System Dynamics Backend - Technical Documentation

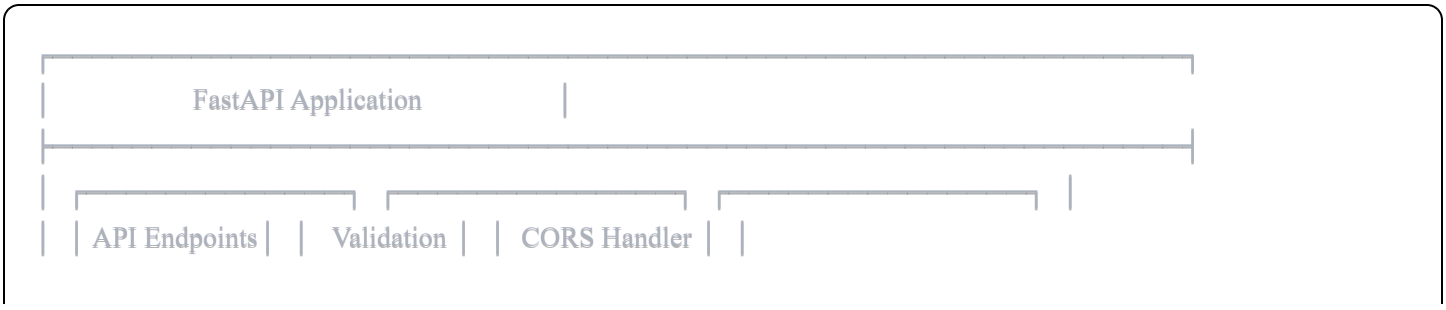
📋 Table of Contents

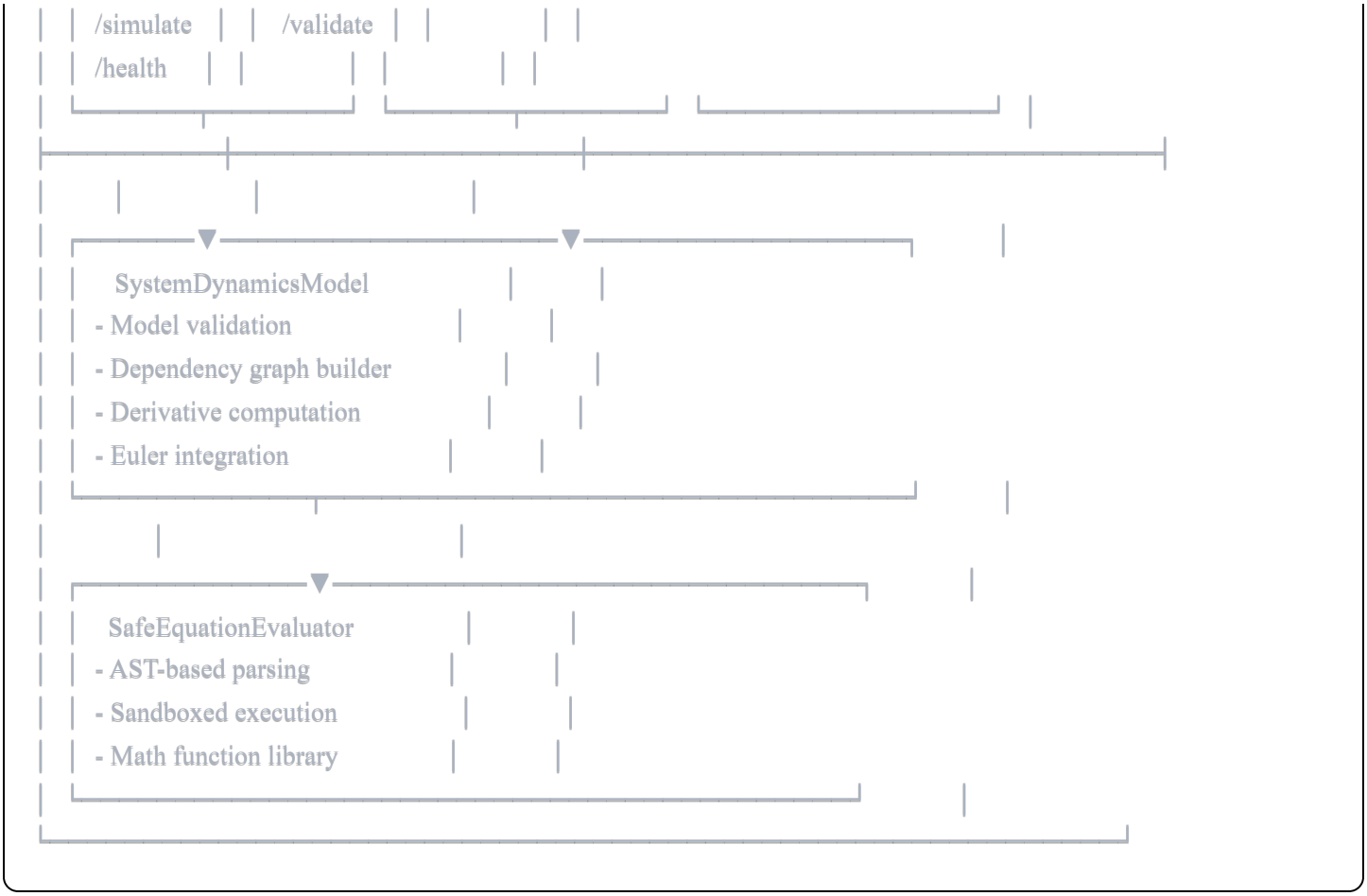
Architecture Overview

Technology Stack

- **Framework**: FastAPI 0.104.1
- **Server**: Uvicorn with ASGI
- **Numerical Computing**: NumPy 1.24.3, SciPy 1.11.4
- **Data Validation**: Pydantic 2.5.0
- **Python Version**: 3.8+

High-Level Architecture





System Requirements

Minimum Requirements

- **Python**: 3.8 or higher
- **RAM**: 512 MB minimum, 2 GB recommended
- **CPU**: Any modern processor
- **Disk**: 100 MB for dependencies
- **Network**: Port 8000 available

Operating Systems

- ✅ Windows 10/11
- ✅ macOS 10.14+
- ✅ Linux (Ubuntu 18.04+, Debian 10+, RHEL 8+)

Python Dependencies

```
fastapi==0.104.1          # Web framework
uvicorn[standard]==0.24.0 # ASGI server
pydantic==2.5.0           # Data validation
numpy==1.24.3             # Numerical computing
scipy==1.11.4             # Scientific computing
python-multipart==0.0.6   # Form data parsing
```

Installation & Setup

Development Environment Setup

bash

```
# 1. Create project directory
mkdir system-dynamics-backend
cd system-dynamics-backend

# 2. Create virtual environment (recommended)
python -m venv venv

# On Windows:
venv\Scripts\activate

# On macOS/Linux:
source venv/bin/activate

# 3. Install dependencies
pip install -r requirements.txt

# 4. Verify installation
python -c "import fastapi, numpy, scipy; print('All dependencies installed')"
```

Running the Server

bash

```
# Development mode (auto-reload enabled)
python main.py

# Production mode
uvicorn main:app --host 0.0.0.0 --port 8000 --workers 4

# With custom configuration
uvicorn main:app --host 127.0.0.1 --port 8080 --log-level info
```

Environment Variables

bash

```
# Optional environment configuration
export BACKEND_HOST="0.0.0.0"      # Server host
export BACKEND_PORT="8000"         # Server port
export BACKEND_LOG_LEVEL="info"    # Logging level
export BACKEND_WORKERS="4"         # Number of worker processes
```

Core Components

1. Data Models (Pydantic)

Element Model

python

```
class Element(BaseModel):
    id: str                  # Unique identifier
    type: str                # 'stock', 'flow', 'parameter', 'variable'
    name: str                # Display name
    initial: Optional[float] = 0.0  # Initial value (stocks)
    equation: Optional[str] = "" # Equation string
    value: Optional[float] = None   # Constant value (parameters)
```

Element Types:

- **Stock**: Accumulation variable with initial value and rate equation
- **Flow**: Rate variable connecting stocks
- **Parameter**: Constant value used in calculations

- **Variable**: Computed auxiliary variable

Link Model

```python
class Link(BaseModel):
    id: str         # Unique identifier
    source: str     # Source element ID
    target: str     # Target element ID
```

SimulationConfig Model

```python
class SimulationConfig(BaseModel):
    start_time: float = 0.0      # Simulation start time
    end_time: float = 100.0      # Simulation end time
    time_step: float = 1.0       # Integration time step
    method: str = "euler"        # Integration method
    verbose: bool = True         # Enable logging
```

SimulationRequest Model

```python
class SimulationRequest(BaseModel):
    elements: List[Element]          # Model elements
    links: List[Link]                # Element connections
    config: SimulationConfig         # Simulation parameters
```

SimulationResponse Model

```python
class SimulationResponse(BaseModel):
    success: bool                    # Success flag
    time: List[float]                # Time points
    results: Dict[str, List[float]]  # Time series data
    error: Optional[str] = None      # Error message
```

---

2. SafeEquationEvaluator Class

Purpose: Safely parse and evaluate mathematical equations without using dangerous `eval()`.

Key Features

- AST-based parsing
- Whitelist of allowed operations
- Sandboxed execution environment
- No code injection vulnerabilities

Allowed Operations

```python
SAFE_OPERATORS = {
    ast.Add: operator.add,       # +
    ast.Sub: operator.sub,       # -
    ast.Mult: operator.mul,      # *
    ast.Div: operator.truediv,   # /
    ast.Pow: operator.pow,       # **
    ast.USub: operator.neg,      # unary -
    ast.UAdd: operator.pos,      # unary +
}
```

Allowed Functions

```python
SAFE_FUNCTIONS = {
    'abs': abs,
    'min': min,
    'max': max,
    'sin': math.sin,
    'cos': math.cos,
    'tan': math.tan,
    'exp': math.exp,
    'log': math.log,
    'log10': math.log10,
    'sqrt': math.sqrt,
    'pow': pow,
}
```

Methods

`parse_equation(equation: str) -> ast.Expression`

- Parses equation string into Abstract Syntax Tree
- Raises `ValueError` on syntax errors

`set_variables(variables: Dict[str, float])`

- Sets available variables for evaluation
- Variables are element IDs, names, and special vars (time, t)

`evaluate(equation: str) -> float`

- Evaluates equation with current variables
- Returns numeric result
- Raises `ValueError` on evaluation errors

Example Usage

```python
evaluator = SafeEquationEvaluator()
evaluator.set_variables({'Population': 1000, 'rate': 0.05})
result = evaluator.evaluate('Population * rate')  # Returns 50.0
```

---

3. SystemDynamicsModel Class

Purpose: Core simulation engine that manages model execution.

Initialization

```python
model = SystemDynamicsModel(
    elements=list_of_elements,
    links=list_of_links,
    verbose=True  # Enable detailed logging
)
```

Key Attributes

- `stocks`: Dictionary of stock elements

- `flows`: Dictionary of flow elements
- `parameters`: Dictionary of parameter elements
- `variables`: Dictionary of variable elements
- `dependencies`: Graph of element dependencies
- `evaluator`: Instance of SafeEquationEvaluator

Methods

`_validate_model()`

- Checks model structure validity
- Ensures at least one stock exists
- Called automatically during initialization

`_build_dependency_graph()`

- Constructs dependency graph from links
- Used for determining evaluation order

`_compute_state_variables(stock_values, t) -> Dict[str, float]`

- Computes all variables, parameters, and flows
- Returns complete state dictionary
- Evaluation order: parameters → variables → flows

`compute_derivatives(stock_values, t) -> Dict[str, float]`

- Computes rate of change for all stocks
- Returns dictionary of derivatives
- Called at each integration step

`simulate_euler(config) -> Dict[str, Any]`

- Main simulation loop using Euler integration
- Returns time series results
- Handles verbose logging if enabled

API Reference

Base URL

```
http://localhost:8000
```

Endpoints

1. Root Endpoint

```http
GET /
```

Response:

```json
{
  "message": "System Dynamics Simulation API",
  "version": "1.0.0",
  "endpoints": {
    "simulate": "/simulate",
    "health": "/health"
  }
}
```

---

2. Health Check

```http
GET /health
```

Response:

```json
{
  "status": "healthy"
}
```

Use Case: Service monitoring, load balancer health checks

---

3. Simulate Model

```http
POST /simulate
Content-Type: application/json
```

Request Body:

```json
```

```
{
 "elements": [
  {
   "id": "stock_1",
   "type": "stock",
   "name": "Population",
   "initial": 1000.0,
   "equation": "births - deaths"
  },
  {
   "id": "flow_1",
   "type": "flow",
   "name": "births",
   "equation": "birth_rate * Population"
  },
  {
   "id": "param_1",
   "type": "parameter",
   "name": "birth_rate",
   "value": 0.03
  }
 ],
 "links": [
  {
   "id": "link_1",
   "source": "param_1",
   "target": "flow_1"
  },
  {
   "id": "link_2",
   "source": "stock_1",
   "target": "flow_1"
  }
 ],
 "config": {
  "start_time": 0.0,
  "end_time": 100.0,
  "time_step": 1.0,
  "method": "euler",
  "verbose": true
 }
}
```

Success Response (200 OK):

```json
{
  "success": true,
  "time": [0.0, 1.0, 2.0, ..., 100.0],
  "results": {
    "stock_1": [1000.0, 1020.0, 1040.4, ...],
    "flow_1": [30.0, 30.6, 31.21, ...],
    "param_1": [0.03, 0.03, 0.03, ...]
  },
  "error": null
}
```

Error Response (200 OK with error):

```json
{
  "success": false,
  "time": [],
  "results": {},
  "error": "Error computing derivative for stock 'Population': Undefined variable: births"
}
```

---

4. Validate Model

```http
POST /validate
Content-Type: application/json
```

Request Body: Same as `/simulate`

Success Response:

```json
```

```
{
  "valid": true,
  "message": "Model is valid",
  "stocks": 2,
  "flows": 3,
  "parameters": 2,
  "variables": 1
}
```

Error Response:

```
json

{
  "valid": false,
  "message": "Model must have at least one stock"
}
```

Equation System

Equation Parsing Flow

```
User Input String
    ↓
Parse to AST (Abstract Syntax Tree)
    ↓
Validate Node Types (whitelist check)
    ↓
Recursive Evaluation
    ↓
Return Numeric Result
```

Supported Syntax

Basic Arithmetic

```
python
```

```
"2 + 3"          # 5
"10 - 4"         # 6
"5 * 6"          # 30
"20 / 4"         # 5.0
"2 ** 3"         # 8
```

Variables

python

```
"Population * 0.05"
"Stock_A + Stock_B"
"rate * time"
```

Functions

python

```
"sqrt(Population)"
"exp(-0.1 * time)"
"log(Stock + 1)"
"sin(2 * 3.14159 * time)"
"max(0, Stock - 100)"
"abs(Stock_A - Stock_B)"
```

Comparisons

python

```
"Population > 1000"     # True/False (1.0/0.0)
"Stock <= threshold"
"value == target"
"x != y"
```

Conditionals (Ternary)

python

```
"10 if Population > 500 else 5"
"rate * (1 if time < 50 else 0.5)"
"0.1 if Stock > threshold else 0.2"
```

Complex Expressions

```python
"birth_rate * Population * (1 - Population / carrying_capacity)"
"sqrt(Stock_A**2 + Stock_B**2)"
"exp(-decay_rate * time) * initial_value"
```

Special Variables

- `time` or `t`: Current simulation time
- Element IDs: Direct reference (e.g., `stock_1`)
- Element names: Also available (e.g., `Population`)

Equation Validation

The evaluator performs multiple checks:

1. **Syntax Check**: Valid Python expression
2. **Operation Check**: Only whitelisted operations
3. **Function Check**: Only allowed functions
4. **Variable Check**: All referenced variables exist
5. **Type Check**: Results in numeric value

Simulation Engine

Integration Methods

Euler Method (Default)

Formula: $x(t+\Delta t) = x(t) + f(x,t) * \Delta t$

Characteristics:

- Simple, fast
- First-order accuracy
- Suitable for smooth systems
- May be inaccurate for stiff systems

Algorithm:

```
python

for each time step:
    1. Compute state (flows, variables)
    2. Compute derivatives (dx/dt for stocks)
    3. Update stocks: stock += derivative * time_step
    4. Store results
```

Simulation Loop Details

```
python

# 1. Initialize
time_points = [start_time, ..., end_time]
stock_values = {stock_id: initial_value}

# 2. For each time step
for t in time_points:
    # Compute current state
    flows = evaluate_all_flows(stock_values, t)
    variables = evaluate_all_variables(stock_values, t)

    # Compute derivatives
    derivatives = {}
    for stock in stocks:
        derivatives[stock] = evaluate(stock.equation)

    # Update stocks (Euler method)
    for stock in stocks:
        stock_values[stock] += derivatives[stock] * time_step

    # Store results
    results[t] = {stock_values, flows, variables}

# 3. Return results
return time_series_data
```

Numerical Stability

Time Step Selection:

- Too large: Instability, inaccuracy

- Too small: Slow execution, rounding errors
- Rule of thumb: `time_step < 1 / (max_rate * 10)`

Example:

```python
# If fastest process has rate 0.5 per time unit
recommended_time_step = 1 / (0.5 * 10) = 0.2
```

Configuration

Server Configuration

In code:

```python
if __name__ == "__main__":
    import uvicorn
    uvicorn.run(
        app,
        host="0.0.0.0",      # Listen on all interfaces
        port=8000,           # Port number
        log_level="info",    # Logging level
        reload=True          # Auto-reload on code changes (dev only)
    )
```

Command line:

```bash
uvicorn main:app \
    --host 0.0.0.0 \
    --port 8000 \
    --workers 4 \
    --log-level info \
    --access-log
```

CORS Configuration

```python
```

```
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],            # Allow all origins (dev)
    # allow_origins=["http://localhost:3000"],  # Specific origin (prod)
    allow_credentials=True,
    allow_methods=["*"],            # Allow all HTTP methods
    allow_headers=["*"],            # Allow all headers
)
```

Logging Configuration

Verbose Logging (Development):

```
python

config = SimulationConfig(verbose=True)
```

Console Output:

- Model initialization details
- Time step values
- Derivative calculations
- Stock updates

Production:

```
python

config = SimulationConfig(verbose=False)
```

Error Handling

Error Categories

1. Validation Errors (400-level)

```
python
```

```
# Model structure errors
"Model must have at least one stock"
"Undefined variable: birth_rate"
"Circular reference detected"
```

2. Equation Errors

```
python

# Syntax errors
"Syntax error in equation: 2 +"

# Undefined variables
"Undefined variable: missing_stock"

# Type errors
"Function not allowed: eval"
```

3. Numerical Errors

```
python

# Division by zero
"Division by zero in equation: 1/x"

# Overflow
"Math range error: exp(1000)"

# Invalid domain
"Math domain error: log(-1)"
```

Error Response Format

```
json

{
  "success": false,
  "time": [],
  "results": {},
  "error": "Detailed error message here"
}
```

Exception Handling Pattern

```python
try:
    # Simulation logic
    model = SystemDynamicsModel(elements, links, verbose)
    result = model.simulate_euler(config)
except ValueError as e:
    # User input errors
    return {"error": str(e)}
except Exception as e:
    # Unexpected errors
    logger.error(f"Unexpected error: {e}")
    return {"error": "Internal server error"}
```

Performance Considerations

Optimization Tips

1. Time Step Selection

```python
# Balance accuracy vs speed
time_step = 0.1   # More accurate, slower
time_step = 1.0   # Faster, less accurate
```

2. Model Complexity

- **Number of stocks**: Linear impact on performance
- **Equation complexity**: Logarithmic impact
- **Time range**: Linear impact on memory

3. Caching

```python
```

```
# Cache parsed equations (future optimization)
equation_cache = {}
if equation not in equation_cache:
    equation_cache[equation] = parse_equation(equation)
```

Memory Usage

Estimation:

```
python

memory_per_step = (
    num_elements * 8 bytes (float64) +
    overhead ~100 bytes
)

total_memory = memory_per_step * num_time_steps
```

Example:

- 10 elements, 1000 time steps
- ~80KB + overhead ≈ 100KB

Scalability

Current Limits:

- Elements: Up to 1000 (practical)
- Time steps: Up to 100,000 (practical)
- Concurrent requests: Depends on server config

Bottlenecks:

- Equation parsing (can be cached)
- Numerical integration (inherently sequential)
- Memory for large time series

Security

Input Validation

Pydantic Models:

- Type validation
- Required field enforcement
- Value range checks

Equation Safety:

- AST parsing (no arbitrary code execution)
- Whitelisted operations only
- Sandboxed execution environment

Threat Mitigation

Code Injection

✅ **Protected**: AST-based parsing, no `eval()`

DoS Attacks

⚠️ **Partial**: Consider adding:

- Request rate limiting
- Maximum simulation time
- Memory limits

Data Validation

✅ **Protected**: Pydantic validation

Production Recommendations

```python
```

```
# 1. Use specific CORS origins
allow_origins=["https://yourdomain.com"]

# 2. Add authentication
from fastapi.security import HTTPBearer

# 3. Add rate limiting
from slowapi import Limiter

# 4. Add request timeouts
uvicorn.run(app, timeout_keep_alive=30)

# 5. Use HTTPS
uvicorn.run(app, ssl_keyfile="key.pem", ssl_certfile="cert.pem")
```

Troubleshooting

Common Issues

Issue: "Address already in use"

bash

```
# Find process on port 8000
# Windows:
netstat -ano | findstr :8000
taskkill /PID <PID> /F

# Linux/Mac:
lsof -ti:8000 | xargs kill -9
```

Issue: "Module not found"

bash

```
# Ensure virtual environment is activated
source venv/bin/activate  # Mac/Linux
venv\Scripts\activate     # Windows

# Reinstall dependencies
pip install -r requirements.txt
```

Issue: "Import errors"

```bash
# Check Python version
python --version  # Should be 3.8+

# Verify NumPy installation
python -c "import numpy; print(numpy.__version__)"
```

Development Workflow

Testing

```bash
# Run test suite
python test_simulation.py

# Test specific endpoint
curl -X GET http://localhost:8000/health

# Test simulation
curl -X POST http://localhost:8000/simulate \
  -H "Content-Type: application/json" \
  -d @test_model.json
```

Debugging

```python
```

```
# Enable verbose logging
config = SimulationConfig(verbose=True)

# Add breakpoints
import pdb; pdb.set_trace()

# Log to file
import logging
logging.basicConfig(
    filename='simulation.log',
    level=logging.DEBUG
)
```

Version Control

bash

```
# Recommended .gitignore
venv/
__pycache__/
*.pyc
*.log
.env
```

API Testing Examples

Using cURL

bash

```
# Health check
curl http://localhost:8000/health

# Simulate
curl -X POST http://localhost:8000/simulate \
  -H "Content-Type: application/json" \
  -d '{
    "elements": [...],
    "links": [...],
    "config": {
      "start_time": 0,
      "end_time": 100,
      "time_step": 1,
      "verbose": false
    }
  }'
```

Using Python requests

```
python
```

```python
import requests

# Simulate
response = requests.post(
    'http://localhost:8000/simulate',
    json={
        'elements': [...],
        'links': [...],
        'config': {
            'start_time': 0,
            'end_time': 100,
            'time_step': 1,
            'verbose': True
        }
    }
)

result = response.json()
if result['success']:
    print(f"Simulation completed: {len(result['time'])} time points")
else:
    print(f"Error: {result['error']}")
```

Appendix

File Structure

```
backend/
├── main.py              # Main application
├── requirements.txt     # Dependencies
├── test_simulation.py   # Test suite
└── README.md            # Documentation
```

Dependencies Explanation

- **FastAPI**: Modern, fast web framework with automatic API documentation
- **Uvicorn**: ASGI server for running FastAPI
- **Pydantic**: Data validation using Python type annotations
- **NumPy**: Efficient numerical arrays and operations

- **SciPy**: Scientific computing (future: advanced integration methods)
- **python-multipart**: Support for form data (future file uploads)

Future Enhancements

1. **Additional Integration Methods**:
   - Runge-Kutta 4th order (RK4)
   - Adaptive step size
   - Stiff solvers
2. **Advanced Features**:
   - Parameter sensitivity analysis
   - Optimization algorithms
   - Stochastic simulation
   - Parallel execution
3. **Database Integration**:
   - Store models
   - Result caching
   - User sessions
4. **Authentication**:
   - JWT tokens
   - User management
   - Access control

Version: 1.0.0

Last Updated: 2024

Authors: System Dynamics Development Team