ARCHITECTURE FINALE - MECAPY (Sans hypothèse de limites)

🎯 PROBLÉMATIQUE

Contraintes confirmées

- ✅ Développeur solo (pas de gestion infra complexe)
- ✅ Provider français/européen
- ⚠️ **Limite probable : 1000 containers/functions chez Scaleway**
- ✅ Besoin : Multi-utilisateurs, multi-calculs simultanés
- ✅ Calculs courts (secondes) ET longs (heures) possibles

🚀 SOLUTION : Architecture Mutualisée (1 Container = N Calculs)

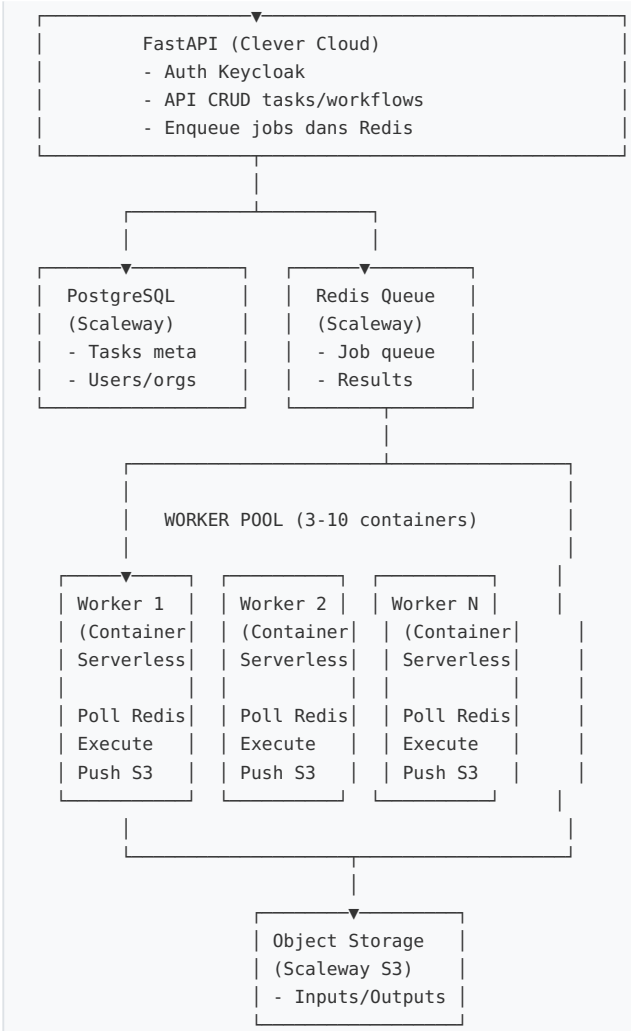
Concept clé : Worker Pool au lieu de 1 container par calcul

```
 ❌ MAUVAISE approche (hit la limite):
1 calcul utilisateur = 1 Serverless Container déployé
→ 1000 utilisateurs × 1 calcul = LIMITE ATTEINTE

✅ BONNE approche (scalable):
1 Serverless Container = Worker qui traite N calculs en séquence
→ 10 workers × 100 calculs chacun = 1000 calculs sans problème
```

🏗️ ARCHITECTURE PROPOSÉE : Worker Pool Serverless

```
┌─────────────────────────────────────────────┐
│              UTILISATEURS (N)               │
└─────────────────────┬───────────────────────┘
                      │
```



🔧 COMPOSANTS DÉTAILLÉS

1. API FastAPI (Clever Cloud)

Rôle : Point d'entrée unique, gestion métadonnées

```
 # api/routes/tasks.py
from fastapi import APIRouter
import redis

router = APIRouter()
redis_client = redis.from_url(os.getenv("REDIS_URL"))

@router.post("/tasks")
async def create_task(task: TaskCreate, user: User):
    """Créer une tâche et l'enqueue"""

    # 1. Sauvegarder metadata en DB
    task_db = await db.tasks.create({
        "id": uuid.uuid4(),
        "user_id": user.id,
        "name": task.name,
```

```
        "code": task.code,
        "status": "queued",
        "created_at": datetime.utcnow()
    })

    # 2. Upload inputs vers S3
    s3_key = f"inputs/{task_db.id}.json"
    await s3.upload(s3_key, task.inputs)

    # 3. Enqueue job dans Redis
    redis_client.rpush("mecapy:jobs", json.dumps({
        "task_id": str(task_db.id),
        "user_id": str(user.id),
        "code_s3_key": f"code/{task_db.id}.py",
        "inputs_s3_key": s3_key,
        "priority": task.priority or 5,
        "timeout": task.timeout or 300  # 5min default
    }))

    return {"task_id": task_db.id, "status": "queued"}


@router.get("/tasks/{task_id}")
async def get_task_status(task_id: str):
    """Récupérer le statut d'une tâche"""
    task = await db.tasks.get(task_id)

    # Si completed, récupérer résultats depuis Redis ou S3
    if task.status == "completed":
        result = redis_client.get(f"result:{task_id}")
        if not result:
            # Fallback S3 si expiré de Redis
            result = await s3.download(f"results/{task_id}.json")

    return {
        "task_id": task_id,
        "status": task.status,
        "result": result if task.status == "completed" else None,
        "error": task.error_message if task.status == "failed" else None
    }
```

---

2. Worker Pool (Serverless Containers)

Architecture : 3-10 containers permanents qui **consomment** la queue Redis

Dockerfile Worker

```
 FROM python:3.12-slim

# Installer dépendances scientifiques
RUN pip install \
    numpy \
    scipy \
    pandas \
    matplotlib \
    redis \
    boto3 \
    RestrictedPython
```

```
COPY worker/ /app/
WORKDIR /app

# Worker daemon qui poll Redis en continu
CMD ["python", "worker.py"]
```

Code Worker

```
 # worker/worker.py
import redis
import boto3
import json
import time
from sandbox import execute_user_code

# Config
REDIS_URL = os.getenv("REDIS_URL")
S3_BUCKET = os.getenv("S3_BUCKET")

redis_client = redis.from_url(REDIS_URL)
s3_client = boto3.client('s3')

def process_job(job_data: dict):
    """Traiter un job de calcul"""

    task_id = job_data['task_id']

    try:
        # 1. Update status = running
        redis_client.hset(f"task:{task_id}", "status", "running")
        redis_client.hset(f"task:{task_id}", "started_at", time.time())

        # 2. Download code et inputs depuis S3
        code = s3_client.get_object(
            Bucket=S3_BUCKET,
            Key=job_data['code_s3_key']
        )['Body'].read().decode()

        inputs = json.loads(
            s3_client.get_object(
                Bucket=S3_BUCKET,
                Key=job_data['inputs_s3_key']
            )['Body'].read()
        )

        # 3. Exécuter code utilisateur (sandboxed)
        result = execute_user_code(
            code=code,
            inputs=inputs,
            timeout=job_data.get('timeout', 300)
        )

        # 4. Upload résultat vers S3
        s3_client.put_object(
            Bucket=S3_BUCKET,
            Key=f"results/{task_id}.json",
            Body=json.dumps(result)
        )

        # 5. Cache résultat dans Redis (TTL 1h)
        redis_client.setex(
            f"result:{task_id}",
            3600,  # 1h expiration
            json.dumps(result)
        )
```

```
        # 6. Update status = completed
        redis_client.hset(f"task:{task_id}", "status", "completed")
        redis_client.hset(f"task:{task_id}", "completed_at", time.time())

    except Exception as e:
        # Gestion erreur
        redis_client.hset(f"task:{task_id}", "status", "failed")
        redis_client.hset(f"task:{task_id}", "error", str(e))


def main():
    """Worker daemon - Poll Redis en continu"""

    print(f"🚀 Worker started - polling Redis queue...")

    while True:
        try:
            # Blocking pop (attend jusqu'à 5s)
            job = redis_client.blpop("mecapy:jobs", timeout=5)

            if job:
                job_data = json.loads(job[1])
                print(f"📋 Processing task {job_data['task_id']}")

                process_job(job_data)

                print(f"✅ Task {job_data['task_id']} completed")

        except Exception as e:
            print(f"❌ Worker error: {e}")
            time.sleep(1)  # Éviter boucle rapide en cas d'erreur

if __name__ == "__main__":
    main()
```

---

3. Déploiement Worker Pool

Option A : Scaleway Serverless Containers (si limites acceptables)

```
 # Build image
docker build -t mecapy-worker:v1 ./worker
docker push registry.scaleway.com/mecapy/worker:v1

# Déployer 5 workers identiques
for i in {1..5}; do
  scw container container create \
    --name mecapy-worker-$i \
    --namespace-id <namespace-id> \
    --registry-image mecapy/worker:v1 \
    --min-scale 1 \
    --max-scale 1 \
    --memory-limit 2048 \
    --cpu-limit 1000 \
    --env REDIS_URL=$REDIS_URL \
    --env S3_BUCKET=$S3_BUCKET
done
```

Avantages : - ✅ 5-10 containers = très loin de la limite 1000 - ✅ Chaque worker traite des centaines de calculs - ✅ Auto-restart si crash - ✅ Coût fixe prévisible

Option B : Scaleway Instances (VMs légères) - SI containers limités

```
 # Déployer 3 VMs DEV1-S (2 vCPU, 2GB RAM)
scw instance server create \
  name=mecapy-worker-1 \
  type=DEV1-S \
  image=ubuntu_jammy \
  cloud-init=worker-init.yaml

# cloud-init.yaml
#cloud-config
runcmd:
  - docker pull registry.scaleway.com/mecapy/worker:v1
  - docker run -d --restart=always \
      -e REDIS_URL=$REDIS_URL \
      -e S3_BUCKET=$S3_BUCKET \
      registry.scaleway.com/mecapy/worker:v1
```

Coût : 3x DEV1-S = ~€18/mois (vs €50/mois pour containers serverless)

📊 COMPARAISON DES OPTIONS BACKEND

Option	Gestion infra	Coût	Limite	Recommandation
Serverless Containers (worker pool)	ϟ Minimal	€30-50/mois	✅ Seulement 5-10 containers	✅✅ **Idéal**
Instances VMs (worker pool)	Moyenne	€18-30/mois	✅ Illimité	✅ Backup si containers limités
Kubernetes	🟡 Élevée	€50+/mois	✅ Illimité	❌ Trop complexe solo
1 Container par calcul	ϟ Zero	€10-20/mois	❌ HIT limite 1000	❌ Non scalable

---

💰 COÛTS DÉTAILLÉS

Architecture Worker Pool (Serverless Containers)

Service	Configuration	Coût mensuel
API FastAPI	Clever Cloud Nano	€7-10/mois
PostgreSQL	Scaleway DB 1GB	€18/mois
Redis	Scaleway DB 512MB	€15/mois
Object Storage	Scaleway S3 (50GB)	€1/mois
5x Workers	Serverless Containers (min=1, max=1)	€30-40/mois
Registry	Container Registry	Gratuit
Total		**€70-85/mois**

Capacité : 1000-5000 calculs/jour selon durée moyenne

Architecture Worker Pool (VMs)

Service	Configuration	Coût mensuel
API + DB + Redis	Identique ci-dessus	€40/mois
3x Instances	DEV1-S (2vCPU, 2GB)	€18/mois
Object Storage	Scaleway S3	€1/mois
Total		**€60-65/mois**

Capacité : Identique (même puissance calcul)

🔥 GESTION DES CALCULS LONGS (> 10min)

Stratégie : Router automatique

```
 # api/services/task_router.py
async def route_task(task: Task) -> str:
    """Router selon estimation de durée"""

    estimated_duration = estimate_duration(task)

    if estimated_duration < 600:  # < 10min
        # Enqueue dans Redis normale
        redis_client.rpush("mecapy:jobs", serialize(task))
```

```
        return "worker_pool"

    else:
        # Lancer instance dédiée on-demand
        instance_id = await launch_dedicated_instance(task)
        return f"dedicated_instance:{instance_id}"


async def launch_dedicated_instance(task: Task) -> str:
    """Lancer une VM on-demand pour calcul long"""

    # Créer instance Scaleway avec cloud-init
    instance = await scw_api.create_instance(
        name=f"compute-{task.id}",
        type="DEV1-M",  # ou GPU1-S si besoin GPU
        image="docker_ubuntu",
        cloud_init=f"""
        #!/bin/bash
        # Download code et inputs
        aws s3 cp s3://{S3_BUCKET}/code/{task.id}.py /tmp/code.py
        aws s3 cp s3://{S3_BUCKET}/inputs/{task.id}.json /tmp/inputs.json

        # Execute
        python3 /tmp/code.py < /tmp/inputs.json > /tmp/output.json

        # Upload résultats
        aws s3 cp /tmp/output.json s3://{S3_BUCKET}/results/{task.id}.json

        # Update status via API
        curl -X PATCH https://api.mecapy.com/tasks/{task.id} \
          -d '{{"status": "completed"}}'

        # Auto-destroy instance
        scw instance server delete {instance.id}
        """
    )

    return instance.id
```

Coût calculs longs : - DEV1-M : €0.024/h → 1h calcul = €0.024 - GPU1-S : €0.50/h → 2h calcul = €1.00

⚡ SCALING STRATÉGIE

Phase 1 : MVP (0-1000 calculs/jour)

```
 3 Workers (Serverless Containers ou VMs)
Coût : €60-70/mois
```

Phase 2 : Growth (1000-5000 calculs/jour)

```
 5-8 Workers
Coût : €80-100/mois
```

Phase 3 : Scale (5000-20 000 calculs/jour)

```
 10-15 Workers + Auto-scaling Redis-based
Coût : €120-150/mois
```

Auto-scaling Worker Pool (si VMs)

```
 # monitoring/autoscaler.py
import redis
import time

def autoscale_workers():
    """Scale workers selon queue depth"""

    queue_depth = redis_client.llen("mecapy:jobs")
    current_workers = get_active_workers()

    if queue_depth > 100 and current_workers < 10:
        # Scale up
        spawn_worker()

    elif queue_depth < 10 and current_workers > 3:
        # Scale down
        kill_idle_worker()

# Run every 30s
while True:
    autoscale_workers()
    time.sleep(30)
```

✅ AVANTAGES ARCHITECTURE FINALE

Pour développeur solo :

1. ✅ **Gestion infra minimale** : API + 5 workers = c'est tout
2. ✅ **Pas de limite artificielle** : Worker pool traite N calculs
3. ✅ **Coût prévisible** : €60-85/mois fixe
4. ✅ **Debugging facile** : Logs centralisés Redis/S3
5. ✅ **Évolutif** : Ajouter workers = 1 commande

Pour utilisateurs :

1. ✅ **Latence basse** : Workers warm (pas de cold start)
2. ✅ **Queue visible** : Position dans la file d'attente
3. ✅ **Isolation** : Sandbox Python par calcul

4. ✅ **Résultats persistants** : S3 + cache Redis

Scalabilité :

- **1000 calculs/jour** : 3 workers suffisent
- **10 000 calculs/jour** : 10 workers suffisent
- **100 000 calculs/jour** : 50 workers + K8s à considérer

🚀 PLAN DE MISE EN ŒUVRE (4 SEMAINES)

Semaine 1 : Infrastructure base

```
 # 1. Provisionner services managés
scw rdb instance create name=mecapy-db engine=postgresql-15
scw redis cluster create name=mecapy-redis

# 2. Créer bucket S3
scw object bucket create mecapy-storage

# 3. Déployer API FastAPI sur Clever Cloud
clever create --type python mecapy-api
```

Semaine 2 : Worker Pool

```
 # 1. Build image worker
docker build -t mecapy-worker:v1 ./worker

# 2. Push vers registry
docker push registry.scaleway.com/mecapy/worker:v1

# 3. Déployer 3 workers (Serverless Containers OU VMs)
./scripts/deploy_workers.sh 3
```

Semaine 3 : API Routes + Tests

```
 # Implémenter routes
POST /tasks              # Create task
GET  /tasks/{id}         # Get status
GET  /tasks/{id}/result  # Get result

# Tests end-to-end
pytest tests/test_worker_pool.py
```

Semaine 4 : Monitoring + Production

```
 # Setup monitoring
- Grafana dashboard (queue depth, worker CPU)
- Alerting (Redis down, workers crashed)

# Go live
clever deploy
```

📊 TABLEAU COMPARATIF FINAL

Critère	Worker Pool Serverless	Worker Pool VMs	K8s + Celery
Gestion infra	ϟ Minimal	Moyenne	🟡 Élevée
Coût base	€70/mois	€60/mois	€90/mois
Coût scale	Linéaire	Linéaire	Linéaire
Limite containers	✅ 5-10 seulement	✅ Illimité	✅ Illimité
Complexité	Simple	Simple	🟡 Complexe
Latence	~50ms	~50ms	~50ms
Auto-scaling	Manuel	Script Python	✅ Natif
Recommandé solo	✅✅✅	✅✅	❌

🎯 RECOMMANDATION FINALE

☆ CHOIX #1 : Worker Pool Serverless Containers

```
 API FastAPI + 5-10 Workers Serverless Containers + Redis + S3
Coût : €70-85/mois
Maintenance : ~2h/mois
```

Pourquoi : - ✅ Simple à déployer et maintenir - ✅ Auto-restart des workers si crash - ✅ Pas de gestion VMs - ✅ Très loin de la limite 1000

☆ CHOIX #2 (fallback) : Worker Pool VMs

```
 API FastAPI + 3-5 VMs workers + Redis + S3
Coût : €60-70/mois
Maintenance : ~3h/mois
```

Pourquoi : - ✅ Moins cher (€10/mois économie) - ✅ Pas de limite containers - Gestion SSH + updates

× NE PAS FAIRE : 1 Container par calcul

→ Hit la limite 1000 rapidement

✅ **VERDICT : Worker Pool = Architecture idéale même avec limite 1000**

Document généré le : 2025-09-30 **Version** : 3.0 - Architecture finale réaliste