Agent Orchestrator

Visual Workflow Builder for Agent Zero

Design, connect, and execute multi-agent pipelines using a drag-and-drop canvas. Coordinate agents in sequence, in parallel, and with conditional branching.

Contents

Troubleshooting

Architecture Reference

1. Overview

The Agent Orchestrator is a visual workflow builder for Agent Zero that lets you design, connect, and execute multi-agent pipelines using a drag-and-drop canvas. Instead of writing code to coordinate agents, you build **flows** by placing nodes on a canvas and drawing connections between them.

Flows are executed as **directed acyclic graphs (DAGs)**. The engine walks the graph in topological order, passing each node's output as input to its successors. Agents run within Agent Zero's existing framework, using the same profiles, tools, and capabilities available in chat.

> **Key Concepts**
>
> **Nodes** are the building blocks — inputs, agents, conditions, parallel groups, and outputs.
> **Edges** are the connections between nodes that define data flow.
> **Flows** are complete graphs that can be saved, loaded, and executed.

2. Getting Started

Opening the Orchestrator

Open the Agent Orchestrator from the Agent Zero sidebar or toolbar. The modal presents two views:

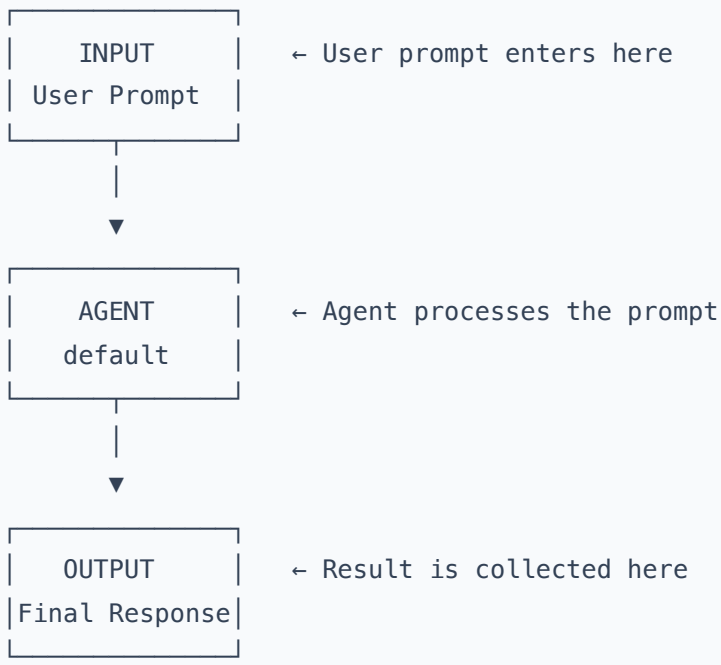
1. **Flow List** — Browse, open, and delete saved flows. Click **New Flow** to create one.
2. **Canvas View** — The visual editor where you build and run flows.

Creating Your First Flow

Every new flow starts with two nodes already placed: an **Input** (User Prompt) and an **Output** (Final Response).

To build a working flow:

1. Drag an **Agent** node from the palette (top-left corner) onto the canvas
2. Connect **Input → Agent** by dragging from the Input's bottom handle to the Agent's top handle
3. Connect **Agent → Output** by dragging from the Agent's bottom handle to the Output's top handle
4. Type a prompt in the toolbar's input field and click the green play button



> **Tip**
>
> You can rename any node by clicking it and editing the Label field in the Inspector panel.

3. Node Types

The orchestrator provides five node types. Each has a distinct color, purpose, and set of configuration options.

INPUT Input Node

The entry point of every flow. Provides the initial text that downstream nodes receive.

Field	Description
Label	Display name on the canvas
Prompt Mode	`Runtime` — uses the prompt typed in the toolbar. `Fixed` — uses a hardcoded prompt.
Fixed Prompt	The text to inject when Prompt Mode is "Fixed"

Connections: One source handle (bottom). Cannot receive incoming connections.

> **When to use Fixed mode**
>
> When the flow always performs the same task regardless of user input — e.g., a daily report generator or a code review pipeline with a predefined checklist.

AGENT Agent Node

The workhorse of every flow. Creates a temporary Agent Zero instance, sends it the combined output from all predecessor nodes, and returns the agent's response.

Field	Description
Label	Display name on the canvas
Agent Profile	Which agent profile to use (matches profiles in Agent Zero settings)
Prompt Override	Optional system prompt prepended to the input. Defines the agent's role.

Connections: One target handle (top), one source handle (bottom). Can receive from and send to multiple nodes.

Multiple inputs: If an Agent has multiple incoming connections, all predecessor outputs are concatenated with double newlines.

> **Prompt Override behavior**
>
> When set, the override is prepended: `{Override}\n\n{Merged input}`. Use it for role instructions like *"You are a senior code reviewer. Analyze the following code."*

PARALLEL Parallel Group Node

Executes multiple Agent nodes simultaneously. Connect the Parallel Group to several Agent nodes, and they all run at the same time. Their outputs are merged according to the selected strategy.

Merge Strategy	Behavior
Concatenate (default)	Joins all branch outputs with double newlines
Summarize	Creates an additional agent to synthesize all outputs into one response
First	Takes only the first branch's output

Connections: One target handle (top), one source handle (bottom).

> **Important**
>
> Only **Agent nodes** directly connected as successors run in parallel. Other node types (Output, Condition) connected as successors execute sequentially after the merge.

CONDITION Condition Node

Routes the flow based on a Python expression. Evaluates the expression against predecessor output, then sends data down either the True or False handle.

Field	Description
Expression	Python expression evaluating to `True` or `False`
True/False Labels	Labels displayed on the two output paths

Available variables: `input` (predecessor text), `len`, `int`, `float`, `str`, `bool`

Example expressions:

```
len(input) > 500                    # Route by length
"error" in input.lower()            # Route by content
bool(input.strip())                 # True if non-empty
```

Connections: One target handle (top). Two source handles: `true` (left) and `false` (right).

If no expression is set or the expression errors, defaults to `True`.

OUTPUT Output Node

The terminal node. Collects combined input from all predecessors and stores it as the flow's final result.

Connections: One target handle (top). Cannot have outgoing connections. A flow should have exactly one Output node.

4. Building Flows

Canvas Interactions

Action	How
Add a node	Drag from the Nodes palette (top-left) onto the canvas
Connect nodes	Drag from a source handle (bottom dot) to a target handle (top dot)
Select a node	Click on it — opens the Inspector panel
Delete a node or edge	Select it and press `Delete`
Pan the canvas	Click and drag on empty space
Zoom	Scroll wheel, or use bottom-right controls
Deselect	Click on empty canvas space

Inspector Panel

Click any node to open the Inspector panel on the right side. It shows:

- **During editing:** Configuration fields specific to the selected node type
- **During/after execution:** The node's status (pending, running, completed, error) and output text

Saving, Loading & Running

- Click **Save** in the footer to persist the flow. Flows are saved as JSON in `usr/flows/`.
- The * asterisk next to the flow name indicates unsaved changes.
- To run: enter a prompt in the toolbar, click the green play button (or press Enter).
- To stop: click the red stop button. Remaining nodes are marked "skipped".

Working with Results

After a flow completes, a result panel appears below the toolbar with these actions:

Button	Action
Copy	Copies full output to clipboard
Send to Chat	Posts the output as a message in your current Agent Zero chat
Toggle	Expands or collapses the output panel
Dismiss	Clears results and resets execution state

> **Tip**
>
> Use **Send to Chat** to push orchestrated results into your main conversation for follow-up questions or further processing.

5. Flow Patterns

Sequential Pipeline

Each agent processes the output of the previous one.

```
Input → Agent A (draft) → Agent B (review) → Agent C (polish) → Output
```

Use case: Multi-stage text processing — draft, review, polish.

Parallel Fan-Out

Multiple agents process the same input simultaneously.

```
                      ┌→ Agent A (researcher)
Input → Parallel →├→ Agent B (critic)     → Output
                      └→ Agent C (editor)
```

Use case: Getting multiple perspectives, or performing independent analyses concurrently.

Conditional Branching

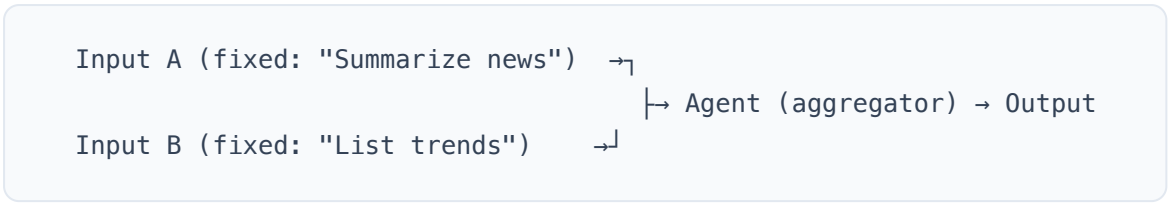
Route data to different agents based on content.

```
Input → Agent (classifier) → Condition
                                  ├→ [True]  → Agent (handler A) → Output
                                  └→ [False] → Agent (handler B) → Output
```

Use case: Classify input first, then route to a specialized handler.

Multi-Input Aggregation

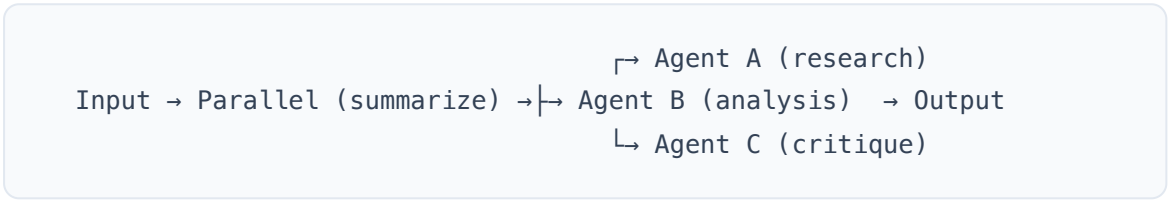
Multiple inputs feed into a single agent.



Use case: Combining data from multiple fixed sources into one synthesis.

Parallel with Summarization

Fan out to specialists, then auto-summarize their combined output.



With the **Summarize** merge strategy, the Parallel Group automatically creates an additional agent to synthesize all branch outputs into a coherent response.

6. Execution Model

How the Engine Works

1. **Topological sort** — The engine uses Kahn's algorithm to determine execution order. All nodes are sorted so every node runs after its predecessors.
2. **Sequential walk** — Nodes execute one at a time in topological order, except for Parallel Group branches which run concurrently via `asyncio.gather`.
3. **Input gathering** — Before each node runs, the engine collects outputs from all predecessors. For edges from Condition nodes, only the active branch's output is included.
4. **Agent execution** — Each Agent node creates a temporary context, sends the prompt, waits for the response, then destroys the context. The agent has full access to Agent Zero's tools during execution.
5. **Output collection** — The Output node stores its combined input as the flow's final result.

Node Execution States

State	Visual	Meaning
Pending	Dashed border, dimmed	Waiting to execute
Running	Blue glow + spinner	Currently executing
Completed	Green border	Finished successfully
Error	Red border + glow	Execution failed
Skipped	Dotted border, very dim	Cancelled before execution

Error Handling

- If any node throws an error, execution **stops immediately**. The failed node is marked "error" and remaining nodes are not executed.
- Condition expression errors default to `True` and log a warning (they do not stop the flow).

7. Best Practices

Do

- Start simple: Input → Agent → Output. Add complexity only when needed.
- Name your nodes with descriptive labels for readability.
- Use Prompt Override for agent role assignment instead of putting instructions in the Input node.
- Save frequently — the asterisk reminds you of unsaved changes.
- Test incrementally — build and test one section at a time.
- Use "Send to Chat" to push results into your main conversation.

Don't

- Create cycles — the engine will abort with an error.
- Connect Output nodes as sources (they are terminal).
- Connect to Input nodes as targets (they are entry points).
- Put non-Agent nodes after a Parallel Group expecting parallel execution.
- Leave Agent nodes unconnected (they receive no input and return nothing).
- Use complex Python in Condition expressions — no imports, no file I/O.
- Expect flow agents to appear in the chat sidebar — use "Send to Chat" instead.

8. Troubleshooting

Problem	Solution
Flow loads but canvas is empty	The flow may have been saved before serialization fixes. Delete and recreate it.
Agent node shows "error" immediately	Check that the Agent Profile exists in Agent Zero settings. The "default" profile is always available.
Condition always takes the True path	Verify the expression is set in the Inspector. Ensure it references `input`. Check backend logs for expression errors.
"Send to Chat" does nothing	Ensure you have an active chat context open in Agent Zero.
Flow won't save	The flow must have a name. The `usr/flows/` directory must be writable.
Nodes don't show execution status	Status updates arrive via 1-second polling. Check the browser console for errors.

9. Architecture Reference

File Structure

```
webui/components/modals/orchestrator/
  orchestrator.html              # Modal template + CSS
  orchestrator-store.js          # Alpine store (state + API calls)
  orchestrator-bridge.js         # Alpine ↔ React event bridge
  react/
    OrchestrationCanvas.js       # React Flow canvas + node components

python/helpers/
  flow_engine.py                 # FlowRun, FlowEngine, graph walker

python/api/
  flow_list.py                   # List saved flows
  flow_load.py                   # Load a flow from disk
  flow_save.py                   # Save a flow to disk
  flow_delete.py                 # Delete a saved flow
  flow_execute.py                # Start execution (background)
  flow_status.py                 # Poll execution status
  flow_stop.py                   # Cancel a running flow
  flow_result.py                 # Post results to chat

usr/flows/                       # Saved flow JSON files
```

Data Flow

```
User clicks Play
    │
    ▼
orchestrator-store.js → POST /flow_execute { flow, user_prompt }
    │
    ▼
flow_execute.py → FlowEngine.start_background(flow, prompt)
    │
    ▼
FlowEngine._walk_graph(run)     ← Topological order
    ├── InputNode   → returns prompt text
    ├── AgentNode   → creates context, runs monologue, returns response
    ├── ParallelGroupNode → asyncio.gather on Agent branches, merges
    ├── ConditionNode → evaluates expression, routes via handles
    └── OutputNode  → stores final_output
    │
    ▼
orchestrator-store.js polls /flow_status every 1s
    │
    ▼
Canvas updates node states + edge animations
    │
    ▼
Flow completes → result panel appears
```