3

```python
#!/usr/bin/env python3
"""
constructor.py - CLI Interface para ConstructorAgent
Vision Wagon - Automatización SDLC

Uso:
    python constructor.py init
    python constructor.py build --blueprint blueprints/agents.yml --target SecurityAgent
    python constructor.py status
    python constructor.py task --name scaffold_agent --agent-name TestAgent --agent-type operational
"""

import argparse
import asyncio
import json
import sys
from pathlib import Path
from typing import Dict, Any
import logging

# Importar ConstructorAgent (asumiendo que está en el mismo directorio o instalado)
from constructor_agent import ConstructorAgent, ConstructorCLI, BlueprintParser, AgentSpec

# Configurar logging
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)
```

```
logger = logging.getLogger(__name__)


class VisionWagonCLI:
    """CLI Principal de Vision Wagon ConstructorAgent"""

    def __init__(self):
        self.constructor_cli = ConstructorCLI()

    def create_parser(self) -> argparse.ArgumentParser:
        """Crea el parser de argumentos"""
        parser = argparse.ArgumentParser(
            description="Vision Wagon ConstructorAgent - Automatización SDLC",
            formatter_class=argparse.RawDescriptionHelpFormatter,
            epilog="""
Ejemplos:
  %(prog)s init                              # Inicializar proyecto
  %(prog)s build --blueprint agents.yml          # Construir desde blueprint
  %(prog)s build --target SecurityAgent          # Construir agente específico
  %(prog)s task --name init_project_structure     # Ejecutar tarea específica
  %(prog)s status                            # Ver estado del proyecto
  %(prog)s validate --blueprint agents.yml       # Validar blueprint
            """
        )

        subparsers = parser.add_subparsers(dest='command', help='Comandos disponibles')

        # Comando init
        init_parser = subparsers.add_parser('init', help='Inicializar estructura del proyecto')
        init_parser.add_argument(
            '--project-root',
            type=str,
            default='.',
            help='Directorio raíz del proyecto (default: directorio actual)'
        )

        # Comando build
        build_parser = subparsers.add_parser('build', help='Construir desde blueprint')
        build_parser.add_argument(
            '--blueprint',
            type=str,
            help='Ruta al archivo blueprint YAML'
        )
        build_parser.add_argument(
            '--target',
            type=str,
            help='Objetivo específico a construir (agente, módulo, etc.)'
        )
        build_parser.add_argument(
            '--dry-run',
            action='store_true',
            help='Simular ejecución sin realizar cambios'
        )

        # Comando status
        status_parser = subparsers.add_parser('status', help='Ver estado del proyecto')
        status_parser.add_argument(
            '--detailed',
            action='store_true',
            help='Mostrar información detallada'
        )

        # Comando task
```

```
task_parser = subparsers.add_parser('task', help='Ejecutar tarea específica')
task_parser.add_argument(
    '--name',
    type=str,
    required=True,
    help='Nombre de la tarea a ejecutar'
)
task_parser.add_argument(
    '--agent-name',
    type=str,
    help='Nombre del agente (para tareas de agentes)'
)
task_parser.add_argument(
    '--agent-type',
    type=str,
    choices=['executive', 'operational'],
    default='operational',
    help='Tipo de agente'
)
task_parser.add_argument(
    '--params',
    type=str,
    help='Parámetros adicionales en formato JSON'
)

# Comando validate
validate_parser = subparsers.add_parser('validate', help='Validar blueprint')
validate_parser.add_argument(
    '--blueprint',
    type=str,
    required=True,
        help='Ruta al archivo blueprint a validar'
    )

    # Comando generate
    generate_parser = subparsers.add_parser('generate', help='Generar blueprints o
plantillas')
    generate_parser.add_argument(
        '--type',
        type=str,
        choices=['blueprint', 'agent', 'api', 'database'],
        required=True,
        help='Tipo de elemento a generar'
    )
    generate_parser.add_argument(
        '--name',
        type=str,
        required=True,
        help='Nombre del elemento'
    )
    generate_parser.add_argument(
        '--output',
        type=str,
        help='Archivo de salida'
    )

    # Comando list
    list_parser = subparsers.add_parser('list', help='Listar elementos del proyecto')
    list_parser.add_argument(
        '--type',
        type=str,
        choices=['agents', 'tasks', 'blueprints', 'all'],
            default='all',
            help='Tipo de elementos a listar'
        )

        return parser

    async def handle_init_command(self, args) -> Dict[str, Any]:
        """Maneja el comando init"""
        try:
            # Configurar directorio del proyecto
            project_root = Path(args.project_root).resolve()
            self.constructor_cli.constructor.project_root = project_root

            print(f"🚀 Inicializando Vision Wagon en: {project_root}")

            result = await self.constructor_cli.run_command('init')

            if 'error' not in result:
                print("✅ Proyecto inicializado exitosamente")
                print("\nEstructura creada:")
                for task_result in result.get('results', []):
                    if task_result.success:
                        print(f"  ✓ {task_result.task_name}")
                        for artifact in task_result.artifacts_created:
                            print(f"    📁 {artifact}")
                    else:
                        print(f"  ❌ {task_result.task_name}: {task_result.error}")

            return result

        except Exception as e:
            logger.error(f"Error en comando init: {e}")
            return {"error": str(e)}

    async def handle_build_command(self, args) -> Dict[str, Any]:
        """Maneja el comando build"""
        try:
            # Determinar blueprint
            if args.blueprint:
                blueprint_path = Path(args.blueprint)
            else:
                # Buscar blueprint por defecto
                possible_blueprints = [
                    Path("blueprints/default.yml"),
                    Path("blueprints/agents.yml"),
                    Path("blueprint.yml")
                ]
                blueprint_path = None
                for bp in possible_blueprints:
                    if bp.exists():
                        blueprint_path = bp
                        break

                if not blueprint_path:
                    return {"error": "No se encontró un blueprint. Use --blueprint para especificar
uno."}

            if not blueprint_path.exists():
                return {"error": f"Blueprint no encontrado: {blueprint_path}"}

            print(f"🔨 Construyendo desde blueprint: {blueprint_path}")
            if args.target:
```

```
    print(f"🎯 Objetivo específico: {args.target}")

if args.dry_run:
    print("🧪 Modo simulación (dry-run)")
    # TODO: Implementar dry-run
    return {"message": "Simulación completada"}

result = await self.constructor_cli.run_command(
    'build',
    blueprint=str(blueprint_path),
    target=args.target
)

# Mostrar resultados
if 'error' not in result:
    results = result.get('results', {})
    summary = results.get('summary', {})

    print(f"\n📊 Resumen de construcción:")
    print(f"  ✅ Tareas exitosas: {summary.get('successful_tasks', 0)}")
    print(f"  ❌ Tareas fallidas: {summary.get('failed_tasks', 0)}")
    print(f"  📈 Tasa de éxito: {summary.get('success_rate', '0%')}")
    print(f"  ⏱  Tiempo total: {summary.get('total_execution_time', '0s')}")

    print(f"\n📝 Detalle de tareas:")
    for task in results.get('tasks', []):
        status = "✅" if task['success'] else "❌"
        print(f"  {status} {task['name']} ({task['execution_time']})")
        if task.get('artifacts_created', 0) > 0:
            print(f"    📦 {task['artifacts_created']} archivos creados")
```

```
                if task.get('error'):
                    print(f"    🚨 Error: {task['error']}")

        return result

    except Exception as e:
        logger.error(f"Error en comando build: {e}")
        return {"error": str(e)}

async def handle_status_command(self, args) -> Dict[str, Any]:
    """Maneja el comando status"""
    try:
        result = await self.constructor_cli.run_command('status')

        print("📊 Estado del proyecto Vision Wagon")
        print(f"📁 Directorio: {result.get('project_root', 'No especificado')}")

        history = result.get('execution_history', {})
        summary = history.get('summary', {})

        if summary.get('total_tasks', 0) > 0:
            print(f"\n📈 Historial de ejecución:")
            print(f"  🎯 Total de tareas: {summary.get('total_tasks', 0)}")
            print(f"  ☑️ Exitosas: {summary.get('successful_tasks', 0)}")
            print(f"  ❌ Fallidas: {summary.get('failed_tasks', 0)}")
            print(f"  📊 Tasa de éxito: {summary.get('success_rate', '0%')}")

            if args.detailed:
                print(f"\n📝 Detalle de tareas:")
                for task in history.get('tasks', []):
                        status = "✅" if task['success'] else "❌"
                        print(f"  {status} {task['name']} - {task['execution_time']}")
                        if task.get('error'):
                            print(f"    🚨 {task['error']}")
            else:
                print("\n💤 No hay historial de ejecución disponible")

            # Verificar estructura del proyecto
            project_root = Path(result.get('project_root', '.'))
            structure_status = self._check_project_structure(project_root)

            print(f"\n🏗️ Estado de la estructura:")
            for item, exists in structure_status.items():
                status = "✅" if exists else "❌"
                print(f"  {status} {item}")

            return result

        except Exception as e:
            logger.error(f"Error en comando status: {e}")
            return {"error": str(e)}

    async def handle_task_command(self, args) -> Dict[str, Any]:
        """Maneja el comando task"""
        try:
            print(f"⚙️ Ejecutando tarea: {args.name}")

            # Preparar parámetros adicionales
            params = {}
            if args.params:
                try:
```

```
                params = json.loads(args.params)
            except json.JSONDecodeError:
                return {"error": "Parámetros JSON inválidos"}

        # Para tareas de agentes, crear especificación
        if args.name == "scaffold_agent" and args.agent_name:
            from constructor_agent import AgentSpec, ScaffoldAgentTask

            agent_spec = AgentSpec(
                name=args.agent_name,
                type=args.agent_type,
                description=f"Agente {args.agent_name} generado por CLI",
                dependencies=[],
                methods=[],
                config=params
            )

            # Registrar tarea temporal
            task = ScaffoldAgentTask(agent_spec)
            constructor = self.constructor_cli.constructor
            constructor.tasks_registry[f"scaffold_agent_{args.agent_name.lower()}"] = task

            result = await
constructor.execute_task(f"scaffold_agent_{args.agent_name.lower()}")

            if result.success:
                print(f"✅ Agente {args.agent_name} creado exitosamente")
                for artifact in result.artifacts_created:
                    print(f"  📄 {artifact}")
            else:
                print(f"❌ Error creando agente: {result.error}")
            return {"task": args.name, "result": result}

        # Para otras tareas
        result = await self.constructor_cli.constructor.execute_task(args.name)

        if result.success:
            print(f"✅ Tarea {args.name} completada")
            if result.artifacts_created:
                print("📦 Archivos creados:")
                for artifact in result.artifacts_created:
                    print(f"  📄 {artifact}")
        else:
            print(f"❌ Tarea {args.name} falló: {result.error}")

        return {"task": args.name, "result": result}

    except Exception as e:
        logger.error(f"Error en comando task: {e}")
        return {"error": str(e)}

async def handle_validate_command(self, args) -> Dict[str, Any]:
    """Maneja el comando validate"""
    try:
        blueprint_path = Path(args.blueprint)

        if not blueprint_path.exists():
            return {"error": f"Blueprint no encontrado: {blueprint_path}"}

        print(f"🔍 Validando blueprint: {blueprint_path}")
```

```
# Parsear blueprint
blueprint_data = BlueprintParser.parse_blueprint(blueprint_path)

if not blueprint_data:
    print("❌ Blueprint inválido o vacío")
    return {"error": "Blueprint inválido"}

# Validaciones
validation_results = []

# Validar estructura básica
required_sections = ['metadata', 'agents']
for section in required_sections:
    if section in blueprint_data:
        validation_results.append(f"✅ Sección '{section}' presente")
    else:
        validation_results.append(f"⚠️ Sección '{section}' faltante")

# Validar agentes
if 'agents' in blueprint_data:
    agents = BlueprintParser.extract_agent_specs(blueprint_data)
    validation_results.append(f"📊 {len(agents)} agentes encontrados")

    for agent in agents:
        if agent.name and agent.type in ['executive', 'operational']:
            validation_results.append(f" ✅ Agente '{agent.name}' válido")
        else:
            validation_results.append(f" ❌ Agente '{agent.name or 'sin nombre'}' inválido")

# Mostrar resultados
print("\n📋 Resultados de validación:")
```

```
        for result in validation_results:
            print(f"  {result}")

        return {
            "blueprint": str(blueprint_path),
            "valid": True,
            "validation_results": validation_results
        }

    except Exception as e:
        logger.error(f"Error en comando validate: {e}")
        return {"error": str(e)}

async def handle_generate_command(self, args) -> Dict[str, Any]:
    """Maneja el comando generate"""
    try:
        print(f"🎨 Generando {args.type}: {args.name}")

        if args.type == 'blueprint':
            return await self._generate_blueprint(args)
        elif args.type == 'agent':
            return await self._generate_agent_blueprint(args)
        else:
            return {"error": f"Tipo de generación no implementado: {args.type}"}

    except Exception as e:
        logger.error(f"Error en comando generate: {e}")
        return {"error": str(e)}

async def handle_list_command(self, args) -> Dict[str, Any]:
    """Maneja el comando list"""
```

```
try:
    print(f"📋 Listando: {args.type}")

    result = {"type": args.type, "items": []}

    if args.type in ['agents', 'all']:
        agents = self._list_agents()
        result["agents"] = agents

        print(f"\n🤖 Agentes encontrados ({len(agents)}):")
        for agent in agents:
            print(f"  📄 {agent['name']} ({agent['type']}) - {agent['path']}")

    if args.type in ['blueprints', 'all']:
        blueprints = self._list_blueprints()
        result["blueprints"] = blueprints

        print(f"\n📋 Blueprints encontrados ({len(blueprints)}):")
        for bp in blueprints:
            print(f"  📄 {bp}")

    if args.type in ['tasks', 'all']:
        tasks = list(self.constructor_cli.constructor.tasks_registry.keys())
        result["tasks"] = tasks

        print(f"\n⚙️ Tareas disponibles ({len(tasks)}):")
        for task in tasks:
            print(f"  🔧 {task}")

    return result
    except Exception as e:
        logger.error(f"Error en comando list: {e}")
        return {"error": str(e)}

def _check_project_structure(self, project_root: Path) -> Dict[str, bool]:
    """Verifica la estructura del proyecto"""
    required_dirs = [
        "agents/executive",
        "agents/operational",
        "agents/core",
        "database",
        "api",
        "orchestrator",
        "tests",
        "config",
        "blueprints"
    ]

    status = {}
    for dir_path in required_dirs:
        full_path = project_root / dir_path
        status[dir_path] = full_path.exists()

    return status

def _list_agents(self) -> List[Dict[str, str]]:
    """Lista agentes existentes"""
    agents = []
    project_root = Path(self.constructor_cli.constructor.project_root)

    for agent_type in ['executive', 'operational']:
        agent_dir = project_root / "agents" / agent_type
        if agent_dir.exists():
            for py_file in agent_dir.glob("*.py"):
                if py_file.name != "__init__.py":
                    agents.append({
                        "name": py_file.stem,
                        "type": agent_type,
                        "path": str(py_file.relative_to(project_root))
                    })

    return agents

def _list_blueprints(self) -> List[str]:
    """Lista blueprints disponibles"""
    blueprints = []
    project_root = Path(self.constructor_cli.constructor.project_root)

    blueprints_dir = project_root / "blueprints"
    if blueprints_dir.exists():
        for yml_file in blueprints_dir.glob("*.yml"):
            blueprints.append(str(yml_file.relative_to(project_root)))
        for yaml_file in blueprints_dir.glob("*.yaml"):
            blueprints.append(str(yaml_file.relative_to(project_root)))

    return blueprints

async def _generate_blueprint(self, args) -> Dict[str, Any]:
    """Genera un blueprint básico"""
    blueprint_template = {
        "metadata": {
            "name": args.name,
            "version": "1.0.0",
            "description": f"Blueprint generado para {args.name}",
            "created_at": "2025-06-14",
            "created_by": "ConstructorAgent CLI"
        },
        "config": {
            "database_type": "sqlite",
            "api_port": 8000,
            "log_level": "INFO"
        },
        "agents": [
            {
                "name": f"{args.name}Agent",
                "type": "operational",
                "description": f"Agente principal para {args.name}",
                "dependencies": [],
                "methods": [
                    {
                        "name": "process",
                        "description": "Método principal de procesamiento"
                    }
                ],
                "config": {}
            }
        ]
    }

    # Determinar archivo de salida
    if args.output:
        output_path = Path(args.output)
    else:
            output_path = Path(f"blueprints/{args.name.lower()}.yml")

        # Crear directorio si no existe
        output_path.parent.mkdir(parents=True, exist_ok=True)

        # Escribir blueprint
        import yaml
        with open(output_path, 'w', encoding='utf-8') as f:
            yaml.dump(blueprint_template, f, default_flow_style=False, allow_unicode=True)

        print(f"✅ Blueprint generado: {output_path}")

        return {
            "generated": str(output_path),
            "type": "blueprint",
            "name": args.name
        }

    async def _generate_agent_blueprint(self, args) -> Dict[str, Any]:
        """Genera un blueprint específico para un agente"""
        agent_blueprint = {
            "metadata": {
                "name": f"{args.name} Agent Blueprint",
                "version": "1.0.0",
                "description": f"Blueprint específico para el agente {args.name}"
            },
            "agents": [
                {
                    "name": args.name,
                    "type": "operational",
                    "description": f"Agente {args.name}",
                "dependencies": [],
                "methods": [
                    {
                        "name": "initialize",
                        "description": "Inicialización del agente"
                    },
                    {
                        "name": "process",
                        "description": "Procesamiento principal"
                    },
                    {
                        "name": "validate_input",
                        "description": "Validación de entrada"
                    },
                    {
                        "name": "cleanup",
                        "description": "Limpieza de recursos"
                    }
                ],
                "config": {
                    "timeout": 30,
                    "retry_attempts": 3
                }
            }
        ]
    }

    # Determinar archivo de salida
    if args.output:
        output_path = Path(args.output)
    else:
            output_path = Path(f"blueprints/agent_{args.name.lower()}.yml")

        # Crear directorio si no existe
        output_path.parent.mkdir(parents=True, exist_ok=True)

        # Escribir blueprint
        import yaml
        with open(output_path, 'w', encoding='utf-8') as f:
            yaml.dump(agent_blueprint, f, default_flow_style=False, allow_unicode=True)

        print(f"✅ Blueprint de agente generado: {output_path}")

        return {
            "generated": str(output_path),
            "type": "agent_blueprint",
            "name": args.name
        }

    async def run(self):
        """Ejecuta la CLI"""
        parser = self.create_parser()
        args = parser.parse_args()

        if not args.command:
            parser.print_help()
            return

        # Mapear comandos a handlers
        handlers = {
            'init': self.handle_init_command,
            'build': self.handle_build_command,
            'status': self.handle_status_command,
            'task': self.handle_task_command,
            'validate': self.handle_validate_command,
            'generate': self.handle_generate_command,
            'list': self.handle_list_command
        }

        if args.command in handlers:
            try:
                result = await handlers[args.command](args)

                # Mostrar errores si existen
                if 'error' in result:
                    print(f"\n❌ Error: {result['error']}")
                    sys.exit(1)

            except KeyboardInterrupt:
                print(f"\n⏹️ Operación cancelada por el usuario")
                sys.exit(1)
            except Exception as e:
                print(f"\n💥 Error inesperado: {e}")
                logger.exception("Error inesperado en CLI")
                sys.exit(1)
        else:
            print(f"❌ Comando no reconocido: {args.command}")
            parser.print_help()
            sys.exit(1)

def main():
    """Punto de entrada principal"""
    cli = VisionWagonCLI()
    asyncio.run(cli.run())


if __name__ == "__main__":
    main()


4


# ========== agents/core/base_agent.py ==========
"""
BaseAgent - Clase base para todos los agentes de Vision Wagon
"""


import asyncio
import logging
from abc import ABC, abstractmethod
from typing import Dict, Any, Optional, List
from datetime import datetime
import uuid


logger = logging.getLogger(__name__)


class BaseAgent(ABC):
    """Clase base abstracta para todos los agentes"""


    def __init__(self, agent_id: str, agent_type: str, config: Optional[Dict[str, Any]] = None):
        self.agent_id = agent_id
        self.agent_type = agent_type
        self.config = config or {}
        self.status = "inactive"
        self.created_at = datetime.utcnow()
        self.last_activity = None
        self.execution_count = 0

        # Configurar logging específico del agente
        self.logger = logging.getLogger(f"agent.{agent_id}")

    @abstractmethod
    async def initialize(self) -> bool:
        """Inicializa el agente - debe ser implementado por cada agente"""
        pass

    @abstractmethod
    async def process(self, context: Dict[str, Any]) -> Dict[str, Any]:
        """Procesa una tarea - debe ser implementado por cada agente"""
        pass

    @abstractmethod
    async def validate_input(self, data: Dict[str, Any]) -> bool:
        """Valida datos de entrada - debe ser implementado por cada agente"""
        pass

    @abstractmethod
    async def cleanup(self) -> None:
        """Limpia recursos - debe ser implementado por cada agente"""
        pass

    async def log_action(self, action: str, data: Dict[str, Any]) -> None:
        """Registra una acción del agente"""
        self.logger.info(f"Acción {action}: {data}")
        self.last_activity = datetime.utcnow()

        # TODO: Integrar con base de datos para logging persistente
        # await self._save_log_to_db(action, data)

    async def get_status(self) -> Dict[str, Any]:
        """Obtiene el estado actual del agente"""
        return {
            "agent_id": self.agent_id,
            "agent_type": self.agent_type,
            "status": self.status,
            "created_at": self.created_at.isoformat(),
            "last_activity": self.last_activity.isoformat() if self.last_activity else None,
            "execution_count": self.execution_count,
            "config": self.config
        }

    async def start(self) -> bool:
        """Inicia el agente"""
        try:
            self.logger.info(f"Iniciando agente {self.agent_id}")

            if await self.initialize():
                self.status = "active"
                await self.log_action("agent_started", {"agent_id": self.agent_id})
                return True
            else:
                self.status = "error"
                return False

        except Exception as e:
            self.logger.error(f"Error iniciando agente {self.agent_id}: {e}")
            self.status = "error"
            return False
```

```
async def stop(self) -> None:
    """Detiene el agente"""
    try:
        self.logger.info(f"Deteniendo agente {self.agent_id}")
        self.status = "stopping"

        await self.cleanup()

        self.status = "inactive"
        await self.log_action("agent_stopped", {"agent_id": self.agent_id})

    except Exception as e:
        self.logger.error(f"Error deteniendo agente {self.agent_id}: {e}")
        self.status = "error"

async def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
    """Ejecuta el procesamiento principal del agente"""
    if self.status != "active":
        return {
            "status": "error",
            "error": f"Agente {self.agent_id} no está activo",
            "agent_status": self.status
        }

    try:
        # Validar entrada
        if not await self.validate_input(context):
            return {
                "status": "error",
                "error": "Validación de entrada falló",
                    "agent_id": self.agent_id
                }

            # Procesar
            self.execution_count += 1
            result = await self.process(context)

            await self.log_action("execution_completed", {
                "execution_count": self.execution_count,
                "result_status": result.get("status", "unknown")
            })

            return result

        except Exception as e:
            self.logger.error(f"Error ejecutando agente {self.agent_id}: {e}")
            await self.log_action("execution_error", {"error": str(e)})

            return {
                "status": "error",
                "error": str(e),
                "agent_id": self.agent_id
            }

# ========== database/database_models.py ==========
"""
Modelos de base de datos para Vision Wagon
"""

from sqlalchemy import Column, Integer, String, DateTime, Text, Boolean, ForeignKey, JSON
from sqlalchemy.orm import relationship, declarative_base
```

```python
from sqlalchemy.dialects.postgresql import UUID

from datetime import datetime

import uuid


Base = declarative_base()


class Campaign(Base):

    """Modelo de campaña"""

    __tablename__ = 'campaigns'


    id = Column(Integer, primary_key=True, index=True)

    uuid = Column(UUID(as_uuid=True), default=uuid.uuid4, unique=True, index=True)

    name = Column(String(255), nullable=False)

    description = Column(Text)

    status = Column(String(50), default='pending')

    campaign_type = Column(String(100))

    target_audience = Column(String(255))

    budget = Column(Integer)  # En centavos

    start_date = Column(DateTime)

    end_date = Column(DateTime)

    created_at = Column(DateTime, default=datetime.utcnow)

    updated_at = Column(DateTime, default=datetime.utcnow, onupdate=datetime.utcnow)


    # Configuración como JSON

    config = Column(JSON)

    metadata = Column(JSON)


    # Relaciones

    logs = relationship("AgentLog", back_populates="campaign")

    executions = relationship("CampaignExecution", back_populates="campaign")
    def __repr__(self):
        return f"<Campaign(id={self.id}, name='{self.name}', status='{self.status}')>"

    def to_dict(self):
        return {
            "id": self.id,
            "uuid": str(self.uuid),
            "name": self.name,
            "description": self.description,
            "status": self.status,
            "campaign_type": self.campaign_type,
            "target_audience": self.target_audience,
            "budget": self.budget,
            "start_date": self.start_date.isoformat() if self.start_date else None,
            "end_date": self.end_date.isoformat() if self.end_date else None,
            "created_at": self.created_at.isoformat(),
            "updated_at": self.updated_at.isoformat(),
            "config": self.config,
            "metadata": self.metadata
        }

class AgentLog(Base):
    """Modelo de logs de agentes"""
    __tablename__ = 'agent_logs'

    id = Column(Integer, primary_key=True, index=True)
    agent_id = Column(String(100), nullable=False, index=True)
    campaign_id = Column(Integer, ForeignKey('campaigns.id'), nullable=True)
    action = Column(String(100), nullable=False)
    status = Column(String(50), default='info')
    message = Column(Text)
```

```
data = Column(JSON)
timestamp = Column(DateTime, default=datetime.utcnow, index=True)
execution_time = Column(Integer)  # En milisegundos

# Relaciones
campaign = relationship("Campaign", back_populates="logs")

def __repr__(self):
    return f"<AgentLog(id={self.id}, agent='{self.agent_id}', action='{self.action}')>"

def to_dict(self):
    return {
        "id": self.id,
        "agent_id": self.agent_id,
        "campaign_id": self.campaign_id,
        "action": self.action,
        "status": self.status,
        "message": self.message,
        "data": self.data,
        "timestamp": self.timestamp.isoformat(),
        "execution_time": self.execution_time
```