Informe de Pruebas de Carga

Grupo 05

Miguel Bueno González
Ainhoa Carbajo Orgaz
Marcos De Diego Martín
Paula Negro Asegurado

Índice

1. Introducción

En esta práctica, se ha diseñado una página web de un supermercado habitual para poder evaluar sobre ella diferentes medidas ejecutando pruebas de carga y análizandolas con Locust y Grafana.

El objetivo de la práctica es identificar cuellos de botella, tiempos de respuesta y la capacidad del sistema para escalar ante un aumento significativo de usuarios o transaccionesa.

2. Diseño de la web

A continuación se describirá brevemente el proceso de diseño de la web

2.1. Elección de la propuesta

La página web representará un supermercado habitual con opción a compra y funcionalidades habituales en las webs de este tipo de servicios como: regsitro y acceso al portal privado, consulta de los productos del catálogo y proceso de compra online.

2.2. Desarrollo inicial

Para implementar la interfaz de la página web se han creado una serie de documentos html para las siguientes vistas de la aplicación:

- Una vista de registro y otra de login.
- Una página principal con un un menú.
- Una vista de selector de productos.
- Una vista para visualizar todos los artículos que están actualmente en el carrito.
- Una vista de todos los pedidos realizados por el usuario.

2.3. Implementación de la API

Además se ha implementado una API con la ayuda de Flask para llevar a cabo lafuncionalidad del supermercado.

```
nuevaApp/
├── run.py
├── locustfile.py
├── requirements.txt
└── app/
    ├── __init__.py
    ├── db.py
    ├── config.py
    ├── inicio.py
    ├── registro.py
    ├── menu.py
    ├── productos.py
    ├── carrito.py
    ├── pedidos.py
    ├── metrics.py
    ├── routes.py
    ├── static/
    │   ├── estilo.css
    │   └── estilo2.css
    └── templates/
        ├── inicio.html
        ├── registro.html
        ├── menu.html
        ├── productos.html
        ├── carrito.html
        ├── checkout.html
        ├── pedidos.html
        ├── detalles_pedido.html
        └── editarProducto.html
```

Descripción de la estructura:

- **app**/: Contiene la lógica de negocio del servidor Flask. Cada fichero representa un módulo funcional, como productos, pedidos o carrito.
- **templates**/: Plantillas HTML que renderiza Flask en las vistas.
- **static**/: Archivos estáticos como hojas de estilo (.css).
- **Archivos raíz** (como `run.py`, `locustfile.py`): Scripts de arranque y pruebas, fuera del paquete principal.

app/: Contiene toda la lógica de negocio y organización del servidor Flask. Cada módulo se encarga de una parte funcional (carrito, pedidos, productos).
templates/: Archivos HTML que se renderizan desde las rutas de Flask para mostrar la interfaz al usuario.

static/: Archivos estáticos como hojas de estilo (.css) que se aplican a las plantillas HTML.
Archivos raíz (run.py, locustfile.py, etc.): Scripts de arranque o pruebas, fuera del paquete app/. **db.py :**

```
    load_dotenv()

def get_connection():
    return pymysql.connect(
            host="localhost",
            user="admin2",
            password="123412341234.",
            database="tienda_comida",
            cursorclass=pymysql.cursors.DictCursor  # para obtener resultados como diccionario
        )


def close_connection(e=None):
    db = g.pop('db', None)
    if db is not None:
        db.close()
```

get_connection(): Crea una nueva conexión a la base de datos tienda_comida con credenciales específicas. Devuelve un cursor en forma de diccionario (DictCursor), lo que facilita acceder a los resultados por nombre de columna.
close_connection(): Cierra una conexión almacenada temporalmente en g.db, si existe. Esta función puede ser usada en los teardown handlers de Flask para liberar recursos al terminar una petición. **inicio.py :**

```
    inicio_routes = Blueprint("inicio", __name__)

@inicio_routes.route("/inicio", methods=["GET"])
def mostrar_inicio():
    return render_template("inicio.html")


@inicio_routes.route('/inicio', methods=['POST'])
def login():
    data = request.get_json()
    username = data.get("username")
    password = data.get("password")

    conn = get_connection()
    cursor = conn.cursor()
    cursor.execute("SELECT id FROM usuarios WHERE nombre=%s AND contrasena_hash=%s",
                   (username, password))
    user = cursor.fetchone()
    if user:
        session["user_id"] = user["id"]
        session["username"] = username
        return jsonify({"mensaje": "Login exitoso"}), 200
    return jsonify({"error": "Credenciales invalidas"}), 401
```

Muestra el formulario de login (inicio.html) al usuario. Procesa el login:
Recibe un JSON con username y password.
Verifica si hay un usuario con esas credenciales en la tabla usuarios.

Si es válido, guarda user_id y username en la sesión (session) para mantener la autenticación.
Si no es válido, responde con un error 401.

run.py :

```
app = create_app()

if __name__ == '__main__':
    app.config['DEBUG'] = True
    app.config['ENV'] = 'development'  # Esto asegura que Flask corra en modo de desarrollo
    app.run(debug=True, host='0.0.0.0')
```

Llama a create_app() que configura e inicia la app y sus blueprints.
Ejecuta el servidor en modo debug (útil para desarrollo) y lo expone en host='0.0.0.0' para permitir conexiones externas.

registro.py :

```
@bp.route('/registro', methods=['POST'])
def registro():
    data = request.get_json()
    username = data.get("username")
    password = data.get("password")
    email = data.get("email", f"{username}@fake.com")

    if not username or not password:
        return jsonify({"error": "Faltan datos"}), 400

    try:
        conn = get_connection()
        cursor = conn.cursor()
        cursor.execute(
            "INSERT INTO usuarios (nombre, email, contrasena_hash) VALUES (%s, %s, %s)",
            (username, email, password)
        )
        conn.commit()
        session["user_id"] = cursor.lastrowid
        session["username"] = username
        return jsonify({"mensaje": "Usuario registrado"}), 201
    except Exception as e:
        if "Duplicate" in str(e):
            return jsonify({"error": "Usuario ya existe"}), 409
        return jsonify({"error": "Error interno"}), 500
    finally:
        cursor.close()
        conn.close()
```

Este endpoint permite registrar un nuevo usuario. Toma un username, password y opcionalmente un email, y crea un nuevo registro en la base de datos usuarios. Si el nombre de usuario ya existe, devuelve error 409. Si el registro tiene éxito, se guarda la sesión del usuario y se responde con un mensaje de confirmación.

```
@bp.route('/registro', methods=['GET'])
def mostrar_registro():
    return render_template('registro.html')
```

Este endpoint devuelve la plantilla HTML del formulario de registro. Se usa cuando el usuario accede por navegador para visualizar la página de alta de cuenta.

productos.py :

```
@productos_routes.route("/productos", methods=["GET"])
def mostrar_productos():
    return render_template("productos.html")
```

Este método devuelve la vista productos.html, que permite al usuario visualizar los productos del catálogo desde el navegador. Se activa cuando se accede a la URL /productos mediante una solicitud GET. Es una ruta pensada para la navegación web tradicional, no para el consumo por clientes API.

```
@productos_routes.route("/productos", methods=["GET"])
def listar_productos():
    try:
        categoria = request.args.get("categoria")
        conn = get_connection()
        cursor = conn.cursor()

        if categoria:
            cursor.execute("SELECT * FROM productos WHERE stock > 0
     AND categoria = %s", (categoria,))
        else:
            cursor.execute("SELECT * FROM productos WHERE stock >
    0")

        productos = cursor.fetchall()
        cursor.close()
        return jsonify(productos)
    except Exception as e:
        return jsonify({"error": str(e)}), 500
```

Este endpoint proporciona una lista de productos disponibles en formato JSON. Si se proporciona un parámetro categoria, filtra los productos por dicha categoría. Solo devuelve productos con stock mayor a 0. Es útil para aplicaciones frontend o scripts que consumen la API y necesitan acceder dinámicamente al catálogo de productos.

```
@productos_routes.route("/productos/<int:producto_id>", methods=["
    GET"])
def detalle_producto(producto_id):
    try:
        conn = get_connection()
        cursor = conn.cursor()
        cursor.execute("SELECT * FROM productos WHERE id = %s", (
    producto_id,))
        rows = cursor.fetchall()
        cursor.close()
        return jsonify(rows)
    except Exception as e:
        return jsonify({"error": str(e)}), 500
```

Este endpoint permite obtener los detalles de un producto concreto a partir de su ID. Es muy útil en interfaces dinámicas donde al hacer clic sobre un producto se desea mostrar su información detallada sin recargar toda la página.

carrito.py :

```
@carrito_routes.route("/carrito/agregar", methods=["POST"])
def agregar_al_carrito():
    try:
        user_id = get_user_id()
        if not user_id:
            return jsonify({"error": "No has iniciado sesion"}),
    401

        data = request.get_json()
        producto_id = data.get("producto_id")
        cantidad = data.get("cantidad", 1)

        if not producto_id:
            return jsonify({"error": "Falta producto_id"}), 400

        carrito_id = obtener_o_crear_carrito(user_id)

        conn = get_connection()
        cursor = conn.cursor()

        cursor.execute("""
            SELECT id FROM carrito_productos
            WHERE carrito_id = %s AND producto_id = %s
        """, (carrito_id, producto_id))
        row = cursor.fetchone()

        if row:
            cursor.execute("""
                UPDATE carrito_productos
                SET cantidad = cantidad + %s
                WHERE carrito_id = %s AND producto_id = %s
            """, (cantidad, carrito_id, producto_id))
        else:
            cursor.execute("""
                INSERT INTO carrito_productos (carrito_id,
    producto_id, cantidad)
                VALUES (%s, %s, %s)
            """, (carrito_id, producto_id, cantidad))

        conn.commit()
        cursor.close()
        conn.close()
        return jsonify({"mensaje": "Producto agregado al carrito"})
    , 200
    except Exception as e:
        import traceback
        traceback.print_exc()
        return jsonify({"error": f"Error interno: {str(e)}"}), 500
```

Este endpoint agrega un producto al carrito del usuario. Si el producto ya estaba en el carrito, actualiza su cantidad. Si no estaba, lo inserta. La función también se encarga de crear el carrito si no existía previamente. Es un pilar fundamental del flujo de compra de la aplicación.

```
@carrito_routes.route("/carrito/vaciar", methods=["DELETE"])
def vaciar_carrito():
    try:
        user_id = get_user_id()
        if not user_id:
```

```
            return jsonify({"error": "No has iniciado sesion"}),
    401

        carrito_id = obtener_o_crear_carrito(user_id)

        conn = get_connection()
        cursor = conn.cursor()
        cursor.execute("DELETE FROM carrito_productos WHERE
    carrito_id = %s", (carrito_id,))
        conn.commit()
        cursor.close()
        return jsonify({"mensaje": "Carrito vaciado"}), 200
    except Exception as e:
        import traceback
        traceback.print_exc()
        return jsonify({"error": "Error interno"}), 500
```

Este endpoint elimina todos los productos del carrito del usuario actual. Es útil tanto si el usuario decide cancelar la compra como si desea empezar de nuevo con un carrito limpio. También se utiliza al finalizar una compra (checkout).

pedidos.py :

```
@pedidos_routes.route("/pedidos", methods=["GET"])
def listar_pedidos():
    user_id = session.get("user_id")
    if not user_id:
        return jsonify({"error": "No has iniciado sesion"}), 401

    try:
        conn = get_connection()
        cursor = conn.cursor()
        cursor.execute("SELECT * FROM pedidos WHERE usuario_id = %s
     ORDER BY fecha_pedido DESC", (user_id,))
        pedidos = cursor.fetchall()
        cursor.close()
        return jsonify(pedidos)
    except Exception as e:
        return jsonify({"error": str(e)}), 500
```

Este endpoint lista todos los pedidos que ha realizado el usuario autenticado. Devuelve un listado ordenado por fecha (del más reciente al más antiguo) y permite a los usuarios consultar el historial de sus compras.

```
@pedidos_routes.route("/pedidos/<int:pedido_id>", methods=["GET"])
def detalle_pedido(pedido_id):
    user_id = get_user_id()
    if not user_id:
        return jsonify({"error": "No has iniciado sesion"}), 401

    conn = get_connection()
    cursor = conn.cursor()

    cursor.execute("SELECT * FROM pedidos WHERE id = %s AND
    usuario_id = %s", (pedido_id, user_id))
    pedido = cursor.fetchone()
    if not pedido:
        cursor.close()
        return jsonify({"error": "Pedido no encontrado"}), 404

```

```
    cursor.execute("""
        SELECT pp.producto_id, p.nombre, pp.cantidad, pp.precio_unitario
        FROM pedido_productos pp
        JOIN productos p ON pp.producto_id = p.id
        WHERE pp.pedido_id = %s
    """, (pedido_id,))
    productos = cursor.fetchall()

    cursor.close()

    return jsonify({
        "pedido": pedido,
        "productos": productos
    })
```

Este método devuelve los detalles completos de un pedido específico, incluyendo los productos que lo componen, sus cantidades y precios. Es útil para visualizar resúmenes de pedidos pasados o generar comprobantes.

```
@pedidos_routes.route("/checkout", methods=["POST"])
def checkout():
    user_id = get_user_id()
    if not user_id:
        return jsonify({"error": "No has iniciado sesion"}), 401

    conn = get_connection()
    cursor = conn.cursor()

    cursor.execute("""
        SELECT cp.producto_id, cp.cantidad, p.precio
        FROM carrito_productos cp
        JOIN carritos c ON cp.carrito_id = c.id
        JOIN productos p ON cp.producto_id = p.id
        WHERE c.usuario_id = %s
    """, (user_id,))
    items = cursor.fetchall()

    if not items:
        cursor.close()
        return jsonify({"error": "El carrito esta vacio"}), 400

    total = sum(item["cantidad"] * float(item["precio"]) for item in items)

    time.sleep(2)  # Simula el procesamiento del pago

    cursor.execute("INSERT INTO pedidos (usuario_id, total) VALUES (%s, %s)", (user_id, total))
    pedido_id = cursor.lastrowid

    for item in items:
        cursor.execute("""
            INSERT INTO pedido_productos (pedido_id, producto_id, cantidad, precio_unitario)
            VALUES (%s, %s, %s, %s)
        """, (pedido_id, item["producto_id"], item["cantidad"], item["precio"]))

        cursor.execute("""
            UPDATE productos
```

```
            SET stock = stock - %s
            WHERE id = %s
        """, (item["cantidad"], item["producto_id"]))

    cursor.execute("""
        DELETE cp FROM carrito_productos cp
        JOIN carritos c ON cp.carrito_id = c.id
        WHERE c.usuario_id = %s
    """, (user_id,))

    conn.commit()
    cursor.close()

    return jsonify({
        "mensaje": "Compra realizada correctamente",
        "pedido_id": pedido_id
    })
```

Este es uno de los endpoints más importantes. Se encarga de procesar el pago (simulado con time.sleep(2)), generar un nuevo pedido, registrar sus productos asociados, actualizar el stock y vaciar el carrito del usuario. Representa la culminación del flujo de compra.

locustfile.py :

```
def generar_usuario_aleatorio():
    nombre = "user" + ''.join(random.choices(string.ascii_lowercase
     + string.digits, k=6))
    contrasena = "test1234"
    return nombre, contrasena
```

Esta función genera dinámicamente un nombre de usuario aleatorio concatenando el prefijo üserçon una cadena de 6 caracteres aleatorios (letras y dígitos). Se devuelve junto a una contraseña fija para simular el registro de un nuevo usuario en cada ejecución de prueba.

```
class UsuarioSimulado(HttpUser):
    wait_time = between(1, 3)

    def on_start(self):
        self.username, self.password = generar_usuario_aleatorio()
        self.registrar_usuario()
        self.login()
```

Esta clase simula el comportamiento de un usuario que interactúa con la interfaz web. Al comenzar (on_start), se genera un usuario nuevo que inmediatamente se registra y hace login. El parámetro wait_time simula un tiempo de espera aleatorio entre peticiones para imitar una navegación humana más realista.

```
    def registrar_usuario(self):
        email = f"{self.username}@test.com"
        payload = {
            "username": self.username,
            "password": self.password,
            "email": email
        }
        headers = {'Content-Type': 'application/json'}

```

```
        with self.client.post("/registro", json=payload, headers=
    headers, catch_response=True) as response:
            if response.status_code == 201:
                response.success()
                print(f"[OK] Registrado: {self.username}")
            else:
                response.failure(f"Fallo en registro: {response.
    status_code}")
```

Este método realiza la petición POST al endpoint /registro simulando que el usuario se registra en la plataforma. Si la respuesta tiene un código 201, se considera un registro exitoso. La opción catch_response=True permite marcar manualmente la petición como exitosa o fallida.

```
    def login(self):
        payload = {
            "username": self.username,
            "password": self.password
        }
        headers = {'Content-Type': 'application/json'}

        with self.client.post("/inicio", json=payload, headers=
    headers, catch_response=True) as response:
            if response.status_code == 200:
                response.success()
                print(f"[OK] Sesion iniciada con {self.username}")
            else:
                response.failure("Login fallido")
                print(f"[ERROR] Fallo login con {self.username}")
```

Este método inicia sesión con el usuario recién creado haciendo un POST al endpoint /inicio. En caso de éxito (código 200), se considera que la sesión fue iniciada correctamente y se imprime un mensaje en consola.

```
    @task(2)
    def ver_menu(self):
        self.client.get("/menu", name="GET /menu")

    @task(3)
    def ver_productos(self):
        self.client.get("/productos", name="GET /productos")

    @task(2)
    def ver_carrito(self):
        self.client.get("/carrito", name="GET /carrito")
```

Estos métodos simulan la navegación por la interfaz gráfica del sitio. Cada @task(n) indica la frecuencia relativa con la que se ejecutará esa acción en relación con las demás. Se accede al menú, a la vista de productos y al carrito de compras.

```
    def obtener_producto_id(self):
        return random.randint(13, 50)

    @task(1)
    def agregar_carrito(self):
        producto_id = self.obtener_producto_id()
        self.client.post("/carrito/agregar", json={
```

```
            "producto_id": producto_id,
            "cantidad": 1
        }, name=f"POST /carrito/agregar")
```

El método obtener_producto_id selecciona aleatoriamente un ID de producto válido. Luego, agregar_carrito envía una petición POST al endpoint /carrito/agregar para añadir dicho producto al carrito del usuario. La cantidad por defecto es 1.

```
    @task(1)
    def procesar_compra(self):
        self.agregar_carrito()
        self.client.post("/checkout", json={}, name="POST /checkout
    ")
```

Esta tarea simula el flujo de compra completo: primero se añade un producto al carrito y luego se realiza una petición POST al endpoint /checkout para procesar la compra. Se envía un JSON vacío porque el backend ya asocia el carrito con el usuario autenticado.

```
    @task(1)
    def ver_pedidos(self):
        self.client.get("/pedidos", name="GET /pedidos")
```

2.4. Base de Datos

De forma complementaria se ha realizado una base de datos con mysql para representar las principales entidades necesarias y poder ejecutar las consultas que supondrán las pruebas de carga. La base de datos cuenta con las siguientes entidades que se describirán de forma resumida a continuació:

```

CREATE TABLE `usuarios` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `nombre` varchar(100) NOT NULL,
  `email` varchar(100) NOT NULL,
  `contraseña_hash` varchar(255) NOT NULL,
  `fecha_registro` datetime DEFAULT current_timestamp(),
  PRIMARY KEY (`id`),
  UNIQUE KEY `email` (`email`)
)


CREATE TABLE `carritos` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `usuario_id` int(11) DEFAULT NULL,
  `fecha_creacion` datetime DEFAULT current_timestamp(),
  PRIMARY KEY (`id`),
  KEY `usuario_id` (`usuario_id`),
  CONSTRAINT `carritos_ibfk_1` FOREIGN KEY (`usuario_id`)
    REFERENCES `usuarios` (`id`) ON DELETE CASCADE
)


CREATE TABLE `carrito_productos` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `carrito_id` int(11) DEFAULT NULL,
```

```
  `producto_id` int(11) DEFAULT NULL,
  `cantidad` int(11) NOT NULL,
  PRIMARY KEY (`id`),
  KEY `carrito_id` (`carrito_id`),
  KEY `producto_id` (`producto_id`),
  CONSTRAINT `carrito_productos_ibfk_1` FOREIGN KEY (`carrito_id`)
    REFERENCES `carritos` (`id`) ON DELETE CASCADE,
  CONSTRAINT `carrito_productos_ibfk_2` FOREIGN KEY (`producto_id`)
     REFERENCES `productos` (`id`) ON DELETE CASCADE
)


CREATE TABLE `pedido_productos` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `pedido_id` int(11) DEFAULT NULL,
  `producto_id` int(11) DEFAULT NULL,
  `cantidad` int(11) NOT NULL,
  `precio_unitario` decimal(10,2) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `pedido_id` (`pedido_id`),
  KEY `producto_id` (`producto_id`),
  CONSTRAINT `pedido_productos_ibfk_1` FOREIGN KEY (`pedido_id`)
    REFERENCES `pedidos` (`id`) ON DELETE CASCADE,
  CONSTRAINT `pedido_productos_ibfk_2` FOREIGN KEY (`producto_id`)
    REFERENCES `productos` (`id`) ON DELETE CASCADE
)


CREATE TABLE `pedidos` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `usuario_id` int(11) DEFAULT NULL,
  `fecha_pedido` datetime DEFAULT current_timestamp(),
  `total` decimal(10,2) DEFAULT NULL,
  `estado` varchar(50) DEFAULT 'pendiente',
  PRIMARY KEY (`id`),
  KEY `usuario_id` (`usuario_id`),
  CONSTRAINT `pedidos_ibfk_1` FOREIGN KEY (`usuario_id`) REFERENCES
     `usuarios` (`id`) ON DELETE CASCADE
)


CREATE TABLE `productos` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `nombre` varchar(100) NOT NULL,
  `descripcion` text DEFAULT NULL,
  `precio` decimal(10,2) NOT NULL,
  `stock` int(11) NOT NULL,
  `categoria` varchar(50) NOT NULL DEFAULT 'Sin categoría',
  PRIMARY KEY (`id`)
)
```

2.4.1. Usuarios

Almacena el nombre, email, fecha de registro y un hash de la contraseña de los usuarios registrados.

2.4.2. Productos

Almacena un identificador único, el nombre, descripción, precio y cantidad en stock de los productos añadidos a la tienda. En este caso 11.

2.4.3. Pedidos

Guarda un identificador de cada pedido además de la fecha de realización, el total del coste y el estado. Además guarda el id del usuario que lo realizó como clave foránea.

2.4.4. Carritos

Esta tabla almacena todos los carritos existentes por los usuarios registardos mediante un id, la fecha de creación y el id del usuario correspondiente.

2.4.5. Extras

Finalmente se almacenan dos tablas que sirven para la relación entre los productos asociados a un pedido y los productos asociados a cada carrito.

- **pedido_productos** contiene un identifiador único, la cantidad y el precio unitario, relacionando un identificador de un pedido con uno de un producto que está contenido en él.
- **carrito_productos** guarda de la misma forma los identificadores y la cantidad d eun producto añadido a un carrito.

Esta base de datos se ha realizado con la intención de facilitar las pruebas de carga que se realizarán sobre la página web especificando el comportamiento en cuanto a cantidad de usuarios, carritos, etc

3. Métricas y Análisis Pruebas de Carga

3.1. Métricas de red

3.1.1. Conexiones de red activas

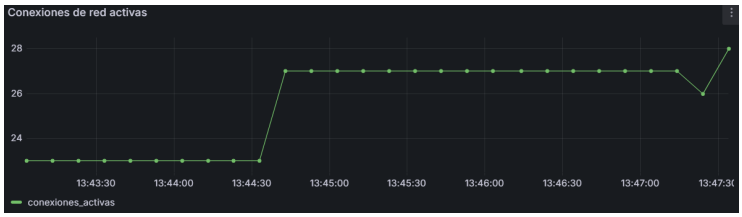

Figura 1: *Conexiones de red activas*

La gráfica anterior muestra la evolución del número de conexiones activas en la máquina virtual durante un periodo de aproximadamente 8 minutos.
Se puede observar lo siguiente:
El sistema comienza con 29 conexiones activas de forma estable.
Aproximadamente a las 19:22:30, se produce una caída repentina hasta 24 conexiones, seguida de una bajada gradual hasta alcanzar un mínimo de 20 conexiones activas alrededor de las 19:24:30.
A partir de ahí, se observa una recuperación progresiva de conexiones hasta estabilizarse nuevamente en torno a 26.

Finalmente, en los últimos minutos del registro, se presenta un incremento sostenido, alcanzando las 33 conexiones activas al final del periodo.
Este comportamiento esta relacionado con los accesos de usuarios y procesos automáticos que se conectan o desconectan de la red. El patrón observado indica un entorno dinámico en el que el número de conexiones varía con el tiempo, influido por tareas programadas o actividad del sistema.

Figura 2: *Conexiones de red activas*

Durante una prueba de carga es esperable que el número de conexiones de red activas en la máquina virtua aumente significativamente. Cada usuario simulado por Locust suele mantener su propia sesión HTTP, lo que implica una conexión TCP individual al servidor. Por lo tanto, si se simulan 500 usuarios concurrentes, se podrían establecer hasta 500 conexiones TCP simultáneas entre los clientes virtuales y el servidor Flask. Este incremento en las conexiones activas puede observarse en la siguiente gráfica. Durante la fase de prueba las conexiones activas aumentan rápidamente y se mantengan elevadas mientras dure la prueba. Una vez finalizada la prueba, el número de conexiones disminuye gradualmente a medida que se cierran las sesiones.
Es importante destacar que, aunque cada usuario mantiene una conexión persistente, la reutilización de conexiones puede variar dependiendo de la configuración del cliente HTTP utilizado por Locust y del comportamiento de la aplicación.

3.1.2. Latencia

El script diseñado en Bash tiene como objetivo medir la latencia de la red a través de la ejecución periódica de pings hacia el servidor DNS de Google (8.8.8.8). El proceso es el siguiente:
Obtenemos la latencia mediante el comando *ping*, que se ejecuta tres veces consecutivas (ping -c 3), y luego extraemos el valor de la latencia promedio usando *awk* y *cut*.
El valor de latencia obtenido se almacena en una base de datos MariaDB (latencia_red) junto con el timestamp correspondiente, lo que permite realizar un seguimiento del comportamiento de la red a lo largo del tiempo.

El script ejecuta este proceso cada 10 segundos, lo que permite tener una medición frecuente y precisa de la latencia de la red.

Figura 3: *Latencia*

La gráfica anterior muestra la evolución de la latencia de red en milisegundos (ms) durante un período de aproximadamente 24 minutos. El eje X representa el tiempo y el eje Y la latencia medida en milisegundos.
Se observan algunos puntos clave del comportamiento de la latencia:
- Fluctuaciones constantes: La latencia se mantiene en un rango relativamente bajo, entre 3.85 ms y 4.25 ms, lo cual es indicativo de una red estable.
- Pico a las 19:43:00: Se puede observar un pico de latencia de alrededor de 4.2 ms, lo que podría estar relacionado con un evento puntual de congestión en la red, como una actualización o aumento en la cantidad de tráfico.
A pesar de los picos esporádicos, la latencia generalmente se mantiene estable y dentro de un rango pequeño, lo que indica que la red está funcionando de manera eficiente. Para mostrar una gráfica de la latencia, se espera obtener

Figura 4: *Latencia*

un claro pico en la misma en respuesta al aumento de procesos en curso en la MV. está gráfica no resulta tan explicativa si no la ponemos en contexto. La

pruebas de carga se han realizado de 12:42 a 12:44 y de 12:51 a 12:55. Si nos fijamos, los picos más alto se alcanzan a las 12:43:50 aproximadamente ya las 12:52, es decir durante los últimos instantes de cada prueba de carga en el que aumenta la cantidad de request y por tanto aumenta la latencia (ver gráficas locust). Esto es esperable aunque los picos sucesivos pueden indicar problemas de recuperación aunque nada tan indicativo como para asegurar que la máquina sufre una saturación persistente.

3.1.3. Otras Métricas de Red

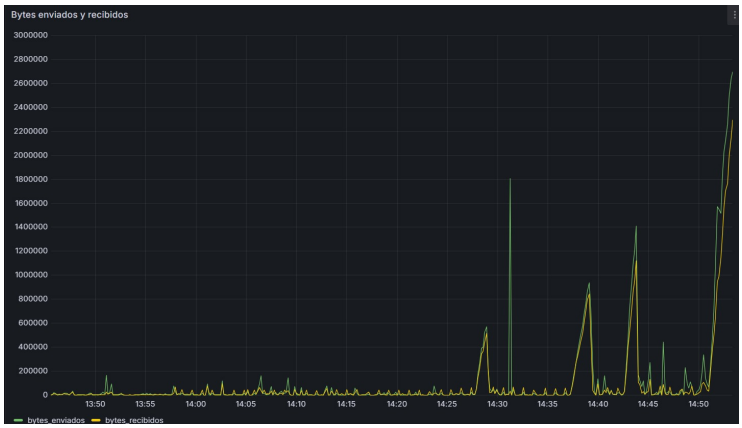

Figura 5: *Cantidad de Paquetes*

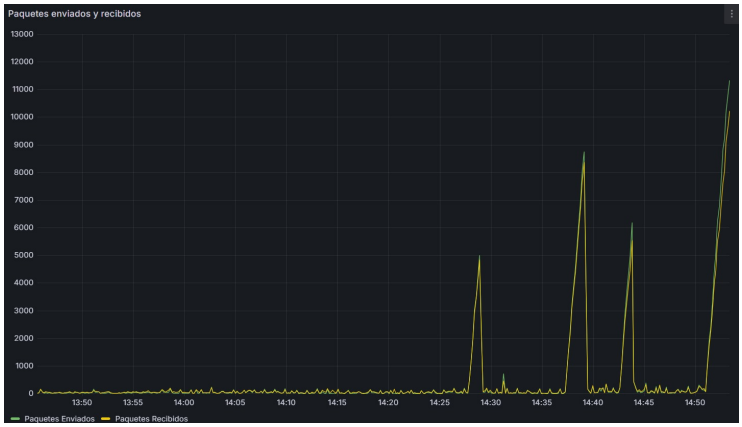

Figura 6: *Bytes enviados y recibidos*

Las dos gráficas anteriores muestran la cantidad de paquetes y bytes enviados y recibidos antes y durante la realización de las pruebas de carga. Ambas tienen a simple vista una estructura similar pues los picos altos se corresponden con el alto numero de solicitudes enviadas a la 'página web, que aumenta exponencialmente con el número de usuarios simulados. Así mismo como en otras ocasiones los picos inferiores o de nivel 0 pueden mostrar la inactividad ocasionada debido

a fallos (la tasa de fallos de las pruebas era del 20) o a periodos de inactividad o de pausa de las pruebas.

3.2. Metricas de Negocio

A continuación se muestran unas gráficas obtenidas a partir de los datos recogidos por el propio locust que permiten visualizar el comportamiento de los usuarios simulados en el locustfile descrito anteriormente, estas gráficas no son realmente relevantes para comprobar el rendimiento de la máquina virtual pero sí interesantes desde un punto de vista económico y de gestión de nuestro supermercado.

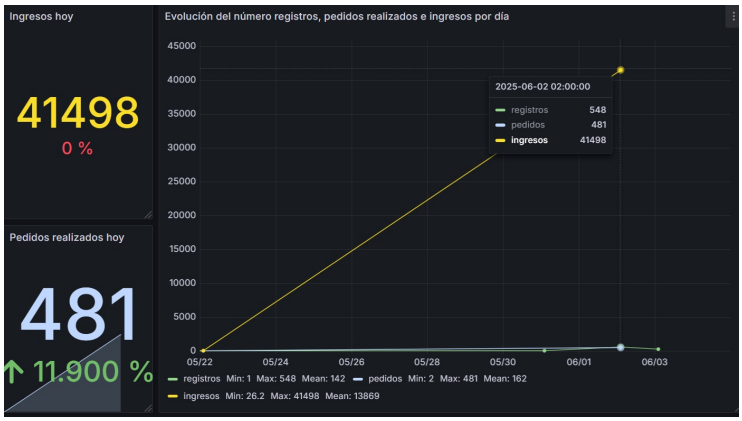

Figura 7: *Cantidad de Pedidos*

Este primer dashboard muestra la visualización de la cantidad de pedidos realizados el día de una de las pruebas de carga, así como los ingresos. La gráfica muestra un incremento lineal en los registros en ingresos que simula la prueba de carga con el registro de 548 usuarios en la página web de ventas.

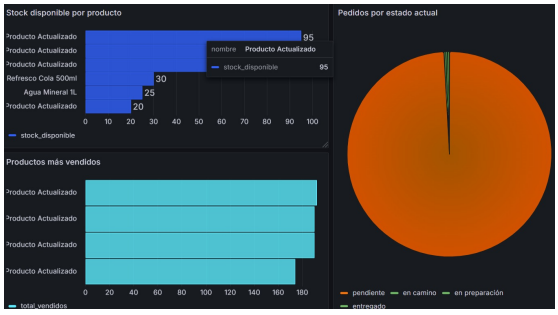

Figura 8: *Medidas de productos*

Por otro lado, este panel del dashboard ofrece métricas relacionadas con los productos y su actualización en la base de datos. Dado que las tareas simuladas por los usuarios de Locust incluyen la creación y modificación de productos con nombres generados dinámicamente, las gráficas muestran repetidamente la

etiqueta "Producto Actualizado". Esto explica por qué no aparecen productos del catálogo real, como los refrescos o artículos identificables.

3.3. Disco

A continuación comparamos al situación del espacio en disco antes y después de las pruebas de carga.

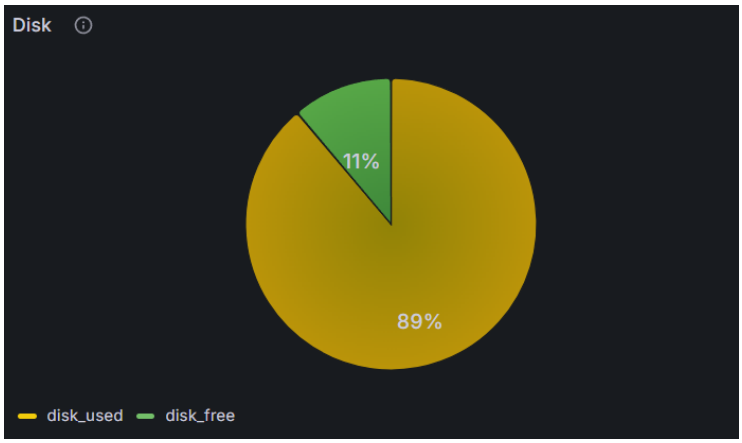

Figura 9: *Espacio en el disco sin carga*

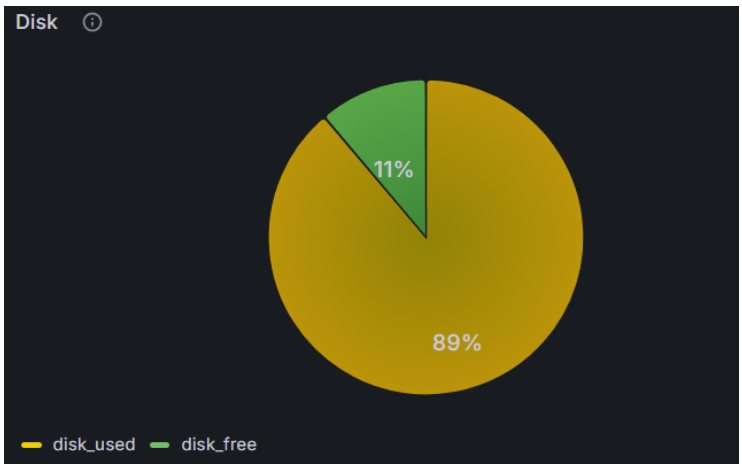

Figura 10: *Espacio en el disco con carga*

Este gráfico circular representa el porcentaje de uso del disco durante el monitoreo del sistema. Se observa que el 89 % del espacio está en uso, mientras que solo el 11 % permanece libre. Este patrón se ha mantenido constante tanto durante las pruebas de carga como en estado inactivo, lo que indica que las operaciones simuladas no han tenido un impacto significativo en el almacenamiento del sistema.
La razón de esta estabilidad puede deberse a que las tareas realizadas por los usuarios simulados (consultas, inserciones temporales, registros, etc.) no generan

archivos grandes o permanentes en el disco. Esta métrica es útil para verificar que, al menos en esta fase, el almacenamiento no representa un cuello de botella en el rendimiento general de la aplicación.

3.4. Memoria RAM

A continuación comparamos al situación del espacio en memoria antes y después de las pruebas de carga.

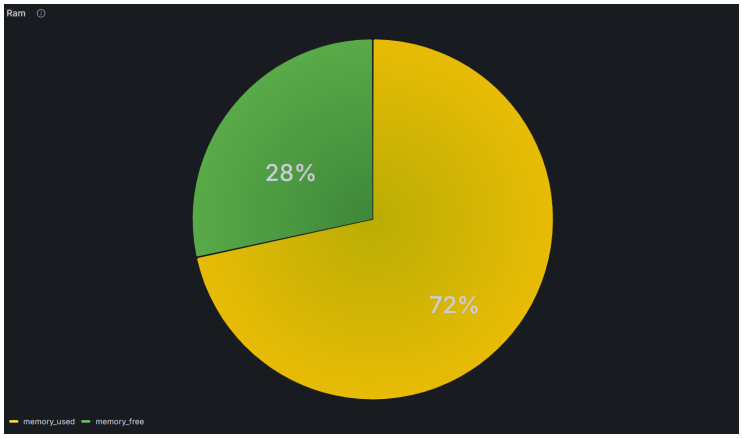

Figura 11: *Memoria RAM sin carga*

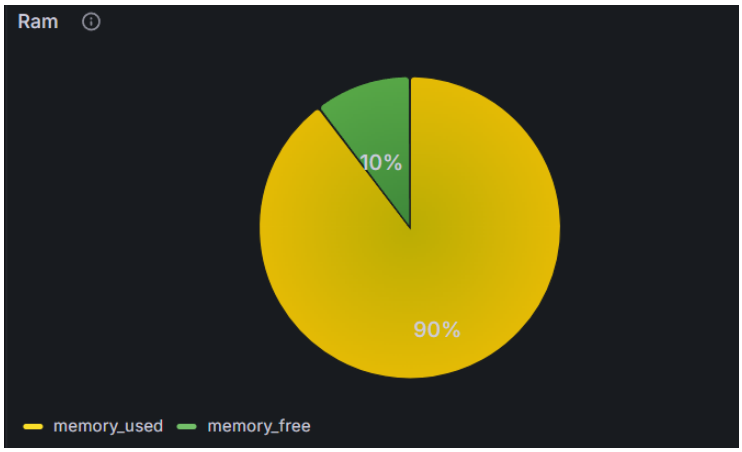

Figura 12: *Memoria RAM con carga*

Durante las pruebas de carga, se observó una diferencia notable en el uso de memoria RAM del sistema. En la primera gráfica (con carga), el 90 % de la memoria RAM está en uso y solo un 10 libre, lo que indica una demanda intensa de recursos debido a la simulación de múltiples usuarios y operaciones simultáneas. En cambio, la segunda gráfica (sin carga) refleja un estado mucho más relajado del sistema, con solo un 72 % de memoria usada y un 28 % libre. Esta diferencia del 18 en el uso de RAM demuestra el impacto directo que tiene la ejecución

de pruebas de carga sobre los recursos del sistema, particularmente en entornos con memoria limitada. Este análisis permite concluir que, aunque el sistema sigue operando correctamente, la carga simulada ejerce una presión significativa sobre la memoria, lo que debe considerarse para escalar la infraestructura en un entorno de producción.

3.5. Metricas generales

Figura 13: *Metricas generales*

Las anteriores gráficas se han obtenido directamente de locust. Se adjuntan en este informe para mostrar dos de las pruebas de carga realizadas. la primera de menor valor, llegó a los 74 usuarios, con intervalos de un segundo. Se aprecia claramente el pico en el tiempo de respuesta a los 47 usuarios. La segunda muestra una prueba más larga que llegó a los 200 usuarios, en ella el tiempo de respuesta crece de manera lineal desde los 21 a los 160 usuarios, podemos establecer entonces en ese rango la pero evolución de la web.

3.6. Metricas de CPU

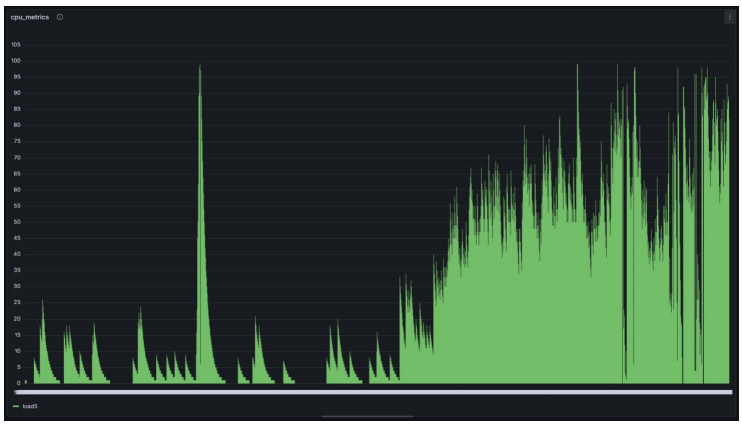

Figura 14: *CPU metrics*

La gráfica muestra el comportamiento del load5 de una máquina virtual durante una prueba de carga de 1 minuto con 500 usuarios por segundo usando Locust. Al inicio, la carga es baja con picos aislados, pero rápidamente aumenta de forma sostenida hasta alcanzar niveles muy altos, superiores a 80, con variaciones bruscas y caídas puntuales. Esto indica una saturación del sistema, donde hay más procesos esperando CPU que los que pueden ser atendidos, lo cual puede provocar lentitud o fallos en el servicio. El patrón sugiere que la infraestructura no está adecuadamente dimensionada para la carga simulada, y se recomienda analizar y escalar los recursos o mejorar la eficiencia del backend.

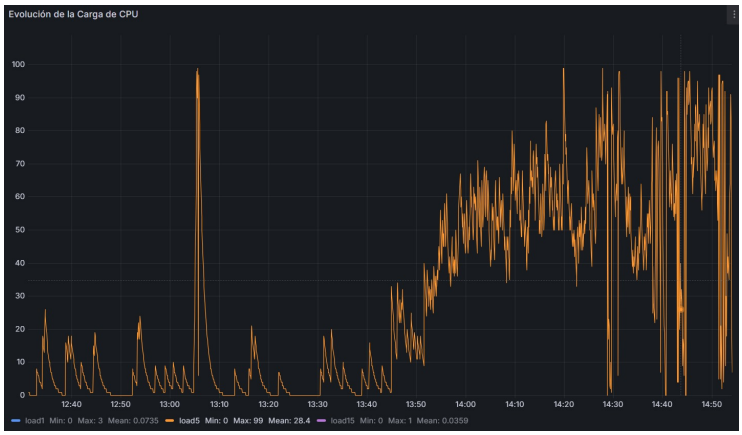

Figura 15: *CPU metrics*

Esta otra gráfica también muestra la evolución del load5 durante dos pruebas de carga realizadas. En la primera mitad del gráfico (hasta las 13:45 aproximadamente), se observan picos intermitentes con una media baja, lo que sugiere que la carga era puntual y posiblemente parte de pruebas iniciales o calibraciones. A partir de las 13:45, el load5 se incrementa de forma sostenida y alcanza valores

máximos cercanos a 100, manteniéndose con alta variabilidad y frecuentes picos extremos. Esto indica una segunda prueba con una carga mucho más agresiva y sostenida en el tiempo, saturando completamente la capacidad de la máquina. El comportamiento caótico y los picos constantes reflejan un sistema bajo fuerte presión, posiblemente con múltiples procesos compitiendo por recursos, lo que puede derivar en degradación del rendimiento o cuellos de botella graves. Coincidiendo con la gráfica anterior.

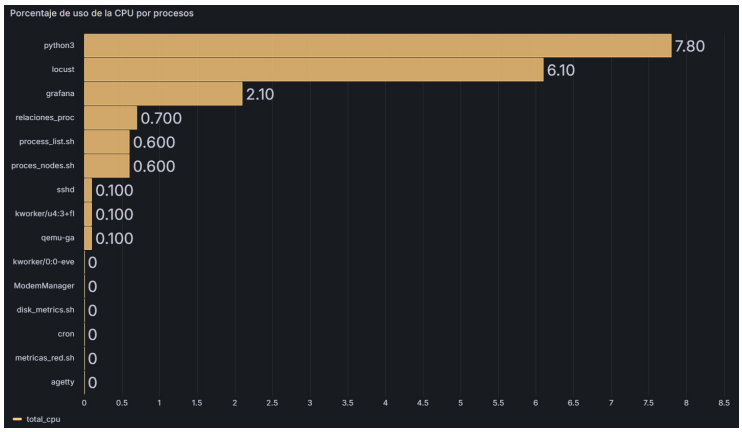

Figura 16: *Porcentaje de CPU*

La gráfica muestra el porcentaje de uso de CPU por cada proceso activo en una máquina virtual durante la ejecución de una prueba de carga. Esta visualización permite identificar de forma clara qué procesos están generando mayor carga en el sistema.
Destacan dos procesos como principales consumidores de CPU:
python3: 7.8 %
locust: 6.1 %
Ambos están directamente relacionados con la ejecución de las pruebas (ya que Locust está escrito en Python), y juntos suman un 13.9 del uso total de la CPU, lo que indica que la simulación de usuarios está generando una carga computacional considerable.
Otros procesos como grafana, relaciones_proc o los scripts de monitorización (`process_list.sh`, `proces_nodes.sh`) muestran un consumo mucho menor, no superior al 0,7. El resto de procesos del sistema (`sshd`, `kworker`, `cron`, `agetty`, etc.) presentan un uso nulo o muy bajo, lo que es esperable en condiciones normales.
En una máquina sin carga artificial, el uso de CPU suele estar próximo a cero o dominado por procesos del sistema con actividad puntual. En contraste, esta gráfica evidencia cómo la actividad de prueba impone una carga tangible y medible sobre el sistema, confirmando la efectividad de las pruebas realizadas con Locust.

3.7. Métricas de Procesos

3.7.1. Número de procesos por usuario

Aquí mostramos medidas relacionadas con el número de procesos por usuario.

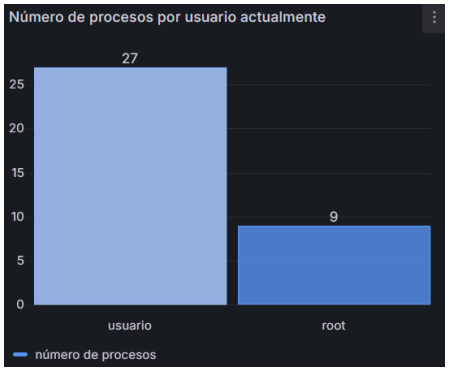

Figura 17: *Con prueba de carga*

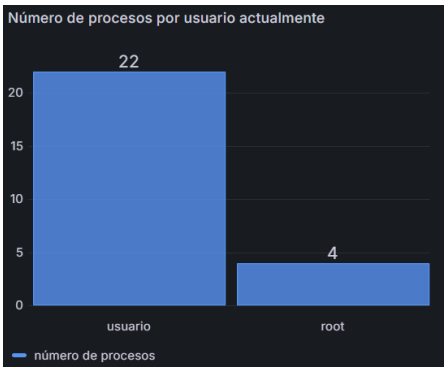

Figura 18: *Datos iniciales*

En estas dos imágenes podemos observar los cambios que ha habido en el número de procesos por usuario en un instante de tiempo determinado. La imagen a la izquierda muestra los datos en un instante mientras se estaban realizando las pruebas de carga, y la imagen de la derecha muestra los datos que había inicialmente antes de ejecutar las pruebas de carga.

Se puede ver cómo, al realizar las pruebas de carga, el número de procesos ha aumentado en comparación con cuando la máquina tiene menos volumen de actividad o peticiones, ya que la carga hace que se generen más procesos para atender las solicitudes.

3.7.2. Número de procesos por comando

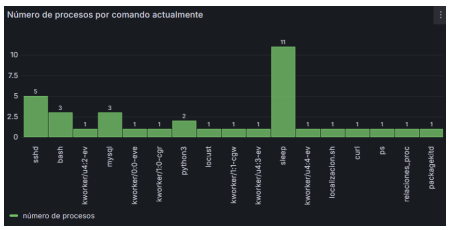

Figura 19: *Con prueba de carga*

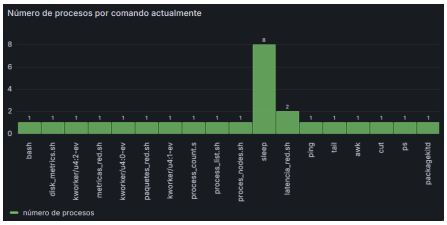

Figura 20: *Datos iniciales*

En este caso, al igual que el anterior, podemos observar un claro aumento en el número de procesos por comando que se están ejecutando. A la izquierda vemos la gráfica con los datos mientras se está procesando la prueba de carga, y a la derecha, la gráfica con los datos iniciales sin pruebas de carga en proceso.

Podemos destacar los comandos que han aparecido como 'locust', donde se realizan las pruebas de carga, o 'python3' y 'mysql', que aparecen con mayor número de procesos, ya que están trabajando con las pruebas de carga.

3.7.3. Numero de procesos activos

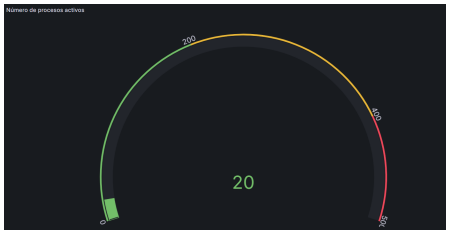

Figura 21: *Con prueba de carga*

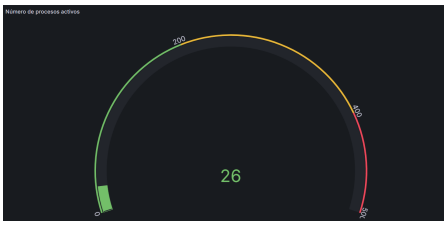

Figura 22: *Datos iniciales*

Por último, en las métricas de procesos hemos comparado el número de procesos activos cuando se están realizando las pruebas de carga (imagen de la izquierda) y, en el estado inicial, mientras no se estaban realizando las pruebas de carga (imagen de la derecha).

Ciertamente, al igual que en las otras dos métricas de procesos, en este caso también hay un aumento en el número de procesos activos.

4. Conclusiones

El diseño del backend basado en Flask se ha dividido adecuadamente en módulos independientes (productos.py, carrito.py, pedidos.py, etc.), lo que facilita el mantenimiento y extensión futura. Cada módulo gestiona claramente rutas GET y POST específicas, separando responsabilidades.
Los métodos POST permiten interacciones clave como registrar usuarios, iniciar sesión, agregar productos al carrito y realizar compras. Por su parte, los GET permiten consultar productos, pedidos o ver contenido del carrito, proporcionando una interfaz completa para el cliente web y las pruebas de carga.
Tras ejecutar pruebas de carga con Locust sobre la página web desplegada, se ha podido observar un impacto significativo en el rendimiento del sistema. La métrica load5 evidenció una saturación progresiva, alcanzando valores extremos que indican un número de procesos muy superior a la capacidad de la CPU para atenderlos en tiempo razonable. Aunque el uso de CPU por proceso individual no fue excesivo siendo locust y python3 los principales consumidores, el alto número de conexiones simultáneas generó un cuello de botella por concurrencia. Las gráficas de latencia y conexiones de red confirmaron un deterioro en el tiempo de respuesta de la aplicación y un aumento considerable de conexiones activas durante el test. Estos resultados sugieren que la infraestructura actual no está dimensionada para soportar una carga elevada sostenida, por lo que se recomienda optimizar la aplicación, mejorar la gestión de concurrencia y escalar los recursos de la máquina virtual para garantizar la estabilidad bajo condiciones de alta demanda.