COMPILE LOCALLY, QUERY GLOBALLY: LEARNING REUSABLE GRAPH PROGRAMS

Anonymous authors
Paper under double-blind review

ABSTRACT

When familiar graph components are connected in new ways, reusing local structural knowledge can support new predictions and avoid repeated computation. However, standard message-passing representations depend on current connections, while boundary-only summaries omit information needed to query internal nodes. We address these challenges by learning how information propagates within each component and reusing this computation across different assemblies. We introduce the Compositional Graph Compiler (CoGraC), an end-to-end framework that turns local structure into reusable numerical programs. Its queryable programs preserve both boundary interactions and recoverable internal responses, allowing them to be connected and jointly executed for prediction. Prediction gradients through this computation train the compiler to retain local computations that remain useful after composition. Once compiled, unchanged components can support new connections and queries without rerunning the encoder. Experiments show that CoGraC improves compositional link prediction and transfers across domains with a frozen compiler. With task-specific training, the same interface supports competitive node classification and improved graph classification, while program reuse reduces computation as real graphs change.

1 INTRODUCTION

Graphs often change through new connections between familiar components. For example, a link between two communities creates new paths without changing the structure inside either community. A prediction model must account for these new paths and the local structures they connect. Reusing knowledge of each community could help the model make these predictions and avoid repeatedly processing the same structure. We study what a model should retain about a component so that it can be reused under new connections and queries.

Existing methods retain different aspects of graph structure for different purposes. Message-passing networks and structural link predictors extract features for nodes or node pairs in an observed graph (Gilmer et al., 2017; Zhang & Chen, 2018; Chamberlain et al., 2023; Wang et al., 2024a). Structural vocabularies and graph foundation models aim to transfer knowledge across graphs and tasks (Wang et al., 2024b; Sun et al., 2025). Numerical reduction preserves a given operator's behavior at a component's connection points, allowing connected components to be analyzed together (Dörfler & Bullo, 2013). These approaches leave open how to learn reusable local computations for prediction under new connections and queries; our goal is to close this gap.

This goal raises two challenges. First, standard message-passing models build node representations from information that arrives through the current connections. Changing those connections can change the representations inside a component, even when its own structure is unchanged. Computing these new representations repeats work on the same local structure. Reuse therefore requires a way to retain local knowledge and apply it to the new interactions.

Second, predictions can involve nodes deep inside a component. Consider a queried link between internal nodes in two communities. For such a query, the model needs to determine how information reaches the two internal endpoints. A summary of interactions at the connection points does not contain enough information to recover those internal responses. In fact, two components can have identical boundary behavior and different internal responses. A reusable representation must

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

therefore preserve both the boundary behavior needed to connect components and the information needed to query their internal nodes (Section 2.1).

Our idea is to learn how information propagates within each component and reuse that knowledge under different connections. Consider a community that connects to a different neighbor while keeping its internal graph unchanged. The new connection changes how the community interacts with the rest of the graph. The model can retain what it has learned about the community's internal structure, then combine that knowledge with information from other components according to the current connections. Predictions for both boundary and internal nodes can thus adapt to the new graph while the learned local relationships stay fixed.

We introduce the *Compositional Graph Compiler* (CoGraC; Figure 1) with two key designs. First, a *queryable program* stores each component's interactions at its connection points and the information needed to recover internal-node responses. This lets CoGraC connect the programs and make predictions involving nodes throughout the assembled graph. Second, CoGraC learns the local programs end to end through prediction on assembled graphs. During training, it connects the programs in different configurations and propagates prediction errors through the joint computation to update the compiler. The compiler thus learns which local information remains useful when components interact in a larger graph. Together, these designs let CoGraC learn local knowledge for global prediction and reuse it under new connections and queries.

Experiments spanning controlled compositions and eight real graph datasets demonstrate CoGraC's predictive value and practical reuse. CoGraC outperforms fixed operators and independently trained local programs in compositional link prediction, exceeding the full-graph GPR64 baseline by 7.1–8.2 AP points. The learned structural knowledge also transfers to new graph domains with a frozen compiler and few-shot readout adaptation. With task-specific training, the same program interface achieves competitive node classification accuracy and improves graph classification on Peptides-func by 2.25 AP points over matched local execution. Program reuse also accelerates local structural updates on Cora by 2.4–5.0$\times$ over full recompilation, with identical predictions. On Photo, response-column caching gives CoGraC a 2.7$\times$ overall speedup for repeated attribute updates to the same nodes.

Our main contributions are:

- **We identify a fundamental issue in graph prediction**: existing models often couple local representations with the current global connectivity, leading to redundant computation when graph compositions change. We address this by learning reusable local computations whose joint execution adapts to each assembly, allowing unchanged components to be compiled once and reused under unseen connections and queries.
- **We propose CoGraC, a new end-to-end framework that learns reusable local computation for graph prediction.** It introduces a queryable program representation that preserves both boundary behavior and recoverable internal responses. The compositional compiler uses prediction gradients through joint execution to learn which local computations remain useful after composition.
- Experiments on controlled compositions and eight real graph datasets across diverse domains show that CoGraC improves compositional link prediction and cross-domain transfer, achieves competitive node classification, and improves graph classification through joint execution. Further analysis demonstrates its computational efficiency under repeated graph updates.

2 Compositional Graph Compiler

Preliminaries and task. We learn a shared compiler and readout for prediction on graphs assembled from local components. Each component G_j contains observed edges and two sets of nodes: boundary nodes B_j and internal nodes I_j. The boundary nodes serve as connection points between components and are called *ports*. Compilation uses only (G_j, B_j), with fixed decomposition metadata included in G_j; connections between components and queries enter at execution. For link prediction, we predict whether a queried pair $q = (u, v)$ forms an edge. Either endpoint may be a boundary or internal node. Targets include hidden cross-component edges and removed real-graph edges. Attribute inputs extend the same interface to node and graph classification (Section 2.4).

Overview (Figure 1). A shared encoder produces a matrix operator describing information propagation within each component. The compiler encodes it as a program using shared finite coefficient dictionaries, retaining boundary interactions and information for recovering internal responses. CoGraC connects the programs according to the current assembly, then solves the resulting system to obtain responses for the queried nodes. A readout maps these responses to predictions. Prediction gradients through the joint computation train the encoder and dictionaries. Once the compiler is fixed, unchanged components reuse their programs across connections and queries.

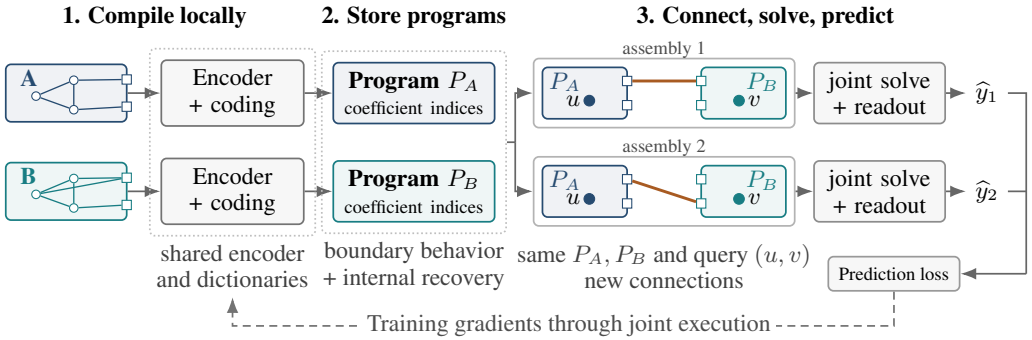

Figure 1: **(1)** A shared encoder and coefficient dictionaries compile components A and B independently. **(2)** The resulting programs P_A and P_B encode boundary interactions and the information needed to recover internal responses. **(3)** Both assemblies reuse P_A and P_B for the same internal-node query (u, v) (filled circles), with different connections between ports (orange edges between squares). The joint solve and readout produce predictions $\widehat{y}_1$ and $\widehat{y}_2$. The dashed arrow shows prediction gradients training the shared compiler.

2.1 Compiling Queryable Local Programs

We compile G_j by learning a rule that can respond to inputs from new connections and queries (Figure 1, steps 1–2). A normalized positive definite matrix M_j represents this rule: solving $M_j x = f$ gives node responses x to inputs f. To construct it, a message-passing encoder assigns positive weights to observed edges and positive diagonal regularization (*grounding*) to internal nodes, controlling neighbor interactions and node values. We normalize the resulting matrix and add fixed diagonal regularization to obtain M_j (Appendix A).

To use this rule in a larger graph, we derive a port interaction matrix Q_j from M_j. Components with the same Q_j can have different internal responses (Proposition 1). To recover these responses, we retain a matrix T_j mapping port values to their contribution at internal nodes. The internal block K_j of M_j supplies the remaining information: its inverse gives changes due to internal inputs with port values fixed. Ordering coordinates as (B_j, I_j), block elimination (Dörfler & Bullo, 2013) gives

$$K_j = M_{j,II}, \qquad T_j = -K_j^{-1} M_{j,IB}, \qquad Q_j = M_{j,BB} - M_{j,BI} K_j^{-1} M_{j,IB}. \tag{1}$$

The normalized relation $i = T_j b + K_j^{-1} f_I$ then recovers internal values from port values b and internal inputs f_I.

These factors define the local computation to be stored as program P_j. We encode them using shared finite tables of learned coefficient values, so each component records indices into the same tables. Normalization gives components a common scale for these tables; we store the visible-input scale $a_j > 0$ separately and restore it during assembly. Boundary connections sharing a port are jointly coded in groups of up to three; other coefficients use scalar entries (Appendix A.4). Together with support masks locating connections and node addresses, the indices reconstruct approximate $\widehat{Q}_j, \widehat{T}_j, \widehat{K}_j$. These factors then give the program's energy contribution to the assembly as follows:

$$\widehat{E}_j(b, i) = a_j \left[b^\top \widehat{Q}_j b + (i - \widehat{T}_j b)^\top \widehat{K}_j (i - \widehat{T}_j b) \right]. \tag{2}$$

The first term describes port interactions; the second penalizes deviations of internal values from $\widehat{T}_j b$. Nonnegative connection coefficients and positive diagonal regularization keep $\widehat{Q}_j$ and $\widehat{K}_j$ positive definite, so the decoded programs can be jointly executed.

2.2 Assembly and Node-Pair Queries

We connect the stored programs through their ports (Figure 1, step 3). Ports in different components may share a global node address; each internal node belongs to one component. Let z collect global port values and $A_j z$ select those of component j. Placing each local port operator $a_j \widehat{Q}_j$ into these global coordinates gives the assembled system

$$S_\lambda = \sum_j A_j^\top (a_j \widehat{Q}_j) A_j + C + \lambda I. \tag{3}$$

Here C encodes external connections and port grounding, while the load $\lambda \geq 0$ adds diagonal regularization to the global ports.

For query $q = (u, v)$, we apply a unit input at each endpoint in turn and record both endpoint responses. These responses capture how the queried nodes interact through the assembled graph and therefore provide structural features for link prediction. The resulting 2×2 matrix $R_{q,\lambda}$ contains self-responses on its diagonal and cross-responses off the diagonal. To recover these responses from the port system, write each internal state as its port-driven value plus a local residual, $i_j = \widehat{T}_j A_j z + \eta_j$. Substituting into Eq. (2) and adding the external connections and port load gives

$$\widehat{E}(z, \eta) = z^\top S_\lambda z + \sum_j \eta_j^\top (a_j \widehat{K}_j) \eta_j. \tag{4}$$

The energy separates into independent blocks in these coordinates. We recover query responses by combining S_λ^{-1} for ports with $(a_j \widehat{K}_j)^{-1}$ for internal residuals.

A two-row map J_q connects the query endpoints to the port system. For a boundary endpoint, its row selects the global port coordinate; for an internal endpoint, it is the corresponding row of $\widehat{T}_j A_j$. Thus $J_q^\top$ maps the two unit inputs onto ports. We solve $S_\lambda X_q = J_q^\top$ and map the port responses back through J_q, then add the conditional internal response D_q:

$$R_{q,\lambda} = J_q S_\lambda^{-1} J_q^\top + D_q \in \mathbb{R}^{2\times2}. \tag{5}$$

For internal endpoints, D_q contains the corresponding entries of $(a_j \widehat{K}_j)^{-1}$; its off-diagonal can be nonzero only when both endpoints belong to the same component. Entries involving boundary endpoints are zero. This recovers the exact query block of the inverse operator assembled from the decoded programs. New connections change the assembly, and new queries select recovery factors from the same stored programs.

To predict a link, we summarize R by four quantities: total self-response, the absolute difference of self-responses, cross-response, and endpoint-difference response. Concatenating these summaries at loads $\lambda \in \{0, 0.1, 1, 10\}$ gives 16 response features. A learned readout combines them with public graph statistics and task-specific features. In the controlled task, local alignment compares each endpoint's response to the bridge port in its own component. These local features complement responses through the full assembly. This is a strong reference under the task's port-distance-based hidden-edge rule. Real-graph link prediction uses ordinary path features. Appendix A.5 defines the summaries and both feature interfaces.

2.3 Learning through Prediction

During source training, each component is compiled and decoded once per batch, and its factors are shared across that batch's connection configurations. Prediction gradients pass through joint execution to train the shared encoder and dictionaries (Figure 1, dashed arrow). This teaches the compiler which information matters after composition. Binary cross-entropy jointly trains the encoder, dictionaries, and readout for prediction. Since finite coding introduces approximation error, we also use a relative reconstruction auxiliary to keep the decoded operator close to the continuous operator produced by the encoder (Appendix A.6). Proposition 2 (Appendix A.2) relates local coding error to the stability of assembled responses. The auxiliary's weight is calibrated on unlabeled source inputs. Hard coefficient selection is nondifferentiable, so we use a straight-through estimator (Bengio et al., 2013). The forward pass executes the decoded matrix, while the backward pass routes task gradients to the selected dictionary values and the continuous operator produced by the encoder.

2.4 Attribute Inputs and Repeated Use

For link prediction, we apply unit inputs at the queried nodes to measure their structural interactions. To support node and graph classification, we encode node attributes as inputs and propagate them through the assembled programs. Joint execution combines information across components, producing node representations that reflect both attributes and connections. A node readout predicts individual labels from these representations. A graph readout pools node features and program responses to predict graph labels. Each task jointly trains the compiler, attribute encoder, and readout, so supervision shapes the local propagation rules (Appendix D).

Separating structural programs from attribute inputs also enables reuse after training. With model parameters fixed, new attributes and queries use the existing programs. New connections change the global assembly while preserving unchanged local programs. When a component's structure or ports change, we refresh affected programs and retain the rest. This allows learned local computations to serve successive predictions as the graph evolves. Algorithm 1 (Appendix A.9) gives the full procedure; Appendix E.2 describes execution reuse under repeated updates.

3 Experiments

3.1 Experimental setup

Datasets and tasks. Amazon Computers and Coauthor Physics (Computers and Physics; Shchur et al., 2018) provide co-purchase and co-authorship graphs for controlled compositional link prediction. We hold component interiors fixed, add synthetic ports, and vary cross-component connections to predict hidden edges (Appendix A.7). Training uses six components with three ports each. Sixteen-component and five-port assemblies appear only at evaluation. Original and New-node are disjoint evaluation pools that also exclude training-component nodes. For cross-domain transfer, we freeze the source-trained compiler and evaluate on PubMed (Namata et al., 2012; Yang et al., 2016), USA air traffic (USA; Ribeiro et al., 2017), and email-Eu-core (Yin et al., 2017; Leskovec et al., 2007), covering citation, transport, and communication networks beyond the source domains. Classification uses task-specific training: Cora (Sen et al., 2008) and Amazon Photo (Photo; Shchur et al., 2018) supply attributed graphs with node labels, and Peptides-func supplies molecular graphs with function labels (Dwivedi et al., 2022).

Baselines. Fixed Operator tests operator learning with an unlearned graph operator and the same readout architecture. CoGraC-Local trains the same encoder architecture and coefficient coding with independent component execution and local alignment, testing the joint training and execution design. For full-assembly link prediction, we compare with GPR64, our adaptation of generalized PageRank propagation (Chien et al., 2021), and NCN (Wang et al., 2024a), which uses the authors' GNN and common-neighbor pair predictor. For transfer, we compare with Fixed Operator and Public/path only using CoGraC's readout architecture, and with NCN (adapted) using its frozen source-trained representation. All three follow CoGraC's readout-training and adaptation protocol; classical structural scores provide further references. For classification, we compare with a GPR-GNN-based node predictor (GPR), PolyFormer (Ma et al., 2024), and a standalone GINE graph classifier (Hu et al., 2020). Shared-frontend comparisons also test what program responses add to a GINE encoder. In classification, CoGraC-Full denotes joint execution; CoGraC denotes the complete node predictor.

Evaluation. Compositional link prediction reports ROC-AUC (AUC), average precision (AP), and binary cross-entropy (BCE). Transfer and graph classification report AP, and node classification reports accuracy. AP gains are percentage-point differences computed from unrounded scores.

Implementation. The compositional comparison trains all models on the same 24,000-update stream and evaluates final checkpoints. Appendices A.7 and A.8 detail task construction and real-graph decomposition. Appendices B–D detail baseline configurations, training, and selection.

Table 1: **Learned, jointly executed programs improve compositional link prediction.** Five-seed means with sample SD in subscripts, after averaging 12 panels per pool. Appendix B gives variant definitions, complete panels, and coding comparisons.

	Original			New-node		
Method	AUC ↑	AP ↑	BCE ↓	AUC ↑	AP ↑	BCE ↓
CoGraC	$.7427_{\pm.0006}$	$.7164_{\pm.0007}$	$.5824_{\pm.0026}$	$.7307_{\pm.0027}$	$.6954_{\pm.0035}$	$.5889_{\pm.0029}$
Fixed Operator	$.7234_{\pm.0025}$	$.7039_{\pm.0019}$	$.6000_{\pm.0013}$	$.7123_{\pm.0006}$	$.6858_{\pm.0013}$	$.6015_{\pm.0013}$
CoGraC-Local	$.7129_{\pm.0018}$	$.6914_{\pm.0021}$	$.6060_{\pm.0028}$	$.7027_{\pm.0011}$	$.6750_{\pm.0019}$	$.6132_{\pm.0020}$
GPR64	$.6519_{\pm.0149}$	$.6347_{\pm.0071}$	$.6549_{\pm.0077}$	$.6440_{\pm.0120}$	$.6242_{\pm.0090}$	$.6611_{\pm.0051}$
NCN	$.5235_{\pm.0015}$	$.5268_{\pm.0016}$	$.6920_{\pm.0002}$	$.5238_{\pm.0012}$	$.5225_{\pm.0020}$	$.6923_{\pm.0001}$

3.2 Main results

Learned programs improve compositional link prediction. As Table 1 shows, CoGraC leads Fixed Operator, CoGraC-Local, GPR64, and NCN in macro AUC, AP, and BCE across both pools and all seeds. These gains demonstrate the predictive value of learned local programs executed jointly. The matched-readout analyses in Section 3.3 then examine the contributions of stored internal information, joint execution, and compiler training. The advantage over GPR64 also persists when its training budget is doubled (Appendix B.6).

A frozen compiler transfers across graph domains. To test cross-domain reuse, we next freeze the compiler and adapt source-trained readouts using twenty labeled pairs per class in each target graph. Query and support positives are removed before feature computation.

Table 2: **Frozen-backbone transfer across graph domains.** AP means and SDs across three source-trained backbones, averaging readouts and support draws within each. Adapted rows use 20 labels per class; † denotes fixed representations. Resource allocation uses raw scores. Appendix C gives the protocol and complete metrics.

Method	PubMed	USA	email-Eu-core
CoGraC	$\mathbf{0.8442}_{\pm 0.0014}$	$\mathbf{0.7309}_{\pm 0.0022}$	$0.9274_{\pm 0.0010}$
NCN (adapted)	$0.6745_{\pm 0.0182}$	$0.6355_{\pm 0.0438}$	$0.8485_{\pm 0.0687}$
Fixed Operator†	0.8396	0.7217	0.9240
Public/path only†	0.8243	0.7027	0.9111
Resource allocation	0.6076	0.7153	**0.9607**

Table 2 shows that CoGraC improves AP over Fixed Operator, Public/path only, and NCN (adapted) on all three targets. These gains demonstrate that learned local responses remain useful across graph domains (Appendix C).

The same interface supports competitive node classification. Beyond link prediction, we train CoGraC with node labels and achieve higher mean accuracy than PolyFormer and accuracy comparable to the GPR-based reference on Cora and Photo (Table 3; Appendix D.3). Matched Cora controls further show benefits from both operator learning and joint training and execution (Appendix D.1). Section 3.4 evaluates how these classifiers reuse programs as graphs change.

Joint execution improves peptide function prediction. For graph classification, we likewise compare CoGraC-Full and CoGraC-Local using the same two-layer GINE frontend and prediction-head architectures. The models also match capacity and training budget, with each trained under its own execution rule (Appendix D.2).

As Table 3 shows, Full leads the graph-classification comparisons. In particular, it improves over matched Local and zero response in every initialization, showing that joint execution improves prediction with matched frontend and readout architectures.

Table 3: **Programs support real graph and node classification.** Left: Peptides-func test AP (%), mean ± SD over five initializations. Frontend rows use the same two-layer GINE frontend and prediction-head architectures; Full and Local also match capacity and budget. Right: final node test accuracy (%), with SD over five initializations (Cora) or five splits averaged over three initializations each (Photo). GPR and PolyFormer include graph post-processing; Appendix D.3 defines the final node predictors.

Peptides-func: graph classification

Method	Test AP
Atom/bond attribute readout	46.96 ± 0.82
GINE (standalone)	50.09 ± 2.01
Frontend + zero response	52.34 ± 0.58
Frontend + fixed local response	53.30 ± 1.20
Frontend + CoGraC-Local	53.71 ± 1.04
Frontend + CoGraC-Full	$\mathbf{55.96 \pm 1.18}$

Node classification

Method	Cora	Photo
CoGraC	82.66 ± 0.72	95.27 ± 0.43
GPR	82.34 ± 0.53	95.33 ± 0.72
PolyFormer	80.76 ± 0.87	95.02 ± 0.72

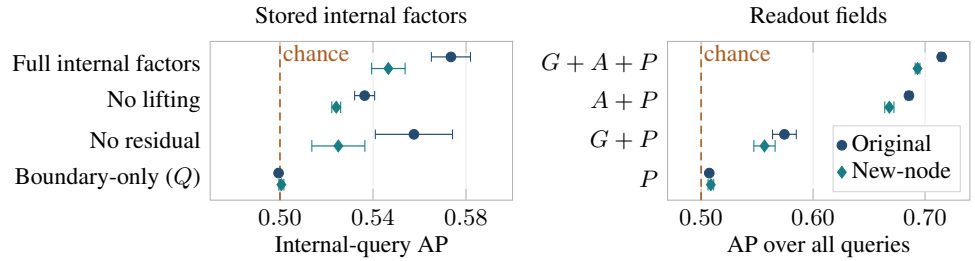

Figure 2: **Stored internal factors and joint execution each add predictive information.** Five-seed AP means and sample SD for fresh readouts on frozen compilers. Left: internal queries, response features and public graph statistics; the boundary-only interface maps internal endpoints uniformly to ports. Right: all queries with global responses G, local alignment A, and public graph statistics P. Dashed lines mark chance.

3.3 Ablations

To identify what drives compositional gains, we freeze the compilers and fit fresh readouts to controlled feature interfaces with matched initialization and budget (Figure 2; Appendix B.4).

Both internal factors improve prediction on internal queries. For internal queries, the boundary-only interface maps endpoints uniformly to ports and performs at chance. Adding the complete internal factors improves AP in both pools, while removing either lifting or the conditional residual reduces AP in every seed. Since all conditions exclude alignment, these gains isolate the complementary information the factors preserve about internal nodes.

Joint execution adds information beyond local alignment. We next cross global responses G from the assembled system with local alignment A, which compares query-to-port responses along direct bridges; all conditions retain public graph statistics P. Adding G improves macro AP over both $A+P$ and P in every seed, with $G+A+P$ performing best. These gains show that assembled responses complement local endpoint information. Further matched-readout comparisons confirm execution gains under both Local and Full training; Full-trained programs achieve the highest mean AP at the common global-response interface (Appendix B.5).

Finite coding preserves predictive performance. Turning to coefficient coding, grouped programs stay within 0.3 macro AP points of continuous coding in each pool. With the same Frobenius auxiliary and boundary-dictionary parameter count, Grouped–F also gives more consistent boundary-query AP than scalar coding (Q48) on sixteen-component assemblies across five initializations (Appendix B).

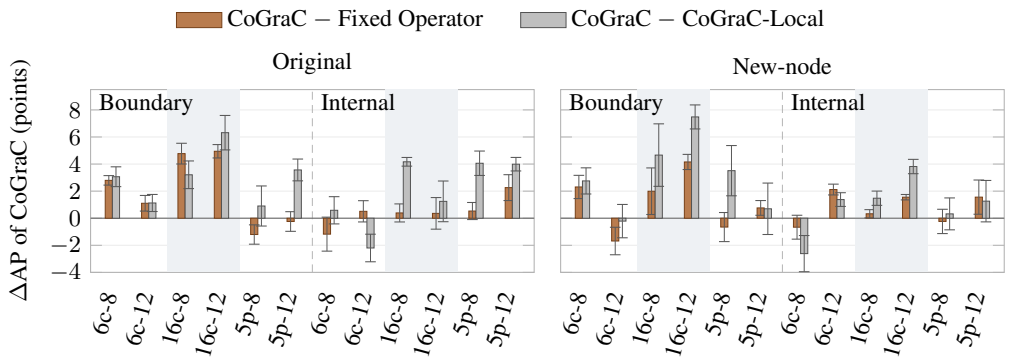

Figure 3: **CoGraC's advantage is most consistent in the largest assemblies.** Bars show five-seed mean AP differences (points) between CoGraC and each reference, with paired sample SD. Shaded: sixteen-component assemblies. 6c/16c: six/sixteen components; 5p: six components with five ports; 8/12: internal nodes per component. Appendix Table 6 lists absolute AP for every method.

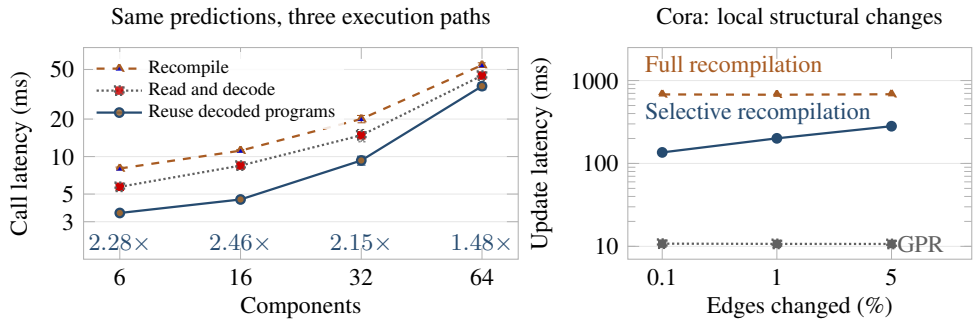

Figure 4: **Program reuse preserves predictions and reduces computation.** Left: reconnection calls answering 64 queries; mean and SD of three graph-draw medians. Right: complete Cora updates via full or selective recompilation and GPR; mean of three model-seed medians. Both CoGraC comparisons preserve predictions exactly.

3.4 Further analysis: generalization and reuse

The strongest gains occur in larger assemblies. As Figure 3 shows, CoGraC outperforms both Fixed Operator and CoGraC-Local on all sixteen-component panels, with the largest gains on boundary queries. These results demonstrate generalization to larger assemblies of independently compiled components.

Reuse under new connections and queries. Program reuse also saves computation: under new connections and queries, decoded programs give 1.48–2.46× speedups over full recompilation with identical logits (Figure 4, left). All three paths answer 64 queries per call on one CPU thread and recompute features, assembly, solves, and readout. Even including initial compilation and decoding, reuse saves 24–38% over 100 calls on average (Appendix E.1).

Local structural changes on a real graph. Next, we delete or restore Cora edges while preserving the partition and each component's spanning forest. Here, recompiling only affected components accelerates complete classifier updates by 2.43–5.01× over full recompilation for 0.1–5% edge changes (Figure 4, right). Both routes use the same LU solver and give identical scores.

Repeated attribute updates with a small cache. When only attributes change, the structural operator stays fixed, allowing response reuse. We exploit this on Photo by caching response columns online for repeated updates to the same 1% node set. This reduces preparation-plus-64-request cost by 62.4% relative to CoGraC's factor-cache execution, with unchanged accuracy and a 2.22 MiB

cache. Once populated, the cache enables millisecond-level updates. Appendix E.2 gives full protocols, GPR comparisons, and uniform-node results.

4 Related Work

Graph representations and transferable structural knowledge. Message-passing networks aggregate neighborhoods (Gilmer et al., 2017). SEAL encodes enclosing subgraphs (Zhang & Chen, 2018), ELPH/BUDDY sketch neighborhood overlap (Chamberlain et al., 2023), NCN pools common neighbors (Wang et al., 2024a), and NBFNet learns Bellman–Ford aggregation (Zhu et al., 2021). Cross-graph transfer uses text-attributed prompting in OFA (Liu et al., 2024), graph coordinators in GCOPE (Zhao et al., 2024), recurring trees and cycles in RiemannGFM (Sun et al., 2025), and attention over analytical LinearGNN predictors in GraphAny (Zhao et al., 2025). For link prediction, UniLP adapts through target-graph examples (Dong et al., 2025), while PALP learns to combine frozen experts using target supervision (Song et al., 2025). For computational reuse, GNNAutoScale caches historical embeddings during training (Fey et al., 2021), while InstantGNN incrementally updates propagated features on dynamic graphs (Zheng et al., 2022). CoGraC reuses local operators that remain executable under new connections and queries.

Learned propagation and graph classification. OptBasisGNN constructs polynomial bases from the graph and input signal to improve optimization (Guo & Wei, 2023), while PolyFormer uses attention to learn node-specific polynomial filters (Ma et al., 2024). For graph-level prediction, GNN+ strengthens classical GNN architectures through feature encoding and layer design (Luo et al., 2025).

Graph reduction and composable numerical methods. Static condensation eliminates internal coordinates (Guyan, 1965); Craig–Bampton synthesis retains interface coordinates and internal modes for coupling and recovery (Craig & Bampton, 1968). Kron reduction preserves boundary resistance (Dörfler & Bullo, 2013), and spectral reduction preserves spectral and cut properties (Loukas, 2019). Resistance and commute time motivate inverse-Laplacian pair features (Klein & Randić, 1993; Fouss et al., 2007). Learned numerical elements include SPSD interface displacement-to-force maps (Parish et al., 2024) and convex boundary energies with exact internal lifting from element matrices (Jiang et al., 2026), and LatentDDM composes pretrained local operators through subdomain exchange and corrected-input reevaluation (Chen et al., 2026). CoGraC learns operators for graph prediction and exports finite boundary and internal-query factors; endpoint differences of their assembled responses recover grounded port resistance.

Finite representations and learned computation. VQ-VAE learns a shared discrete codebook (van den Oord et al., 2017). VQGraph tokenizes neighborhoods for transfer to an MLP (Yang et al., 2024); GFT quantizes computation trees into a cross-domain vocabulary (Wang et al., 2024b); RGVQ regularizes assignments through topology and feature similarity (Zhai et al., 2026). Interaction Networks share object and relation computations across systems (Battaglia et al., 2016), and neural algorithm execution learns graph processors from algorithm states (Veličković et al., 2020). CoGraC learns finite coefficient dictionaries for boundary interactions, internal lifting, and conditional responses through the assembled prediction task.

5 Discussion and Conclusion

When graph connections change, prediction models often recompute representations of components whose internal structure remains unchanged. To address this problem, we introduced CoGraC, which learns queryable local programs through prediction on assembled graphs. These programs can be jointly executed under new connections without re-encoding unchanged components. Experiments show improved compositional link prediction and frozen cross-domain transfer, while task-specific training yields competitive node classification and improved graph classification. Program and response reuse further reduce computation as graphs change. Together, these results establish reusable local computation as a way to carry learned structural knowledge into new graph contexts. Future work could learn component boundaries that suit prediction and update patterns, and make program reuse more efficient on larger dynamic graphs.

Reproducibility Statement

Code is available at https://anonymous.4open.science/r/CoGraC-Code-FFC3/. Appendix A specifies local operators, finite encoding, features, training, task generation, and real-graph decomposition. Appendix B reports all compositional panels, coding stability, frozen-interface comparisons, and extended-budget controls. Appendix C gives the frozen-transfer protocol and complete reference comparisons. Appendix D details classification models, matched execution controls, validation-based selection, and partition sensitivity. Appendix E specifies timing workloads, preparation and cumulative costs, cache sizes, and numerical agreement. Section 3 presents the main comparisons.

AI Use Statement

In this work, we used generative AI tools for coding assistance, manuscript revision and language polishing, and the analysis and organization of experimental results. We reviewed the AI-assisted material for accuracy, consistency, and alignment with our intended meaning. We take full responsibility for the final content of this work, including its text, experimental results, code, and supporting materials.

References

Lada A. Adamic and Eytan Adar. Friends and neighbors on the web. *Social Networks*, 25(3): 211–230, 2003. doi: 10.1016/S0378-8733(03)00009-1.

Peter W. Battaglia, Razvan Pascanu, Matthew Lai, Danilo Jimenez Rezende, and Koray Kavukcuoglu. Interaction networks for learning about objects, relations and physics. In *Advances in Neural Information Processing Systems*, volume 29, 2016. URL https://proceedings.neurips.cc/paper/2016/hash/3147da8ab4a0437c15ef51a5cc7f2dc4-Abstract.html.

Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013. URL https://arxiv.org/abs/1308.3432.

Benjamin Paul Chamberlain, Sergey Shirobokov, Emanuele Rossi, Fabrizio Frasca, Thomas Markovich, Nils Hammerla, Michael M. Bronstein, and Max Hansmire. Graph neural networks for link prediction with subgraph sketching. In *International Conference on Learning Representations*, 2023. URL https://arxiv.org/abs/2209.15486.

Zituo Chen, Baiming Zhang, and Sili Deng. Learnable composition for neural operators. *arXiv preprint arXiv:2609.03069*, 2026. URL https://arxiv.org/abs/2609.03069.

Eli Chien, Jianhao Peng, Pan Li, and Olgica Milenkovic. Adaptive universal generalized PageRank graph neural network. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=n6jl7fLxrP.

Roy R. Craig, Jr. and Mervyn C. C. Bampton. Coupling of substructures for dynamic analyses. *AIAA Journal*, 6(7):1313–1319, 1968. doi: 10.2514/3.4741. URL https://doi.org/10.2514/3.4741.

Kaiwen Dong, Haitao Mao, Zhichun Guo, and Nitesh V. Chawla. Universal link predictor by in-context learning on graphs. *Transactions on Machine Learning Research*, 2025. ISSN 2835-8856. URL https://openreview.net/forum?id=EYpqmoejB8.

Florian Dörfler and Francesco Bullo. Kron reduction of graphs with applications to electrical networks. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 60(1):150–163, 2013. doi: 10.1109/TCSI.2012.2215780. URL https://ieeexplore.ieee.org/document/6316101/.

Vijay Prakash Dwivedi, Ladislav Rampášek, Mikhail Galkin, Ali Parviz, Guy Wolf, Anh Tuan Luu, and Dominique Beaini. Long range graph benchmark. In *Advances in Neural Information Processing Systems*, volume 35,

2022. URL https://papers.nips.cc/paper_files/paper/2022/hash/8c3c666820ea055a77726d66fc7d447f-Abstract-Datasets_and_Benchmarks.html.

Matthias Fey, Jan E. Lenssen, Frank Weichert, and Jure Leskovec. GNNAutoScale: Scalable and expressive graph neural networks via historical embeddings. In *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pp. 3294–3304. PMLR, 2021. URL https://proceedings.mlr.press/v139/fey21a.html.

François Fouss, Alain Pirotte, Jean-Michel Renders, and Marco Saerens. Random-walk computation of similarities between nodes of a graph with application to collaborative recommendation. *IEEE Transactions on Knowledge and Data Engineering*, 19(3):355–369, 2007. doi: 10.1109/TKDE.2007.46. URL https://doi.org/10.1109/TKDE.2007.46.

Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pp. 1263–1272, 2017. URL https://proceedings.mlr.press/v70/gilmer17a.html.

Yuhe Guo and Zhewei Wei. Graph neural networks with learnable and optimal polynomial bases. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 12077–12097, 2023. URL https://proceedings.mlr.press/v202/guo23i.html.

Robert J. Guyan. Reduction of stiffness and mass matrices. *AIAA Journal*, 3(2):380, 1965. doi: 10.2514/3.2874. URL https://doi.org/10.2514/3.2874.

Weihua Hu, Bowen Liu, Joseph Gomes, Marinka Zitnik, Percy Liang, Vijay Pande, and Jure Leskovec. Strategies for pre-training graph neural networks. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=HJlWWJSFDH.

Qian Huang, Horace He, Abhay Singh, Ser-Nam Lim, and Austin R. Benson. Combining label propagation and simple models out-performs graph neural networks. In *International Conference on Learning Representations*, 2021. URL https://arxiv.org/abs/2010.13993.

Hongyue Jiang, Jianjiang Zhan, Chenzhuo Zhang, and Fan Wang. Convex neural energy elements: Monolithic finite-element assembly of geometry-parameterized neural operators with stability and error guarantees. *arXiv preprint arXiv:2608.02036*, 2026. URL https://arxiv.org/abs/2608.02036.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, 2015. URL https://arxiv.org/abs/1412.6980.

Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=SJU4ayYgl.

D. J. Klein and M. Randić. Resistance distance. *Journal of Mathematical Chemistry*, 12(1):81–95, 1993. doi: 10.1007/BF01164627. URL https://doi.org/10.1007/BF01164627.

Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM Transactions on Knowledge Discovery from Data*, 1(1), 2007. doi: 10.1145/1217299.1217301. URL https://doi.org/10.1145/1217299.1217301. Article 2, 41 pages.

Hao Liu, Jiarui Feng, Lecheng Kong, Ningyue Liang, Dacheng Tao, Yixin Chen, and Muhan Zhang. One for all: Towards training one graph model for all classification tasks. In *International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/57faf5642eb06e0602b95f6aa989b38a-Abstract-Conference.html.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=Bkg6RiCqY7.

Andreas Loukas. Graph reduction with spectral and cut guarantees. *Journal of Machine Learning Research*, 20(116):1–42, 2019. URL https://jmlr.org/papers/v20/18-680.html.

Yuankai Luo, Lei Shi, and Xiao-Ming Wu. Can classic GNNs be strong baselines for graph-level tasks? Simple architectures meet excellence. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 41290–41310, 2025. URL https://proceedings.mlr.press/v267/luo25h.html.

Jiahong Ma, Mingguo He, and Zhewei Wei. PolyFormer: Scalable node-wise filters via polynomial graph transformer. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pp. 2118–2129, 2024. doi: 10.1145/3637528.3671849. URL https://doi.org/10.1145/3637528.3671849.

Galileo Namata, Ben London, Lise Getoor, and Bert Huang. Query-driven active surveying for collective classification. In *Proceedings of the Workshop on Mining and Learning with Graphs*, 2012. URL https://linqs.org/assets/resources/namata-mlg12.pdf.

Eric Parish, Payton Lindsay, Timothy Shelton, and John Mersch. Embedded symmetric positive semi-definite machine-learned elements for reduced-order modeling in finite-element simulations with application to threaded fasteners. *Computational Mechanics*, 74:1357–1381, 2024. doi: 10.1007/s00466-024-02481-5. URL https://link.springer.com/article/10.1007/s00466-024-02481-5.

Leonardo F. R. Ribeiro, Pedro H. P. Savarese, and Daniel R. Figueiredo. struc2vec: Learning node representations from structural identity. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 385–394, 2017. doi: 10.1145/3097983.3098061. URL https://arxiv.org/abs/1704.03165.

Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Gallagher, and Tina Eliassi-Rad. Collective classification in network data. *AI Magazine*, 29(3):93–106, 2008. doi: 10.1609/aimag.v29i3.2157. URL https://doi.org/10.1609/aimag.v29i3.2157.

Oleksandr Shchur, Maximilian Mumme, Aleksandar Bojchevski, and Stephan Günnemann. Pitfalls of graph neural network evaluation. *arXiv preprint arXiv:1811.05868*, 2018. URL https://arxiv.org/abs/1811.05868.

Yu Song, Zhigang Hua, Harry Shomer, Yan Xie, Jingzhe Liu, Bo Long, and Hui Liu. A scalable pretraining framework for link prediction with efficient adaptation. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2*, pp. 2621–2632, 2025. doi: 10.1145/3711896.3736822. URL https://doi.org/10.1145/3711896.3736822.

Li Sun, Zhenhao Huang, Suyang Zhou, Qiqi Wan, Hao Peng, and Philip Yu. RiemannGFM: Learning a graph foundation model from Riemannian geometry. In *Proceedings of the ACM on Web Conference 2025*, pp. 1154–1165, 2025. doi: 10.1145/3696410.3714952. URL https://arxiv.org/abs/2502.03251.

Aaron van den Oord, Oriol Vinyals, and Koray Kavukcuoglu. Neural discrete representation learning. In *Advances in Neural Information Processing Systems*, volume 30, 2017. URL https://proceedings.neurips.cc/paper/2017/hash/7a98af17e63a0ac09ce2e96d03992fbc-Abstract.html.

Petar Veličković, Rex Ying, Matilde Padovano, Raia Hadsell, and Charles Blundell. Neural execution of graph algorithms. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=SkgKO0EtvS.

Xiyuan Wang, Haotong Yang, and Muhan Zhang. Neural common neighbor with completion for link prediction. In *International Conference on Learning Representations*, 2024a. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/3efb4bdc6bfe13e1ff95b4407c37961d-Abstract-Conference.html.

Zehong Wang, Zheyuan Zhang, Nitesh V. Chawla, Chuxu Zhang, and Yanfang Ye. GFT: Graph foundation model with transferable tree vocabulary. In *Advances in Neural Information Processing Systems*, volume 37, 2024b. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/c23ccf9eedf87e4380e92b75b24955bb-Abstract-Conference.html.

Ling Yang, Ye Tian, Minkai Xu, Zhongyi Liu, Shenda Hong, Wei Qu, Wentao Zhang, Bin Cui, Muhan Zhang, and Jure Leskovec. VQGraph: Rethinking graph representation space for bridging GNNs and MLPs. In *International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/474ada926b331d78f06d95e8913111cc-Abstract-Conference.html.

Zhilin Yang, William W. Cohen, and Ruslan Salakhutdinov. Revisiting semi-supervised learning with graph embeddings. In *Proceedings of the 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pp. 40–48, 2016. URL https://proceedings.mlr.press/v48/yanga16.html.

Hao Yin, Austin R. Benson, Jure Leskovec, and David F. Gleich. Local higher-order graph clustering. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 555–564, 2017. doi: 10.1145/3097983.3098069. URL https://doi.org/10.1145/3097983.3098069.

Zian Zhai, Fan Li, Xingyu Tan, Xiaoyang Wang, and Wenjie Zhang. Graph is a natural regularization: Revisiting vector quantization for graph representation learning. In *International Conference on Machine Learning*, 2026. URL https://icml.cc/virtual/2026/poster/62731.

Muhan Zhang and Yixin Chen. Link prediction based on graph neural networks. In *Advances in Neural Information Processing Systems*, volume 31, 2018. URL https://proceedings.neurips.cc/paper/2018/hash/53f0d7c537d99b3824f0f99d62ea2428-Abstract.html.

Haihong Zhao, Aochuan Chen, Xiangguo Sun, Hong Cheng, and Jia Li. All in one and one for all: A simple yet effective method towards cross-domain graph pretraining. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pp. 4443–4454, 2024. doi: 10.1145/3637528.3671913. URL https://arxiv.org/abs/2402.09834.

Jianan Zhao, Zhaocheng Zhu, Mikhail Galkin, Hesham Mostafa, Michael Bronstein, and Jian Tang. Fully-inductive node classification on arbitrary graphs. In *International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/3bdeb28a531f7af94b56bcdf8ee88f17-Abstract-Conference.html.

Yanping Zheng, Hanzhi Wang, Zhewei Wei, Jiajun Liu, and Sibo Wang. Instant graph neural networks for dynamic graphs. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pp. 2605–2615. ACM, 2022. doi: 10.1145/3534678.3539352. URL https://doi.org/10.1145/3534678.3539352.

Tao Zhou, Linyuan Lü, and Yi-Cheng Zhang. Predicting missing links via local information. *The European Physical Journal B*, 71:623–630, 2009. doi: 10.1140/epjb/e2009-00335-8. URL https://arxiv.org/abs/0901.0553.

Zhaocheng Zhu, Zuobai Zhang, Louis-Pascal Xhonneux, and Jian Tang. Neural Bellman-Ford networks: A general graph neural network framework for link prediction. In *Advances in Neural Information Processing Systems*, volume 34, 2021. URL https://proceedings.neurips.cc/paper/2021/hash/f6a673f09493afcd8b129a0bcf1cd5bc-Abstract.html.

A METHOD DETAILS

A.1 ENCODER, GROUNDING, AND NORMALIZATION

All encoder inputs are local to the visible component, with saturation $s(x) = x/(1+x)$. Internal nodes receive ten fields: a constant, saturated degree/weighted degree/triangle count, clustering, port-incident weight fraction, saturated internal/port counts, saturated nearest-port distance, and a query-marked-port weight fraction fixed to zero. Ports receive a constant, the same three saturated statistics, clustering, and a query marker fixed to zero.

The ten internal fields concatenate with a normalized weighted aggregate of six adjacent-port fields. A $16 \to 32$ map with SiLU initializes the internal states. Two residual blocks concatenate each state with its weighted internal-neighbor aggregate and apply a $64 \to 32 \to 32$ SiLU network; aggregation denominators are clamped below at one. A separate $6 \to 32$ SiLU map embeds ports. The shared edge head reads endpoint-state sums and absolute differences, $\log(1 + w^0_{uv})$, and edge type. For components without ports, the saturated nearest-port distance is one and the port-feature aggregate is zero. The port-incident weight fraction is zero when the weighted degree is zero.

Let s_{uv} be the edge-head output and t_i the internal grounding-head output. The multiplier for an existing edge is $\exp(\log(4)\tanh(s_{uv}))$, and internal grounding is $g_i = 0.1\exp(\log(4)\tanh(t_i))$. Thus, the encoder changes edge strengths within the positive range $[1/4, 4]$ times their visible values and assigns positive internal grounding. Write $W^0 \in \mathbb{R}^{r\times r}$ for the symmetric visible internal adjacency and $U^0 \in \mathbb{R}^{p\times r}$ for visible port-to-internal weights. Their learned counterparts W, U and the grounding vector g define

$$H^0 = \begin{bmatrix} \operatorname{diag}(U\mathbf{1}) & -U \\ -U^\top & \operatorname{diag}(W\mathbf{1} + U^\top\mathbf{1} + g) - W \end{bmatrix}. \tag{6}$$

Normalization gives coefficient dictionaries a common operator scale. For the bounded components with nonempty boundary and internal sets used in compiler training, the input-defined scale is

$$a = \frac{\sum_{uv} W^0_{uv} + 2\sum_{bi} U^0_{bi} + 0.1r}{p + r}. \tag{7}$$

The sum of W^0 counts each undirected internal edge twice. Define $N_a(X) = aX/(\operatorname{tr}(X)/(p+r))$ and $P_B = \operatorname{diag}(\mathbf{1}_p, \mathbf{0}_r)$. The normalized operator used for coding is

$$H = N_a\big(N_a(H^0) + 0.01aP_B\big), \qquad M = H/a + 0.01I. \tag{8}$$

The two fixed grounding terms are specified before training and shared across executable comparison methods. The complete-coordinate grounding is incorporated before factorization, and the decoded energy is reconstructed directly from those factors.

A.2 CONDITIONAL FACTORIZATION AND QUERY RESPONSE

Proposition 1 (Boundary behavior does not determine internal responses)**.** *Positive definite grounded graph operators can share the same port interaction matrix while yielding different internal responses.*

Proof. Take one boundary and one internal coordinate with

$$M_t = \begin{bmatrix} 1 + t^2 & -t \\ -t & 1 \end{bmatrix}, \qquad 0 < t < 1. \tag{9}$$

This matrix has determinant one and positive grounding $1 + t^2 - t$ and $1 - t$. Every member has $Q = K = 1$ and $T = t$. Since $M_t^{-1} = \left[\begin{smallmatrix} 1 & t \\ t & 1+t^2 \end{smallmatrix}\right]$, its boundary self-response is always one, while its internal self-response is $1 + t^2$. □

Retaining T and K carries this internal information into later assemblies.

Conditional energy and recovery. The conditional form separates the energy transmitted through the boundary from the remaining internal energy. For a positive definite matrix M with nonempty boundary and internal blocks, completing the square gives

$$\begin{bmatrix} b \\ i \end{bmatrix}^\top M \begin{bmatrix} b \\ i \end{bmatrix} = b^\top Q b + (i - Tb)^\top K (i - Tb), \tag{10}$$

$$M = \begin{bmatrix} Q + T^\top K T & -T^\top K \\ -KT & K \end{bmatrix}, \tag{11}$$

with Q, T, K from Eq. (1). Replacing them with the decoded factors yields $\widehat{M}$. Positive definiteness follows directly: if $b \neq 0$, the boundary term is positive; if $b = 0$ and $i \neq 0$, the internal term is positive. Hence $\widehat{Q} \succ 0$ and $\widehat{K} \succ 0$ imply $\widehat{M} \succ 0$.

Equation (4) applies the same change of coordinates to all components. Its block inverse maps back to Eq. (5): the scale multiplies Q and K once, and T is unchanged. An internal endpoint always contributes its local inverse diagonal to D_q; cross terms occur only for two internal endpoints in the same component.

Connection to resistance distance. Let $\widehat{H}_\lambda$ be the full assembled operator on ports and internal nodes, and let $d = (1, -1)^\top$. The endpoint-difference response is

$$r_1 + r_2 - 2c = (e_u - e_v)^\top \widehat{H}_\lambda^{-1} (e_u - e_v) = d^\top J_q S_\lambda^{-1} J_q^\top d + d^\top D_q d. \tag{12}$$

This is a squared distance in the inner product induced by $\widehat{H}_\lambda^{-1}$. For port endpoints, $D_q = 0$ and S_λ is a grounded graph Laplacian. Connecting its grounding conductances to a common reference node gives the effective-resistance interpretation for every $\lambda \geq 0$ (Klein & Randić, 1993; Dörfler & Bullo, 2013). For internal endpoints, independent coding of Q, T, K preserves positive definiteness and can produce reconstructed operators outside the grounded-Laplacian class. The inverse-kernel interpretation still applies, including the internal residual term.

Proposition 2 (Stability under finite coding)**.** *Let $M_j \succ 0$, $a_j > 0$, and $L_j L_j^\top = M_j$. Suppose the decoded operators satisfy*

$$\epsilon_j = \left\| L_j^{-1} \widehat{M}_j L_j^{-\top} - I \right\|_2, \qquad \epsilon = \max_j \epsilon_j < 1.$$

Let $H_\lambda^{\mathrm{ref}} \succ 0$ and $\widehat{H}_\lambda$ be assembled from the continuous and decoded operators, respectively, using the same component scales, connections, grounding, and nonnegative port load. Then

$$(1 - \epsilon) H_\lambda^{\mathrm{ref}} \preceq \widehat{H}_\lambda \preceq (1 + \epsilon) H_\lambda^{\mathrm{ref}}.$$

For every query q, the corresponding response blocks satisfy

$$\frac{1}{1 + \epsilon} R_{q,\lambda}^{\mathrm{ref}} \preceq R_{q,\lambda} \preceq \frac{1}{1 - \epsilon} R_{q,\lambda}^{\mathrm{ref}}.$$

Proof. The local error bound gives $(1 - \epsilon) M_j \preceq \widehat{M}_j \preceq (1 + \epsilon) M_j$. Positive scaling, mapping into global coordinates, and summation preserve this order. Adding the shared positive semidefinite connection and load terms gives the assembled bound. Inversion reverses the order, and restriction to query coordinates gives the response bound. □

The response bound applies to self-responses and the endpoint-difference response. It depends on the largest local error, with no explicit sum over components. The per-component relative auxiliary in Eq. (16) upper-bounds ϵ_j^2 / n_j, connecting the training objective to this stability analysis.

A.3 Numerical execution

The controlled implementation assembles disjoint port blocks and the external connection energy, then solves the dense port system for the queried endpoints. Its external matrix C contains a fixed grounding of 0.1 per port. The real-graph implementation assembles a sparse port system using shared port addresses and uniquely owned internal nodes. Each visible edge has one component

owner. Its local energies supply grounding, and the additional load is charged once per global port address.

The sparse solver uses diagonally preconditioned conjugate gradients with relative tolerance 10^{-10}, zero absolute tolerance, and at most 5,000 iterations per right-hand side. It checks the attained relative residual against $\max(10\times10^{-10}, 2\times10^{-12})$ and raises an error on nonconvergence. Empty factor blocks use the corresponding reduced interface, and singleton components use the fixed unit shape with their input scale.

A.4 Dictionary construction and indexing

The dictionaries encode nonnegative coefficients so that decoding preserves the conditional energy structure. For a grounded factor $F \in \{Q, K\}$, define $c_{ab} = -F_{ab}$ for $a < b$ and $g = F\mathbf{1}$. Decoding gives $\widehat{F} = \mathcal{L}(\widehat{c}) + \operatorname{diag}(\widehat{g})$, where $\mathcal{L}$ is the weighted graph Laplacian. The scalar dictionaries for Q grounding, K connections, K grounding, and T each contain a fixed zero and 15 learned positive levels. Positive levels are represented as cumulative softplus increments. A zero coefficient selects the zero level, while a positive coefficient selects the nearest positive level in logarithmic coordinates.

The boundary connection dictionary groups coefficients that share a port. At each step, select the port with the largest number of unassigned positive connections, take up to three of those connections, and remove them from the unassigned set. Ties between ports are resolved by local index, and edges are visited in lexicographic endpoint order. For three fully connected ports, the resulting groups have sizes two and one.

Let $w_1, \dots, w_{16} \in \mathbb{R}^3_{>0}$ be the learned triples, and let $\pi_0, \dots, \pi_5$ denote the six permutations of the three coefficient slots within each group. For a group c_g with occupancy mask m_g, selection is

$$(\ell_g, k_g) = \arg\min_{\ell\in\{0,\dots,5\},\, k\in\{1,\dots,16\}} \sum_{s=1}^{3} m_{g,s}\big(c_{g,s} - (\pi_\ell w_k)_s\big)^2. \tag{13}$$

The 96 candidates share 48 learned scalar parameters. With zero-based word indices, the stored ID is $16\ell_g + (k_g - 1)$. Support masks identify active connections and occupied slots during reconstruction.

The first 24 unlabeled source input batches initialize the dictionaries and feature moments. Boundary triples are initialized by clustering the logarithms of sorted coefficient groups. Short groups are filled by their mean for initialization. Sixteen farthest-point initial centers undergo eight assignment/mean updates. Each scalar dictionary initializes 15 logarithmically spaced positive levels from half the smallest positive source value to twice the largest, then learns their increments. Deployment stores coefficient indices, supports, node addresses, and component scales together with the shared dictionaries.

A.5 Feature definitions

Response features summarize how the assembled operator couples the query endpoints. For $R = R_{q,\lambda}$, let $r_1 = R_{11}$, $r_2 = R_{22}$, and $c = (R_{12} + R_{21})/2$. Apply $\operatorname{slog}(x) = \operatorname{sign}(x)\log(1 + |x|)$ elementwise to

$$(r_1 + r_2,\ |r_1 - r_2|,\ c,\ r_1 + r_2 - 2c). \tag{14}$$

These four quantities capture total self-response, endpoint asymmetry, cross-response, and the response along the endpoint difference. Concatenating them in load-major order over $0, 0.1, 1, 10$ gives 16 fields. Increasing port grounding attenuates propagation through the assembled ports; as $\lambda \to \infty$, the response approaches the conditional term D_q in Eq. (5).

The controlled alignment features compare how each endpoint couples to the ports used by a direct connection between its component and the other endpoint's component. At a given load, solve the decoded local energy with loading on port coordinates: $V_j = (a_j \widehat{M}_j + \lambda P_{B_j})^{-1}$. For endpoint u and port b in component j, define

$$d_{u,b} = -\log\left[\operatorname{clip}\left(\frac{(V_j)_{ub}}{\sqrt{(V_j)_{uu}(V_j)_{bb}}}, 10^{-8}, 1\right)\right]. \tag{15}$$

Normalize the observed direct port-bridge weights between the endpoint components to sum to one. For each bridge (b, c), form $(d_{u,b} + d_{v,c})/2$, $|d_{u,b} - d_{v,c}|$, $(d_{u,b} - d_{v,c})^2$, and $\exp(-|d_{u,b} - d_{v,c}|)$. Their bridge-weighted means, followed by $\log(1 + x)$, give four features per load. The alignment fields are zero when the endpoint components have no direct bridge or when both endpoints have the same owner. The global response fields capture connections through the assembled system.

Public graph statistics provide the readout with visible endpoint and graph statistics. For each endpoint, compute saturated degree, weighted degree, and triangle count, plus clustering coefficient. Their means and squared differences give eight endpoint fields. Eight graph fields apply $\log(1 + x)$ to node count, edge count, degree mean and standard deviation, weighted-degree mean and standard deviation, mean triangle count, and public port count. In the real-graph interface, the port-count statistic follows the fixed public decomposition used to construct these fields.

The real-graph interface also includes nine ordinary path fields: $\log(1 + \mathrm{CN})$, Adamic–Adar, resource allocation, Jaccard similarity, $\log(1 + d_u d_v)$, $\log(1 + \min(d_u, d_v))$, $\log(1 + \max(d_u, d_v))$, reachability, and reciprocal shifted shortest-path distance $1/(1 + \mathrm{dist}(u, v))$, set to zero when unreachable. Here CN is the common-neighbor count and d_u, d_v are visible degrees. All fields use the visible graph after the held-out positive pairs have been removed. The complete layouts are

Interface	First 16	Next fields	Remaining fields
Controlled	Response	16 local alignment	16 public
Real graph	Response	9 path + 16 public	7 zeros
CoGraC-Local	Alignment	16 transformed alignment	16 public

CoGraC-Local uses the feature vector $[A, \tanh(A)^{\odot 2}, P]$, where A is the 16-dimensional alignment vector and P contains the 16 public graph statistics. Here A already includes the $\log(1+x)$ transform above; the elementwise $\tanh$ and square are applied before source standardization. Thus the local interface combines independently computed alignment responses, their nonlinear transform, and public graph statistics.

The readout standardizes each field, applies a $48 \to 128$ linear map and SiLU, then a $128 \to 128$ linear map with ℓ_2 output normalization. A final linear classifier gives the logit. Controlled feature moments are fixed from the first 24 source input batches. A standard deviation at most 10^{-5} is replaced by one. Real-graph readouts use moments from their source fitting set and the same standardization rule.

A.6 Training and reuse

Binary cross-entropy and a relative reconstruction auxiliary jointly train the compiler and readout. Let $L_j L_j^\top = \mathrm{stopgrad}(M_j)$, let $\widehat{M}_j$ be the normalized matrix represented by Eq. (2), and let n_j be the component size. The objective is

$$\mathcal{L} = \mathcal{L}_{\mathrm{BCE}} + \beta\, \mathbb{E}_j \left[\frac{1}{n_j} \left\| L_j^{-1} \widehat{M}_j L_j^{-\top} - I \right\|_F^2 \right]. \tag{16}$$

For the relative auxiliary, the source-calibrated weight in Eq. (16) is $\beta \approx 2.005 \times 10^{-3}$. An initial unlabeled calibration on 32 source inputs sets its gradient scale relative to a Frobenius auxiliary weighted by 0.02. The reference matrix is detached when computing the whitening transform.

Hard assignments are held fixed during the backward pass. Writing sg for stop-gradient, the straight-through surrogate is conceptually

$$\widetilde{M} = \widehat{M} + M - \mathrm{sg}(M). \tag{17}$$

We apply this estimator to the matrix reconstructed from the decoded factors, then refactor it for execution. The forward pass uses $\widehat{M}$, while the backward pass routes the same gradient to the decoded and continuous branches. These task gradients train the selected dictionary values and encoder, respectively. The auxiliary in Eq. (16) uses $\widehat{M}$ directly and a detached reference; with hard assignments fixed, it updates only the dictionaries.

The encoder, dictionaries, and controlled readout are optimized jointly using AdamW (Loshchilov & Hutter, 2019) with initial learning rate 0.002, weight decay 10^{-4}, gradient clipping at 5, and a whole-horizon cosine schedule ending at one tenth of the initial learning rate. Training runs for 24,000 updates, alternating 12,000 synthetic-component batches and 12,000 source-derived batches with real interiors, constructed ports, and constructed labels. Evaluation uses the final checkpoint.

Compilation and reuse. Training decodes each component once per batch and shares its factors across connection configurations. After training, stored programs and shared dictionaries serve new connections and queries. Execution recomputes public features from the current visible graph; local graph or port changes refresh the affected programs.

A.7 Constructed task generation

Components and ports. The source-derived components use connected induced subgraphs from the visible Computers and Physics graphs. These subgraphs form the component interiors. Starting from a randomly selected available node, we grow an interior by repeatedly sampling a node from its frontier; its nodes are then excluded from further interior extraction in the same pool. Training interiors have 4, 8, or 12 nodes, while the reported evaluation panels use 8 or 12. Each component receives three synthetic ports, or five in the expanded-interface condition. Each port connects to two distinct, uniformly sampled internal nodes through unit-weight edges. A bank samples component interiors with replacement, alternates between the two source graphs, and fixes the resulting local graphs and port attachments before generating external connections or queries. The 24,000-update training stream interleaves 12,000 synthetic-component batches with 12,000 source-derived batches. The synthetic branch contains 6,000 boundary-query and 3,000 internal-query batches from weighted motifs with six internal nodes, plus 3,000 internal-query batches from path, cycle, tree, and community interiors with 4, 8, or 12 nodes. The source-derived branch contains 6,000 boundary-query and 6,000 internal-query batches. Both branches use the boundary- and internal-query sampling rules described below. The source-derived branch uses 16 banks per internal-node count, each containing six components and three ports per component; each batch contains two sampled assemblies.

Assemblies. The standard condition places six components on a triangular-prism scaffold. The deeper condition uses 16 components on a Möbius-ladder scaffold, and the expanded-interface condition retains six components with five ports each. Component positions are randomly permuted on the scaffold. External edges join ports of adjacent components, with weights sampled uniformly from $[0.75, 1.25]$. Internal and boundary queries use the following two sampling rules.

Internal queries. For every scaffold edge (j, k), we first sample one port uniformly from each component and add the corresponding visible bridge (b_j, b_k). Let $d_j(u)$ and $d_k(v)$ be the unweighted hop distances from these ports to internal nodes $u \in I_j$ and $v \in I_k$, computed within their respective local graphs. We sample one hidden edge with probability

$$\Pr\big((u, v) \mid b_j, b_k\big) = \frac{\exp[-6\,|\log(1 + d_j(u)) - \log(1 + d_k(v))|]}{\sum_{x \in I_j} \sum_{y \in I_k} \exp[-6\,|\log(1 + d_j(x)) - \log(1 + d_k(y))|]}. \tag{18}$$

This sampled edge is a positive query and is omitted from the visible graph. Its negative is an absent edge between the same two component interiors. We shuffle eligible candidates and choose the one minimizing a lexicographic cost: first the ℓ_1 difference between sorted pairs of visible endpoint degrees, then the indicator that its visible-graph hop distance differs from the positive's. Thus exact matching is used when available, with degree discrepancy taking priority otherwise.

Boundary queries. For each scaffold edge, we select two distinct ports in each incident component. The probability of the two unordered port pairs is proportional to the exponential in Eq. (18), with d_j and d_k now denoting the within-component distances between the two ports on each side. An independently sampled orientation pairs the left and right endpoints into two external bridges. After constructing the full assembly, we uniformly select eight external edges, remove them from the common visible graph, and use them as positive queries. Eight negatives are selected from original cross-component port nonedges using the same degree-first, hop-second matching rule, without

repeating negatives. For both query types, the labels indicate whether a candidate belongs to the sampled full graph; local path compatibility determines sampling probabilities.

Evaluation pools. The Original pool extracts a fresh population of interiors after excluding all nodes in the training component pool. The New-node pool additionally excludes the Original population. Both draw from the same two source graphs. Each pool contains twelve panels: two internal-node counts, three assembly conditions, and two query types. The Original pool has 12 banks per panel and the New-node pool has 16; each bank is evaluated under two separately sampled external connection configurations. The same bank supplies both query types within an assembly condition.

A.8 Decomposition of real graphs

Edge ownership and shared ports. We construct local components from the visible graph using overlapping patches of at most 16 nodes, with every visible edge assigned to exactly one patch. Nodes are ordered by decreasing original visible degree, with node identifiers resolving ties. The next patch starts at the first node in this order with an unassigned incident edge. We grow the patch by choosing the frontier node with the most remaining connections to already selected nodes; original degree and then node identifier resolve ties. Growth stops at 16 nodes or an empty frontier. The patch receives all still-unassigned edges between its selected nodes, and these edges are removed from the remaining graph. The process repeats until every edge has an owner; isolated nodes receive singleton patches.

A node occurring in multiple patches becomes a shared port in each occurrence; a node occurring in one patch is an internal node of that component. Local node order lists ports first, followed by internal nodes, each in increasing identifier order. The compiler operates on the resulting local edge records, and assembly identifies copies of each shared port by their common address. For these patches, the frozen encoder applies its existing port embeddings and shared edge head to owned port-to-port edges as well as port-to-internal and internal edges. Writing B^0 for the owned port-to-port adjacency, its input scale includes every owned edge:

$$a = \frac{\sum W^0 + 2\sum U^0 + \sum B^0 + 0.1\max(r,1)}{p+r}. \tag{19}$$

The operator uses the same two trace rescalings, compile-time port grounding, and pre-factorization $0.01I$, together with the frozen coefficient dictionaries. All-port patches use Q alone, all-internal patches use K alone, and singleton patches use the fixed unit shape. For the Fixed Operator reference, a node occurring in c_u patches receives grounding $0.1/c_u$ in each occurrence, giving total grounding 0.1 per graph node. On the visible PubMed graph, this yields 6,612 patches, 4,546 distinct shared ports, and 15,171 internal nodes; on USA, it yields 465 patches, 603 shared ports, and 587 internal nodes.

A.9 DETAILED TRAINING AND REUSE ALGORITHM

Algorithm 1 separates compilation, end-to-end training, and reuse with a fixed compiler. Let θ denote encoder parameters, $\mathcal{D}$ the shared coefficient dictionaries, and ψ the task's frontend, readout, and learned propagation parameters. A program P_j stores coefficient indices, support masks, node addresses, and scale a_j; its decoded factors are $\widehat{\mathcal{F}}_j = (\widehat{Q}_j, \widehat{T}_j, \widehat{K}_j)$.

Joint prediction. The PREDICT routine uses the current assembly and task inputs. For link prediction, it assembles S_λ at each load $\lambda \in \{0, 0.1, 1, 10\}$. For each query, it constructs J_q, D_q, solves $S_\lambda X_q = J_q^\top$, and recovers $R_{q,\lambda} = J_q X_q + D_q$. The readout combines response features with the local alignment or path features and public graph statistics defined in Appendix A.5.

For classification, PREDICT maps attributes to right-hand sides and executes Eq. (20), injecting each global port input once. Node classification applies the row-normalized response polynomial and task-specific output rule; graph classification pools attributes and responses. Loading follows the task configuration: Photo distributes its all-node load across component owners and recondenses the loaded factors (Appendix D.3 also specifies the node-classification output rules). The task loss is binary cross-entropy for link prediction, training-node cross-entropy for node classification, and binary cross-entropy over valid labels for graph classification.

Gradient and cache lifetimes. Within each backward pass, hard assignments stay fixed; the straight-through estimator routes gradients to the continuous operator produced by the encoder and selected dictionary values. Each new batch uses updated parameters. During reuse, changes to local graphs, ports, node addresses, or grounding shares refresh affected programs; Cora refreshes affected packed groups. Operator changes invalidate factorizations and response caches; propagation-coefficient changes also invalidate response-column caches. New queries or attributes reuse valid caches (Appendix E.2). Readout adaptation retains the frozen compiler's programs (Appendix C).

Algorithm 1 CoGraC compilation, end-to-end training, and program reuse

Require: Local components and ports; assembly configurations; queries or attributes; training labels; task configuration.

Ensure: Learned $(\theta, \mathcal{D}, \psi)$ and predictions from reusable local programs.

procedure COMPILE($G_j, B_j; \theta, \mathcal{D}$)
  Compute scale a_j and normalized local operator M_j from (G_j, B_j).
  Obtain Q_j, T_j, K_j by conditional factorization (Eq. (1)).
  Select grouped/scalar dictionary entries and form P_j (Appendix A.4).
  Decode P_j into $\widehat{\mathcal{F}}_j$; **return** $(P_j, \widehat{\mathcal{F}}_j, M_j)$.
end procedure

Training

Initialize parameters and dictionaries using the task's training protocol.
for each training batch **do**
  for each distinct local component j in the batch **do**
    $(P_j, \widehat{\mathcal{F}}_j, M_j) \leftarrow$ COMPILE$(G_j, B_j; \theta, \mathcal{D})$.
  end for
  Share decoded factors across the batch's assemblies, retaining their computation graph.
  Apply PREDICT to each assembly and its queries or attributes, using ψ.
  Form $\mathcal{L}_{\text{task}} + \beta \mathcal{L}_{\text{rec}}$, with relative reconstruction as in Eq. (16).
  Backpropagate through joint execution with the straight-through estimator; update $(\theta, \mathcal{D}, \psi)$.
end for

Reuse with fixed compiler parameters

Fix $(\theta, \mathcal{D})$ and initialize a program cache for these parameters.
for each new prediction request **do**
  for each component j whose program is missing or whose local input has changed **do**
    $(P_j, \widehat{\mathcal{F}}_j, \cdot) \leftarrow$ COMPILE$(G_j, B_j; \theta, \mathcal{D})$; refresh its cache entry.
  end for
  Retain all other programs; update the assembly and dependent caches as needed.
  Apply PREDICT with the current connections, queries or attributes, and readout.
  Output predictions and retain valid cache entries for subsequent requests.
end for

B COMPOSITIONAL LINK PREDICTION AND CODING COMPARISONS

We use five training seeds, 107001–107003 and 107101–107102, and a common 24,000-update source stream. Each seed averages twelve panels per pool (Section 3); subscripts give sample SD across seeds. Fixed Operator seeds vary its readout. AUC/AP are higher-is-better and BCE is lower-is-better.

B.1 COMPARISON DESIGN

Table 4: Executable variants. Relative and Frobenius denote reconstruction auxiliaries. Learned variants train the encoder and readout; finite variants also train their dictionaries. Fixed Operator trains only its readout.

Method	Local operator	Coding	Execution
CoGraC	Learned	Grouped; relative	Full
Fixed Operator	Visible graph	None	Full
CoGraC-Local	Learned	Grouped; relative	Local
Scalar coding	Learned	Scalar; Frobenius	Full
Continuous coding	Learned	Continuous; Frobenius	Full
Grouped–F	Learned	Grouped; Frobenius	Full

Fixed Operator uses visible edge weights, fixed grounding, and no learned encoder or coefficient quantization. For constructed tasks it follows the normalization in Appendix A with unchanged edge weights and internal grounding 0.1; for real graphs its assembled energy is the visible-graph Laplacian plus $0.1I$ and the specified port load.

GPR64 is our full-graph link predictor built around generalized PageRank propagation (Chien et al., 2021). It operates on the full visible graph, using 16 node fields describing visible topology, component statistics, and port roles. Its encoder combines node, neighbor, and component-pooled features, followed by a learned mixture of propagation orders 0–64. The pair representation concatenates endpoint means and squared differences, a graph mean, eight graph statistics, and 36 pairwise walk features from twelve propagation orders, giving 236 fields. The walk features summarize cross-responses and endpoint self-responses. Executable readouts take 48 fields; both systems project their inputs to a 128-dimensional prediction representation.

NCN is a full-graph link predictor based on the authors' `cn1` implementation (Wang et al., 2024a). It operates on the full visible graph, using 16 public node fields describing visible topology, component statistics, and port roles. Its one-layer GCN (Kipf & Welling, 2017) encodes nodes using the visible edge weights. The native common-neighbor predictor then constructs a learned representation for each queried node pair, with common neighbors identified through binary adjacency. The resulting representation is passed to the final linear prediction layer to produce a link score. This pair representation also supplies the frozen features for the transfer comparison in Appendix C.

The scalar and continuous variants use the Frobenius auxiliary, so Grouped–F supplies the matched coding comparison.

B.2 COMPLETE-MODEL AND CODING COMPARISONS

Grouped–F uses the same six-permutation boundary dictionary as CoGraC with the Frobenius auxiliary $\|\widehat{M} - M\|_F^2/n$, weighted by 0.02. Q48 encodes boundary connections independently with 48 positive levels plus zero, matching Grouped–F's boundary-coefficient parameter count. Its log-space initialization clusters the first 24 source inputs with eight assignment/mean updates; assignment then minimizes squared error, as in grouped coding. Grounding and internal factors retain their scalar dictionaries. Grouped–F, Q48, and continuous coding share the reconstruction auxiliary; Grouped–F and Q48 also share encoder/readout initialization.

CoGraC improves all three macro metrics over Fixed Operator, CoGraC-Local, GPR64, and NCN in every paired seed. Grouped and continuous programs have similar macro AP, while Grouped–F versus Q48 isolates grouped coefficient selection.

Table 5: Complete-model and coding comparisons across five training seeds. Each seed first averages the twelve panels in a pool; subscripts give sample SD across seeds. Grouped–F and Q48 match the reconstruction auxiliary, encoder/readout initialization, and boundary-coefficient parameter count.

	Original			New-node		
Method	AUC ↑	AP ↑	BCE ↓	AUC ↑	AP ↑	BCE ↓
CoGraC	$.7427_{\pm.0006}$	$.7164_{\pm.0007}$	$.5824_{\pm.0026}$	$.7307_{\pm.0027}$	$.6954_{\pm.0035}$	$.5889_{\pm.0029}$
Fixed Operator	$.7234_{\pm.0025}$	$.7039_{\pm.0019}$	$.6000_{\pm.0013}$	$.7123_{\pm.0006}$	$.6858_{\pm.0013}$	$.6015_{\pm.0013}$
CoGraC-Local	$.7129_{\pm.0018}$	$.6914_{\pm.0021}$	$.6060_{\pm.0028}$	$.7027_{\pm.0011}$	$.6750_{\pm.0019}$	$.6132_{\pm.0020}$
GPR64	$.6519_{\pm.0149}$	$.6347_{\pm.0071}$	$.6549_{\pm.0077}$	$.6440_{\pm.0120}$	$.6242_{\pm.0090}$	$.6611_{\pm.0051}$
NCN	$.5235_{\pm.0015}$	$.5268_{\pm.0016}$	$.6920_{\pm.0002}$	$.5238_{\pm.0012}$	$.5225_{\pm.0020}$	$.6923_{\pm.0001}$
Scalar coding (Q48)	$.7208_{\pm.0306}$	$.6840_{\pm.0299}$	$.6055_{\pm.0160}$	$.7163_{\pm.0274}$	$.6714_{\pm.0286}$	$.6111_{\pm.0131}$
Continuous coding	$.7406_{\pm.0014}$	$.7140_{\pm.0020}$	$.5833_{\pm.0024}$	$.7328_{\pm.0011}$	$.6983_{\pm.0018}$	$.5869_{\pm.0017}$
Grouped–F	$.7430_{\pm.0013}$	$.7146_{\pm.0033}$	$.5822_{\pm.0040}$	$.7310_{\pm.0012}$	$.6973_{\pm.0017}$	$.5885_{\pm.0045}$

B.3 Complete evaluation panels

Table 6 reports all 24 panels: two internal sizes (8 and 12), three assembly conditions, and boundary/internal queries in each pool. Panels contain twelve banks in Original and sixteen in New-node.

Table 6: Every compositional evaluation panel: five-seed mean AP. 6c and 16c denote six- and sixteen-component assemblies; 5p denotes six-component assemblies with five ports. The middle number is the internal-node count; B/I indicate boundary/internal queries. Grouped–F, Q48, and continuous coding use the Frobenius auxiliary.

Panel	CoGraC	Fixed	Local	GPR64	Grouped–F	Q48	Cont.	NCN
Original								
6c-8-B	0.7927	0.7647	0.7620	0.6228	0.7909	0.7612	0.7839	0.5224
6c-12-B	0.7600	0.7489	0.7487	0.6643	0.7559	0.7497	0.7503	0.4979
16c-8-B	0.8934	0.8457	0.8613	0.7701	0.8960	0.7463	0.8903	0.5187
16c-12-B	0.9258	0.8764	0.8626	0.8000	0.9274	0.7845	0.9259	0.5255
5p-8-B	0.6390	0.6510	0.6300	0.5860	0.6322	0.6500	0.6372	0.5052
5p-12-B	0.6845	0.6869	0.6488	0.6087	0.6828	0.6699	0.6791	0.5188
6c-8-I	0.6374	0.6491	0.6315	0.6018	0.6307	0.6326	0.6291	0.5257
6c-12-I	0.6698	0.6647	0.6918	0.6064	0.6725	0.6528	0.6698	0.5744
16c-8-I	0.6465	0.6425	0.6048	0.6049	0.6446	0.6539	0.6440	0.5200
16c-12-I	0.6140	0.6104	0.6016	0.5666	0.6157	0.6208	0.6184	0.5321
5p-8-I	0.6681	0.6628	0.6275	0.5963	0.6622	0.6452	0.6710	0.5490
5p-12-I	0.6660	0.6434	0.6261	0.5882	0.6639	0.6407	0.6691	0.5316
New-node								
Panel	CoGraC	Fixed	Local	GPR64	Grouped–F	Q48	Cont.	NCN
6c-8-B	0.7847	0.7616	0.7571	0.6385	0.7829	0.7535	0.7828	0.5086
6c-12-B	0.7611	0.7779	0.7632	0.6391	0.7649	0.7474	0.7725	0.5322
16c-8-B	0.8765	0.8565	0.8299	0.7723	0.8835	0.7597	0.8738	0.5335
16c-12-B	0.9206	0.8791	0.8458	0.7335	0.9252	0.8265	0.9131	0.5143
5p-8-B	0.6799	0.6864	0.6448	0.6087	0.6808	0.6744	0.6940	0.5276
5p-12-B	0.6601	0.6525	0.6531	0.5762	0.6669	0.6829	0.6631	0.5028
6c-8-I	0.6126	0.6192	0.6387	0.6170	0.6165	0.6169	0.6185	0.5339
6c-12-I	0.6009	0.5797	0.5871	0.5641	0.5983	0.5804	0.6001	0.5182
16c-8-I	0.6090	0.6057	0.5942	0.5917	0.6139	0.6143	0.6097	0.5259
16c-12-I	0.6272	0.6116	0.5890	0.5903	0.6221	0.6245	0.6243	0.5399
5p-8-I	0.6027	0.6050	0.5994	0.5815	0.6018	0.5955	0.6015	0.5201
5p-12-I	0.6097	0.5941	0.5971	0.5769	0.6110	0.5803	0.6260	0.5134

Table 7 shows Grouped–F's stability across all five initializations on the four sixteen-component boundary panels.

Table 7: Boundary-query AP on sixteen-component assemblies by seed. Each pair compares Q48 with Grouped–F (G–F) under the Frobenius auxiliary; 8/12 denotes internal nodes per component.

	Original				New-node			
	16c-8-B		16c-12-B		16c-8-B		16c-12-B	
Seed	Q48	G–F	Q48	G–F	Q48	G–F	Q48	G–F
107001	0.8205	0.9012	0.8919	0.9245	0.8667	0.8850	0.9297	0.9270
107002	0.8364	0.8919	0.8516	0.9324	0.8449	0.8862	0.9095	0.9258
107003	0.7051	0.8925	0.7806	0.9246	0.7072	0.8980	0.8655	0.9271
107101	0.9039	0.8977	0.9190	0.9186	0.8581	0.8701	0.9456	0.9224
107102	0.4657	0.8965	0.4795	0.9369	0.5217	0.8781	0.4824	0.9237

Training cost. All models use 24,000 updates. Five-seed training times (seconds, mean $\pm$ sample SD) are 362.8157 ± 21.1846 for CoGraC, 304.7117 ± 17.8462 for Local, 324.8813 ± 8.1603 for Q48, 427.0383 ± 38.9879 for continuous coding, 92.0443 ± 3.0870 for Fixed, and 256.3457 ± 28.5209 for GPR64.

B.4 Internal query factors and global-response features

We freeze the five source-trained compilers and fit seven fresh readouts per compiler with identical initializations and 24,000 updates on the same source stream. Decoded factors execute at the four standard loads with their original Q and K amplitudes.

Factor and feature interventions. The response is $R_{q,\lambda} = J_q S_\lambda^{-1} J_q^\top + D_q$. Each factor intervention retains Q, the connections, and public graph statistics. No lifting zeros the internal-endpoint rows of J_q while keeping boundary selectors and D_q; no residual sets $D_q = 0$. The Q-only interface maps each internal endpoint uniformly to its component's ports with weight $1/p$ and sets $D_q = 0$. These four conditions omit alignment. Their boundary-query responses are unchanged. Each intervention trains a separate readout on mixed boundary and internal queries, so the learned readout parameters can yield different boundary AP even with identical boundary features.

The three 16-field groups are global responses G, local alignment A, and public graph statistics P. Alignment compares endpoint-to-port responses along direct bridges; G evaluates the assembled system (Appendix A.5). We compare $G + A + P$, $G + P$, $A + P$, and P with identical readout architectures, optimization, and final-checkpoint evaluation. Each condition fits feature moments on the first 24 source entries and zeros excluded channels after standardization. Alignment uses its unexpanded 16 fields.

The complete $G + A + P$ interface has the highest macro AP in both pools, with positive gains over $A + P$ in every seed and on 22 of 24 evaluation panels.

B.5 Training route and execution interface

Five Local-trained and five Full-trained compilers receive fresh $A + P$ and $G + A + P$ readouts while their parameters remain frozen. Within each seed, heads share byte-identical initializations, the 24,000-update source stream, input widths, and normalization. Both compiler routes use the same prediction labels. Local training includes $\tanh(A)^2$ fields; the refitted common interfaces use the unexpanded alignment for both routes.

Global responses improve both training routes in every seed of both pools. For Local-trained compilers, the paired gains are 2.799 ± 0.379 and 1.870 ± 0.380 AP points. At the common $G + A + P$ interface, Full training adds 0.329 ± 0.399 and 0.331 ± 0.408 points, positive in four of five seeds in each pool.

Table 8: Frozen-interface AP: each compiler averages panels within the indicated query group, followed by five-seed mean and sample SD. The bottom block gives paired AP gains in percentage points (internal queries for factor interventions; all queries for feature additions), positive in every seed.

	Original			New-node		
Interface	All	Boundary	Internal	All	Boundary	Internal
$G + A + P$	$.7148_{\pm.0026}$	$.7789_{\pm.0036}$	$.6507_{\pm.0037}$	$.6933_{\pm.0021}$	$.7740_{\pm.0027}$	$.6126_{\pm.0033}$
$G + P$ (full factors)	$.5744_{\pm.0106}$	$.5754_{\pm.0185}$	$.5735_{\pm.0084}$	$.5566_{\pm.0095}$	$.5666_{\pm.0204}$	$.5466_{\pm.0072}$
$A + P$	$.6856_{\pm.0028}$	$.7429_{\pm.0036}$	$.6283_{\pm.0029}$	$.6681_{\pm.0041}$	$.7374_{\pm.0035}$	$.5987_{\pm.0049}$
P	$.5073_{\pm.0015}$	$.5071_{\pm.0021}$	$.5076_{\pm.0012}$	$.5087_{\pm.0025}$	$.5052_{\pm.0034}$	$.5121_{\pm.0016}$
No lifting + P	$.5760_{\pm.0053}$	$.6157_{\pm.0106}$	$.5364_{\pm.0043}$	$.5778_{\pm.0053}$	$.6314_{\pm.0098}$	$.5242_{\pm.0019}$
No residual + P	$.5839_{\pm.0099}$	$.6102_{\pm.0049}$	$.5576_{\pm.0166}$	$.5727_{\pm.0052}$	$.6203_{\pm.0037}$	$.5251_{\pm.0114}$
Boundary-only (Q) + P	$.5583_{\pm.0019}$	$.6172_{\pm.0035}$	$.4994_{\pm.0008}$	$.5582_{\pm.0040}$	$.6158_{\pm.0080}$	$.5006_{\pm.0011}$

Paired gain (points)	Original	New-node
Full factors − no lifting	$3.71_{\pm1.07}$	$2.25_{\pm0.71}$
Full factors − no residual	$1.59_{\pm1.20}$	$2.15_{\pm0.99}$
Full factors − Q only	$7.41_{\pm0.86}$	$4.61_{\pm0.80}$
G added to $A + P$	$2.92_{\pm0.35}$	$2.52_{\pm0.44}$
G added to P	$6.71_{\pm1.01}$	$4.79_{\pm0.82}$

Table 9: Training route × execution interface. AP (%, mean ± sample SD over five compiler seeds), averaging twelve panels per pool.

Compiler training	Readout	Original	New-node
Local	$A + P$	68.349 ± 0.457	67.130 ± 0.422
Local	$G + A + P$	71.148 ± 0.199	68.999 ± 0.357
Full	$A + P$	68.562 ± 0.281	66.806 ± 0.409
Full	$G + A + P$	71.477 ± 0.257	69.330 ± 0.215

B.6 EXTENDING THE GPR64 TRAINING BUDGET

For seeds 107001–107003, GPR64 continues from 24,000 to 48,000 updates by replaying the same source stream with restored model, optimizer, and random-generator states. The learning rate decays from 0.0002 to 0.00002; weight decay is 10^{-4} and gradient clipping is 5. Final-checkpoint comparisons use the same three seeds and panels (Table 10). CoGraC remains ahead by 8.16 ± 0.73 and 7.68 ± 0.56 AP points on Original and New-node, positive in every paired seed.

Table 10: GPR64 budget extension on the same three seeds. Metrics average all twelve panels per pool within each seed; AP subscripts give sample SD across seeds.

Pool	Method	AP ↑	AUC ↑	BCE ↓
Original	CoGraC 24k	$0.7164_{\pm 0.0010}$	0.7426	0.5826
	GPR64 24k	$0.6340_{\pm 0.0099}$	0.6492	0.6561
	GPR64 48k	$0.6348_{\pm 0.0078}$	0.6546	0.6543
New-node	CoGraC 24k	$0.6974_{\pm 0.0006}$	0.7318	0.5882
	GPR64 24k	$0.6203_{\pm 0.0097}$	0.6396	0.6621
	GPR64 48k	$0.6205_{\pm 0.0061}$	0.6428	0.6623

C FROZEN-COMPILER TRANSFER FOR REAL-GRAPH LINK PREDICTION

Visible graphs and support. We freeze the source compiler and fit readouts on Computers and Physics. PubMed, USA, and email-Eu-core (Yin et al., 2017; Leskovec et al., 2007) are held out from compiler development. Each of three target support draws contains 20 positive and 20 negative pairs. All query and support positives are removed before computing features or extracting the edge-owned patches of Appendix A.8. Support and query pairs are disjoint and absent from the visible graph. The query sets contain 4,432 positive and 4,432 negative pairs on PubMed, 1,359 and 1,309 on USA, and 512 and 512 on email-Eu-core, respectively. Negatives are nonedges of the original graph. PubMed and USA replace one endpoint of each positive pair, matching its degree decile in the message graph and expanding to neighboring deciles when needed; email-Eu-core samples node pairs uniformly and rejects edges, self-loops, and duplicates.

Readout fitting and adaptation. CoGraC, Fixed Operator, and Public/path only use the common 48-field interface. Public/path only retains the 16 public graph statistics and nine path features, with response channels set to zero. All readouts have two hidden layers of width 128. Each source graph supplies disjoint fitting and validation sets, each containing 256 positive and 256 negative pairs. All positive edges in both sets are removed before feature computation. Three source readouts train for exactly 500 Adam (Kingma & Ba, 2015) updates at learning rate 0.001, with feature standardization fitted only on source fitting pairs. Source validation is used only for reporting. Target adaptation fits a classifier on centered readout features, using ridge 0.01, temperature 0.25, and support labels alone. Compiler parameters remain frozen throughout. The three compiler seeds are 107001–107003 and the source-readout seeds are 107231–107233.

NCN (adapted) reuses the final 24,000-update checkpoints from Appendix B for seeds 107001–107003. We freeze its GNN and common-neighbor predictor, and use the 64-dimensional pair representation before the final linear prediction layer. On each visible graph, the 16 public input fields retain their source definitions, with component membership and port roles supplied by the edge-owned decomposition. Source readout fitting and target adaptation follow the protocol above.

For CoGraC and NCN (adapted), we compute metrics separately for each of the nine readout/support combinations and average them within each backbone. Table 11 reports means across three backbones, with sample SD for AP. Fixed Operator and Public/path only each have one fixed representation and the same nine combinations. Common neighbors (CN), Adamic–Adar (AA) (Adamic & Adar, 2003), and resource allocation (RA) (Zhou et al., 2009) rank pairs directly from the same visible graph without target support. Their BCE uses source-only logistic calibration.

Table 11: Frozen transfer and classical references. AP subscripts are sample SD across three source-trained backbones, after averaging readouts and support draws. † denotes a fixed representation with no backbone-seed SD. Classical AUC/AP use raw scores; BCE uses source-only calibration.

Target	Method	AP ↑	AUC ↑	BCE ↓
PubMed	CoGraC	$0.8442_{\pm 0.0014}$	0.8357	0.5302
	NCN (adapted)	$0.6745_{\pm 0.0182}$	0.6265	0.7752
	Fixed Operator†	0.8396	0.8326	0.5341
	Public/path only†	0.8243	0.8231	0.5632
	CN	0.6043	0.6079	0.9531
	AA	0.6073	0.6082	0.8884
	RA	0.6076	0.6081	0.6866
USA	CoGraC	$0.7309_{\pm 0.0022}$	0.7383	0.6346
	NCN (adapted)	$0.6355_{\pm 0.0438}$	0.6339	0.7035
	Fixed Operator†	0.7217	0.7319	0.6394
	Public/path only†	0.7027	0.7114	0.6621
	CN	0.6795	0.6518	2.5867
	AA	0.6876	0.6651	6.0791
	RA	0.7153	0.7081	0.8679
email-Eu-core	CoGraC	$0.9274_{\pm 0.0010}$	0.9440	0.3117
	NCN (adapted)	$0.8485_{\pm 0.0687}$	0.8866	0.4362
	Fixed Operator†	0.9240	0.9429	0.3127
	Public/path only†	0.9111	0.9371	0.3248
	CN	0.9395	0.9467	0.7079
	AA	0.9501	0.9521	0.8364
	RA	0.9607	0.9593	0.3700

D SUPERVISED NODE AND GRAPH CLASSIFICATION

D.1 CLASSIFICATION INTERFACE AND CORA CONTROLS

Programs and attribute inputs. Classifiers learn the attribute right-hand sides (RHSs), local compiler, and readout jointly from scratch. They use the edge-owned decomposition of Appendix A.8 and the conditional factors and dictionaries of Appendix A.4. Each node has baseline grounding 0.1, apportioned among its owners; the component amplitude is the mean trace scale of its fixed owned energy. For classification, G_j includes grounding shares $\rho_{j,u} = 1/c_u$, where c_u is the number of components containing u in the visible-graph decomposition. The following construction replaces the grounding and normalization in Appendix A.1. The encoder scales the visible edge weights and the apportioned grounding at every node, including ports, by learned positive multipliers. We normalize the resulting energy H_j^θ as $M_j = H_j^\theta / a_j$ before conditional factorization. Programs depend on local structure, and attributes enter through the RHSs.

An attribute frontend converts node features into an input matrix F. Joint execution produces the response matrix Y. The blocks of F and Y at global ports are F_B, Y_B; those at component j's internal nodes are $F_{j,I}, Y_{j,I}$. The attribute solve is

$$\begin{aligned} S_\lambda Y_B &= F_B + \sum_j A_j^\top \widehat{T}_j^\top F_{j,I}, \\ Y_{j,I} &= \widehat{T}_j A_j Y_B + (a_j \widehat{K}_j)^{-1} F_{j,I}. \end{aligned} \tag{20}$$

The first line maps internal inputs to ports and combines them with F_B to solve for Y_B. The second recovers internal responses from port values and each component's stored factors. Each global port input is injected once, including when several components share its address.

For assembled energy E and port mask D_B, row normalization divides $(E + \lambda D_B)^{-1} F$ by $\mathrm{clamp}((E + \lambda D_B)^{-1}\mathbf{1}, 10^{-8})$, broadcasting across channels. Finite decoding and the straight-through estimator follow Appendix A, with a relative reconstruction penalty against a stopped continuous reference.

Repeated execution and matched controls. Cora uses the Planetoid public split (Yang et al., 2016): graph structure and unlabeled attributes are visible, and cross-entropy uses the training mask. A two-layer MLP produces class logits F. With $R = (E + 0.1 D_B)^{-1}$ and $\Pi = \mathrm{diag}(\mathrm{clamp}(R\mathbf{1}, 10^{-8}))^{-1} R$, the three-step classifier predicts

$$Z = \sum_{j=0}^{3} c_j \Pi^j F. \tag{21}$$

The shared coefficients initialize to $(0.1, 0.09, 0.081, 0.729)$. One compiled energy set and its constant-RHS response serve all depths with gradients intact. The one-step control uses two mixture coefficients. Local keeps all three intermediate states inside each component and averages final outputs at shared nodes; Full assembles and solves jointly. Fixed Operator uses the original grounded operator with three-step joint execution, and the attribute control uses the same MLP directly. Full and Local train under their respective execution rules.

Optimization and selection. All six Cora methods share the attribute frontend and optimizer settings: width 64, dropout 0.5, AdamW learning rate 0.01, and weight decay 5×10^{-4}. Components contain at most 16 nodes. Compiler parameters use 0.1 times the learning rate, zero weight decay, and relative penalty weight 0.002. Training runs for at most 500 epochs, with validation every five epochs and patience 80. GPR (Chien et al., 2021) uses ten propagation steps. Configurations are selected by mean validation accuracy over development seeds 120021 and 120022. They are fixed before the validation-only initialization check on seed 120041 and evaluation on seeds 121101–121105. Validation selects the checkpoint within each run; evaluation runs use the public Planetoid test split. Independent full-coordinate checks agree with repeated program execution to 2.22×10^{-16} in outputs and 3.55×10^{-15} in continuous energy gradients.

Full improves mean accuracy by 2.02 points over Local and 1.38 over one-step execution, with both gains positive in every initialization (Table 12). Its mean gain over Fixed Operator is 1.54 points

Table 12: **Cora matched execution controls.** Test accuracy (%, mean ± sample SD over five initializations). All methods share the attribute frontend and optimizer settings.

Method	Test accuracy
Attribute readout	55.90 ± 0.51
Fixed Operator, three steps	80.18 ± 1.08
CoGraC-Local, three steps	79.70 ± 0.80
CoGraC-Full, one step	80.34 ± 0.36
CoGraC-Full, three steps	81.72 ± 0.59
GPR (matched frontend)	$\mathbf{83.06 \pm 0.69}$

and is positive in four of five runs. These controls quantify the contribution of learned programs and their execution; Appendix D.3 evaluates complete node-classification configurations.

D.2 Peptide function prediction with a shared chemical frontend

Common frontend and execution interfaces. Peptides-func (Dwivedi et al., 2022) uses the official LRGB split and macro AP for checkpoint selection and reporting. Each molecule is an independent system. Categorical atom and bond embeddings feed two residual GINE layers of width 32 with BatchNorm and ReLU. This whole-molecule frontend supplies F to learned Full, learned Local, fixed Local, and zero response. Full assembles the global system; Local solves components independently and averages responses at shared nodes before row normalization. The fixed local response uses the original grounded operator in the local executor, and zero response fills response channels with zero.

Each model concatenates F with responses at loads $\{0, 0.1, 1, 10\}$, applies a projection with LayerNorm and ReLU, and concatenates mean and sum pooling for prediction. A parallel head reads pooled F directly; the final prediction adds its output to the response-head output multiplied by a learned scale initialized to 0.1. Full and Local have identical architectures, initialization rules, and 30,789 parameters. Each jointly trains its whole-molecule frontend, compiler, and heads under its execution rule.

Training and references. The shared-frontend models use components of at most eight nodes, batches of 32 molecules, AdamW learning rate 0.003, weight decay 10^{-4}, zero dropout, gradient clipping at five, and relative penalty weight 0.002. Training follows a 120-epoch cosine schedule, validation every five epochs, and patience 40. The task loss averages valid graph-label entries and the auxiliary averages components. Training-batch membership is seed-fixed, with shuffled epoch order; dictionaries initialize from at most four training batches.

The standalone GINE (Hu et al., 2020) uses five layers with BatchNorm, residual connections, and layerwise bond embeddings (learning rate 0.001). The attribute reference combines atom embeddings with mean incident-bond embeddings, followed by a two-layer MLP, pooling, and zero response slots. Both use AdamW with the batch size, weight decay, dropout, clipping, 120-epoch schedule, and validation rule above. GINE trains on an A40 GPU; the other models use one CPU thread.

The six configurations are fixed after two development initializations. Local is the validation-selected application, and Full is its prespecified joint-execution comparison. All configurations use the same five fresh seeds, 123351–123355; reported means include every run. Full and Local each complete 40,800 updates, taking 22,285 and 22,498 seconds on average. Training checks verify identical initial states, the execution routes, and nonzero compiler task gradients.

Results. Full improves test AP over Local by 2.25 points, zero response by 3.62, and GINE by 5.86, each positive in every initialization. Full–Local validation AP improves by 3.28 points, also positive in all five runs. The 2.65-point gain over fixed Local combines operator learning and joint execution and is positive in four runs. Full also attains the highest mean AUC, 86.66%. Table 13 reports AUC and BCE as auxiliary metrics; the primary AP comparison is in Table 3.

Table 13: Peptides-func supplementary metrics. Means over the same five initializations as Table 3; AUC is in percent.

Method	AUC ↑	BCE ↓
Atom/bond attribute readout	83.22	0.25465
GINE	85.23	0.25211
Frontend + zero response	85.48	0.24243
Frontend + fixed local response	85.87	0.24901
Frontend + CoGraC-Local	86.16	0.25034
Frontend + CoGraC-Full	86.66	0.24986

D.3 Node-classification applications

Selection and evaluation. For each dataset and method, mean validation accuracy over two development initializations selects one complete configuration, including its training and output settings. This configuration stays fixed across evaluation runs; validation accuracy selects checkpoints. Cora uses the public Planetoid split and five new training seeds, 123411–123415. Photo (Shchur et al., 2018) uses three new seeds, 123031–123033, on each of five stratified splits (62001–62005), with 4,590 training, 1,530 validation, and 1,530 test nodes per split. Table 14 evaluates the final configurations defined below. Here, GPR denotes the complete GPR-based predictor, including graph smoothing on Cora and Correct-and-Smooth (C&S; Huang et al., 2021) on Photo. CoGraC uses learned-operator C&S on Cora and direct softmax outputs on Photo. The matched Cora controls in Appendix D.1 separately compare their constituent mechanisms.

PolyFormer. We use the authors' monomial-basis implementation (Ma et al., 2024) on the same graphs, attributes, and splits. Cora uses width 256, polynomial order 6, and four attention heads; Photo uses width 64, order 10, and one head. Both use Adam for at most 2,000 epochs, with validation every epoch and patience 250. The main and attention learning rates are $(0.0005, 0.001)$ for Cora and $(0.0005, 0.0005)$ for Photo; corresponding weight decays are $(0.001, 10^{-5})$ and $(5 \times 10^{-5}, 10^{-4})$. The implementation's `dropout` and `dprate` are $(0.9, 0.6)$ and $(0.2, 0.5)$, respectively. Cora applies ordinary-graph smoothing with coefficient 0.5 and no correction. Photo blends predictions with a propagated training-label field at weight 0.6. Both output procedures use 50 propagation steps with row-normalized $A + I$ and training labels only.

Table 14: Node-classification test accuracy (%) for validation-selected final configurations. Cora reports mean and sample SD over five training seeds. Photo first averages five splits within each seed, then reports mean and sample SD across three seeds.

Dataset	Method	Test accuracy
Cora	CoGraC	82.66 ± 0.72
	GPR	82.34 ± 0.53
	PolyFormer	80.76 ± 0.87
Photo	CoGraC	95.268 ± 0.080
	GPR	95.333 ± 0.114
	PolyFormer	95.024 ± 0.042

Photo: eight response applications. In the final CoGraC configuration, each decoded component receives its ownership share of $3I$, giving assembled response $R = (E+3I)^{-1}$. Loaded components are recondensed for the port solve, with no extra port load. A learned polynomial $Z = \sum_{j=0}^{8} c_j \Pi^j F$, with $\Pi = \mathrm{diag}(\mathrm{clamp}(R\mathbf{1}, 10^{-8}))^{-1} R$, reuses these energies at every depth. Inputs are row-wise ℓ_2 normalized; training uses CG, component size 16, MLP width 64, dropout 0.2, AdamW learning rate 0.01, weight decay 10^{-4}, and relative penalty 0.002. Compiler parameters use 0.1 times the learning rate and zero weight decay. The maximum is 300 epochs, with validation every five epochs and patience 100. Development seeds 123021 and 123022 select the configuration; its output uses softmax directly.

The final GPR configuration uses ten propagation steps, the same input normalization, width, dropout, and training horizon, learning rate 0.03, and weight decay 5×10^{-4}. Its final scores are computed by ordinary-graph C&S with correction and smoothing coefficients 0.5 and correction scale one. Mean training times are 2,163.80 s for CoGraC and 81.84 s for GPR. Appendix E.2 measures reuse of these trained models.

Table 15 averages three initializations per split; the resulting split-level SDs are 0.427, 0.719, and 0.722 points for CoGraC, GPR, and PolyFormer, respectively. These describe split variation, while Table 14 describes initialization variation. The main application table uses split-level SD for Photo.

Table 15: Photo test accuracy (%) by split for validation-selected final configurations, averaged over three training seeds. Each method uses one fixed configuration across all splits.

Split	CoGraC	GPR	PolyFormer
0	94.771	94.771	94.706
1	95.926	96.580	96.296
2	95.251	95.011	94.837
3	95.338	95.272	94.771
4	95.054	95.033	94.510

Cora: learned-operator correction and smoothing. CoGraC's Cora predictor uses the three-step classifier and training settings of Appendix D.1, followed by learned-operator correction and smoothing. On each visible edge, $\mathrm{softplus}(-E_{uv})$ supplies positive weights for a transition matrix with unit self-loops, symmetric degree normalization, and row normalization. Correction propagates training residuals with training rows clamped; smoothing starts from corrected scores with training rows replaced by one-hot labels. Both stages use 50 updates, coefficient 0.5, and correction scale one. GPR's Cora predictor uses ordinary-graph smoothing with coefficient 0.5 and zero correction.

Mean validation accuracy over seeds 123401 and 123402 selects the final output configuration. Candidates comprise direct predictions, training-label field blends with weights $\{0.2, 0.4, 0.6, 0.8\}$, and C&S using ordinary-graph transitions or, for CoGraC, its learned transition. C&S candidates use correction coefficients $\{0.5, 0.8\}$, scales $\{0, 1, 2\}$, and smoothing coefficients $\{0.5, 0.8\}$; duplicate zero-correction configurations are counted once. Label propagation uses the training mask, and test labels evaluate the fixed predictions.

Sensitivity to component partitioning. Three frozen Cora models (123411–123413) use component sizes 8, 16, and 32 with three label-free tie-breaking choices per size. Edge ownership remains unique and grounding shares sum to one. These forward-only evaluations change layouts, port roles, and port-load placement while keeping trained parameters fixed. Mean validation accuracy spans 81.733–82.067% across the nine layouts (Table 16); individual models span 1.2, 0.8, and 1.0 points.

Table 16: Cora partition sensitivity: validation accuracy (%, mean $\pm$ sample SD over three frozen models). GPR's mean is 80.867% for every layout.

Component size	Partition seed	Components	Ports	Accuracy
8	0	922	1157	81.867 ± 1.301
8	123901	911	1176	82.000 ± 0.721
8	123902	919	1172	81.733 ± 0.757
16	0	612	998	81.867 ± 0.462
16	123901	603	973	81.933 ± 0.808
16	123902	607	986	82.000 ± 0.529
32	0	473	867	82.067 ± 0.306
32	123901	456	861	81.867 ± 0.503
32	123902	457	852	81.800 ± 0.346

E REPEATED EXECUTION: CONNECTIONS AND ATTRIBUTES

E.1 PROGRAM REUSE UNDER NEW CONNECTIONS AND QUERIES

Execution paths and protocol. *Recompile* runs the encoder, selects finite coefficients, and decodes on every call; *Read and decode* reads and decodes a saved program; *Reuse decoded programs* keeps the decoded matrices in memory. All three share the dictionary, interpreter, and readout, and give identical logits in every timing case. We use compiler seed 107001 and three graph draws per size (109301–109303), with eight internal nodes and three ports per component. Each call answers 64 queries at loads $\{0, 0.1, 1, 10\}$. Under both fixed and new connections, queries change on every call; paths are randomly interleaved for 100 calls after two warm-up calls per regime.

The benchmark uses AMD EPYC 7763 CPUs, PyTorch 2.0.0, and one active Torch/BLAS thread per job. Every timed call recomputes local factorizations, assembly, four dense solves, local alignment and public graph statistics, and the readout. The dictionary stays resident; graph and query generation precedes timing. The *Read and decode* mode uses an actual `torch.load` file read with operating-system caching enabled. Both stored-program paths use the same NumPy decoder, converting FP64 output to the common FP32 executor. The inference compiler omits the unused training loss; checks confirm identical payloads and energies to the training implementation at every size.

Table 17: Execution under new connections. Call latency is the mean of three graph-draw medians, with sample SD. Cumulative times average the three draws and include measured preparation and all indicated calls.

Components	Execution	Call latency (ms)	Prepare + first call (ms)	Prepare + 100 calls (s)
6	Recompile	$8.020_{\pm 0.214}$	8.037	0.8368
	Read and decode	$5.717_{\pm 0.291}$	146.541	0.7350
	Reuse decoded programs	$3.525_{\pm 0.072}$	153.903	0.5193
16	Recompile	$11.150_{\pm 0.245}$	11.202	1.1738
	Read and decode	$8.461_{\pm 0.374}$	405.016	1.4487
	Reuse decoded programs	$4.524_{\pm 0.035}$	418.709	0.8909
32	Recompile	$20.026_{\pm 1.351}$	20.216	2.0042
	Read and decode	$14.814_{\pm 1.680}$	404.752	1.9507
	Reuse decoded programs	$9.303_{\pm 0.765}$	419.799	1.3391
64	Recompile	$54.243_{\pm 2.334}$	53.272	5.4571
	Read and decode	$44.670_{\pm 2.458}$	138.604	4.5824
	Reuse decoded programs	$36.753_{\pm 2.362}$	139.821	3.7912

Latency and preparation. Reusing decoded programs speeds up calls under new connections by 2.28, 2.46, 2.15, and $1.48\times$ as component count increases. Preparation includes initial compilation plus decoding for *Reuse decoded programs*, and compilation plus serialization for *Read and decode*; process startup and dictionary loading are outside this accounting. Including preparation, reusing decoded programs saves 24–38% over 100 calls on average; one of the three draws at each of 16 and 32 components has no observed crossover within that window. Calls under fixed connections give 1.48–$2.47\times$ steady-call speedups. Six-component checks over all five training seeds give 2.22–$2.32\times$ across both regimes; the four additional seeds use ten calls each on graph draw 109301.

Numerical scale and execution cost. The four sizes contain $18, 48, 96, 192$ global ports and use 128 right-hand sides per load; the largest recorded relative solve residual is 9.45×10^{-7}. Serialized programs contain coefficient indices, supports, node addresses, and scales, with a shared dictionary of 3,504 serialized bytes. Reuse removes repeated compilation. Under new connections, mean feature time rises from 1.993 to 30.952 ms between six and 64 components, while global solves rise from 1.015 to 4.609 ms. These measurements identify feature construction and assembled execution as the work remaining after compilation is reused. Stage means include all 100 calls per draw and are distinct from the median-based call latencies.

E.2 Reuse under real-graph structure and attribute updates

We evaluate structural and attribute updates with fixed parameters from the Cora and Photo application models in Appendix D.3. All routes use one CPU thread, rotated execution order, and identical input changes for CoGraC and GPR. Cora measures selective recompilation of the complete classifier; Photo measures response-column reuse under the common core-output interface specified below.

Cora: selective recompilation. Using the original graph and partition with seeds 123411–123413, requests cumulatively delete or restore original edges while retaining each component's deterministic spanning forest. Each seed receives six warm-up requests followed by 18 timed requests, cycling through changes to 0.1%, 1%, and 5% of edges. Accuracy uses the existing validation mask. Both CoGraC routes maintain local records, pack components, assemble and factor the updated global matrix, execute the classifier, and rebuild its topology-dependent label operator. Selective recompilation reuses unchanged packed groups; changing any group input invalidates that group to preserve hard coefficient assignments. Both routes use the same full-coordinate LU solver for the decoded energy whose block elimination gives the port equations. GPR rebuilds its propagation state from the updated graph.

Table 18: Cora structural updates: means of per-seed median latency and request accuracy, with six timed requests per fraction and seed. Recompiled fractions include packed-group invalidation.

Changed edges (%)	Recompiled components (%)	CoGraC full recompilation (ms)	CoGraC selective recompilation (ms)	GPR (ms)	Validation accuracy (%)	
					CoGraC	GPR
0.1	12.19	680.33	135.89	10.78	81.989	81.144
1	32.75	674.15	200.93	10.71	81.978	81.189
5	40.82	683.60	281.29	10.69	81.922	81.156

Selective recompilation yields paired mean speedups of 5.01, 3.36, and $2.43\times$, with zero prediction-score difference across all requests and seeds. An independent dense assembly check gives zero matrix difference and a residual of 8.44×10^{-14}. Table 18 reports complete update costs for both models.

Photo: online response-column reuse. All three selected models (seeds 123031–123033) run on split 62001. Each replay contains 64 requests updating 1% of nodes, chosen uniformly per request or held to the same repeated set. Requests independently mask each selected node's original feature coordinates with probability 0.1, renormalize its row, and retain other nodes' current attributes. Both methods admit a response-column cache after observing the same changed-node set on two consecutive requests. The policy uses past and current addresses and retains one set of at most 2% of graph nodes. Cache misses and column construction count toward every cumulative total.

For the learned polynomial response $M = \sum_j c_j \Pi^j$ and changed nodes $\mathcal{U}$, fixed structure permits the exact update

$$Z_{\text{new}} = Z_{\text{old}} + M_{:,\mathcal{U}}\big(F_{\text{new},\mathcal{U}} - F_{\text{old},\mathcal{U}}\big), \tag{22}$$

where F contains attribute-frontend outputs. CoGraC's factor cache retains LU factors and normalization; GPR's CSR cache retains its propagator. Both receive the same opportunity to cache columns, using 2.218 MiB for the repeated set.

Repeated-set reuse reduces CoGraC's cumulative time from 15.888 to 5.975 s and reduces per-request latency to 2.094 ms. Under uniform requests, the same policy retains factor-based execution because no node set meets its admission condition. Table 19 includes both workloads and all preparation and request costs. CoGraC's factor-cache and response-column-cache outputs have zero classification disagreements and agree within absolute and relative tolerances of 10^{-4}.

Timing scope. Photo measures core inference plus softmax for both methods, which is CoGraC's final prediction interface. GPR's final C&S interface gives 94.771% accuracy on the repeated-set

Table 19: Photo attribute updates with identical local CPU libraries and core inference plus softmax for both models. Latency averages per-seed medians; preparation and cumulative time average per-seed totals. Column cache denotes the online response-column cache. All 64 requests are included. Accuracy scores the original test nodes under the simulated stream.

Workload	Execution	Latency (ms)	Prepare (s)	Prepare + 64 calls (s)	Accuracy (%)
Uniform	CoGraC, factor cache	191.704	3.668	16.003	94.684
	CoGraC, column cache	192.494	3.846	16.250	94.684
	GPR, CSR cache	14.528	0.045	1.013	94.827
	GPR, column cache	14.609	0.053	1.004	94.827
Repeated set	CoGraC, factor cache	192.398	3.492	15.888	94.768
	CoGraC, column cache	2.094	3.942	5.975	94.768
	GPR, CSR cache	14.450	0.046	0.977	94.917
	GPR, column cache	2.537	0.052	0.352	94.917

stream with 155.541 ms response-column-cache latency. Table 19 uses the common core-output interface for all three frozen models. Preparation includes execution caches and initial frontend/output state for a resident service; model/data loading, library staging, and request generation precede measurement for both methods. Photo totals include preparation and every request without discarded warm-up; Cora excludes the six stated warm-up requests.