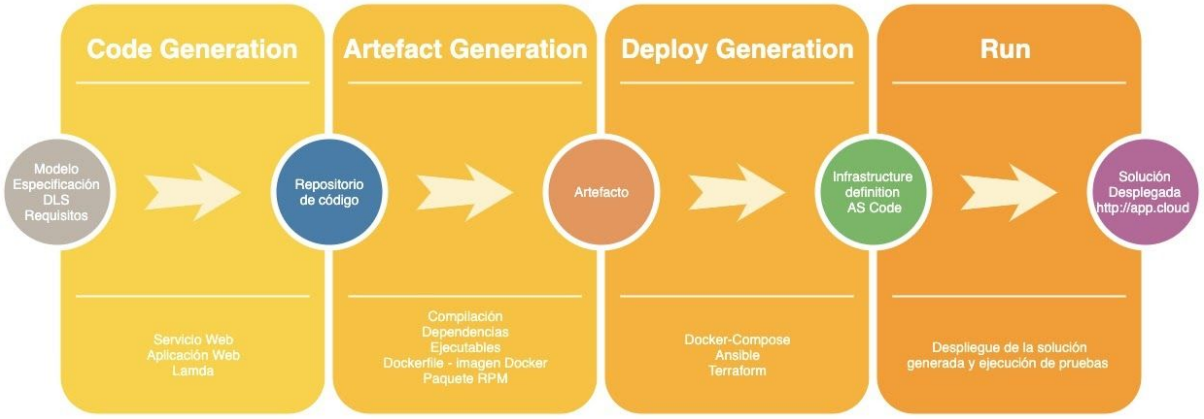
Cloud-Generative-Manager

(Code-Runner o ya veremos)

En el ámbito de estudio de la programación generativa y las arquitecturas basadas en servicios http/api-rest, se propone como caso de estudio la implementación de un pipeline generativo de soluciones web. El objetivo sería cubrir todo el ciclo de vida de este tipo de aplicaciones desde un aspecto generativo y automatizado. El siguiente diagrama aterriza el concepto propuesto, dividiendo en etapas el proceso generativo que se desea implementar:



1º- Code Generation:

Generación del código asociado a la solución http que se desea alcanzar:

- Web Estática (html + css)
- Servicio: Backend REST API
- Web Dinámica: Frontal Dinámico + Backend REST API
- Lambda: Fragmento de código ejecutado en la Nube

El objetivo de esta fase es la generación de código de la solución especificada, así como la creación de un repositorio de código donde se publicaría. Este repositorio será utilizado en la siguientes etapas del pipeline. En esta fase, una propuesta de input podría ser la siguiente:

nature: backend-api
bussiness_model: <fichero de especificación o DLS>
technology:

- Golang
- Python
- Node

peristence_layer:
- null
- MongoDB
- Mysql
- MemDB

path: /api/_bussinesmodel
port: 8000

El fichero de especificación _bussinesmodel establecería el modelo de datos del servicio, así como las relaciones entre los modelos (si las hubiera), por ejemplo para generar un servicio que manejara profesores y alumnos, una primera aproximación sería:

```
entities:
  - id: profesor
    description: texto descriptivo
    attributes:
      - first_name: Rubén
      - departament: Programación Generativa
  - id: alumno
    description: texto descriptivo
      - first_name: Hugo
      - dni: 77777777
  relations:
      - id: profesor_alumno
        type: one_to_many
```

Otro planteamiento podría ser jerárquico:

```
model:
    -   id: asignatura
        description: texto descriptivo
        attributes:
            - title: programación generativa
        contains:
            model:
              - id: profesor
                description: texto descriptivo
                attributes:
                    - first_name: Pepe
            model:
              - id: profesor
                description: texto descriptivo
```

```
                attributes:
                    - first_name: Iván
```

El resultado sería el código necesario para ejecutar operaciones básicas sobre estos modelos de datos vía REST. De manera adicional, se generaría las herramientas de construcción necesarias para compilar, instalar, testear y ejecutar en local dicha solución, generando por ejemplo un Makefile. Toda esta producción quedaría almacenada en un repositorio de código. Se utilizará un proveedor de repositorios de código en la nube como por ejemplo https://github.com/

2º- Artefact Generation

En esta etapa se utilizará el código generado en la etapa de anterior para construir un artefacto autocontenido con todo lo necesario para ejecutar la solución generada.

Para ello, las aplicaciones generadas han de ir acompañadas de una especificación definiendo el conjunto de acciones comunes a todas las aplicaciones generadas por el sistema para continuar con la siguiente etapa del pipeline.Podría denominarse profile.env de la aplicación generada:

```
task:
   _build: < make build | ./configure.sh | go build >
   _test: <make test | python test.py | go test >
   _run: < make run | ./binary.linux | node server.js >
env:
    -   DB_URI: <config>
    -   CONFIG_DIR: <config>
    -   PASSWORD: <config>
```

En este profile.env se modelan las operaciones que podemos llevar a cabo con el código generado, operaciones como compilar, ejecutar pruebas unitarias, generar ejecutable, ejecutar la aplicación o configurar lo necesario para la posterior ejecución del producto software. Llegados a este punto, sería posible soportar como entrada un repositorio de código NO generado por este sistema en la etapa 1, para ello, el repositorio de código deberá contener un profile.env y ser acorde a las capacidades de este sistema. Es necesario dotar de más semántica al deploy.env para saber las partes que componen la aplicación. La naturaleza de los artefactos generados podría ser:

- Dockerfile o Imagen Docker
- Ejecutable
- RPM
- Tar.gz

- Configuración (xml, yml, json)

Estos artefactos serán aprovechados en la siguiente etapa del pipeline y almacenados en el repositorio de entrada o publicados en la cloud para su posterior uso como por ejemplo Docker Hub

3º- Deploy Generation

En esta etapa se utilizará el pofile.env junto con el artefacto generado en la anterior etapa para generar una definición de la infraestructura virtual. Esta definición ha de contener todo lo necesario para desplegar la solución sobre un entorno virtual. La aproximación en esta etapa sería utilizar contenedores (docker-compose) para representar la infraestructura virtual necesaria. Por ejemplo, una definición de infraestructura virtual generada podría ser:

```
version: '2'

networks:
  internalnet

services:
  generated_service:
    image: <imagen generada en la segunda etapa>
    command: [" ./ejecutable --config <configuración generada en la segunda etapa>"]
    ports:
      - 8000
    links:
      - mongodb
    networks:
      internalnet:
        aliases:
          - generated_service.internal
    depends_on:
      - mongodb
  mongodb:
    image: mongo:3.4.11
    networks:
      - internalnet
```

En esta definición de infraestructura se relaciona el servicio generado con la capa de persistencia (mongoDB por ejemplo) que se definió en la primera etapa.

4º- RUN

Esta etapa consiste en la ejecución de la definición de infraestructura virtual producida en la etapa 3, ejecutando el docker-compose sobre una máquina virtual alojada en la nube. El resultado sería una url tal que http://cloud.generative.manager/api/_bussinesmodel:8080 donde se podría probar el servicio generados. La idea es gestionar la operativa de la solución generada desde la interfaz del sistema.

Objetivos

El primer objetivo sería conseguir implementar el pipeline generativo. El sistema estaría compuesto por una interfaz de usuario donde gestionar las soluciones generadas (el repositorio de código), así como cada una de las etapas por las que pasa el pipeline para ofrecer soluciones End-2-End. Cada una de las etapas sería implementada por un servicio web distinto, favoreciendo así el desacoplamiento de las partes del pipeline. Como punto de partida se soportaría un tipo de de solución sencilla, como por ejemplo una web estática, el objetivo es implementar las 4 etapas.

El siguiente objetivo (o iteración) sería el de integrar nuevos generadores para soportar diversos tipos de tecnologías, distintos tipos de modelos de negocio (transformaciones a modelo de datos), distintos DSL´s, capas de persistencia, estilos, interfaces…
El resultado es una factoría de código generativa y automatizada.

Consideraciones

Especificación Etapa 1

Para las distintas soluciones que el sistema soporte, la especificación o dls de la etapa 1 variarán, por ejemplo, si el tipo de solución es una web estática la especificación se reduciría al siguiente ejemplo, ya que no se producirían modelo de datos, no se necesitaría capa de persistencia, y la tecnología sería html:

```
  nature: static-web
  port: 80
```

Esta especificación generaría la estructura de una web estática:

```
html
  index.html
assets/
  images/
    logo.png
```

```
  stylesheets/
    base.css
javascripts/
  index.js
```

El artefacto generado sería un simple fichero comprimido (un tar.gz) con la estructura de directorios, y el dockerfile simplemente montaría el árbol de la web en el directorio esperado por el servidor de aplicación:

```
FROM httpd:2.4
COPY ./html/ /usr/local/apache2/htdocs/
```

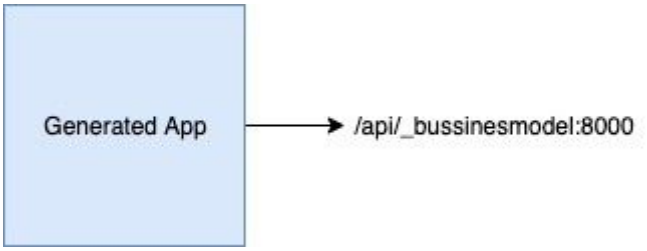
Lo ideal sería comenzar con aplicaciones sencillas e ir evolucionando el sistema hasta completar un catálogo de aplicaciones complejas y soluciones reales.

EXTENSIONES

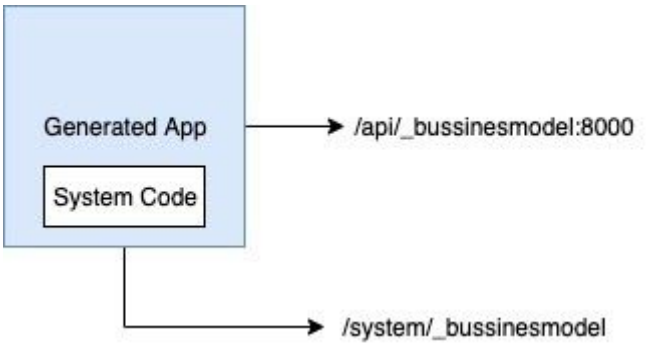
Una vez alcanzados los objetivos propuestos (pipeline End2End de diversas soluciones, en diversos lenguajes) se podría trabajar en las siguientes mejoras o dar lugar a nuevos estudios.

Interfaz sistema de las soluciones generadas

El resultado inicial del pipeline sería una url tal que http://cloud.generative.manager/api/_bussinesmodel:8080 donde probar la solución generada. Es decir:



La extensión que se propone sería la de incluir, en la etapa de generación de código, todo lo necesario para servir una api interna o de sistema de manera común a todas las soluciones generadas, sirviendo end-points que den cierta información sobre la solución:



Los end-points de la interfaz /system/_bussinesmodel podrían servir información como:

- /system/_bussinesmodel/version → versión del software
- /system/_bussinesmodel/config → configuración del sistema
- /system/_bussinesmodel/metrics → ciertas métricas, como número de peticiones atendidas, fallidas, uptime

La idea es aprovechar en la medida de lo posible esta interfaz para obtener información de las soluciones generadas y desplegadas y servir esta información desde la interfaz de usuario del sistema generativo. Incluso las aplicaciones con esta interfaz podrían consumir recursos comunes al dominio cloud, comportamiento de plataforma.

Incluir 5º etapa de Validación y Verificación

Incluir una nueva etapa de generación y ejecución de pruebas. La idea sería generar pruebas automáticas que ejecutar contra las aplicaciones generadas después de desplegarlas, validando así que la api generada se comporta de manera correcta. Para ello, los productos software generados deben ir acompañados de una rigurosa documentación de como ser consumidos. El problema está en la diversidad de la naturaleza de las aplicaciones que el sistema consiga generar.

Mejoras en la Etapa 3: Infrastructure AS Code

Esta etapa mejoraría adoptando estrategias como blue-green-deployment para producir código de herramientas como https://www.ansible.com/, https://www.terraform.io/, https://puppet.com/ y ejecutando despliegues reales de infraestructura, como por ejemplo máquinas virtuales, servicios de bases de datos gestionados y demás soluciones cloud empresariales reales. Esta mejora es demasiado ambiciosa.