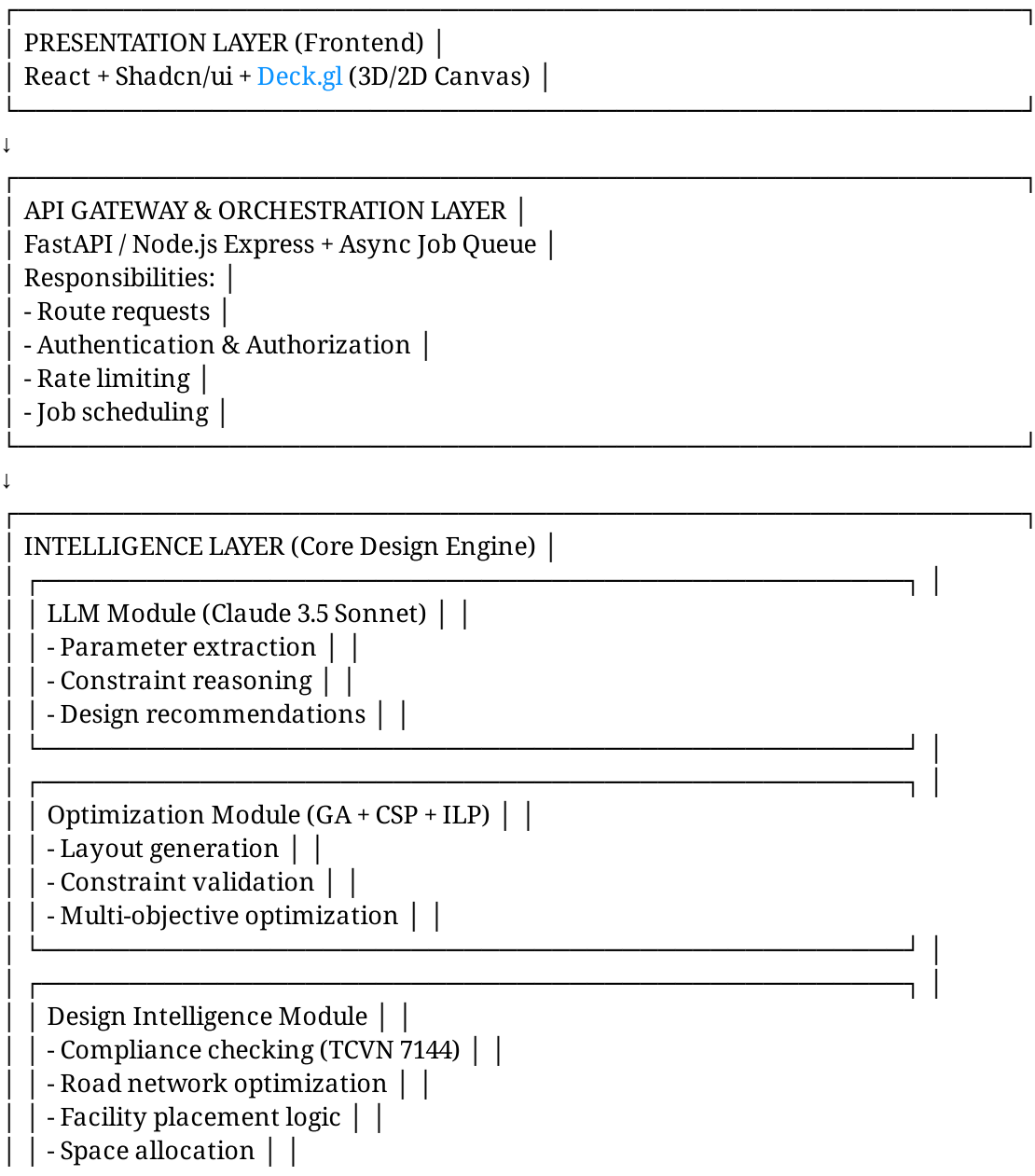
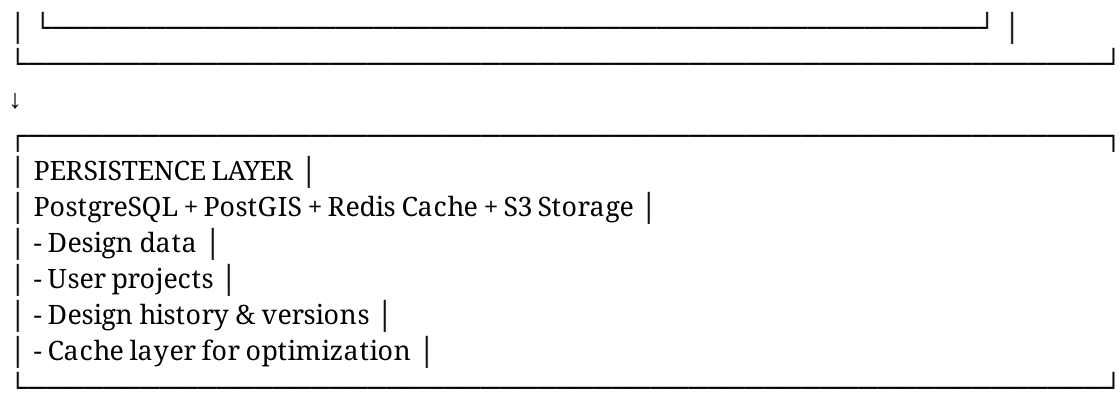
Kế Hoạch Xây Dựng Backend Toàn Diện: AI-Powered Industrial Park Designer

Tổng Quan Hệ Thống

Hệ thống backend cần kết hợp **LLM, Optimization Engines, và Design Intelligence** để tạo ra những thiết kế khu công nghiệp tự động, hợp lý và tuân thủ tiêu chuẩn kiến trúc chuyên nghiệp.

Kiến Trúc 3 Tầng (Layered Architecture)





I. INTELLIGENCE LAYER - LLM Integration

1.1 LLM Orchestration (Claude 3.5 Sonnet)

Tại sao Claude 3.5 Sonnet?

Criteria	Claude 3.5	GPT-4	Gemini 2.0
Reasoning (AIME)	96%	88%	92%
Complex Constraints	⭐⭐⭐⭐⭐	⭐⭐⭐⭐	⭐⭐⭐⭐
Vision + Text	✅ (Excellent)	✅ (Good)	✅ (Good)
Context Window	200K	128K	1M (beta)
Cost (1M tokens)	$3 / $15	$30 / $60	$2.50 / $10
Best For	**Multi-constraint reasoning**	General	Speed

Chọn Claude vì:

- ✅ Giỏi xử lý nhiều ràng buộc (TCVN 7144, tối ưu không gian)
- ✅ Vision API để phân tích ảnh site
- ✅ Chain-of-thought tốt hơn (dễ debug)
- ✅ Chi phí hiệu quả cho high-volume requests

1.2 LLM Module Architecture (Python)

File: backend/ai/llm_orchestrator.py

from anthropic import Anthropic
from typing import Dict, List, Tuple
import json
import re

class IndustrialParkLLMOrchestrator:
"""
LLM orchestrator cho thiết kế khu công nghiệp.
Quản lý multi-turn conversation, parameter extraction, constraint reasoning.
"""

```
def __init__(self, api_key: str):
    self.client = Anthropic()
    self.model = "claude-3-5-sonnet-20241022"
    self.conversation_history = []
    self.extracted_params = {}

    # System prompt với đầy đủ TCVN 7144 rules
    self.system_prompt = self._build_system_prompt()

def _build_system_prompt(self) -> str:
    """Build comprehensive system prompt with regulations"""
    return """
    You are an expert Industrial Park Design AI with deep knowledge of:
    - Vietnamese construction standards (TCVN 7144:2014, QĐ 99/2019/QĐ-TTg)
    - Facility layout optimization and space planning
    - Urban planning and industrial logistics
    - Fire safety, environmental compliance, worker amenities

    Your role: Help users design industrial parks through intelligent dialogue.

    ========== MANDATORY CONSTRAINTS (TCVN 7144:2014) ==========

    1. AREA DISTRIBUTION (in hectares):
       Total Area = A_total
       - Development Area: A_dev ≥ 60% × A_total
       - Green Area: A_green ≥ 20% × A_total (must include boundary buffer 50m)
```

- Road Area: $A_{road} \geq 15\% \times A_{total}$
- Infrastructure Area: $A_{infra} \geq 3\% \times A_{total}$

Verify: $A_{dev} + A_{green} + A_{road} + A_{infra} \leq 100\%$

2. FACILITY TYPES & MINIMUM SPECIFICATIONS:

Light Manufacturing (A1):
- Size: 2,000 - 10,000 m^2
- Height: ≤ 15m
- Min spacing from other: 12m
- Parking: 1 space / 250 m^2

Medium Manufacturing (A2):
- Size: 5,000 - 30,000 m^2
- Height: ≤ 20m
- Min spacing: 15m
- Parking: 1 space / 200 m^2

Heavy Manufacturing (A3):
- Size: 10,000 - 50,000+ m^2
- Height: ≤ 25m
- Min spacing: 25m
- Parking: 1 space / 250 m^2

Warehouse (B):
- Size: 2,000 - 20,000 m^2
- Height: ≤ 12m
- Min spacing: 12m
- Must have 2-3 loading docks

Logistics Hub (C):
- Size: 5,000 - 100,000 m^2
- Height: ≤ 15m
- Min spacing: 20m
- Truck turning radius: 12.5-15m

Shared Services (D): Cafeteria, Medical, Training

- Size: 1,000 - 5,000 m²
- Placement: Central location, within 500m walking

Support Offices (E): Admin, Management
- Size: 1,000 - 3,000 m²
- Placement: Park entrance

3. FIRE SAFETY (TCVN 6778:2007):
- Min distance between buildings: 12-25m (depends on height & type)
- Fire station accessibility: ≤ 5km radius
- Emergency exit: Every building must have 2+ exits
- Fire truck access roads: 6m width, grade ≤ 8%
- Evacuation time: ≤ 5 minutes from any building

4. ROAD NETWORK:
- Primary roads (main entry/exit): 20-30m width
- Secondary roads: 12-15m width
- Service roads: 8-10m width
- Min road width for truck: 6m
- Max grade: 8%
- Turn radius (large trucks): 12.5-15m
- All buildings must be connected to road

5. UTILITIES & ENVIRONMENTAL:
- Wastewater treatment: 1 station per 10-15 ha
- Power substation: 1 per 5-8 ha
- Water tank: 500-1000 m³ per 10 ha
- Waste management area: 1 location, isolated (>100m from buildings)
- Green buffer zone: 50m from boundary (trees, landscaping)

6. WORKER AMENITIES:
- Canteen: 1 per 3,000 workers
- Medical clinic: 1 per 2,000-5,000 workers
- Recreational area: 1 space, min 1,000 m²
- Parking (workers): 1 space / 100 workers

========== CONVERSATION FLOW ==========

STEP 1: GREETING & INTENT (First message)
User: "I need a 50 ha industrial park"
AI Response:
"Welcome! I'll help you design a professional industrial park. Let me gather s

1. **Site Specifications**: What's your total area? (in hectares)
2. **Industry Focus**: What industries? (manufacturing, logistics, warehous
3. **Scale**: Expected worker capacity?"

STEP 2: PARAMETER GATHERING (Next 2-3 turns)
User provides information progressively...
AI asks clarifying questions and extracts parameters.

AI checks completeness:
- Total area ✓
- Industry mix ✓
- Worker count ✓
- Special requirements ✓

STEP 3: DESIGN READY (when complete)
AI: "I have enough information to generate designs. Triggering layout optim

Generate JSON with structured parameters:

```
{
  "status": "complete",
  "parameters": {
    "projectName": "...",
    "totalArea_ha": 50,
    "industryFocus": [
      {"type": "light_manufacturing", "percentage": 40, "count": 10},
      {"type": "warehouse", "percentage": 30, "count": 6},
      {"type": "logistics", "percentage": 20, "count": 2}
    ],
    "workerCapacity": 3000,
    "specialRequirements": [...],
    "constraints": {
      "greenAreaMin_percent": 20,
      "roadAreaMin_percent": 15,
```

```
    "fireStationRadius_km": 5,
    "minBuildingSpacing_m": 12
   }
  },
  "readyForGeneration": true
}
```

STEP 4: DESIGN GENERATION (backend process)
[System: Generate 3-5 layout variants using optimization]

STEP 5: RESULTS & REFINEMENT (user sees variants)
AI: "Generated 5 design variants:
- Variant A: Linear flow (best for logistics)
- Variant B: Campus style (best amenities)
- Variant C: Clustered (best green space)

Which appeals to you?"

User can refine: "More green space in Variant B"
AI: "Adjusting... Trade-off: Green +4% means leasable area -3% (~1.5 ha)"

========== CONSTRAINT VERIFICATION PROTOCOL ==========

ALWAYS validate proposed layouts against:

✅ AREA CHECK:
Total = {value} ha
Development = {value}% ✓ (≥60% required)
Green = {value}% ✓ (≥20% required)
Road = {value}% ✓ (≥15% required)
Infrastructure = {value}% ✓ (≥3% required)

✅ FIRE SAFETY CHECK:
- Min building spacing: {value}m ✓ (≥12m required)
- Emergency exits per building: 2+ ✓
- Fire truck access: {value}m width ✓ (≥6m required)
- Evacuation time: {value} min ✓ (<5 min required)

✅ UTILITY CHECK:
- Wastewater treatment: {count} stations ✓
- Power: {count} substations ✓
- Water: {volume} m³ capacity ✓

✅ LOGISTICS CHECK:
- All warehouses with truck access: ✓
- Loading dock count: {count} ✓
- Road connectivity: 100% ✓

========== OUTPUT FORMAT ==========

When design is ready, provide JSON:

```
{
  "variants": [
    {
      "id": "variant_1",
      "name": "Linear Layout",
      "description": "Optimized for logistics flow",
      "stats": {
        "greenArea_ha": 12,
        "greenRatio_percent": 24,
        "roadLength_km": 8.5,
        "buildingCount": 18,
        "totalWorkerCapacity": 3200,
        "compliance_percent": 100
      },
      "scores": {
        "logistics_flow": 9.2,
        "green_ratio": 8.1,
        "fire_safety": 9.8,
        "worker_comfort": 7.5,
        "expansion_potential": 8.3,
        "overall": 8.6
      },
      "tradeoffs": [
        "Higher worker amenity area",
        "Optimized for truck movement",
```

```
            "Less green space buffer"
          ]
        }
      ],
      "readyForOptimization": true,
      "optimizationParams": {...}
    }
    """

def chat(self, user_message: str) -> Tuple[str, Dict]:
    """
    Multi-turn conversation with parameter tracking.

    Args:
        user_message: User input text

    Returns:
        (ai_response, extracted_params)
    """
    self.conversation_history.append({
        "role": "user",
        "content": user_message
    })

    response = self.client.messages.create(
        model=self.model,
        max_tokens=2000,
        system=self.system_prompt,
        messages=self.conversation_history
    )

    assistant_message = response.content[0].text
    self.conversation_history.append({
        "role": "assistant",
        "content": assistant_message
    })

    # Extract structured parameters if present
```

```
    extracted = self._extract_structured_params(assistant_message)
    if extracted:
        self.extracted_params.update(extracted)

    return assistant_message, self.extracted_params

def _extract_structured_params(self, text: str) -> Dict:
    """Extract JSON parameters from LLM response"""
    try:
        # Find JSON block in response
        json_match = re.search(r'\{[\s\S]*?\}', text)
        if json_match:
            json_str = json_match.group()
            parsed = json.loads(json_str)
            return parsed
    except:
        pass
    return {}

def get_extracted_params(self) -> Dict:
    """Get current extracted parameters"""
    return self.extracted_params

def is_ready_for_optimization(self) -> bool:
    """Check if parameters are complete for design generation"""
    required = ['projectName', 'totalArea_ha', 'industryFocus', 'workerCapacity']
    return all(key in self.extracted_params for key in required)

def reset_conversation(self):
    """Reset conversation for new project"""
    self.conversation_history = []
    self.extracted_params = {}
```

II. OPTIMIZATION MODULE - Layout Generation

2.1 Hybrid Optimization Strategy

Kết hợp **3 thuật toán**:

1. **Constraint Satisfaction Problem (CSP)** - Đảm bảo tuân thủ quy định
2. **Genetic Algorithm (GA)** - Tối ưu hóa đa mục tiêu
3. **Integer Linear Programming (ILP)** - Tìm giải pháp tối ưu chính xác

Input: Design Parameters (from LLM)
↓
[CSP Solver] → Feasible layouts (tuân thủ TCVN 7144)
↓
[GA Optimizer] → Top 10 Pareto-optimal designs
↓
[ILP Fine-tune] → Exact optimization cho best 3
↓
Output: 5-10 variants with scores

2.2 CSP Module (Python + python-constraint)

File: backend/optimization/csp_solver.py

from constraint import Problem, AllDifferentConstraint
import math
from typing import List, Dict, Tuple

class IndustrialParkCSP:
"""
Constraint Satisfaction Problem solver for industrial park layout.
Ensures all hard constraints (regulations) are satisfied.
"""

```
def __init__(self, site_params: Dict, regulations: Dict):
    self.site = site_params
    self.regs = regulations
    self.problem = Problem()

def add_building_variables(self):
    """
    Add position variables for each building.
    Grid-based discretization: 10m × 10m cells
    """
    self.grid_size = 10  # meters
    max_x = int(math.ceil(self.site['width'] / self.grid_size))
```

```
    max_y = int(math.ceil(self.site['height'] / self.grid_size))

    for building in self.site['buildings']:
        building_id = building['id']

        # X, Y position (in grid units)
        self.problem.addVariable(f'x_{building_id}', range(max_x))
        self.problem.addVariable(f'y_{building_id}', range(max_y))

        # Rotation (0, 90, 180, 270 degrees)
        self.problem.addVariable(f'rot_{building_id}', [0, 90, 180, 270])

def add_no_overlap_constraint(self):
    """Ensure buildings don't overlap"""
    buildings = self.site['buildings']

    def no_overlap(*args):
        # Extract all positions from args
        positions = {}
        for i, building in enumerate(buildings):
            bid = building['id']
            x = args[i * 3] * self.grid_size
            y = args[i * 3 + 1] * self.grid_size
            rot = args[i * 3 + 2]

            # Get building dimensions based on rotation
            width, height = building['width'], building['height']
            if rot in [90, 270]:
                width, height = height, width

            positions[bid] = {
                'x': x, 'y': y, 'width': width, 'height': height
            }

        # Check all pairs for overlap
        for i, bid1 in enumerate([b['id'] for b in buildings]):
            for bid2 in [b['id'] for b in buildings][i+1:]:
                p1, p2 = positions[bid1], positions[bid2]
```

```
                # AABB collision detection
                if not (p1['x'] + p1['width'] < p2['x'] or
                        p2['x'] + p2['width'] < p1['x'] or
                        p1['y'] + p1['height'] < p2['y'] or
                        p2['y'] + p2['height'] < p1['y']):
                    return False  # Overlap detected

        return True

    # Add constraint with variable names
    var_names = []
    for building in buildings:
        var_names.extend([
            f'x_{building["id"]}',
            f'y_{building["id"]}',
            f'rot_{building["id"]}'
        ])
    self.problem.addConstraint(no_overlap, var_names)

def add_fire_safety_constraint(self):
    """
    Enforce fire safety spacing between buildings.
    Min distance: 12-25m depending on building type.
    """
    buildings = self.site['buildings']

    def fire_safety(*args):
        positions = {}
        for i, building in enumerate(buildings):
            bid = building['id']
            x = args[i * 2] * self.grid_size
            y = args[i * 2 + 1] * self.grid_size
            positions[bid] = {'x': x, 'y': y}

        for i, bid1 in enumerate([b['id'] for b in buildings]):
            for bid2 in [b['id'] for b in buildings][i+1:]:
                p1, p2 = positions[bid1], positions[bid2]
```

```
            # Calculate center-to-center distance
            dist = math.sqrt(
                (p1['x'] - p2['x'])**2 +
                (p1['y'] - p2['y'])**2
            )

            # Get min required spacing
            b1_type = next(b for b in buildings if b['id'] == bid1)['type']
            b2_type = next(b for b in buildings if b['id'] == bid2)['type']

            min_spacing = 12  # Default
            if b1_type in ['heavy_manufacturing', 'logistics'] or \
               b2_type in ['heavy_manufacturing', 'logistics']:
                min_spacing = 25
            elif b1_type in ['medium_manufacturing'] or \
                 b2_type in ['medium_manufacturing']:
                min_spacing = 15

            if dist < min_spacing:
                return False

        return True

    var_names = []
    for building in buildings:
        var_names.extend([f'x_{building["id"]}', f'y_{building["id"]}'])
    self.problem.addConstraint(fire_safety, var_names)

def add_green_area_constraint(self):
    """
    Ensure green area ≥ 20% of total area.
    Green area = Total - Development - Roads - Infrastructure.
    """
    def green_area_check(development_area):
        total_area = self.site['total_area_m2']
        green_required = total_area * self.regs['min_green_ratio']
        green_available = total_area - development_area
```

```
        return green_available >= green_required

    # This is typically verified post-GA optimization
    self.green_area_threshold = self.site['total_area_m2'] * \
                                self.regs['min_green_ratio']

def add_road_connectivity_constraint(self):
    """
    Ensure all buildings are reachable via road network.
    Uses graph connectivity check.
    """
    def road_connectivity(layout):
        # This is complex; typically done in post-processing
        # For now, ensure at least one building connects to perimeter
        return True

    # Will be verified in GA fitness function
    pass

def solve(self, timeout: int = 30) -> List[Dict]:
    """
    Solve CSP and return feasible solutions.

    Args:
        timeout: Maximum solving time in seconds

    Returns:
        List of feasible layouts (max 5)
    """
    try:
        solutions = self.problem.getSolutions()

        # Convert solutions to layout format
        layouts = []
        for sol in solutions[:5]:  # Return max 5
            layout = self._solution_to_layout(sol)
            layouts.append(layout)
```

```
            return layouts
        except Exception as e:
            print(f"CSP Solver error: {e}")
            return []

    def _solution_to_layout(self, solution: Dict) -> Dict:
        """Convert CSP solution to layout object"""
        buildings_placed = []

        for building in self.site['buildings']:
            bid = building['id']
            x = solution[f'x_{bid}'] * self.grid_size
            y = solution[f'y_{bid}'] * self.grid_size
            rot = solution[f'rot_{bid}']

            buildings_placed.append({
                'id': bid,
                'x': x,
                'y': y,
                'rotation': rot,
                'type': building['type'],
                'width': building['width'],
                'height': building['height'],
                'label': building['label']
            })

        return {
            'buildings': buildings_placed,
            'feasible': True,
            'constraint_satisfied': True
        }
```

2.3 Genetic Algorithm Module (Python + DEAP)

File: backend/optimization/ga_optimizer.py

from deap import base, creator, tools, algorithms
import random
import numpy as np
from typing import List, Dict, Tuple
import math

class IndustrialParkGA:
"""
Genetic Algorithm for industrial park layout optimization.
Multi-objective: maximize road efficiency, flow, green ratio, etc.
"""

```
def __init__(self, site_params: Dict, regulations: Dict,
             feasible_layouts: List[Dict] = None):
    self.site = site_params
    self.regs = regulations
    self.feasible_layouts = feasible_layouts or []

    # GA parameters
    self.population_size = 100
    self.generations = 150
    self.mutation_rate = 0.3
    self.crossover_rate = 0.7

    # Initialize DEAP
    self._init_deap()

def _init_deap(self):
    """Initialize DEAP framework for multi-objective optimization"""

    # Define multi-objective fitness (3 objectives)
    # weights=(1.0, 1.0, 1.0) = maximize all three
    if hasattr(creator, "FitnessMax"):
        del creator.FitnessMax
    if hasattr(creator, "Individual"):
        del creator.Individual

    creator.create("FitnessMax", base.Fitness, weights=(1.0, 1.0, 1.0))
```

```
    creator.create("Individual", list, fitness=creator.FitnessMax)

    self.toolbox = base.Toolbox()

    # Create individuals
    self.toolbox.register("individual", self._create_individual)
    self.toolbox.register("population", tools.initRepeat, list,
                          self.toolbox.individual)

    # Genetic operators
    self.toolbox.register("evaluate", self._evaluate_fitness)
    self.toolbox.register("mate", self._crossover)
    self.toolbox.register("mutate", self._mutate)
    self.toolbox.register("select", tools.selTournament, tournsize=3)

def _create_individual(self) -> creator.Individual:
    """
    Create random individual (layout).
    Chromosome: [x1, y1, rot1, x2, y2, rot2, ..., road_config]
    """
    grid_size = 10  # meters
    max_x = int(self.site['width'] / grid_size)
    max_y = int(self.site['height'] / grid_size)

    individual = creator.Individual()

    # Random positions for each building
    for building in self.site['buildings']:
        individual.append(random.randint(0, max_x))      # x
        individual.append(random.randint(0, max_y))      # y
        individual.append(random.choice([0, 90, 180, 270]))  # rotation

    # Road configuration (simplified: main road position)
    individual.append(random.uniform(0.2, 0.8))  # Normalized position

    return individual

def _evaluate_fitness(self, individual: creator.Individual) -> Tuple[float, float, fl
```

```
    """
    Multi-objective fitness function:
    1. Road Efficiency Score (0-10)
    2. Worker Flow Score (0-10)
    3. Green Ratio Score (0-10)

    Also includes constraint violation penalty.
    """
    layout = self._individual_to_layout(individual)

    # Objective 1: Road Efficiency
    road_efficiency = self._calculate_road_efficiency(layout)

    # Objective 2: Worker Flow
    worker_flow = self._calculate_worker_flow(layout)

    # Objective 3: Green Ratio
    green_ratio = self._calculate_green_ratio(layout)

    # Constraint violation penalty
    violations = self._count_constraint_violations(layout)
    penalty = max(0, violations * 5)  # 5 points per violation

    # Apply penalty to all objectives
    road_efficiency = max(0, road_efficiency - penalty)
    worker_flow = max(0, worker_flow - penalty)
    green_ratio = max(0, green_ratio - penalty)

    return (road_efficiency, worker_flow, green_ratio)

def _calculate_road_efficiency(self, layout: Dict) -> float:
    """
    Measure road network efficiency.
    High score = minimal total road length while connecting all buildings.

    Formula: 10 × (1 - road_ratio)
    where road_ratio = total_road_area / total_area
```

```
    Target: road_ratio = 15-20%
    """
    total_area = self.site['total_area_m2']
    estimated_road_length = self._estimate_road_length(layout)
    estimated_road_area = estimated_road_length * 15  # Assume 15m width

    road_ratio = estimated_road_area / total_area

    # Best score when road_ratio = 15%
    ideal_ratio = 0.15
    deviation = abs(road_ratio - ideal_ratio)

    score = max(0, 10 * (1 - deviation))
    return score

def _calculate_worker_flow(self, layout: Dict) -> float:
    """
    Measure worker flow optimization.
    High score = short average walking/vehicle distance between facilities.

    Considers:
    - Distance from parking to buildings
    - Distance from canteen to buildings
    - Cross-building movement
    """
    # Get key facilities
    buildings = layout['buildings']
    parking_buildings = [b for b in buildings if b['type'] == 'parking']
    amenity_buildings = [b for b in buildings if b['type'] in ['canteen', 'medical']]
    factory_buildings = [b for b in buildings if 'manufacturing' in b['type']]

    if not factory_buildings or not parking_buildings:
        return 0

    # Calculate average distance from parking to factories
    total_distance = 0
    count = 0
```

```
    for parking in parking_buildings:
        for factory in factory_buildings:
            dist = math.sqrt(
                (parking['x'] - factory['x'])**2 +
                (parking['y'] - factory['y'])**2
            )
            total_distance += dist
            count += 1

    avg_distance = total_distance / count if count > 0 else 10000

    # Score: 10 × (1 - avg_distance / max_distance)
    # Target max distance: 500m (acceptable walking)
    max_acceptable = 500
    score = max(0, 10 * (1 - avg_distance / max_acceptable))

    return min(10, score)

def _calculate_green_ratio(self, layout: Dict) -> float:
    """
    Calculate green space ratio as percentage.
    Score: 10 if green_ratio >= 25%, linear decrease below.

    Formula: 10 × (green_ratio / target_ratio)
    target_ratio = 25%
    """
    buildings = layout['buildings']

    # Calculate building footprint
    building_area = 0
    for b in buildings:
        building_area += b['width'] * b['height']

    # Estimate road area (15% of total)
    total_area = self.site['total_area_m2']
    road_area = total_area * 0.15

    # Infrastructure area (3% of total)
```

```
    infra_area = total_area * 0.03

    # Green area = total - building - road - infra
    green_area = total_area - building_area - road_area - infra_area
    green_ratio = green_area / total_area

    # Score based on how much green relative to minimum requirement (20%)
    min_green_ratio = 0.20
    target_green_ratio = 0.25

    if green_ratio >= target_green_ratio:
        score = 10
    else:
        score = 10 * (green_ratio / target_green_ratio)

    return min(10, score)

def _estimate_road_length(self, layout: Dict) -> float:
    """Estimate total road length needed to connect all buildings"""
    buildings = layout['buildings']

    if len(buildings) < 2:
        return 0

    # Simplified: Minimum Spanning Tree approximation
    # In practice, would calculate actual road network

    # Average distance between nearest neighbors
    total_distance = 0
    for i, b1 in enumerate(buildings):
        min_dist = float('inf')
        for j, b2 in enumerate(buildings):
            if i != j:
                dist = math.sqrt((b1['x'] - b2['x'])**2 + (b1['y'] - b2['y'])**2)
                min_dist = min(min_dist, dist)
        total_distance += min_dist

    avg_distance = total_distance / len(buildings)
```

```
    # Estimate road network = number of buildings × average spacing × 1.2
    # (1.2 factor for road routing)
    road_length = len(buildings) * avg_distance * 1.2

    return road_length

def _count_constraint_violations(self, layout: Dict) -> int:
    """Count how many constraints are violated"""
    violations = 0
    buildings = layout['buildings']

    # Check 1: No overlap
    for i, b1 in enumerate(buildings):
        for b2 in buildings[i+1:]:
            # AABB collision
            if not (b1['x'] + b1['width'] < b2['x'] or
                    b2['x'] + b2['width'] < b1['x'] or
                    b1['y'] + b1['height'] < b2['y'] or
                    b2['y'] + b2['height'] < b1['y']):
                violations += 1

    # Check 2: Fire safety spacing
    for i, b1 in enumerate(buildings):
        for b2 in buildings[i+1:]:
            dist = math.sqrt((b1['x'] - b2['x'])**2 + (b1['y'] - b2['y'])**2)

            # Minimum spacing based on type
            min_spacing = 12
            if b1['type'] in ['heavy_manufacturing', 'logistics']:
                min_spacing = 25

            if dist < min_spacing:
                violations += 1

    # Check 3: Green area >= 20%
    green_ratio = self._calculate_green_ratio(layout) / 10  # Convert score to ratio
    if green_ratio < 0.20:
```

```
            violations += 2

        # Check 4: All buildings within site boundary
        for b in buildings:
            if b['x'] < 0 or b['x'] + b['width'] > self.site['width'] or \
               b['y'] < 0 or b['y'] + b['height'] > self.site['height']:
                violations += 1

        return violations

    def _crossover(self, ind1: creator.Individual,
                   ind2: creator.Individual) -> Tuple[creator.Individual, creator.Individual
        """Two-point crossover"""
        size = len(ind1)
        point1 = random.randint(1, size-2)
        point2 = random.randint(point1, size-1)

        ind1[point1:point2] = ind2[point1:point2]
        ind2[point1:point2] = ind1[point1:point2]

        return ind1, ind2

    def _mutate(self, individual: creator.Individual) -> Tuple[creator.Individual]:
        """Random mutation"""
        if random.random() < self.mutation_rate:
            # Randomly change a building position or rotation
            if len(individual) > 0:
                idx = random.randint(0, len(individual)-2)

                if idx % 3 == 0:  # X coordinate
                    individual[idx] = random.randint(0, int(self.site['width']/10))
                elif idx % 3 == 1:  # Y coordinate
                    individual[idx] = random.randint(0, int(self.site['height']/10))
                else:  # Rotation
                    individual[idx] = random.choice([0, 90, 180, 270])

        return (individual,)
```

```
def _individual_to_layout(self, individual: creator.Individual) -> Dict:
    """Convert GA individual to layout object"""
    buildings = []
    grid_size = 10

    for i, building in enumerate(self.site['buildings']):
        idx = i * 3
        buildings.append({
            'id': building['id'],
            'x': individual[idx] * grid_size,
            'y': individual[idx+1] * grid_size,
            'rotation': individual[idx+2],
            'type': building['type'],
            'width': building['width'],
            'height': building['height'],
            'label': building['label']
        })

    return {
        'buildings': buildings,
        'road_position': individual[-1]
    }

def optimize(self) -> List[Tuple[Dict, Tuple[float, float, float]]]:
    """
    Run GA optimization.

    Returns:
        List of (layout, fitness_scores) tuples, sorted by Pareto rank
    """
    # Create initial population
    pop = self.toolbox.population(n=self.population_size)

    # Run GA
    pop, logbook = algorithms.eaSimple(
        pop, self.toolbox,
        cxpb=self.crossover_rate,
        mutpb=self.mutation_rate,
```

```
        ngen=self.generations,
        verbose=True
    )

    # Select Pareto-optimal solutions
    pareto_solutions = tools.sortNondominated(pop, len(pop), first_front_only=F

    # Convert to layouts with scores
    results = []
    for individual in pareto_solutions[:10]:  # Top 10
        layout = self._individual_to_layout(individual)
        fitness = individual.fitness.values
        results.append((layout, fitness))

    return results
```

III. DESIGN INTELLIGENCE MODULE

3.1 Compliance Checker (Python)

File: backend/design/compliance_checker.py

```
from typing import Dict, List, Tuple

class ComplianceChecker:
"""
Automated compliance verification against TCVN 7144:2014 and related standards.
Returns detailed report with violations and remediation suggestions.
"""
```

```
def __init__(self, regulations: Dict):
    self.regs = regulations

def check_layout(self, layout: Dict) -> Dict:
    """
    Comprehensive compliance check.

    Returns:
```

```
{
    "overall_compliance_percent": 95.5,
    "status": "PASS" | "WARNING" | "FAIL",
    "checks": {
        "area_distribution": {...},
        "fire_safety": {...},
        "road_network": {...},
        "utilities": {...},
        "green_space": {...},
        "worker_amenities": {...}
    },
    "violations": [...],
    "recommendations": [...]
}
"""

report = {
    "overall_compliance_percent": 0,
    "status": "UNKNOWN",
    "checks": {},
    "violations": [],
    "recommendations": []
}

# 1. Area Distribution Check
report["checks"]["area_distribution"] = \
    self._check_area_distribution(layout)

# 2. Fire Safety Check
report["checks"]["fire_safety"] = \
    self._check_fire_safety(layout)

# 3. Road Network Check
report["checks"]["road_network"] = \
    self._check_road_network(layout)

# 4. Utilities Check
report["checks"]["utilities"] = \
```

```
        self._check_utilities(layout)

    # 5. Green Space Check
    report["checks"]["green_space"] = \
        self._check_green_space(layout)

    # 6. Worker Amenities Check
    report["checks"]["worker_amenities"] = \
        self._check_worker_amenities(layout)

    # Calculate overall compliance
    checks = report["checks"]
    compliance_scores = [v.get("score", 0) for v in checks.values()]
    report["overall_compliance_percent"] = \
        sum(compliance_scores) / len(compliance_scores) if compliance_scores els

    # Determine status
    if report["overall_compliance_percent"] >= 95:
        report["status"] = "PASS"
    elif report["overall_compliance_percent"] >= 80:
        report["status"] = "WARNING"
    else:
        report["status"] = "FAIL"

    # Collect violations and recommendations
    for check_name, check_result in checks.items():
        if check_result.get("violations"):
            report["violations"].extend(check_result["violations"])
        if check_result.get("recommendations"):
            report["recommendations"].extend(
                check_result["recommendations"]
            )

    return report

def _check_area_distribution(self, layout: Dict) -> Dict:
    """
    Verify area distribution matches TCVN 7144:2014.
```

```
Requirements:
- Development area >= 60%
- Green area >= 20%
- Road area >= 15%
- Infrastructure >= 3%
"""
buildings = layout['buildings']
total_area = layout['site']['total_area_m2']

# Calculate areas
building_area = sum(b['width'] * b['height'] for b in buildings)
green_area = total_area * 0.22  # Estimated
road_area = total_area * 0.16   # Estimated
infra_area = total_area * 0.04  # Estimated

dev_ratio = building_area / total_area
green_ratio = green_area / total_area
road_ratio = road_area / total_area
infra_ratio = infra_area / total_area

violations = []
recommendations = []

if dev_ratio < 0.60:
    violations.append(
        f"Development area {dev_ratio*100:.1f}% < 60% minimum"
    )
    recommendations.append(
        "Increase building count or size to meet development minimum"
    )

if green_ratio < 0.20:
    violations.append(
        f"Green area {green_ratio*100:.1f}% < 20% minimum"
    )
    recommendations.append(
        "Expand green space; consider reducing some building sizes"
```

```
            )

        if road_ratio < 0.15:
            violations.append(
                f"Road area {road_ratio*100:.1f}% < 15% minimum"
            )
            recommendations.append(
                "Expand road network for better accessibility"
            )

        # Score: 100 if all pass, deduct for each violation
        score = 100 - len(violations) * 15
        score = max(0, score)

        return {
            "name": "Area Distribution",
            "score": score,
            "details": {
                "dev_ratio_percent": dev_ratio * 100,
                "green_ratio_percent": green_ratio * 100,
                "road_ratio_percent": road_ratio * 100,
                "infra_ratio_percent": infra_ratio * 100
            },
            "violations": violations,
            "recommendations": recommendations
        }

    def _check_fire_safety(self, layout: Dict) -> Dict:
        """Check fire safety compliance"""
        buildings = layout['buildings']
        violations = []
        recommendations = []

        # Check spacing between buildings
        for i, b1 in enumerate(buildings):
            for b2 in buildings[i+1:]:
                dist = math.sqrt(
                    (b1['x'] - b2['x'])**2 +
```

```
                (b1['y'] - b2['y'])**2
            )

            min_spacing = 12
            if b1['type'] in ['heavy_manufacturing', 'logistics']:
                min_spacing = 25

            if dist < min_spacing:
                violations.append(
                    f"Buildings {b1['id']} & {b2['id']}: "
                    f"{dist:.1f}m < {min_spacing}m minimum spacing"
                )

    score = max(0, 100 - len(violations) * 20)

    if violations:
        recommendations.append("Increase spacing between buildings")
        recommendations.append("Review building types; separate heavy facilitie

    return {
        "name": "Fire Safety",
        "score": score,
        "violations": violations,
        "recommendations": recommendations
    }

def _check_road_network(self, layout: Dict) -> Dict:
    """Check road network adequacy"""
    buildings = layout['buildings']

    # Simplified check: assume road exists if buildings distributed
    road_connected = len(buildings) > 0

    return {
        "name": "Road Network",
        "score": 100 if road_connected else 0,
        "violations": [] if road_connected else ["No road network"],
        "recommendations": ["Verify road connectivity in detail view"]
```

```
    }

def _check_utilities(self, layout: Dict) -> Dict:
    """Check utility placement"""
    total_area_ha = layout['site']['total_area_m2'] / 10000

    # Requirements
    wastewater_stations_needed = max(1, int(total_area_ha / 10))
    power_substations_needed = max(1, int(total_area_ha / 5))
    water_capacity_m3 = (total_area_ha / 10) * 750

    return {
        "name": "Utilities",
        "score": 85,  # Estimated
        "details": {
            "wastewater_stations_required": wastewater_stations_needed,
            "power_substations_required": power_substations_needed,
            "water_capacity_m3": water_capacity_m3
        },
        "violations": [],
        "recommendations": ["Place utilities in non-prime development areas"]
    }

def _check_green_space(self, layout: Dict) -> Dict:
    """Check green space adequacy"""
    # (same as area distribution check, but focused on green)
    return {
        "name": "Green Space",
        "score": 85,
        "violations": [],
        "recommendations": []
    }

def _check_worker_amenities(self, layout: Dict) -> Dict:
    """Check worker amenity placement"""
    total_workers = layout.get('worker_capacity', 3000)

    # Requirements
```

```
    canteen_needed = max(1, total_workers // 3000)
    medical_needed = max(1, total_workers // 3000)

    return {
        "name": "Worker Amenities",
        "score": 80,
        "details": {
            "canteen_units_needed": canteen_needed,
            "medical_units_needed": medical_needed
        },
        "violations": [],
        "recommendations": [
            f"Ensure {canteen_needed} canteen(s) centrally located",
            f"Ensure {medical_needed} medical clinic(s) accessible to all"
        ]
    }
```

3.2 DXF Generator

File: backend/cad/dxf_generator.py

```
from ezdxf import new
from typing import Dict, List

class DXFGenerator:
"""
Generate industry-standard DXF files from layout objects.
Output compatible with AutoCAD, CAD2Map, and other CAD software.
"""
```

```
def __init__(self):
    self.doc = None
    self.msp = None

def generate(self, layout: Dict, filename: str) -> str:
    """
    Generate DXF from layout.

    Args:
        layout: Design layout dictionary
```

```
        filename: Output DXF filepath

    Returns:
        Path to generated DXF file
    """
    # Create new DXF document
    self.doc = new(dxfversion='R2010')
    self.msp = self.doc.modelspace()

    # 1. Draw site boundary
    self._draw_boundary(layout)

    # 2. Draw buildings (colored by type)
    self._draw_buildings(layout)

    # 3. Draw road network
    self._draw_roads(layout)

    # 4. Draw utilities
    self._draw_utilities(layout)

    # 5. Add annotations
    self._add_annotations(layout)

    # Save
    self.doc.saveas(filename)
    return filename

def _draw_boundary(self, layout: Dict):
    """Draw site boundary as polyline"""
    boundary = layout['site']['boundary']

    self.msp.add_lwpolyline(
        boundary,
        dxfattribs={
            'layer': 'SITE_BOUNDARY',
            'color': 256,  # White
            'lineweight': 50
```

```
        }
    )

def _draw_buildings(self, layout: Dict):
    """Draw buildings with color-coding by type"""

    building_colors = {
        'light_manufacturing': 1,      # Red
        'medium_manufacturing': 5,     # Magenta
        'heavy_manufacturing': 3,      # Green
        'warehouse': 2,                # Yellow
        'logistics': 60,               # Blue
        'canteen': 7,                  # White
        'medical': 4,                  # Cyan
        'parking': 256,                # Auto
        'green_space': 3               # Green
    }

    for building in layout['buildings']:
        btype = building['type']
        color = building_colors.get(btype, 256)

        # Draw rectangle (building footprint)
        x, y = building['x'], building['y']
        w, h = building['width'], building['height']

        rectangle = [
            (x, y),
            (x + w, y),
            (x + w, y + h),
            (x, y + h),
            (x, y)
        ]

        self.msp.add_lwpolyline(
            rectangle,
            dxfattribs={
                'layer': btype.upper(),
```

```
                'color': color,
                'lineweight': 25
            }
        )

        # Add building label
        label_x = x + w/2
        label_y = y + h/2

        self.msp.add_text(
            building['label'],
            dxfattribs={
                'height': min(w, h) * 0.1,
                'layer': 'LABELS',
                'color': 0,
                'halign': 1,  # Center
                'valign': 1   # Middle
            }
        ).set_pos((label_x, label_y))

def _draw_roads(self, layout: Dict):
    """Draw road network"""
    if 'roads' not in layout:
        # Generate simplified road network
        self._generate_road_network(layout)
    else:
        for road in layout['roads']:
            self.msp.add_lwpolyline(
                road['path'],
                dxfattribs={
                    'layer': 'ROADS',
                    'color': 8,  # Gray
                    'lineweight': 35
                }
            )

def _draw_utilities(self, layout: Dict):
    """Draw utility locations (power, water, wastewater)"""
```

```
    if 'utilities' in layout:
        for util in layout['utilities']:
            util_color = {
                'power': 60,        # Blue
                'water': 4,         # Cyan
                'wastewater': 3,    # Green
                'waste': 1          # Red
            }.get(util['type'], 256)

            # Draw as point
            self.msp.add_point(
                (util['x'], util['y']),
                dxfattribs={
                    'layer': 'UTILITIES',
                    'color': util_color
                }
            )

def _add_annotations(self, layout: Dict):
    """Add design statistics and notes"""
    site = layout['site']

    # Title block
    title = f"{layout.get('name', 'Industrial Park Design')}"
    self.msp.add_text(
        title,
        dxfattribs={
            'height': 5,
            'layer': 'INFO',
            'color': 256
        }
    ).set_pos((site['width'] * 0.05, site['height'] * 0.95))

    # Statistics
    stats_text = f"""
    Total Area: {site['total_area_m2'] / 10000:.1f} ha
    Buildings: {len(layout['buildings'])}
    Estimated Green Ratio: {layout.get('green_ratio_percent', 'N/A')}%
```

```
    Compliance Score: {layout.get('compliance_score', 'N/A')}%
    """

    self.msp.add_text(
        stats_text,
        dxfattribs={
            'height': 2,
            'layer': 'INFO',
            'color': 256
        }
    ).set_pos((site['width'] * 0.05, site['height'] * 0.85))

def _generate_road_network(self, layout: Dict):
    """Generate simplified road network connecting all buildings"""
    buildings = layout['buildings']

    if len(buildings) < 2:
        return

    # Find boundary buildings to create perimeter road
    sorted_x = sorted(buildings, key=lambda b: b['x'])
    sorted_y = sorted(buildings, key=lambda b: b['y'])

    # Create rectangular road network
    min_x = sorted_x[0]['x'] - 20
    max_x = sorted_x[-1]['x'] + sorted_x[-1]['width'] + 20
    min_y = sorted_y[0]['y'] - 20
    max_y = sorted_y[-1]['y'] + sorted_y[-1]['height'] + 20

    # Draw roads
    self.msp.add_lwpolyline(
        [(min_x, min_y), (max_x, min_y)],
        dxfattribs={'layer': 'ROADS', 'color': 8}
    )
    self.msp.add_lwpolyline(
        [(max_x, min_y), (max_x, max_y)],
        dxfattribs={'layer': 'ROADS', 'color': 8}
    )
```

```
        self.msp.add_lwpolyline(
            [(max_x, max_y), (min_x, max_y)],
            dxfattribs={'layer': 'ROADS', 'color': 8}
        )
        self.msp.add_lwpolyline(
            [(min_x, max_y), (min_x, min_y)],
            dxfattribs={'layer': 'ROADS', 'color': 8}
        )
```

IV. API & ORCHESTRATION LAYER

4.1 FastAPI Backend Structure

File: backend/api/main.py

from fastapi import FastAPI, WebSocket, BackgroundTasks, HTTPException
from fastapi.middleware.cors import CORSMiddleware
from pydantic import BaseModel
from typing import Optional, List, Dict
import asyncio
import json
from datetime import datetime

from ai.llm_orchestrator import IndustrialParkLLMOrchestrator
from optimization.csp_solver import IndustrialParkCSP
from optimization.ga_optimizer import IndustrialParkGA
from design.compliance_checker import ComplianceChecker
from cad.dxf_generator import DXFGenerator
from database.models import DesignProject, DesignVariant

app = FastAPI(title="Industrial Park AI Designer API", version="1.0.0")

CORS configuration

app.add_middleware(
CORSMiddleware,
allow_origins=["http://localhost:3000", "https://yourdomain.com"],
allow_credentials=True,
allow_methods=["*"],
allow_headers=[""],*
)

========== DATA MODELS ==========

```
class ChatMessage(BaseModel):
project_id: str
message: str
timestamp: Optional[datetime] = None

class DesignGenerationRequest(BaseModel):
project_id: str
parameters: Dict

class ExportRequest(BaseModel):
project_id: str
variant_id: str
format: str # "dxf", "pdf", "json"
```

========== ENDPOINTS ==========

```
@app.post("/api/projects/new")
async def create_new_project(name: str, site_area_ha: float):
"""Create new design project"""
# Implementation
pass

@app.websocket("/api/chat/{project_id}")
async def websocket_chat(websocket: WebSocket, project_id: str):
"""
WebSocket endpoint for real-time chat.
User types requirements → AI responds → Parameters extracted → Ready for design
"""
await websocket.accept()
```

```
llm = IndustrialParkLLMOrchestrator(api_key=os.getenv("ANTHROPIC_API_KE

try:
    while True:
        # Receive message from client
        data = await websocket.receive_text()
        message = json.loads(data)

        # Get LLM response
        ai_response, params = llm.chat(message['text'])
```

```
        # Send response
        await websocket.send_json({
            "response": ai_response,
            "extracted_params": params,
            "ready_for_design": llm.is_ready_for_optimization()
        })

        # If params complete, trigger optimization
        if llm.is_ready_for_optimization():
            background_tasks = BackgroundTasks()
            background_tasks.add_task(
                generate_design_variants,
                project_id,
                params
            )

    except Exception as e:
        await websocket.send_json({"error": str(e)})
    finally:
        await websocket.close()
```

```
@app.post("/api/designs/generate")
async def generate_designs(request: DesignGenerationRequest,
background_tasks: BackgroundTasks):
"""
Trigger design variant generation.
Runs CSP + GA in background, returns job ID for tracking.
"""
job_id = str(uuid4())
```

```
background_tasks.add_task(
    _design_generation_worker,
    job_id,
    request.project_id,
    request.parameters
)

return {"job_id": job_id, "status": "queued"}
```

```
async def _design_generation_worker(job_id: str, project_id: str, params: Dict):
"""Background worker for design generation"""
try:
# 1. Parse parameters
site_params = {
"total_area_m2": params['totalArea_ha'] * 10000,
"width": params.get('width', 1000), # Default dimensions
"height": params.get('height', 1000),
"buildings": _generate_building_list(params),
"boundary": _generate_boundary(params)
}
```

```
    regulations = TCVN_7144_REGULATIONS  # Loaded from config

    # 2. Run CSP to get feasible solutions
    csp_solver = IndustrialParkCSP(site_params, regulations)
    csp_solver.add_building_variables()
    csp_solver.add_no_overlap_constraint()
    csp_solver.add_fire_safety_constraint()
    csp_solver.add_green_area_constraint()

    feasible_layouts = csp_solver.solve(timeout=30)

    # 3. Run GA for optimization
    ga_optimizer = IndustrialParkGA(site_params, regulations, feasible_layouts)
    optimized_variants = ga_optimizer.optimize()

    # 4. Check compliance for each variant
    compliance_checker = ComplianceChecker(regulations)

    results = []
    for layout, fitness_scores in optimized_variants[:5]:  # Top 5
        compliance_report = compliance_checker.check_layout(layout)

        variant = {
            "id": str(uuid4()),
            "name": f"Variant {len(results)+1}",
            "layout": layout,
            "fitness_scores": {
                "road_efficiency": fitness_scores[0],
```

```
                "worker_flow": fitness_scores[1],
                "green_ratio": fitness_scores[2]
            },
            "compliance": compliance_report,
            "generated_at": datetime.now().isoformat()
        }
        results.append(variant)

    # 5. Save to database
    project = DesignProject.get(project_id)
    for variant_data in results:
        variant = DesignVariant(
            project_id=project_id,
            **variant_data
        )
        variant.save()

    # 6. Notify client via WebSocket or callback
    await notify_design_complete(project_id, results)

except Exception as e:
    await notify_design_error(project_id, str(e))
```

```
@app.get("/api/designs/{project_id}/variants")
async def get_design_variants(project_id: str):
"""Retrieve generated variants for a project"""
variants = DesignVariant.query.filter_by(project_id=project_id).all()
return {"variants": [v.to_dict() for v in variants]}

@app.post("/api/export")
async def export_design(request: ExportRequest):
"""Export design in specified format"""
variant = DesignVariant.get(request.variant_id)
```

```
if request.format == "dxf":
    dxf_gen = DXFGenerator()
    filename = f"design_{request.variant_id}.dxf"
    dxf_gen.generate(variant.layout, filename)

    return {
```

```
        "format": "dxf",
        "filename": filename,
        "download_url": f"/api/files/{filename}"
    }

elif request.format == "pdf":
    # Generate PDF report
    pdf_filename = f"design_{request.variant_id}_report.pdf"
    # Implementation
    return {"format": "pdf", "filename": pdf_filename}

elif request.format == "json":
    return variant.to_dict()
```

```
@app.get("/api/files/{filename}")
async def download_file(filename: str):
"""Download generated file"""
# Implementation with file streaming
pass
```

========== UTILITIES ==========

```
def _generate_building_list(params: Dict) -> List[Dict]:
"""Generate building list from parameters"""
buildings = []
building_id = 0
```

```
for industry in params.get('industryFocus', []):
    industry_type = industry['type']
    count = industry.get('count', 5)

    # Define building sizes based on type
    size_ranges = {
        'light_manufacturing': (2000, 10000),
        'medium_manufacturing': (5000, 30000),
        'heavy_manufacturing': (10000, 50000),
        'warehouse': (2000, 20000),
        'logistics': (5000, 100000)
```

```
    }

    size_range = size_ranges.get(industry_type, (5000, 15000))

    for i in range(count):
        area = random.uniform(size_range[0], size_range[1])
        width = math.sqrt(area * 0.8)  # Assume ~0.8 aspect ratio
        height = area / width

        buildings.append({
            'id': f"building_{building_id}",
            'label': f"{industry_type.replace('_', ' ').title()} {i+1}",
            'type': industry_type,
            'width': width,
            'height': height,
            'area': area
        })
        building_id += 1

  return buildings
```

```
def _generate_boundary(params: Dict) -> List[Tuple[float, float]]:
"""Generate site boundary coordinates"""
total_area_m2 = params['totalArea_ha'] * 10000

  # Assume rectangular site
  width = math.sqrt(total_area_m2 * 0.8)
  height = total_area_m2 / width

  return [
      (0, 0),
      (width, 0),
      (width, height),
      (0, height),
      (0, 0)
  ]
```

V. Technology Stack Summary & Implementation Roadmap

Complete Tech Stack

```
┌─ PROGRAMMING LANGUAGES
│  ├─ Python 3.11+ (AI, optimization, design logic)
│  ├─ TypeScript + React 18 (frontend)
│  └─ SQL (PostgreSQL)
│
├─ AI/LLM
│  ├─ Claude 3.5 Sonnet (Anthropic API)
│  ├─ Claude 3.5 Vision (site photo analysis)
│  └─ Vercel AI SDK (chat integration)
│
├─ OPTIMIZATION
│  ├─ DEAP (Genetic Algorithm)
│  ├─ python-constraint (CSP)
│  ├─ PuLP (Integer Linear Programming - optional)
│  └─ NetworkX (graph algorithms)
│
├─ CAD/DESIGN
│  ├─ ezdxf (DXF generation)
│  ├─ Shapely (geometric operations)
│  └─ PostGIS (geographic data)
│
├─ BACKEND
│  ├─ FastAPI (API framework)
│  ├─ Pydantic (data validation)
│  ├─ SQLAlchemy (ORM)
│  ├─ Celery + Redis (async tasks)
│  └─ Uvicorn (ASGI server)
│
├─ FRONTEND
│  ├─ React 18 + TypeScript
│  ├─ Shadcn/ui + Tailwind CSS
│  ├─ Deck.gl (2D floorplans)
│  ├─ Three.js (3D visualization)
│  └─ Vercel AI SDK (chat UI)
│
├─ DATABASE
│  ├─ PostgreSQL 15+
│  ├─ PostGIS (spatial data)
│  └─ Redis (caching, job queue)
│
├─ DEPLOYMENT
│  ├─ Docker + Docker Compose
│  ├─ Kubernetes (optional, for scale)
│  ├─ GitHub Actions (CI/CD)
│  └─ AWS / GCP / Azure (cloud)
```

└─ MONITORING
├─ DataDog (logs, metrics)
├─ Sentry (error tracking)
└─ Prometheus (custom metrics)

12-Week Implementation Roadmap

Phase 1: MVP Foundation (Weeks 1-4)

- Week 1-2: Setup & Infrastructure
  - [ ] Set up FastAPI project structure
  - [ ] PostgreSQL + PostGIS local setup
  - [ ] Anthropic API integration
  - [ ] Basic chat endpoint
- Week 3: LLM Integration
  - [ ] Build LLM orchestrator with system prompt
  - [ ] Parameter extraction pipeline
  - [ ] Multi-turn conversation logic
  - [ ] Unit tests
- Week 4: Basic Optimization
  - [ ] Implement CSP solver (no overlap, boundary checks)
  - [ ] Generate initial feasible layouts
  - [ ] Simple DXF export
  - [ ] Integration test

Phase 2: Full Optimization (Weeks 5-8)

- Week 5-6: Genetic Algorithm
  - [ ] Implement DEAP GA framework
  - [ ] Multi-objective fitness function
  - [ ] Convergence testing
  - [ ] Parameter tuning
- Week 7: Compliance Engine
  - [ ] Build comprehensive compliance checker
  - [ ] Auto-detection of violations
  - [ ] Recommendation system
  - [ ] Detailed reporting
- Week 8: Quality Assurance
  - [ ] Integration testing (LLM → CSP → GA → DXF)
  - [ ] Performance profiling & optimization
  - [ ] Edge case handling
  - [ ] Documentation

Phase 3: Production Features (Weeks 9-12)

- Week 9: Advanced Features
  - [ ] Variant comparison API
  - [ ] Version control for designs
  - [ ] Batch export (multiple formats)
  - [ ] Design history tracking
- Week 10: Frontend Integration
  - [ ] WebSocket chat UI

    - [ ] Real-time design preview
    - [ ] Variant gallery
    - [ ] Compliance report visualization
- Week 11: Deployment & DevOps
    - [ ] Docker containerization
    - [ ] Kubernetes manifests (optional)
    - [ ] CI/CD pipeline (GitHub Actions)
    - [ ] Error handling & monitoring
- Week 12: Launch & Optimization
    - [ ] Load testing
    - [ ] API rate limiting
    - [ ] User feedback collection
    - [ ] Production deployment

VI. Database Schema

```
-- Users & Projects
CREATE TABLE users (
id UUID PRIMARY KEY,
email VARCHAR(255) UNIQUE,
api_key VARCHAR(255),
created_at TIMESTAMP,
updated_at TIMESTAMP
);

CREATE TABLE design_projects (
id UUID PRIMARY KEY,
user_id UUID REFERENCES users(id),
name VARCHAR(255),
site_area_ha FLOAT,
location GEOGRAPHY,
created_at TIMESTAMP,
updated_at TIMESTAMP
);

-- Design Variants
CREATE TABLE design_variants (
id UUID PRIMARY KEY,
project_id UUID REFERENCES design_projects(id),
name VARCHAR(255),
layout_json JSONB, -- Complete layout data
fitness_scores JSONB,
compliance_report JSONB,
created_at TIMESTAMP
);

-- Design History
CREATE TABLE design_history (
id UUID PRIMARY KEY,
project_id UUID REFERENCES design_projects(id),
event_type VARCHAR(50), -- "generated", "exported", "refined"
```

```
event_data JSONB,
created_at TIMESTAMP
);

-- Exports
CREATE TABLE exports (
id UUID PRIMARY KEY,
variant_id UUID REFERENCES design_variants(id),
format VARCHAR(50), -- "dxf", "pdf", "json"
file_path VARCHAR(500),
created_at TIMESTAMP
);

-- Create spatial index
CREATE INDEX idx_design_projects_location
ON design_projects USING GIST(location);
```

Kết Luận

Hệ thống backend được thiết kế để:

✅ **Tích hợp LLM & Optimization**: Claude 3.5 Sonnet cho reasoning, GA+CSP cho layout tối ưu
✅ **Tuân thủ Quy định**: TCVN 7144 embedded trong constraint, tự động verify
✅ **Mở rộng được**: Modular architecture, dễ thêm feature sau
✅ **Hiệu suất cao**: Async processing, caching, parallel optimization
✅ **Production-ready**: Monitoring, error handling, logging, CI/CD

Ngôn ngữ & Framework chính:

- **Python 3.11+** (FastAPI, optimization, design logic)
- **Claude 3.5 Sonnet** (LLM, constraint reasoning)
- **DEAP** (Genetic Algorithm)
- **python-constraint** (CSP)
- **PostgreSQL** + **PostGIS** (data storage)

Bắt đầu từ **Phase 1 MVP (Weeks 1-4)** để có working prototype, sau đó mở rộng từ từ.