



API-Chain or Tool-Chain Next-Step Prediction

Chi-En Hong Neeti Tatiya Xiangxiang Li
Advisor: Professor Ali Hirsra, Miao Wang



Abstract

This project studies next-step prediction in tool-augmented dialogue. Given the dialogue history and the API information available at the current turn, the system must decide what should happen next: ask for missing information, retrieve an API, call an API, respond to the user, or stop. We use ToolDial as the main data and evaluation setting because it combines multi-turn dialogue with explicit API calls, retrieval events, action labels, and API dependency information. Our work connects two parts of the problem: next-action prediction on dialogue data and structural modeling of API dependencies.

Research Question & Motivation

Research question. Given the user’s request, dialogue history, available API information, required parameters, and previous tool interactions, can a model predict the correct next system action and identify the relevant API when a tool call is needed?

Why this is difficult. Tool use is not only an API-selection problem. The system must first decide whether enough information is available to make a call. A missing parameter may require `request`, an unclear intent may require `clarify`, and an unavailable tool may require `retriever_call`.

Research gap.

- ▶ Sequential execution can carry an early mistake into later steps.
- ▶ Static plans are difficult to revise when information is missing or changes during dialogue.
- ▶ Some graph methods connect tools through semantic relatedness rather than strict input-output compatibility.
- ▶ Constrained function calling can produce valid syntax but does not decide whether to ask, retrieve, call, respond, or stop.

Method landscape.

Approach	Limitation for our task
Sequential rollout	Flexible, but early tool errors can propagate through later steps.
Static plan decomposition	Gives an execution plan up front, but is harder to revise when re-quired information is missing.
Graph-augmented search	Can restrict tool paths, but the graph must reflect real input-output compatibility.
Constrained function call-ing	Improves output format, but does not by itself decide whether to ask, retrieve, call, retry, or stop.

Our project therefore focuses on the **decision point before tool execution** rather than full response generation.

ToolDial Dataset and Task Setting

ToolDial is the common data and evaluation setting for our project. It provides multi-turn tool-augmented dialogues with API calls, retrieval events, action labels, API information, and dialogue states.

Dataset role. Main dialogue and evaluation setting; ToolDial itself is not treated as a prediction model.

Test split. 9,200 samples from 1,166 dialogues.

Action mix. Roughly one quarter `request`, one quarter `retriever_call`, and 19% `call`; the rest are other dialogue actions.

Prediction target. Next system action; API identity and arguments are evaluated when supported.

Graph layer. API endpoints and API-to-API structural dependencies; dialogue actions are not graph nodes.

Why ToolDial fits. ToolDial connects **dialogue policy** with **tool chaining**. Even when the correct API is known, the next action may still be to ask for a required parameter or retrieve another API before making the call.

Official ToolDial Action Prediction Pipeline

This phase maps Action Prediction to the official ToolDial implementation. The main script is `experiments/action_train_dist_normal.py`, configured through `action_train.yaml`; `prompts.py` and `nlp.py` handle formatting, while `action_error_analysis.py` supports accuracy and per-action analysis.

Dialogue history. Conversation context available up to the current turn.

Prompt formatting. Converts dialogue turns into model input.

Target action. Next gold system action used for training/evaluation.

Loss masking. Ignores prompt-context tokens and trains on the action target.

Generation. Produces the predicted next-action string.

Normalization. Cleans generated and gold actions before comparison.

Evaluation. Reports overall accuracy and per-action performance.

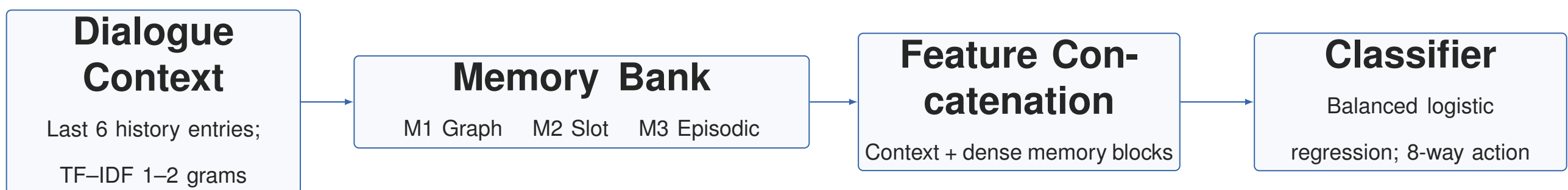
History → Prompt → Model → Next Action → Evaluation

Repository mapping.

File	Role
<code>action_train_dist_normal.py</code>	Main Action Prediction training/evaluation script.
<code>action_train.yaml</code>	Dataset paths, model choice, and setting flags.
<code>prompts.py/nlp.py</code>	Prompt and dialogue formatting helpers.
<code>action_error_analysis.py</code>	Accuracy and per-action analysis.

Graph-Grounded Memory Design

The technical implementation augments the dialogue context with small, structured memory feature blocks. The memory features are **appended** to the context representation rather than blended into the raw dialogue text.



Three memory stores.

M1 – Graph memory (6 features). *Does the graph already know a supplier for what this API is missing?* Static structural evidence: target-known flag, log in/out degree, source-predecessor check, supplier evidence for missing parameters, and an unsupplied-missing indicator.

M2 – Slot memory (5 features). *Has this parameter been filled with a known value before?* Training-only values keyed by normalized parameter name; measures missing-slot count, fraction previously seen, evidence strength, and all-known / none-known indicators.

M3 – Episodic memory (10 features). *What did the system typically do next in a state like this one?* Next-action distributions for dialogue-state signatures: eight smoothed action probabilities plus evidence size and backoff level when a specific signature is sparse.

Compact footprint. The 5,000-feature TF-IDF context is augmented by only 21 dense memory features, producing 5,021 total features. The fitted stores contain 145 normalized slot keys and three episodic backoff tables with 27/24/17 observed signatures.

Implementation. `tooldial_graph.py` reconstructs the graph; `tooldial_memory.py` builds the memory blocks and evaluates 8-way next-action prediction.

Memory Evaluation Output: Diagnostic Ablation

Protocol. Official ToolDial train/test split with 70,266 training instances and 9,200 test instances. Context uses a 5,000-feature TF-IDF representation; memory blocks are evaluated with class-balanced logistic regression over seeds 7, 17, and 27.

	Accuracy	Macro-F1	
	94.54% → 99.26%	96.84% → 99.57%	
	Context only → all memory; +4.72 pp	Context only → all memory; +2.73 pp	
Condition	Accuracy	Macro-F1	Δ Acc.
Context only	94.54%	96.84%	–
+ Graph memory	98.35%	98.95%	+3.81 pp
+ Slot memory	96.00%	97.13%	+1.46 pp
+ Episodic memory	98.26%	98.68%	+3.72 pp
+ All memory	99.26%	99.57%	+4.72 pp
Memory only, no context	93.15%	85.21%	-1.39 pp

Largest per-action gains. With all memory added, `request` F1 rises from 89.68% to 98.52% (+8.84 pp), `retriever_call` from 92.37% to 100.00% (+7.63 pp), and `response` from 96.74% to 100.00% (+3.26 pp).

Interpretation. Graph and episodic memory each provide a large gain over the context-only baseline, while the combined memory bank performs best. The largest class-level gains are on `request` and `retriever_call`, the actions most directly tied to missing information and tool availability. Memory without dialogue context underperforms the baseline, so the memory signals are complementary rather than a replacement for the current dialogue context. This remains a diagnostic feature ablation, separate from the final teammate model results.

Raw API Dependency Graph

Graph semantics. Each node is one API endpoint. A directed edge

$$API_A \rightarrow API_B$$

means that at least one output field from API_A can fill a required input parameter of API_B .

LCS surface similarity.

For two normalized short descriptions s_1, s_2 :

$$\text{LCSsim}(s_1, s_2) = \frac{\text{LCS}(s_1, s_2)}{(|s_1| + |s_2|)/2}$$

A field-level dependency is kept only when

$$\text{LCSsim} > 0.83 \wedge \text{KeywordCos} > 0.50 \wedge \text{DescriptionCos} > 0.77.$$

LCS filter. Threshold > 0.83 on `object_ad`; checks surface similarity of short descriptions.

Keyword cosine. Threshold > 0.50 on `description`; compares documentation text with keywords.

Description cosine. Threshold > 0.77 on `object_long`; checks semantic similarity using longer descriptions.

Adjacency matrix. $\text{adjacency}[i, j] = 1.0$ means that API node i has a directed dependency edge to API node j .

Matrix axes. Rows are the 34,714 API output fields (“who can supply data”) and columns are the 4,929 required parameters (“who needs data”). Self-edges and page/limit-style pagination matches are removed before field matches are collapsed to API-level edges.

Cosine similarity.

The two semantic filters use:

$$\cos(\mathbf{u}, \mathbf{v}) = \frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\| \|\mathbf{v}\|}$$

Graph Construction and Key Findings

The full graph was reconstructed from ToolDial API metadata using the reported filtering rules.

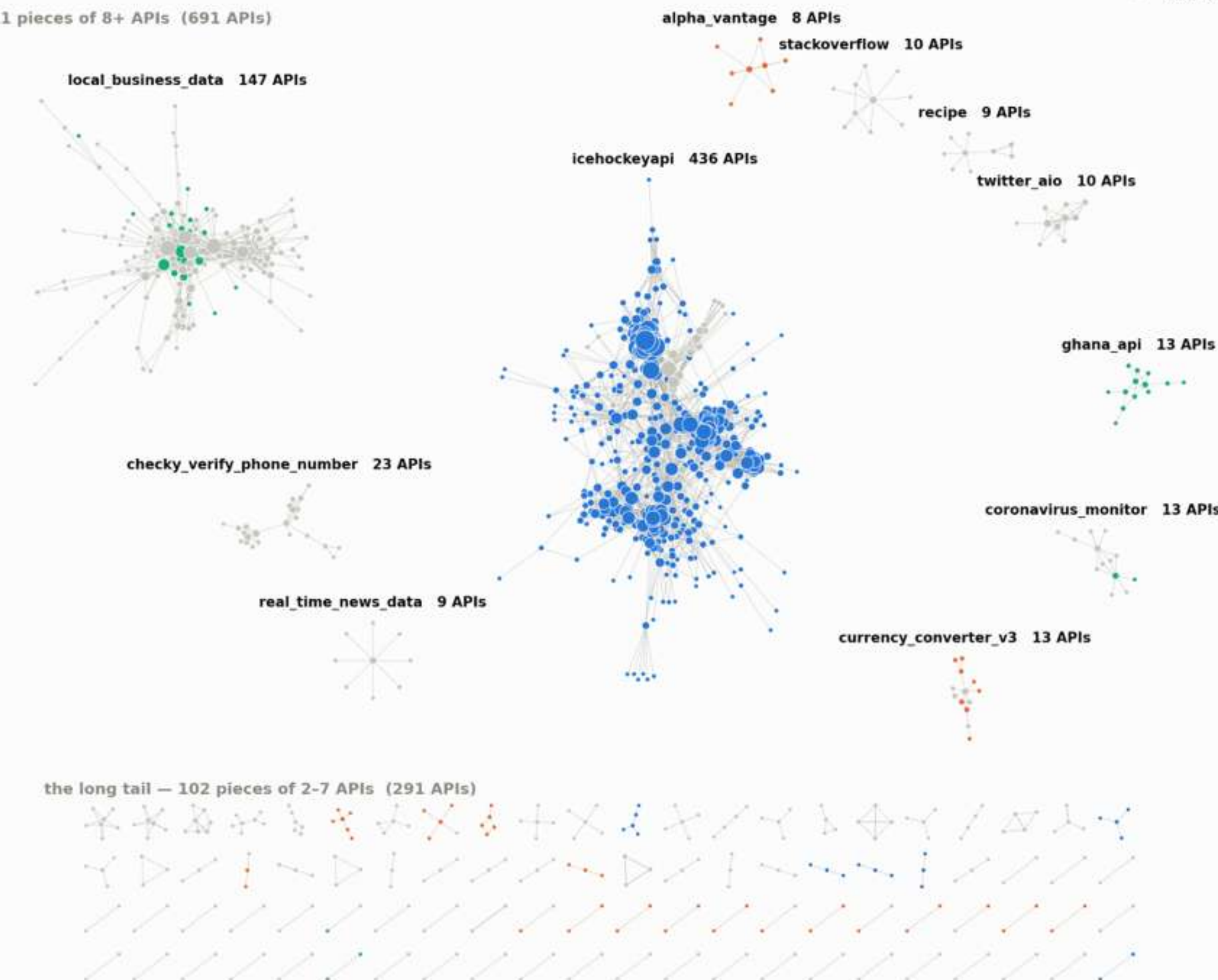


Figure 1. Full ToolDial raw API dependency graph. The final graph contains 982 connected API nodes and 4,857 directed API-level edges.

Stage	Count	Interpretation
API specifications	4,474	Starting API endpoints
Output fields	34,714	Possible data suppliers
Required parameters	4,929	Required API inputs
Candidate pairs	171,105,306	All output-input combinations
After LCS > 0.83	19,325	Surface-compatible pairs
After masks	17,844	Self/page/limit removed
After all filters	7,681	Valid field-level dependencies
Final API graph	4,857 edges	982 connected nodes

Graph structure. Only 982 of 4,474 APIs remain connected under the strict rule (78.05% independent API ratio). These 982 nodes form 113 weakly connected components: 11 components with 8+ APIs contain 691 nodes, while 102 smaller components contain 291 nodes.

Dialogue-edge validation. In the provided evaluation, all 132 released dialogue source→target API pairs were recovered by the reconstructed graph: 132/132, or 100% edge recall.

Example dependency. `Data|youtube_v3_v3|Playlists` produces `items|id`, which can fill the required `id` of `Search|youtube_search_v3|getPlaylist`. The recorded scores are `Keyword` = 0.773, `Description` = 0.933, and `LCS` = 1.000.

Role of the graph. Provides structurally valid API-to-API dependencies and candidate-transition constraints, but does not determine the correct dialogue action, user state, or missing-information decision by itself.

Current Progress and Research Direction

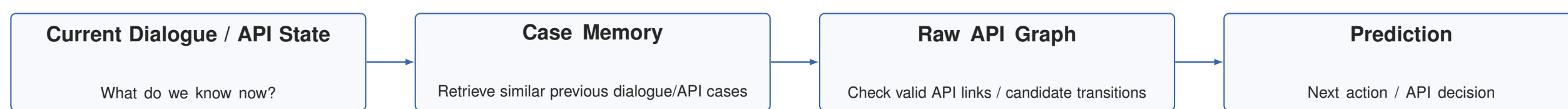
Completed. Reviewed the official Action Prediction pipeline, reconstructed the raw API graph, and implemented graph, slot, and episodic memory. The diagnostic ablation reached 99.26% accuracy / 99.57% macro-F1, while graph validation recovered 132/132 evaluated dialogue API pairs.

Next direction. Use retrieved dialogue/API case evidence together with graph-validity checks in the downstream prediction model, and compare future model results under the same evaluation protocol.

GRASP: Forward-Looking Proposal

The technical ablation shows that structured graph and memory evidence can improve next-action prediction. The final direction, **GRASP** (Graph-grounded, Retrieval-Augmented Skill Procedures), retrieves relevant dialogue/API cases, validates candidate transitions with the raw graph, and passes the combined evidence to a decision layer with an explicit recovery policy.

Simplified pipeline (GRASP Stages 1–3).



Full GRASP design. Five stages in total: schema-verified graph, procedural memory, this decision layer, an explicit Ask / Clarify / Retry / Stop recovery policy, and a continual few-shot adaptation loop — see the accompanying report for the complete specification.

Immediate next steps.

- ▶ Build structured dialogue/API cases and graph constraints around ToolDial decision points.
- ▶ Evaluate the downstream model on the official test split using the metrics above.