ISR Agent Tools Reference

Complete Tool Documentation

```
TOOLS OVERVIEW

The ISR Agent has 12 tools organized into two categories:

PLANNING TOOLS (9 tools)
  Used during mission planning to gather information, calculate routes,
  and validate solutions. These are called automatically by the agent.

OPTIMIZATION TOOLS (3 tools)
  Used to improve existing routes. Only called when user explicitly
  requests optimization (e.g., "optimize", "swap closer", etc.)


TOOL EXECUTION PATTERN

  1. Agent decides which tool(s) to call based on user query
  2. Tool receives parameters (drone_id, waypoints, routes dict, etc.)
  3. Tool accesses shared context (_current_env, _drone_configs, _distance_matrix)
  4. Tool returns formatted string with results
  5. Agent parses string and decides next action


SHARED CONTEXT (Global Variables)

  _current_env       Environment with airports, targets, SAMs
  _drone_configs     Per-drone configuration (fuel, airports, type access)
  _distance_matrix   Pre-computed SAM-aware distances between all waypoints
  _sam_paths         Pre-computed SAM-avoiding paths for visualization
```

Planning Tools (1/2)

get_mission_overview

Get complete mission overview. CALL THIS FIRST.

INPUTS: None

OUTPUTS:
- All airports with positions
- All targets grouped by type with priorities
- All enabled drones with constraints
- Current sequences if any
- Total possible points

get_drone_info

Get detailed constraints for a specific drone.

INPUTS: `drone_id: str ("1", "D1", etc.)`

OUTPUTS:
- Fuel budget
- Start/end airports
- Accessible target types
- List of accessible targets with priorities
- Current assigned sequence

get_distance

Look up distance between two waypoints.

INPUTS:
`from_point: str (e.g., "A1", "T3")`
`to_point: str`

OUTPUTS: Distance in fuel units (SAM-aware)

calculate_route_fuel

Calculate total fuel consumption for a route.

INPUTS:
`drone_id: str`
`waypoints: List[str] (e.g., ["A1", "T3", "T7", "A1"])`

OUTPUTS:
- Each leg with distance
- Total fuel used
- Fuel budget
- Remaining fuel
- VALID/INVALID status

Planning Tools (2/2)

`validate_drone_route`

Validate a route against ALL drone constraints.

INPUTS: `drone_id: str, waypoints: List[str]`

OUTPUTS: Fuel/airport/type/duplicate checks, PASSED/FAILED status

`find_accessible_targets`

Find targets drone can reach, sorted by efficiency.

INPUTS: `drone_id: str, from_point: str`

OUTPUTS: Table: Target | Pts | Type | Dist | Round | Eff

`suggest_drone_route`

Generate a greedy baseline route for a drone.

INPUTS: `drone_id: str`

OUTPUTS: Route sequence, targets, points, fuel, ROUTE_Dx format

`check_target_conflicts`

Check if targets are assigned to multiple drones.

INPUTS: `routes: Dict[str, str]  e.g., {"1": "A1,T3,A1"}`

OUTPUTS: Targets per drone, conflict list, OK/WARNING status

`get_mission_summary`

Get summary stats for a multi-drone plan.

INPUTS: `routes: Dict[str, str]  e.g., {"1": "A1,T3,A1"}`

OUTPUTS: Per drone stats, totals, coverage %, unvisited list

Optimization Tools

(Only called when user explicitly requests optimization)

optimize_assign_unvisited

UI Button: "Insert Missed"

Insert unvisited targets into routes with spare fuel capacity.

INPUTS: `routes: Dict[str, str]`
`e.g., {"1": "A1,T3,A1", "2": "A2,T5,A2"}`

OUTPUTS:
- Updated routes with new targets inserted
- Points and fuel per drone
- ROUTE_Dx format for copy

ALGORITHM:

```
1. Find all unvisited targets
2. For each unvisited target:
   - Find drone with type access and spare fuel
   - Find optimal insertion position (min added distance)
   - Insert if within fuel budget
3. Return updated routes
```

optimize_reassign_targets

UI Button: "Swap Closer"

Move targets to drones whose trajectories pass closer.

INPUTS: `routes: Dict[str, str]`
`e.g., {"1": "A1,T3,A1", "2": "A2,T5,A2"}`

OUTPUTS:
- Number of swaps made
- Swap details (target, from, to, savings)
- Updated routes
- ROUTE_Dx format for copy

ALGORITHM:

```
1. For each target in each route:
   - Calculate removal cost (fuel saved)
   - For each other drone with type access:
     - Calculate insertion cost at each segment
     - Check fuel budget constraint
   - If net savings > 0, perform swap
2. Repeat until no beneficial swaps
```

optimize_remove_crossings

UI Button: "Cross Remove"

Fix self-crossing trajectories using 2-opt algorithm.

INPUTS: `routes: Dict[str, str]`
`e.g., {"1": "A1,T3,A1", "2": "A2,T5,A2"}`

OUTPUTS:
- Number of crossings fixed
- Fix details (drone, segment positions)
- Updated routes
- ROUTE_Dx format for copy

ALGORITHM:

```
1. For each drone route:
   - Check all pairs of non-adjacent segments
   - If segments cross (line intersection):
     - Reverse the middle portion
     - Check if distance improved
   - Repeat until no crossings remain
2. Return optimized routes
```

Tool Usage Rules & Examples

TOOL USAGE RULES

```
1. ALWAYS call get_mission_overview first
2. Use planning tools freely during planning
3. Optimization tools ONLY when user asks:
   - "optimize" -> all three
   - "insert missed" -> optimize_assign_unvisited
   - "swap closer" -> optimize_reassign_targets
   - "remove crossings" -> optimize_remove_crossings
4. After ANY optimization, OUTPUT routes and STOP
5. optimize_reassign_targets: max 8 calls
6. Always validate routes before presenting
```

TYPICAL WORKFLOW

```
User: "Plan a mission for all drones"

Agent calls:
  1. get_mission_overview()
  2. suggest_drone_route("1")
  3. suggest_drone_route("2")
  4. check_target_conflicts(routes)
  5. validate_drone_route("1", route1)
  6. validate_drone_route("2", route2)
  7. get_mission_summary(routes)

Agent outputs:
  ROUTE_D1: A1,T3,T7,A1
  ROUTE_D2: A2,T5,T8,A2
```

INPUT/OUTPUT FORMAT EXAMPLES

Route Dictionary Input:

```
routes = {
    "1": "A1,T3,T7,A1",
    "2": "A2,T5,T8,A2",
    "3": "A1,T1,T2,A1"
}
```

Waypoint List Input:

```
waypoints = ["A1", "T3", "T7", "A1"]
```

Tool String Output:

```
=== VALIDATION PASSED for D1 ===

Route: A1 -> T3 -> T7 -> A1
Targets visited: 2
Total points: 15
Total fuel: 87.5 / 200

ROUTE_D1: A1,T3,T7,A1
```