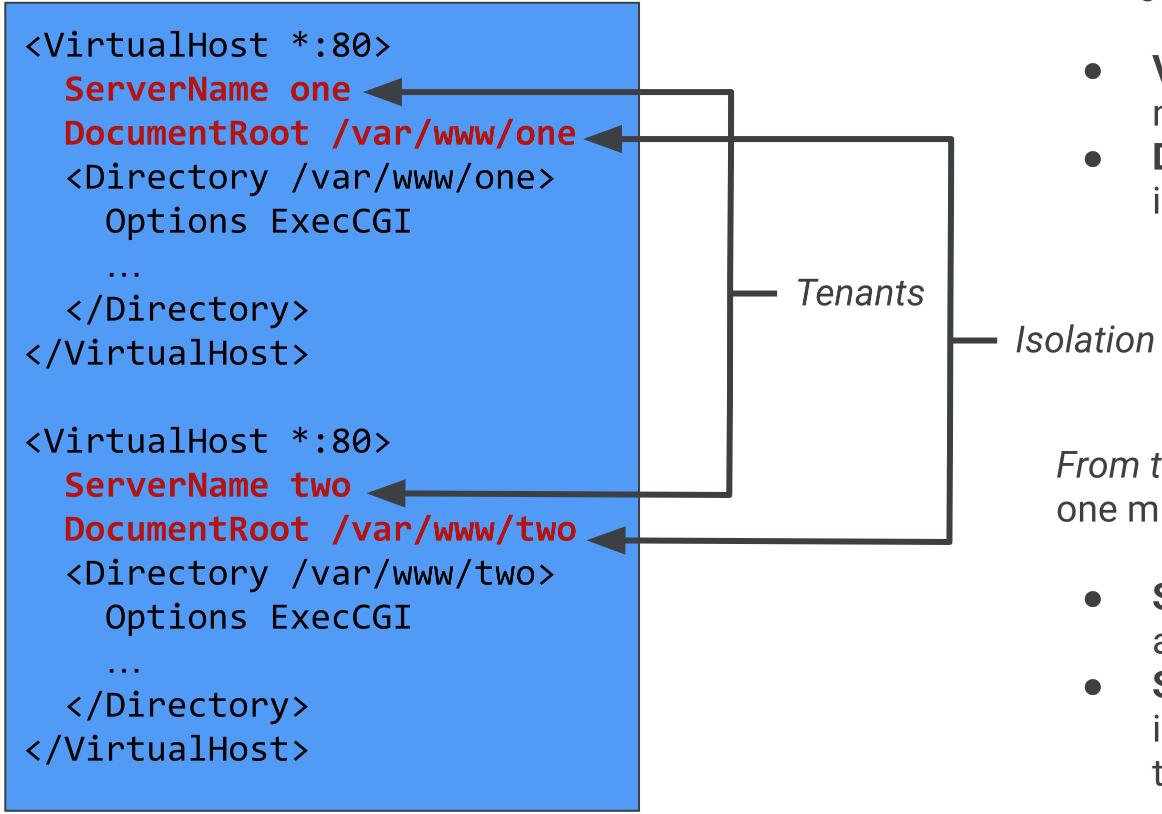
- Every request is a cold start (execution of an arbitrary program)
- This becomes a problem with interpreted languages, which take longer to start

Two recipes materialized to address this:

- Embed the language runtime in the HTTP server (NSAPI, Servlet API, …)
- Split the language runtime into its own long-lived process (FastCGI)

Both involve specialization of the execution environment

- The web server becomes part of an “opinionated platform”
- This specialization enables agility (faster startup, lower CPU and memory usage).
- This is the backdrop against which App Engine was created.
- “A scalable container” - Guido van Rossum (App Engine emeritus)



```
<VirtualHost *:80>
  ServerName one
  DocumentRoot /var/www/one
  <Directory /var/www/one>
    Options ExecCGI
    …
  </Directory>
</VirtualHost>

<VirtualHost *:80>
  ServerName two
  DocumentRoot /var/www/two
  <Directory /var/www/two>
    Options ExecCGI
    …
  </Directory>
</VirtualHost>
```

Let's go back to the '90's one more time…

- **VirtualHost** provided a form of multi-tenancy for web hosting.
- **DocumentRoot** provided a form of isolation between tenants

From this starting point, to build a platform, one must add:

- **Security**: provide effective sandboxing around mutually untrusting workloads.
- **Scale**: support large # of tenants, large individual tenant size, rapid changes in tenant resource usage.

Security for Serverless workloads

- Many sandboxing options out there. Some are specific to particular workloads, some are not.
- Some tradeoff between the level of isolation, and the overhead* of the sandbox.

Dedicated machines

- Putting the “server” in “Serverless”?
- Typically slowest to provision
- Best isolation

IaaS VMs

- Provision in O(1 minute)
- Very good isolation
- Also relevant: core scheduling

Virtualized sandboxes

- gVisor, crosvm…
- Optimized for high density, fast startup (100 ms to 1 second setup time)
- Good security isolation, fair performance isolation

OS level isolation (namespaces, seccomp, jail)

- Even higher density
- 10-100 ms setup time
- Variable security and performance isolation

Runtime level isolation

- v8::Isolate (very fast setup time: less than 10 ms)
- java.lang.SecurityManager (nontrivial CPU cost)

* overhead means a lot of things: memory overhead, CPU overhead, and/or provisioning overhead

Scaling Serverless Platforms

- Scaling to a **large number of tenants** (more specifically: high tenant density)
  - Inevitable consequence of “Serverless” billing models (pay for what you use)
- Supporting **large individual tenants** (for example, rapid redeployment)
  - Rapid application delivery (image pulling / mounting)
  - Traffic migration (perhaps uncomfortably fast)
- Handling **rapid bursts of load**
  - Some predictable, some not
  - Balance queueing with overshoot
  - Instance concurrency limits make this harder

Tenant density and agility are more challenging as the platform becomes more generic

Concrete example: FaaS vs. CaaS

- FaaS optimizations: shared base layers, pre-spun instances ⇒ low node affinity, high agility
- CaaS challenge: the 10+ GB container image ⇒ high node affinity, low agility

Operating Serverless Platforms

Debugging applications

- Debugging can be challenging: common tools (ssh, gdb, …) may not be available.
- Less ability for customer to diagnose issues ⇒ higher support load.
- Billing model affects this (e.g. issues caused by throttle-while-idle).
- Tendency to overwhelm dependencies ⇒ scaling-driven feedback loops.

Updating applications (security patches, etc)

- Highly opinionated platforms ⇒ security patches are easier to auto-apply.
- Can we replace base layers? Sometimes (need library compatibility).
- Can we rebuild the container? Sometimes (need source code).
- Did the update work? Not always clear… “it compiles” may not suffice.

Grading Serverless

Security

- Many sandboxes and tenancy models.
- Less dependent on specialization.
- Good progress in the past five years.

Grade: B

Scaling

- Reasonably good with specialized platforms.
- Fair to poor with more generic platforms.
- Recent progress improving image pulls.

Grade: C

Operating

- Debugging leaves a lot to be desired.
- Not much of a story around auto-updates.
- Need investment here to drive adoption.

Grade: D

Future Developments

Service Mesh

- Should enable better tracing (e.g. to identify overloaded dependencies).
- Would like to see Redis, MySQL and others participating in the mesh.
- Should also enable customers to mix and match execution environments.
- Example: develop on IaaS platform, deploy to a K8S cluster, migrate to S8S.

Software Supply Chain

- Need application source code and dependencies, to enable auto-updates.
- Container builds need to be hermetic and reproducible.
- Expect considerable effort invested in this area going forward.
- Expect that effort to drive Serverless adoption in the years to come.

Questions