EXÉCUTION SÉCURISÉE DU CODE UTILISATEUR - ANALYSE APPROFONDIE

🎯 PROBLÉMATIQUE

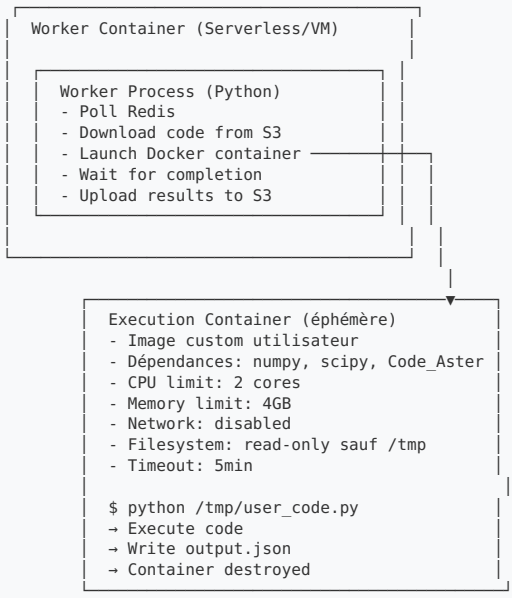
Défis à résoudre

1. **Sécurité** : Empêcher code malveillant (accès filesystem, réseau, fork bomb, etc.)
2. **Isolation** : Chaque calcul ne doit pas affecter les autres
3. **Dépendances** : Worker ne doit PAS avoir les dépendances utilisateur installées
4. **Performance** : Overhead minimal pour calculs courts
5. **Ressources** : Limiter CPU/RAM/temps par calcul

🏗️ SOLUTIONS POSSIBLES (5 approches)

Approche 1 : Docker-in-Docker (DinD) ⭐ RECOMMANDÉ

Concept : Le worker lance un conteneur Docker temporaire pour chaque calcul



Code Worker avec Docker-in-Docker

```
 # worker/docker_executor.py
import docker
import json
import tempfile
import time
from pathlib import Path

class DockerExecutor:
    """Execute user code in isolated Docker containers"""

    def __init__(self):
        self.docker_client = docker.from_env()

    def execute(self, task: dict) -> dict:
        """
        Execute user code in isolated container.

        task = {
            "task_id": "uuid",
            "code": "def calculate(inputs): ...",
            "inputs": {"param1": 10},
            "runtime": "python:3.12",  # ou custom image
```

```
            "requirements": ["numpy==1.24.0", "scipy==1.10.0"],
            "timeout": 300,
            "memory_limit": "2g",
            "cpu_limit": 2.0
        }
        """

        task_id = task['task_id']

        # 1. Créer répertoire temporaire pour le code
        with tempfile.TemporaryDirectory() as tmpdir:
            tmpdir_path = Path(tmpdir)

            # 2. Écrire le code utilisateur
            code_file = tmpdir_path / "user_code.py"
            code_file.write_text(self._wrap_user_code(task['code']))

            # 3. Écrire les inputs
            inputs_file = tmpdir_path / "inputs.json"
            inputs_file.write_text(json.dumps(task['inputs']))

            # 4. Créer requirements.txt si besoin
            if task.get('requirements'):
                req_file = tmpdir_path / "requirements.txt"
                req_file.write_text('\n'.join(task['requirements']))

            # 5. Lancer container Docker avec restrictions
            try:
                container = self.docker_client.containers.run(
                    image=task.get('runtime', 'python:3.12-slim'),
                    command=self._get_execution_command(task),

                    # Volumes: mount code en read-only
                    volumes={
                        str(tmpdir_path): {
                            'bind': '/workspace',
                            'mode': 'ro'  # Read-only !
                        }
                    },

                    # Limits de ressources
                    mem_limit=task.get('memory_limit', '2g'),
                    cpu_quota=int(task.get('cpu_limit', 2.0) * 100000),
                    cpu_period=100000,

                    # Sécurité
                    network_mode='none',  # Pas d'accès réseau !
                    read_only=True,       # Filesystem read-only
                    security_opt=['no-new-privileges'],
                    cap_drop=['ALL'],     # Drop toutes les capabilities

                    # Tmpfs pour /tmp (RAM disk)
                    tmpfs={'/tmp': 'size=512m,mode=1777'},

                    # Timeout
                    detach=True,
                    remove=False  # On nettoie manuellement après
                )

                # 6. Attendre la fin (avec timeout)
                result = container.wait(timeout=task.get('timeout', 300))

                # 7. Récupérer les logs (stdout/stderr)
                logs = container.logs(stdout=True, stderr=True).decode()

                # 8. Récupérer output.json depuis le container
                output_data = self._extract_output(container, '/tmp/output.json')

                # 9. Nettoyer le container
                container.remove(force=True)

                # 10. Retourner résultat
                return {
                    'status': 'success' if result['StatusCode'] == 0 else 'error',
                    'output': output_data,
                    'logs': logs,
                    'exit_code': result['StatusCode']
                }

            except docker.errors.ContainerError as e:
                return {
                    'status': 'error',
                    'error': f"Container error: {str(e)}",
                    'logs': e.stderr.decode() if e.stderr else ''
                }

            except Exception as e:
                return {
                    'status': 'error',
                    'error': str(e)
                }

    def _wrap_user_code(self, user_code: str) -> str:
        """Wrap user code with safety harness"""

        wrapper = f'''
import json
```

```
import sys
import signal

# Timeout handler
def timeout_handler(signum, frame):
    raise TimeoutError("Execution timeout")

signal.signal(signal.SIGALRM, timeout_handler)
signal.alarm(300)  # 5min max

try:
    # Load inputs
    with open('/workspace/inputs.json', 'r') as f:
        inputs = json.load(f)

    # User code
{self._indent_code(user_code, 4)}

    # Execute calculate function
    result = calculate(inputs)

    # Write output
    with open('/tmp/output.json', 'w') as f:
        json.dump(result, f)

    sys.exit(0)

except Exception as e:
    error_output = {{"error": str(e), "type": type(e).__name__}}
    with open('/tmp/output.json', 'w') as f:
        json.dump(error_output, f)
    sys.exit(1)
'''
        return wrapper

    def _indent_code(self, code: str, spaces: int) -> str:
        """Indent user code"""
        indent = ' ' * spaces
        return '\n'.join(indent + line for line in code.split('\n'))

    def _get_execution_command(self, task: dict) -> list:
        """Get Docker command to execute"""

        commands = []

        # Install requirements if specified
        if task.get('requirements'):
            commands.append('pip install --no-cache-dir -r /workspace/requirements.txt')

        # Execute user code
        commands.append('python /workspace/user_code.py')

        return ['sh', '-c', ' && '.join(commands)]

    def _extract_output(self, container, file_path: str) -> dict:
        """Extract output file from container"""
        try:
            bits, stat = container.get_archive(file_path)

            # Extract tar archive
            import tarfile
            import io

            tar_stream = io.BytesIO(b''.join(bits))
            tar = tarfile.open(fileobj=tar_stream)

            # Read output.json
            output_file = tar.extractfile('output.json')
            return json.loads(output_file.read().decode())

        except Exception as e:
            return {'error': f'Failed to extract output: {str(e)}'}
```

Avantages Docker-in-Docker

- ✅ **Isolation parfaite** : Kernel namespaces + cgroups
- ✅ **Dépendances custom** : Chaque user peut avoir son image
- ✅ **Limits strictes** : CPU, RAM, réseau, filesystem
- ✅ **Nettoyage automatique** : Container détruit après
- ✅ **Sécurité prouvée** : Technologie mature

Inconvénients

- ⚠️ **Overhead** : ~2-5s pour spawn container (acceptable si calcul > 10s)
- ⚠️ **Complexité** : Besoin de Docker daemon sur le worker
- ⚠️ **Ressources** : ~200MB RAM par container

Approche 2 : gVisor (runsc) ⭐⭐ TRÈS SÉCURISÉ

Concept : Sandbox ultra-sécurisé avec syscall interception

gVisor = Runtime container de Google qui intercepte TOUS les syscalls

```
 # worker/gvisor_executor.py
import subprocess
import json

class GVisorExecutor:
    """Execute code with gVisor sandbox"""

    def execute(self, task: dict) -> dict:
        # 1. Créer OCI bundle (config.json + rootfs)
        bundle_path = self._create_oci_bundle(task)

        # 2. Lancer avec runsc (gVisor runtime)
        result = subprocess.run(
            [
                'runsc',
                '--network=none',
                '--platform=ptrace',  # ou kvm pour meilleures perfs
                'run',
                '--bundle', bundle_path,
                task['task_id']
            ],
            capture_output=True,
            timeout=task.get('timeout', 300)
        )

        # 3. Récupérer résultat
        return self._parse_output(result)
```

Avantages gVisor : - ✅ **Sécurité maximale** : Syscall interception - ✅ **Pas besoin VM** : Plus léger que VMs - ✅ **Compatible Docker** : Drop-in replacement

Inconvénients : - ⚠️ **Overhead** : 10-20% plus lent que Docker natif - ⚠️ **Complexité setup** : Installer gVisor sur workers

---

Approche 3 : Firecracker MicroVMs ⭐⭐⭐ ULTRA RAPIDE

Concept : MicroVMs ultra-légères (AWS Lambda utilise ça)

```
 Worker lance une MicroVM Firecracker par calcul
→ Boot en 125ms
→ Isolation kernel complète
→ Destroy après exécution
```

```
 # worker/firecracker_executor.py
import requests
import json

class FirecrackerExecutor:
    """Execute code in Firecracker microVM"""

    def __init__(self):
        self.firecracker_socket = '/tmp/firecracker.sock'

    def execute(self, task: dict) -> dict:
        # 1. Configure microVM
        self._configure_vm(
            vcpu_count=task.get('cpu_limit', 2),
            mem_size_mib=task.get('memory_mb', 2048),
            kernel_image='/path/to/vmlinux',
            rootfs='/path/to/rootfs.ext4'
        )

        # 2. Boot microVM (125ms)
        self._boot_vm()

        # 3. Execute code via vsock
        result = self._execute_in_vm(task['code'], task['inputs'])

        # 4. Shutdown VM
        self._shutdown_vm()

        return result

    def _configure_vm(self, **config):
        """Configure Firecracker via API"""
        requests.put(
            f'http://localhost/machine-config',
            json={
                'vcpu_count': config['vcpu_count'],
                'mem_size_mib': config['mem_size_mib']
```

```
            }
        )
```

Avantages Firecracker : - ✅ **Boot ultra-rapide** : 125ms (vs 2-5s Docker) - ✅ **Isolation kernel complète** : Vraie VM - ✅ **Léger** : 5MB RAM overhead - ✅ **Production-proven** : AWS Lambda, Fly.io

Inconvénients : - 🟡 **Complexité élevée** : Setup non-trivial - 🟡 **Linux only** : Nécessite KVM - **Pas de support Windows/Mac**

Approche 4 : RestrictedPython ⚠ DÉCONSEILLÉ SEUL

Concept : Sandbox Python natif (pas de conteneur)

```
 # worker/restricted_executor.py
from RestrictedPython import compile_restricted, safe_builtins
import RestrictedPython.Guards

def execute_user_code(code: str, inputs: dict) -> dict:
    """Execute in Python sandbox (NOT SECURE ENOUGH ALONE)"""

    # Compile code en mode restreint
    byte_code = compile_restricted(
        code,
        filename='<user_code>',
        mode='exec'
    )

    # Whitelist imports
    safe_globals = {
        '__builtins__': safe_builtins,
        '__name__': 'user_module',
        '__metaclass__': type,
        '_getattr_': RestrictedPython.Guards.safe_getattr,

        # Modules autorisés
        'numpy': __import__('numpy'),
        'scipy': __import__('scipy'),
        'math': __import__('math'),
    }

    # Execute
    exec(byte_code, safe_globals)

    # Call calculate()
    return safe_globals['calculate'](inputs)
```

Avantages : - ✅ **Très rapide** : Pas d'overhead - ✅ **Simple** : Pas de Docker/VM

Inconvénients : - 🟡 **INSUFFISANT SEUL** : Bypasses possibles - 🟡 **Pas de limite CPU/RAM** : Doit être combiné - 🟡 **Imports limités** : Difficile de whitelister tout

→ Peut être utilisé EN COMBINAISON avec Docker pour double protection

---

Approche 5 : Subprocess avec ulimit ⚠ FAIBLE ISOLATION

```
 # worker/subprocess_executor.py
import subprocess
import resource

def execute_user_code(code: str, inputs: dict) -> dict:
    """Execute in subprocess with resource limits (WEAK)"""

    def set_limits():
        # Limite CPU: 5min
        resource.setrlimit(resource.RLIMIT_CPU, (300, 300))

        # Limite mémoire: 2GB
        resource.setrlimit(resource.RLIMIT_AS, (2*1024*1024*1024, 2*1024*1024*1024))

        # Limite processus
        resource.setrlimit(resource.RLIMIT_NPROC, (1, 1))

    result = subprocess.run(
        ['python', '-c', code],
        input=json.dumps(inputs),
        capture_output=True,
        timeout=300,
        preexec_fn=set_limits  # Linux only
    )

    return json.loads(result.stdout)
```

→ NE PAS UTILISER en production : Isolation insuffisante

📊 COMPARAISON DES APPROCHES

Approche	Sécurité	Performance	Complexité	Isolation deps	Coût dev
Docker-in-Docker	⭐⭐⭐⭐	⚠️ (2-5s overhead)	⚠️ Moyenne	✅ Parfaite	⚠️ 1 semaine
gVisor	⭐⭐⭐⭐⭐	⚠️ (10-20% slower)	⚠️ Moyenne	✅ Parfaite	⚠️ 2 semaines
Firecracker	⭐⭐⭐⭐⭐	✅ (125ms boot)	🔴 Élevée	✅ Parfaite	🔴 1 mois
RestrictedPython	⭐⭐	✅ (natif)	✅ Simple	❌ Partagée	✅ 3 jours
Subprocess	⭐	✅ (natif)	✅ Simple	❌ Partagée	✅ 1 jour

🎯 RECOMMANDATION FINALE

Pour démarrer (MVP) : Docker-in-Docker ✅

Pourquoi : 1. ✅ **Bon compromis** sécurité/performance/complexité 2. ✅ **Isolation parfaite** des dépendances 3. ✅ **Technologie mature** : Docker = production-proven 4. ✅ **Facile à debugger** : `docker logs`, `docker inspect` 5. ✅ **Évolutif** : Peut migrer vers Firecracker plus tard

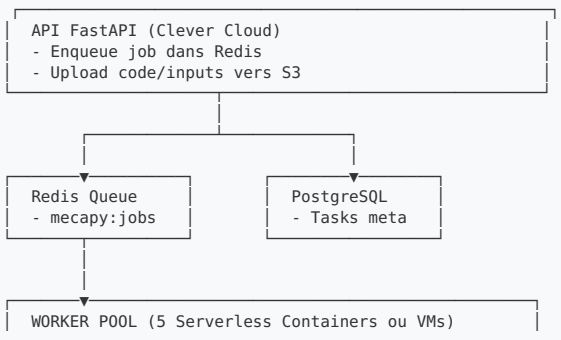
Setup Worker avec Docker :

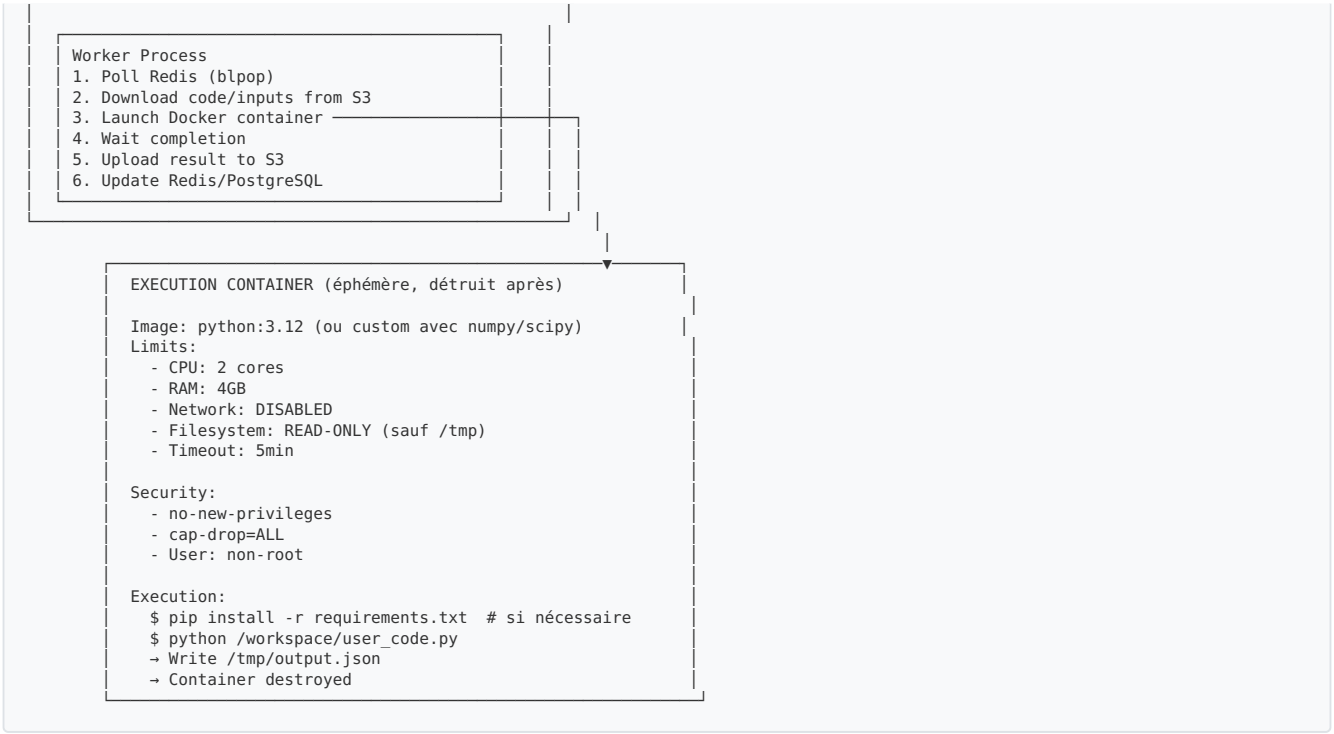
```
 # Dockerfile du Worker
FROM python:3.12-slim

# Installer Docker CLI
RUN apt-get update && apt-get install -y \
    docker.io \
    && rm -rf /var/lib/apt/lists/*

# Installer dépendances worker
RUN pip install redis boto3 docker

COPY worker/ /app/
WORKDIR /app

CMD ["python", "worker.py"]
```

Déploiement :

```
 # Sur Scaleway Serverless Containers
scw container container create \
  --name mecapy-worker \
  --privileged true \  # Nécessaire pour Docker-in-Docker
  --registry-image mecapy/worker:v1

# OU sur VMs
docker run -d \
  -v /var/run/docker.sock:/var/run/docker.sock \  # Socket Docker
  mecapy-worker:v1
```

🔥 ARCHITECTURE RECOMMANDÉE FINALE





🔒 SÉCURITÉ MULTICOUCHE

Couche 1 : Validation API

```
 # api/validation.py
def validate_user_code(code: str) -> bool:
    """Validation statique du code"""

    # Blacklist imports dangereux
    forbidden = ['os', 'subprocess', 'sys', 'socket', '__import__']
    for module in forbidden:
        if f'import {module}' in code:
            raise SecurityError(f"Forbidden module: {module}")

    # Taille max code
    if len(code) > 100_000:  # 100KB
        raise ValueError("Code too large")

    return True
```

Couche 2 : RestrictedPython (optionnel, défense en profondeur)

```
 # worker/restricted_wrapper.py
from RestrictedPython import compile_restricted

def wrap_code(user_code: str) -> str:
    """Double validation avec RestrictedPython"""

    # Tenter compilation restreinte
    compile_restricted(user_code, '<string>', 'exec')

    # Si OK, retourner code wrapped
    return user_code
```

Couche 3 : Docker isolation (principal)

- Network disabled
- Filesystem read-only
- Capabilities dropped
- Cgroups limits

Couche 4 : Monitoring

```
 # worker/monitor.py
def monitor_execution(container):
    """Monitor container pour détection anomalies"""

    stats = container.stats(stream=False)

    # Check CPU spike
    if stats['cpu_stats']['cpu_usage']['total_usage'] > THRESHOLD:
        container.kill()

    # Check memory
    if stats['memory_stats']['usage'] > MEMORY_LIMIT:
        container.kill()
```

💰 COÛTS OVERHEAD DOCKER

Par calcul :

- Spawn container : 2-5s
- Execution : Variable (10s-5min)
- Destroy : 0.5s
- **Total overhead : 2.5-6s**

→ Acceptable si calcul > 10s (overhead < 30%)

Optimisation possible :

```
 # Pool de containers pré-créés (optionnel)
class ContainerPool:
    def __init__(self, size=3):
        self.pool = [
            docker_client.containers.create('python:3.12')
            for _ in range(size)
        ]

    def get_container(self):
        """Réutiliser container au lieu de créer"""
        container = self.pool.pop()
        container.restart()
        return container
```

→ Réduit overhead à ~500ms mais complexité accrue

✅ PLAN D'IMPLÉMENTATION

Semaine 1 : Proof of Concept

```
 # Test Docker executor localement
python test_docker_executor.py

# Vérifier isolation
- Code malveillant bloqué ?
- Limits CPU/RAM respectées ?
- Timeout fonctionne ?
```

Semaine 2 : Intégration Worker

```
 # worker/worker.py
from docker_executor import DockerExecutor

executor = DockerExecutor()

while True:
    job = redis.blpop('mecapy:jobs')
    result = executor.execute(job)
    upload_to_s3(result)
```

Semaine 3 : Tests de charge

```
 # 100 calculs simultanés
python benchmark.py --concurrent 100
```

Semaine 4 : Production

```
 # Deploy 5 workers
./deploy_workers.sh 5
```

📋 ALTERNATIVE SI DOCKER-IN-DOCKER IMPOSSIBLE

Option : Workers dédiés par runtime

```
 Worker Pool A (3 workers) : Python + NumPy/SciPy
Worker Pool B (2 workers) : Python + Code_Aster
Worker Pool C (2 workers) : Python + Custom deps

→ API route selon requirements
```

Avantages : - ✅ Pas besoin Docker-in-Docker - ✅ Dépendances pré-installées (plus rapide)

Inconvénients : - ❌ Moins flexible (limité aux runtimes prédéfinis) - ❌ Maintenance de plusieurs images

---

✅ **VERDICT : Docker-in-Docker = Meilleur compromis pour MVP**

Document généré le : 2025-09-30 **Version** : 1.0 - Analyse exécution sécurisée